IMPLEMENTING PROPOSED IEEE 1588 INTEGRATED SECURITY MECHANISM

BY

DRAGOS MAFTEI

B.A. & Sc., McGill University, 2012

THESIS

Submitted to the University of New Hampshire

in Partial Fulfillment of

the Requirements for the Degree of

Master of Science

in

Computer Science

May, 2018

This thesis has been examined and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science by:

Thesis Director, Radim Bartos, Professor of Computer Science

Philip J. Hatcher, Professor of Computer Science

Robert Noseworthy, Chief Engineer, UNH-IOL

On May 1, 2018

Original approval signatures are on file with the University of New Hampshire Graduate School.

ACKNOWLEDGEMENTS

This work was sponsored in part by award 60NANB16D292 from the National Institute of Standards and Technology.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

IMPLEMENTING PROPOSED IEEE 1588 INTEGRATED SECURITY MECHANISM

by

Dragos Maftei

University of New Hampshire, May, 2018

The IEEE 1588 Precision Time Protocol is the industry standard for precise time synchronization, used in applications such as the power grid, telecommunications, and audio-video bridging, among many others. However, the standard's recommendations on how to secure the protocol are lacking, and thus have not been widely adopted. A new revision of IEEE 1588 is currently being developed, which will include revised specifications regarding security. The aim of this thesis is to explore the feasibility of the proposed security mechanism, specifically as it would apply to use in the power grid, through implementation and evaluation.

The security mechanism consists of two verification approaches, immediate and delayed; we implemented both approaches on top of PTPd, an existing open source implementation of PTP. We support the immediate verification security approach using manual key management at startup, and we support the delayed verification security approach emulating automated key management for a set of security parameters corresponding to one manually configured time period. In our experiments, we found that added performance cost for both verification approaches was within 30 μs, and PTP synchronization quality remained intact when security was enabled. This work should increase awareness and accelerate the adoption of the proposed security mechanism in the power industry.

CHAPTER 1
INTRODUCTION

1.1 Importance of Time

Historically, time has been a tricky concept to precisely define, and it remains so to this day. Despite this, for the sake of completeness before considering its importance, it is worth considering a definition. Practically speaking, time is what the clock reads. However, there are several features that we associate with time that can help us arrive at something more meaningful: it does not appear to be a physical thing, it seems to be continuous, it seems to have directionality, always moving 'forward,' and it seems to be objective, existing independent of one's personal consciousness. With these observations, we can make sense of a clear dictionary definition: time is "the indefinite continued progress of existence and events in the past, present, and future regarded as a whole." [4]

Having addressed what time is — albeit rather cursorily — we can address the second, more immediately relevant question posed above: why is it important? Simply put, time is important in order to coordinate actions or events. This coordination implies multiple parties, as in the case of two people meeting somewhere at an agreed time, and can extend to machines as well, as in the case of computers connected over the internet all over the world. However, this coordination does not necessarily have to include multiple parties; it could be for the purpose of doing something with some degree of regularity. Examples range from farming, wherein crops must be planted at the right time of year, to getting weekly exercise, to walking the dog once a day... All of these examples are using time in slightly different ways, which is an important observation as it serves to illustrate three fundamental requirements of any usage of time to coordinate actions or events: there must be a time source, a quantification of time based on the time source, and a way of tracking or counting the quantified units. The source of time must simply be some regularly occurring event, where each event has the same duration. The duration of such events can vary drastically between different time sources, as evidenced by the examples above: the changing of the seasons, based on the earth's orbit around the sun, occurs much less frequently — and thus produces an event of much longer duration — than the passing of a day, based on the rotation of the earth around its own axis. Similarly, stability of the time source (in other words, the degree to which the duration of each event produced by the same time source varies) may vary among time sources. The length of a day may seem as stable as it gets to a typical observer, yet the days are getting progressively longer, due to the slowing of the Earth's rotation around its axis. Compared with the frequency emitted from a crystal oscillator, for example, an Earth day is not very stable. Once we have a sufficiently stable source of time, we must quantify it, or create units based on the regularly occurring event. Revisiting the Earth day example, we call one such event a *day*, and further split the event into smaller units, namely hours, minutes, and seconds. If using a crystal oscillator as a time source, one could pick some number, count that many cycles in the signal captured from the vibrating crystal, and thus a unit of time is born. Finally, we must track these events, or count the desired units; humans have done this with various calendars and clocks for centuries. Today, our source of time is based on the frequency of the radiation emitted by a Caesium-133 atom's electron changing energy levels; we quantified 1 second as 9,192,631,770 cycles of the Caesium-133 atom's radiation (that is how many cycles happened to equal 1 second as it was once quantified based on a measurement of the Earth's rotation in 1958); and finally, we track it using Universal Coordinated Time (UTC), a timescale that counts ticks or seconds as defined by the Caesium atom, but is adjusted every so often with leap seconds to accurately reflect astronomical time (otherwise, the start of each day as informed by UTC time would seem to happen progressively later than the start of each day as perceived by observing the sun — again, due to the Earth's slowing rotation).

1.2 Time Synchronization

Ever since humans figured out how to track time, it has been an important organizational tool for human networks to operate efficiently. Unsurprisingly, it is no less important of a tool in computer networks, as evidenced by the ubiquity and concurrent rise of the time synchronization protocol Network Time Protocol (NTP) [5] alongside the explosion of the Internet. For most computer applications that rely on separate entities being synchronized to the same source of time, only a certain degree of accuracy is required; timestamps on emails, bank transactions, digital signatures, etc. As the ability to measure time has become increasingly more accurate, so have emerged more applications in which increased accuracy is necessary. IEEE 1588 Precision Time Protocol (PTP) was released in 2002 [6] and revised in 2008 [1] and provided a standard for developers to follow in order to satisfy the nanosecond level time synchronization needs of their applications.

1.3 Security in PTP

Given how crucial it can be for certain applications to have precise time synchronization, it follows that in order to protect the functioning of those applications, one should protect the functioning of PTP. Securing PTP was not initially a high priority due to it being utilized in small, private networks and due to the limited number of applications that used it. However, as PTP networks are growing in number and size, securing PTP is becoming more important. The second and current version of the standard is already almost 10 years old; with a new version of the standard currently being worked on, now is an opportune moment to take a closer look at the new security proposal to assess its feasibility.

1.4 Thesis Statement

The focus of this thesis is on establishing the feasibility of the newly proposed security mechanism for IEEE 1588 Precision Time Protocol, specifically, its integrated security mechanism, in the context of usage in the power grid, as specified by the IEEE C37.238 Power Profile [7].

CHAPTER 2
BACKGROUND AND PREVIOUS WORK

In this chapter, we will introduce the background topics and previous work related to this thesis, starting from the broadest topics, and focusing in further and further on the topic at hand. First, we look at the basics of time transfer. Next, a brief comparison of the two main time synchronization protocols, NTP and PTP, followed by a more in depth look at the workings of IEEE 1588 (PTP). We switch the focus onto security, specifically as it applies to timing, before finally delving into the crux of this work, the latest security proposal for IEEE 1588.

2.1 Time Transfer

Time transfer refers to the transfer of data with the goal of synchronizing one or more clocks to a time reference. Although the basics of time transfer are relatively straightforward, it is important to have a solid understanding of them as they are foundational to any discussion of time synchronization protocols. Let us assume, at a given instant in time, a time reference with an accurate clock reading T_{ref}, and a local clock that reads T_{local}, which may be ahead of or behind the reference clock by some amount. This offset of the local clock with respect to the reference clock at a given point in time is defined as $o = T_{local} - T_{ref}$. The goal of time transfer is to synchronize clocks; a means to that goal is to figure out what this offset is, so that the local clock may adjust accordingly to be synchronized with the reference. If the reference could send its time to the local clock in a timestamp t_1 by some unknown means that would have it arrive instantaneously (see Figure 2.1), then the local clock could solve for offset and adjust its clock accordingly.

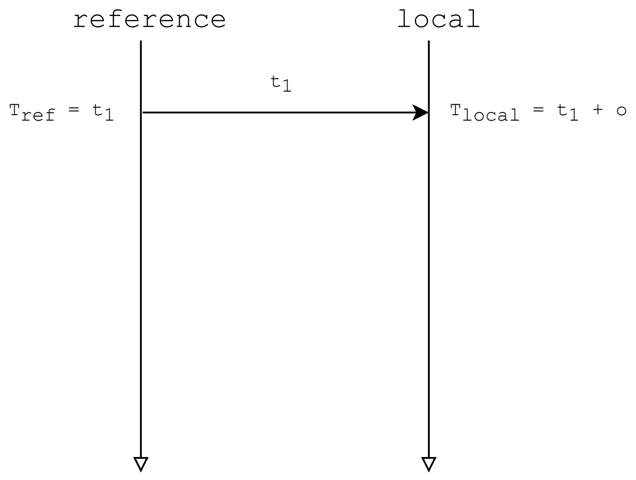


Figure 2.1: hypothetical instantaneous transfer

Of course, there are no such instantaneous means of transferring information given finite signal propagation speed, so in reality we must account for the propagation delay d, as shown in Figure 2.2. If we let t_2 be the local clock's timestamp for the arrival of t_1 from the reference, then $t_2 = t_1 + o + d$, and we can no longer solve for o (unless we also have d).

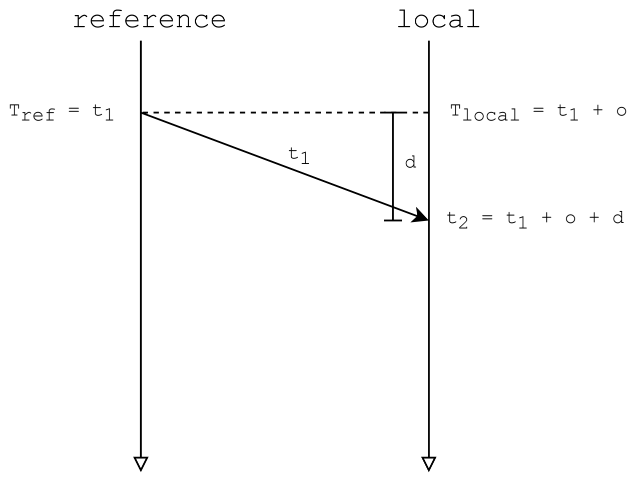


Figure 2.2: accounting for propagation delay d

If, however, we make the assumption that the propagation delay is symmetrical (which is reasonable given a direct wired connection, for example) then we can do a two-way transfer to cancel out the propagation delay and solve for offset. We have already seen in Figure 2.2

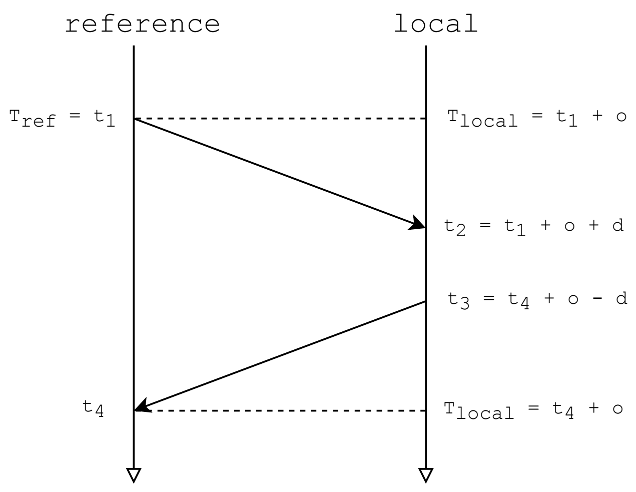


Figure 2.3: two-way transfer

that after receiving t_1 from the reference at local time t_2, we can write our first equality $t_2 = t_1 + o + d$. Figure 2.3 shows the second part of the two-way transfer that leads to our second equality, $t_3 = t_4 + o - d$. If the local side can recover t_4 from the reference (by the reference simply sending t_4 back in one final message), then we can combine the two equations to cancel out offset o and solve for the one-way propagation delay d, as shown in Equation 2.1[1]. If we know the propagation delay, we can then solve for offset as shown in Equation 2.2.

$$d = \frac{(t_2 - t_1) + (t_4 - t_3)}{2} \tag{2.1}$$

$$o = t_2 - t_1 - d \tag{2.2}$$

Obtaining these four timestamps and using them to solve for offset and/or delay is the backbone of time synchronization protocols. Real systems are of course more complicated and nuanced, and this is reflected in the detail found in time synchronization protocol standards documents, but the basic ideas outlined above remain the same.

[1]A critical assumption in this equation is that the propagation delays are symmetric.

2.2 Time Synchronization Protocols

There are currently two main protocols used to coordinate time over a network: *Network Time Protocol* (NTP) and *Precision Time Protocol* (PTP).

NTP was originally published in 1985 [5] with the goal of synchronizing network clocks over a distributed network of servers and clients. With the rise of the Internet, there grew a need to synchronize time over the network, and NTP was the natural choice for doing this, as it provided better performance than the existing approaches [5]. NTP has the ability to synchronize to within tens of milliseconds, which has proved sufficient for its most common use cases, which besides general purpose use (having the correct time reflected on one's computer) includes virtually any application that depends upon coordinated timing over a network.

IEEE 1588 (PTP) was published in 2002 with the goal of providing more precise time synchronization than that available via NTP, for applications that required sub-microsecond level synchronization, such as instrumentation, industrial automation, and military applications [8]. Such improvements in precision compared to NTP were achieved largely by the use of hardware timestamping, and due to PTP's use in contained, private networks. The second version was published in 2008 and is the basis for today's current PTP applications, including telecommunications, power grid, audio video bridging, high-speed trading, self driving cars, etc.

Some of these applications require unique customizations to the standard, which are documented as *PTP profiles*: The power profile [7], telecom profile [9] [10], and gPTP profile [11] are three such profiles that are in wide use in their respective applications. A new revision of the standard is being worked on, expected to be released in 2018.

Both NTP and PTP follow a server/master — client/slave model. Aside from the obvious difference in precision, another notable difference between the two protocols is in their respective domains of application, with NTP being used primarily over the internet (WAN) while PTP was intended for private networks (LAN). This meant that initially, security was not a serious concern for PTP networks as they were smaller, localized, and generally protected from attacks. However, as more systems are beginning to utilize PTP, they are becoming increasingly higher risk targets, thus warranting a renewed effort at properly securing the protocol [8]. Before looking at security in timing and the current state of security in PTP however, we will go into a brief overview of the IEEE 1588 protocol.

2.3 IEEE 1588 Overview

In this section we will give a brief overview of IEEE 1588, covering port states, clock types, step modes, message types, delay calculation, and profiles.

2.3.1 Port States

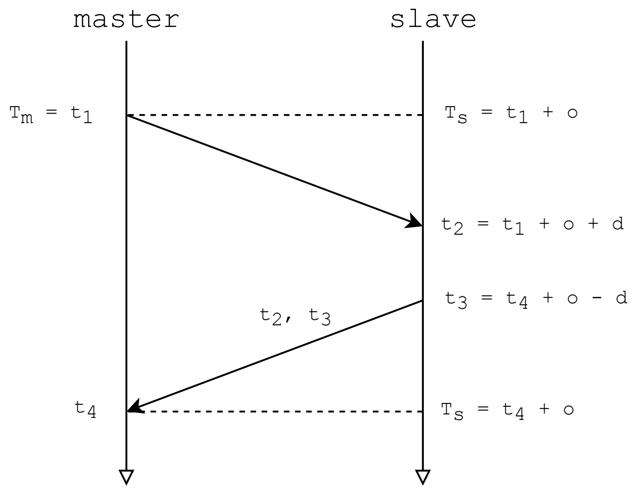


Figure 2.4: two-way transfer

PTP port states define the role or function that a port plays in the exchange of messages. The most important (and most common) states are master and slave, analogous to server and client in NTP. A port in the master state will be the reference time for other (slave) ports, which synchronize to the master. Ports in the master state will initiate most messages, and ports in the slave state will respond to them, and synchronize their clocks accordingly.

There are a few other transient states, but the important ones to note are Master and Slave. Knowing this, we can update Figure 2.3 from earlier to replace "reference" with "master" and "local" with "slave", as shown in Figure 2.4.

2.3.2 Clock Types

In IEEE 1588 language, a 'clock' refers to a network device, or a node, that actively participates in the PTP network. As shown in Figure 2.5, a clock maintains several data sets, has one or more ports, a protocol engine, and a local clock (in the more traditional sense) that keeps the time.

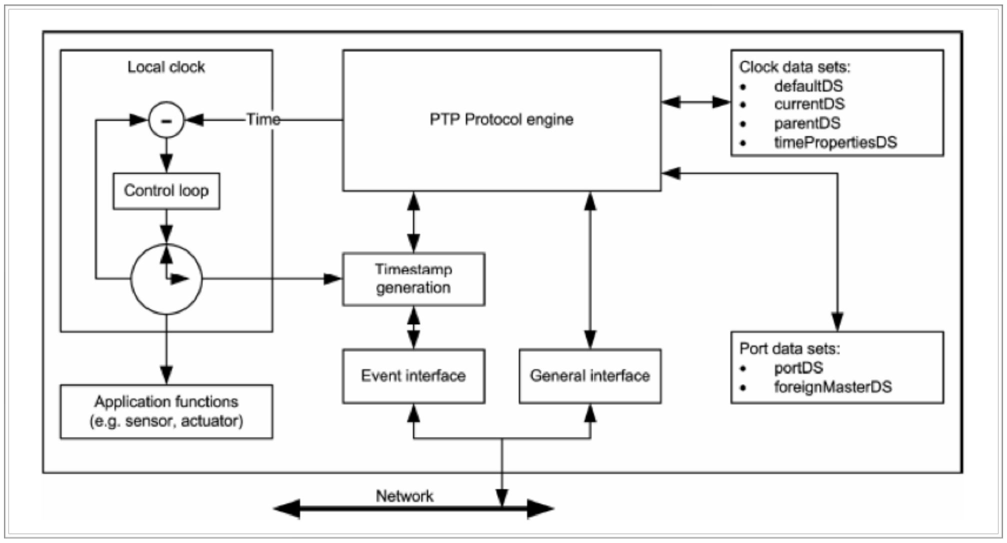


Figure 2.5: IEEE 1588 clock [1]

There are three main clock types, which serve to differentiate nodes in a PTP network based on topology: *Ordinary Clock* (OC), *Boundary Clock* (BC), and *Transparent Clock* (TC). Ordinary clocks are end devices, in that they may serve as a source of time, i.e., be a master clock, or may synchronize to another clock, i.e., be a slave clock; PTP messages thus originate and end with them. Boundary clocks and Transparent clocks, on the other hand, are located in the middle of a topology, acting as a switch or a router. Boundary clocks have multiple ports, one acting as a slave, and the others acting as masters (Figure 2.6); the slave port receives incoming messages from an upstream master which are used to set the shared local clock, and then messages are generated from its master port(s) for downstream slaves to synchronize to. Transparent clocks, as the name suggests, pass messages through, and adjust them to account for *residence time* or the time spent passing through the clock. When downstream clocks use these messages, they can account for the residence time — which may be variable from message to message based on how long the message spends in various internal queues — and thus minimize loss of accuracy.

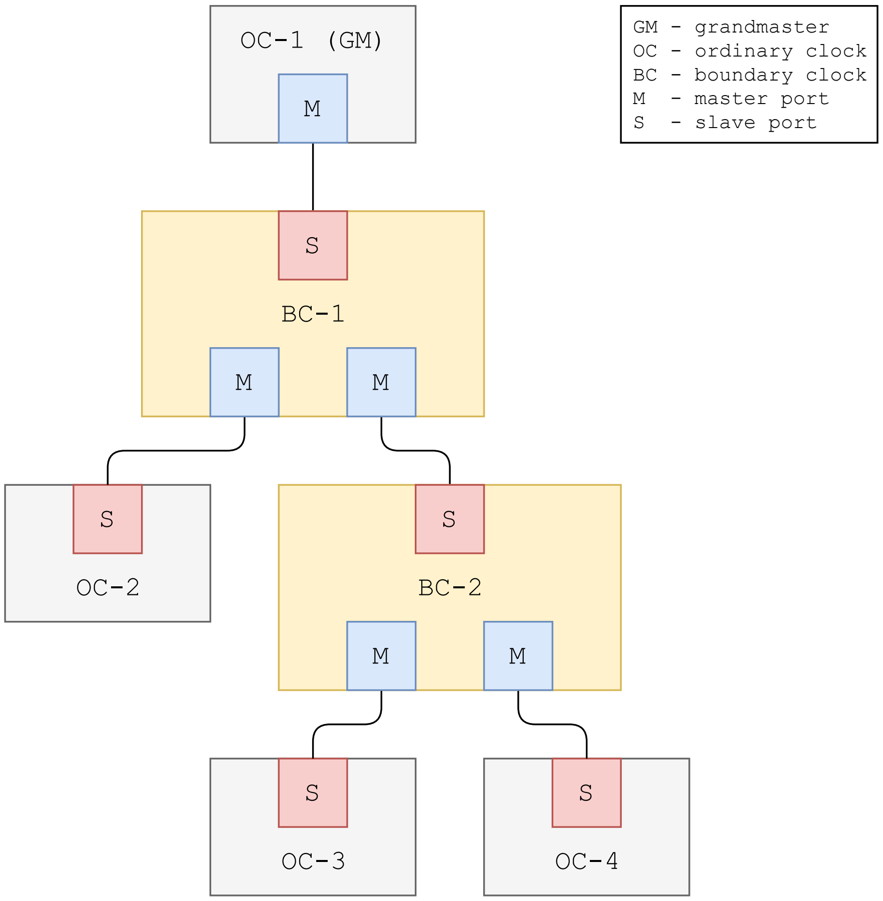
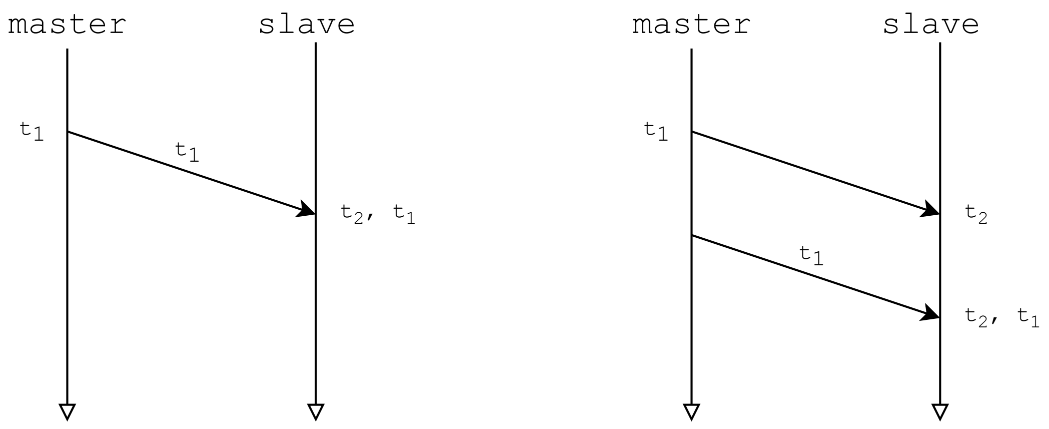


Figure 2.6: clock types

2.3.3 One-Step vs. Two-Step

As we will see in subsequent sections, certain outgoing messages in PTP require egress and ingress timestamps to be collected and transmitted to the receiving node. With respect to this timestamp collection and transmission, there are two different modes of operation that a clock can use. In one-step mode, special hardware is used that can collect a timestamp for an outgoing message and include it in the message itself "on the fly" as it is leaving. In two-step mode, a timestamp is collected as the outgoing message leaves, but is included in a subsequent message. Figure 2.7 illustrates this, with one-step mode shown in Figure 2.7a, and two-step show in Figure 2.7b.



(a) one-step (b) two-step

Figure 2.7: one-step vs. two-step

2.3.4 Message Types

There are ten messages types in IEEE 1588: *Announce*, *Sync*, *Follow_Up*, *Delay_Req*, *Delay_Resp*, *Pdelay_Req*, *Pdelay_Resp*, *Pdelay_Resp_Follow_Up*, *Management*, and *Signaling*. There are a few useful ways to categorize the message types, the broadest being by message class. There are two message classes: *Event* messages, which are required to be timestamped, and *General* messages, which are not. Referring back to Figure 2.7b, we can see that the first message would be considered an *Event* message, since its ingress and egress timestamps were collected. The second message, on the other hand, is a *General* message, as its sole function is to carry timestamp t_1; we do not care about its own egress and ingress timestamps.

The other more natural way to categorize messages is by function: what is each message type used for, and how can they be logically grouped? *Announce* messages are sent by all potential masters, and contain information in them to be used by all other nodes in the network to determine who is the most qualified master, or 'best master.' *Sync* and *Follow_Up* messages are also sent by masters, but are used to establish and transmit timestamps t_1 and t_2 (to be used in offset and/or delay calculations as seen in Section 2.1). The next functional grouping is based on two separate, mutually exclusive methods of calculating propagation delay: either we use the end-to-end method, which use the *Delay_Req* and *Delay_Resp* messages to establish and transmit timestamps t_3 and t_4, or we use the peer-to-peer method, which uses *Pdelay_Req*, *Pdelay_Resp*, and *Pdelay_Resp_Follow_Up* messages independently of *Sync* and *Follow_Up*, to calculate the delay over a single link (these delay calculation methods will be elaborated upon in the following section). Finally, we have *Management* and *Signaling* messages, which are used for other functions beyond the scope of this overview.

2.3.5 Delay Calculation

In IEEE 1588, a slave synchronizes to an upstream master by continuously calculating its offset from the master, and using this value to adjust its local clock accordingly. The offset from master is defined as follows, where all times are measured at the same instant [1]:

$$\text{offsetFromMaster} = \text{time on the slave clock} - \text{time on the master clock}$$

This is shown in Figure 2.1, and when accounting for propagation delay (Figure 2.2), we get Equation 2.2: $o = t_2 - t_1 - d$. As mentioned above, IEEE 1588 offers two methods for calculating this propagation delay d, an end-to-end method, and a peer-to-peer method.

In the end-to-end method (Figure 2.8), the entire link delay between a master and slave is measured, using *Sync*, *Follow_Up*, *Delay_Req*, and *Delay_Resp* messages to obtain timestamps t_1, t_2, t_3, and t_4. In cases where there is a Transparent clock between the master and slave, it adjusts the proper messages for residence time (by modifying a *correctionField* value); for one-step, *Sync* and *Delay_Req* messages get adjusted, while for two-step, *Follow_Up* and *Delay_Resp* messages get adjusted. Once the slave has all four timestamps, the delay is calculated according to a modified Equation 2.1, and offset can be calculated according to a modified Equation 2.2; the equations are modified to account for values in the *correctionField* of the appropriate messages.

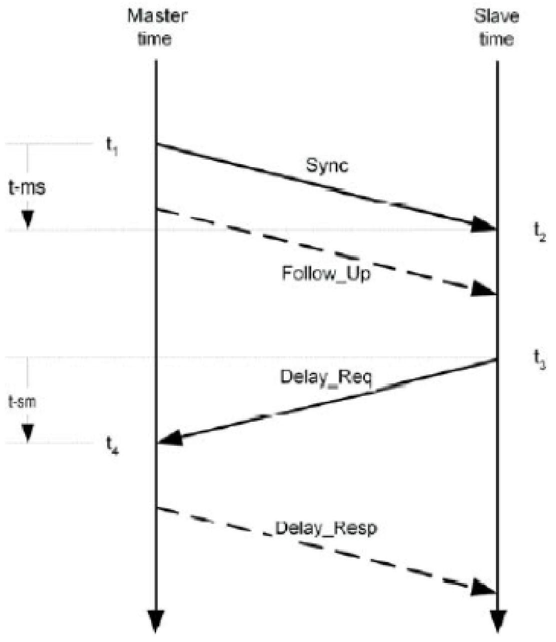


Figure 2.8: delay request-response (end-to-end) [1]

In the peer-to-peer method (Figure 2.9), every node is responsible for calculating the link delay with each of its neighbors, hence the diagram being labeled with *requester* and *responder* rather than *master* and *slave*. Using *Pdelay_Req*, *Pdelay_Resp*, and *Pdelay_Resp_Follow_Up* messages, a requester ends up with a similar set of four timestamps that it can use to calculate propagation delay according to Equation 2.1 (similarly modified as described above, to account for values in the *correctionField*). These calculations happen independently of a master sending out *Sync* and *Follow_Up* messages, between every adjacent pair of ports. When a master does send out *Sync* and *Follow_Up* messages, Transparent clocks adjust these message for the residence time *and* the upstream link delay, so when a slave gets them, they have link delay corrections accumulated in them. These link delays are taken into account, as well as the last link delay, when calculating offset according to a slightly modified version of Equation 2.2 seen above.

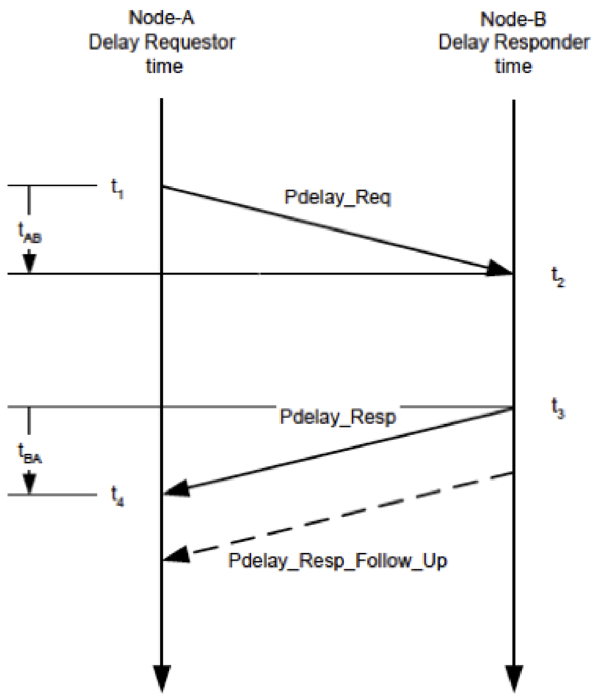


Figure 2.9: peer delay (peer-to-peer) [1]

2.3.6 Profiles

As we have seen in this brief overview, there are quite a few options in IEEE 1588 that allow a PTP network designer some flexibility in setting up the network based available resources, existing infrastructure, or the needs of the given application. Important options include the delay calculation mechanism (end-to-end vs peer-to-peer), whether the network will use one-step or two-step (the former requiring special hardware support), and what the network topology will look like (whether there will be TCs or BCs, etc.). These options will influence what kinds of clocks will be required, what kinds of messages the clocks in the network will use, and how delay and offset will be calculated. Also important is the transport mechanism to be used, most commonly either UDP/IPv4 or Ethernet. As noted in Section 2.2, the specification of which of these options to use or not to use (among a few others not mentioned here) for a given application or industry, is referred to as a *profile*. The work presented in this thesis was done specifically in the context of the power profile [7], as will be described in detail in Section 3.3.

2.4 Security in Timing

Now that we have looked at time transfer in general and PTP in some detail, we can focus on security, as it applies to timing protocols and specifically as it applies to PTP. As usual, the first question to answer when picking up a new topic is "why is this important?" Is security necessary in timing? The reality is that many mission critical applications use PTP: telecom, high speed trading, power grid, etc. PTP is a building block, and so if it fails, so will applications that depend on it. Furthermore, as usage grows, so does exposure to attacks. Therefore, it is fairly clear that security should be considered. If we could add security for free, then of course, we would do it. But are the costs in difficulty of implementation and in performance justified based on the risk level? This is what we will explore in the following chapters. In order to do this, we must first look at what kinds of security threats we may encounter.

2.4.1 The CIA Triad

Cryptography typically has three main goals, which can be abbreviated by the acronym CIA: *confidentiality*, *integrity*, and *authenticity*. This means that for any secure exchange between a sender and a receiver, the message should be encrypted (confidentiality), the receiver should be sure that the message was not modified in transit (integrity), and the receiver should be certain that the message came from the receiver (authenticity). Since the main content of time synchronization protocol messages is time stamps, and since time is not a secret, security solutions to time protocols do not need to provide confidentiality and thus address only the latter two of these components: integrity and authenticity.

2.4.2 Using MAC and HMAC

We can use a message authentication code, or a MAC, to provide message integrity and authenticity. Authenticity can be a special case of integrity, if the message contains a promise of authenticity (e.g., a source address), and so if we can be sure that such a message has not been tampered with, we have integrity, and thus, indirectly, authenticity as well.

The process is illustrated in Figure 2.10.

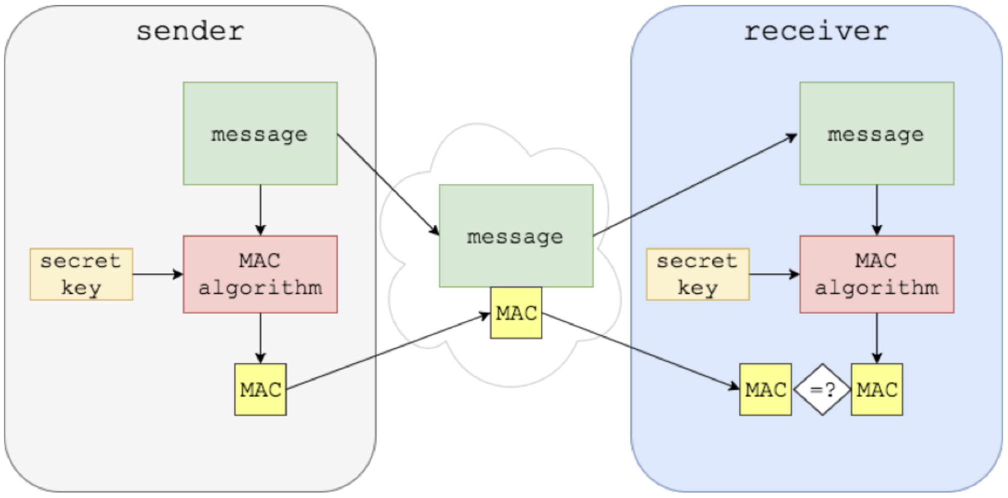


Figure 2.10: using MAC

Prior to beginning, the sender and receiver must share a secret key, and must have agreed on a specific MAC algorithm to use. When sending a message, the sender runs the message and the secret key through the MAC algorithm, which produces a cryptographically strong hash code known as a MAC. The sender sends the message along with the corresponding MAC to the receiver. The receiver runs the received message through the same MAC algorithm using the same secret key, and produces its own MAC for that message. If the calculated MAC matches the received MAC, then the receiver can be sure that the message was created by someone with the secret key, and that the message was not modified in transit. If the message was modified in transit by an attacker who does not have the secret key (and therefore cannot replace the original MAC with a recalculated one that matches the modified message), the receiver will detect this when comparing the received MAC with his own calculated MAC, and thus learn that the integrity and authenticity of the message is compromised (Figure 2.11).

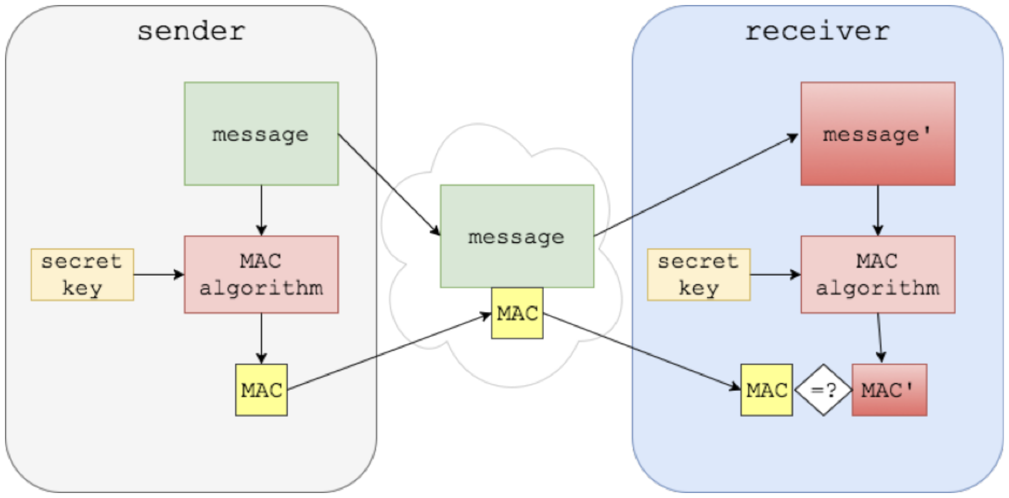


Figure 2.11: detecting a compromised message

We can improve this by using HMAC, which is based on the same logic but uses a hash function in the algorithm, making for a more secure MAC (Figure 2.12). Also of note in Figure 2.12 is that we have renamed the resulting code from "MAC" to "ICV" (integrity check value). This is the terminology that is used in the IEEE 1588 security recommendation documents, and thus will be used here going forward.

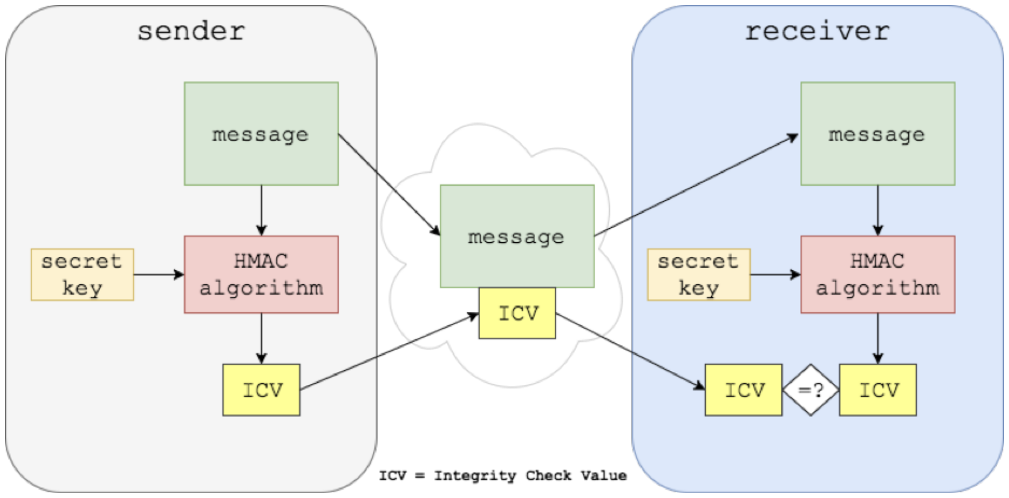


Figure 2.12: using HMAC

However, we must take note that the scope of security in time protocols is not limited to the CIA triad; in 2014, RFC 7384 "Security Requirements of Time Protocols in Packet Switched Networks" was published outlining several other requirements along with their priority levels. In addition to authentication and integrity, *replay protection* is also listed at the priority level "must" and as such should be considered in securing PTP [12].

2.4.3 Annex K

As mentioned previously, security in IEEE 1588 was not a high-level priority in 2008 when the current version of the standard was released, so it is only addressed in the standard as part of an informative, experimental section, Annex K. In keeping with the reasoning outlined above, the goals of Annex K were to provide message integrity and source authentication mechanisms to PTP, as well as replay attack protection [1]. However, due to a number of problems documented by several authors [13] [14] [15], Annex K was not widely adopted. This has led to renewed efforts to define a better security mechanism, which will be included in the next version of IEEE 1588 and forms the basis of this work.

2.5 New 1588 Security Proposal

The security proposal in the new IEEE 1588 standard builds off of Annex K, and maintains the same goals of providing message integrity, source authentication, and replay attack protection. An informative security annex describes a four-pronged approach to security [2]:

- Prong A — deals with integration of security in PTP based on an *AUTHENTICATION TLV* in conjunction with either immediate or delayed security processing
- Prong B — addresses security mechanisms external to PTP such as MACsec and IPsec that can nevertheless still be used to address some of the security requirements for PTP
- Prong C — covers architectural issues, such as network topology choices that can be used to enhance security
- Prong D — addresses security-related issues in monitoring and management

Prong A describes an integrated mechanism that can be built into PTP itself and thus is the core and starting point of the proposed security model as a whole. Also, it is described in detail in the general optional features section of the standard, as opposed to being described only as an informative annex. As such, it is the sole focus of this thesis. More specifically, we aim to explore its feasibility in the context of the power profile.

2.5.1 Prong A Overview

There are three main components to Prong A: the definition and use of an AUTHENTICATION TLV, a verification approach, and an associated key management protocol. TLV stands for Type-Length-Value, and is a common way to encode data in communication protocols. The AUTHENTICATION TLV in this case is used to carry security related information necessary for calculating an ICV in order to provide integrity and authenticity, as described in Section 2.4.2. The verification approach refers to whether ICV verification is done immediately upon reception of a message, or if messages are stored and verified some time later; the former is referred to as immediate verification / security processing, while the latter is referred to as delayed verification / security processing. Finally, there are various parameters and values necessary for doing these calculations, many of which need to be distributed among nodes in the network; the key management protocol refers to the way in which these parameters are established and shared. The realization of the key management protocol is stated to be out of scope by the draft standard. However, due to the interdependence of the verification approach and the key management protocol, certain key management aspects cannot be completely ignored and so they pop up throughout the draft standard. Key management will therefore be discussed here to the extent that it is needed to clarify the usage of the AUTHENTICATION TLV.

2.5.2 Key Management

Key management as described in the security sections of the draft standard is the means by which security parameters are distributed to PTP nodes. These security parameters include keys, key length, integrity algorithm type, ICV length, and verification approach. In the case of delayed processing, there are several other parameters as well that will be discussed later. Key management can be manual or automated. Automated key management protocols use Security Associations (SA) to group these parameters for a given set of senders and receivers, and store such SAs in a Security Association Database (SAD). The SAD thus holds information on *how* a given message gets secured, i.e., which parameters to use. Similarly, a Security Policy (SP) dictates *which* messages require security, and are stored in a Security Policy Database (SPD). Participants in an automated key management protocol will need to store their own copies of the SPD and SAD. The population and management of these databases, the interactions between them, and the specific steps needed to obtain necessary security parameters is out of scope and intentionally left open.

2.5.3 AUTHENTICATION TLV

The structure of the AUTHENTICATION TLV is shown in Figure 2.13. As with any TLV, the first field, tlvType, is the Type (defined here as SECURITY=0x000D) [2], and the second field, lengthField, is the Length of the payload or the Value field of the TLV. The Value field in the AUTHENTICATION TLV consists of several subfields: SPP (security parameter pointer), secParamIndicator, keyID, a conditionally present disclosedKey, sequenceNo and RES (optional, and reserved for use in later revisions), and finally, the ICV.

2	2	1	1	4	D	S	R	K
Type	Length	SPP	secParam Indicator	keyID	disclosedKey (conditional)	sequenceNo (optional)	RES (optional)	ICV

Figure 2.13: AUTHENTICATION TLV (lengths in bytes shown above each field)

The SPP is used to obtain an SA from the SAD. The secParamIndicator is a set of flags indicating the presence or absence of the conditional and optional fields; since sequenceNo and RES are reserved for future editions, it effectively just signals the presence or absence of a disclosedKey, which is only relevant for delayed processing (discussed below). The keyID identifies the key to be used (it is not the key itself, which is stored in the SAD); the details of how it identifies the key depend upon the verification scheme and the associated key management protocol. In delayed processing, certain keys sometimes need to be disclosed via the AUTHENTICATION TLV; when such a key needs to be disclosed, the disclosedKey field is populated with the key. The sequenceNo and RES fields are reserved for future versions and will be ignored hereafter. Finally, the ICV field holds the ICV that protects the integrity and authenticity of the message that its containing AUTHENTICATION TLV is attached to.

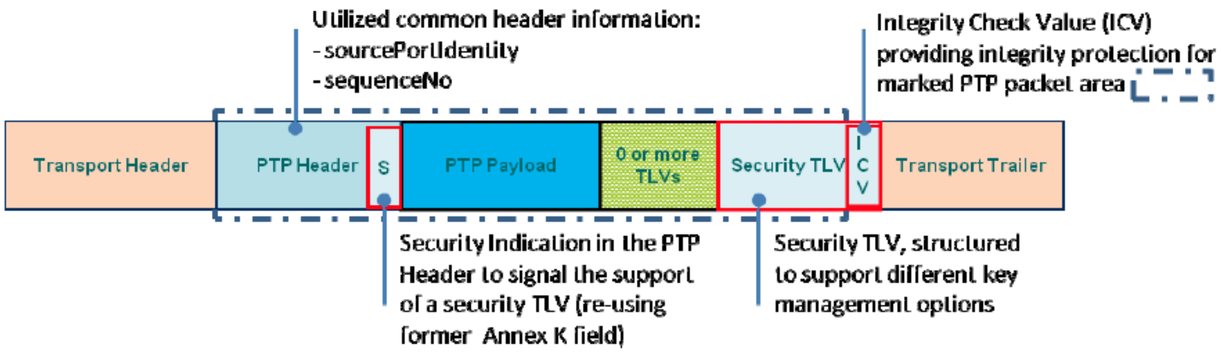

Utilized common header information:
- sourcePortIdentity
- sequenceNo
Integrity Check Value (ICV) providing integrity protection for marked PTP packet area
Transport Header
PTP Header
S
PTP Payload
0 or more TLVs
Security TLV
I C V
Transport Trailer
Security Indication in the PTP Header to signal the support of a security TLV (re-using former Annex K field)
Security TLV, structured to support different key management options

Figure 2.14: packet structure with AUTHENTICATION TLV [2]

The AUTHENTICATION TLV is appended to every PTP message that must be secured. As described in Section 2.4.2, using a secret key and some message as inputs to a given algorithm, a unique ICV can be produced and appended to the message to provide integrity and authenticity; this same model is applied here. Figure 2.14[2] shows the overall packet structure of a PTP message that includes the AUTHENTICATION TLV. The ICV is calculated over the entire PTP message — including the header, payload, and possibly other TLVs — as well as the AUTHENTICATION TLV up to but not including the ICV field itself.

[2]This diagram is taken from the draft standard version that used the term *SecurityTLV* instead of *AUTHENTICATION TLV*.

Having introduced key management and the AUTHENTICATION TLV, we can address control flow: specifically, what are the steps taken on the sending side and on the receiving side in order for this integrated security mechanism to function? Since control flow differs substantially based on the verification approach (immediate vs. delayed) and the closely tied key management, we will look at each one separately.

2.5.4 Immediate Verification

In immediate processing, a key management protocol is utilized by which all members of the network share all the necessary security parameters for calculating an ICV. Thus, both masters and slaves can protect outgoing messages by creating and appending an AUTHENTICATION TLV, and both can, immediately upon reception of such a message, verify the message authenticity and integrity by calculating the ICV and comparing it to the ICV contained in the received message's AUTHENTICATION TLV.

For a sender, the first step is to use the *policy limiting fields* (PLF) to query the SPD in order to obtain the relevant Security Policy. Policy limiting fields include sourcePortIdentity, domainNumber, and messageType, among a few others; these fields should provide enough information in order to determine the security policy, which indicates whether the message needs to be secured. If the message is to be secured, the SPD query should return a Security Parameter Pointer (SPP),[3] which is used to query the SAD and obtain the relevant Security Association (SA) for this message. The SA holds all the necessary parameters for constructing the AUTHENTICATION TLV and calculating the ICV, which in the immediate processing case, are: a keyID (which should indicate a key to use, perhaps as an index into a table), a key (or a key table), key length, integrity algorithm type, ICV length, and a sequenceID window.[4] Knowing that the message is to be secured, the sender can set the

[3]The draft standard indicates that the SPD query should return a *list* of SPPs, pointing to the fact that there may be multiple applicable SAs for a given message, as in the case where multiple AUTHENTICATION TLVs may need to be attached; for simplicity, we have ignored this case in this explanation.

[4]This would be for replay attack protection, which was not considered in this work, in part due to lack of details in the draft standard.

SECURE flag in the flags field of the PTP header, and having all the necessary information from the SA, the sender can begin constructing the AUTHENTICATION TLV: the tlvType is SECURITY; the lengthField is calculated based on ICV length, since there are no optional fields and no other variable length fields when using immediate processing; the SPP is the one obtained previously from the SPD lookup; the secParamIndicator is 0x00 (to indicate the absence of a disclosedKey, which is always the case for immediate processing); and the keyID is provided by the SA. Finally, using the integrity algorithm type and the key given by the SA, the ICV can be calculated over the entire message up to the start of the ICV, and inserted in the ICV field. The AUTHENTICATION TLV for immediate processing is as shown in Figure 2.15.

2	2	1	1	4	K
Type	Length	SPP	secParam Indicator	keyID	ICV

Figure 2.15: AUTHENTICATION TLV with immediate processing (field lengths in bytes)

As with the sending case described above, the first step for a receiver is to use the PLF of the incoming message to query the SPD and obtain the security policy. This should be done before PTP message processing, and after PTP header processing. If the security policy indicates that the message should be processed for security, then the SECURE flag in the message's PTP header should be checked and verified that it is set, indicating that an AUTHENTICATION TLV should be present. The existence of the AUTHENTICATION TLV can then be checked (by verifying the Type field is SECURITY). Next, the SPP field in the received AUTHENTICATION TLV can be verified to match the SPP returned from the SPD query. The SPP can then be used to query the SAD and retrieve the SA for this message, which contains all the necessary security parameters required to calculate the ICV. Although not explicitly stated in the security section of the draft standard, the lengthField should probably be verified to reflect the correct payload length based on the ICV length parameter provided in the SA. Also unspecified (in the immediate case) is if or how the keyID field of the received AUTHENTICATION TLV should be used to identify the appropriate key.[5] Regardless, the receiver should have enough information at this point to use the correct key and integrity algorithm to calculate the ICV over the received message, and compare it with the ICV in the received AUTHENTICATION TLV to determine the integrity and authenticity of the message.

2.5.5 Delayed Verification

The delayed verification approach is dependent on an associated key management protocol that allows for delayed distribution of security parameters. RFC 4082: Timed Efficient Stream Loss-Tolerant Authentication (TESLA) [16] is one such key management protocol, and indeed, the delayed approach described in the security section of the draft standard is designed with TESLA in mind: an Annex to the standard is referenced which describes delayed operation with use of TESLA. As such, we will describe delayed verification here as it would operate when paired with TESLA, or with a key management protocol closely resembling it.

Compared with immediate processing, delayed processing requires a few extra security parameters, and there is more asymmetry between master and slaves: only the master has the parameters (namely, a key chain) necessary to secure outgoing messages, and slaves must be able to buffer messages for later verification. The general idea is that, given a sender and receiver that are loosely time synchronized, and that have agreed upon a way to split time into uniform intervals, the sender uses a new secret key to secure messages sent during each new interval, and discloses each key to the receiver (so the receiver can verify the integrity of the messages secured with that key) only after some delay, after which the sender is no longer using it to secure any new messages. This requires a recursively generated one-way key chain, which will be discussed below.

In the initial bootstrapping phase, the following need to be established first so that time

[5]This may be a key management detail that was considered out of scope.

may be split into uniform intervals:

- a start time T_0,
- an interval duration T_{int}, and
- a number of intervals N

The master then creates a one-way key chain K_0, K_1, ..., K_{N-1}, K_N of size $N + 1$, so that there is a unique key for each time interval, plus one extra key — the *trust anchor* — which is the only key from the chain to be securely distributed to all participating nodes during bootstrapping. Using a pseudorandom function (PRF)[6] f (e.g., a keyed hash function), where $f_k(x)$ denotes hashing over x using key k, TESLA defines the one-way function for key chain construction as follows:

$$F(k) = f_k(0) \tag{2.3}$$

The master picks a random value for K_N, which will be the last key to be used, corresponding to the last time interval N, and recursively generates the keychain by $K_i = F(K_{i+1})$. For example, K_{N-1} will be produced via $F(K_N)$, i.e., using the PRF f to hash over 0 using K_N as a key. This is illustrated in Figure 2.16.

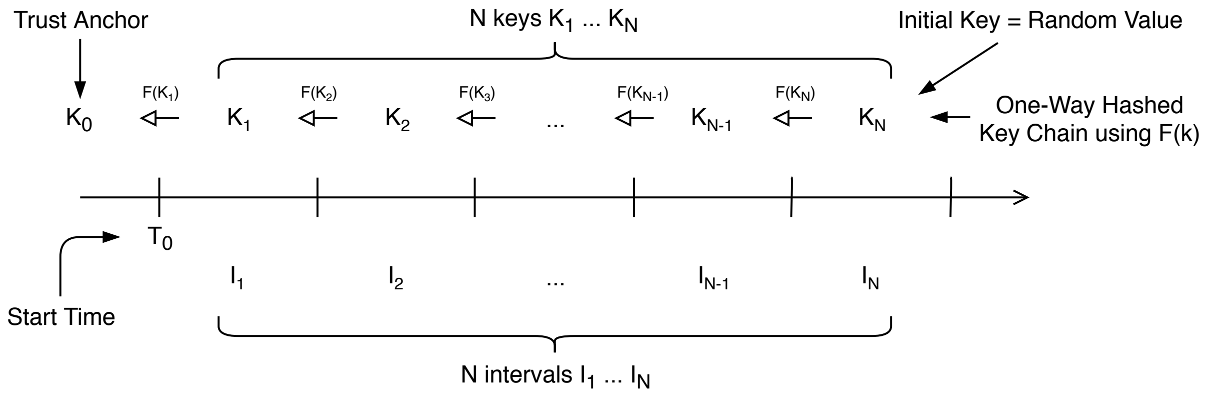


Figure 2.16: key chain construction

[6]A pseudorandom function (PRF) is one that cannot be distinguished from a truly random function; in this example, a keyed hash function may be a PRF, if given an instance of it using a randomly selected key, its output is indistinguishable from that of a truly random function.

An important distinction to make here — and to keep in mind whenever references are made to 'first' and 'last' — is between the order in which keys are generated, and the order in which keys are used. Given the subscript notation used here (and in RFC4082) where the numbers correspond to increasing time intervals, in terms of the *order in which keys are generated*, K_N is the first key, K_1 is the last usable key, and K_0 is the last key, i.e., the trust anchor. However, in terms of the *order in which the keys will be used*, K_1 is the first key, corresponding to the first interval I_1, and K_N is the last key, corresponding to the last interval I_N. Since the chain is recursively generated using a one-way function, K_0 — the last key to be generated — cannot be used to derive any of the other keys; it can only be used as a means to verify the authenticity of previous keys in the key chain (or later keys in terms of time intervals). This is a crucial property of the delayed verification scheme using TESLA for key management: a key K_i corresponding to interval i cannot be used to derive future keys K_{i+x}, where $1 \leq x \leq N - i$.

The other parameters that need to be established at start up are the disclosure delay d, an estimated upper bound on the lag of receiver's clock with respect to the sender's (i.e., offset from master) D_t, and another pseudorandom function f' which will be used to derive ICV-keys from key chain keys. The disclosure delay is measured in number of time intervals, and specifies when each key may be disclosed: K_i, used to secure messages sent during interval i, may only be disclosed in interval $i + d$. D_t is used by slaves when receiving messages to determine whether they are 'safe' or not; this will be discussed below, as we look at the control flow for both sending and receiving. The other pseudorandom function f' is needed because the same key should not be used for multiple cryptographic operations [16], and thus since a key chain key K_i corresponding to interval i was used in the operation of deriving the next key in the chain, it should not be used in the separate operation of calculating an ICV for a message sent during interval i. Thus, we need to derive an ICV-key K_i' from the key chain key K_i. This is done in a similar fashion as with Equation 2.3, using a pseudorandom function f'. The one-way function for ICV-key construction is as follows:

$$F'(k) = f'_k(1) \tag{2.4}$$

The control flow on the sending side for delayed processing begins in the same way as in the immediate case, with several interactions with the SPD and SAD to determine whether the message needs to be secured, and if so, with what parameters (found in the appropriate SA). Similarly, once the sender — which must be the master — knows that the message is to be secured, it can set the SECURE flag the PTP header, and having all the necessary information from the SA, it can begin constructing the AUTHENTICATION TLV. At this point, processing begins to differ from the immediate case. First, the master needs to determine which time interval i it is in, according to Equation 2.5.

$$i = \frac{currentTime - T_0}{T_{int}} \tag{2.5}$$

This current time interval i determines the key to be used in securing the current message. It is also used to calculate the past interval for which a key should be disclosed, by $disclosedKeyInterval = i - d$. This means that during interval i, the master should be disclosing key K_{i-d}. Including a key directly in a message adds overhead to the sending side, and similarly for a receiver, processing such a message also adds overhead; as such, the draft standard specifies that keys should only be disclosed in non-*Event* PTP messages. The master needs to make this determination, and if it is indeed preparing a non-*Event* message, it must adjust the lengthField accordingly, and set the appropriate bit in the secParamIndicator field to indicate that this AUTHENTICATION TLV includes a disclosedKey. As in the immediate processing case, the SPP field is filled with the SPP obtained previously from the SPD lookup, while the keyID field in the delayed processing case is filled with the current time interval i. Finally, as with immediate processing, the ICV is calculated over the entire message up to the start of the ICV, using the integrity algorithm type given by the SA, and inserted into the ICV field; the key to be used, however — ICV-key K'_i — is determined by the current time interval i, derived from K_i as $K'_i = F'(K_i)$. The AUTHENTICATION TLV for delayed processing is shown in Figure 2.17.

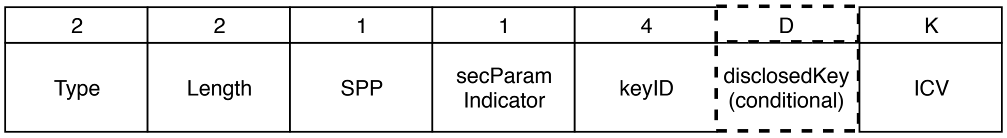


Figure 2.17: AUTHENTICATION TLV with delayed processing

Processing inbound messages with the delayed verification approach begins the same way as with immediate verification, up until the presence of the AUTHENTICATION TLV has been verified, and the appropriate SA has been obtained. Knowing that the message should be processed according to the delayed verification scheme, the secParamIndicator must be checked to find out whether this message contains a disclosedKey. Based on this result, and on the ICV length parameter provided in the SA, the expected length of the AUTHENTICATION TLV payload can be calculated and verified against the lengthField in the received AUTHENTICATION TLV.

At this point, the control flow becomes unique to delayed processing, and can be broadly organized according to the following three steps, which will be discussed in turn below: first, test if the received message is *safe*; second, deal with the disclosed key, if present; and third, buffer the message and proceed according to the mode of operation. The draft standard specifies two such modes of operation when processing inbound messages via the delayed approach: authenticated and unauthenticated. In both modes, received messages are be buffered so that they may be verified later, when the appropriate key is disclosed. In authenticated mode, the contents of a buffered message are not used in any way until *after* the message is verified. This behavior is implicit in the TESLA definition according to RFC 4082, and would require an implementation to be able to manage control flow such that a now-verified old message could be utilized during the security processing of the current message. In unauthenticated mode, after received messages are buffered, they are then used anyway, in spite of their integrity and authenticity not being verified yet. This would require an implementation to be able to roll back the effects of a message that was already used but failed the ICV check.

Revisiting the three steps above, first the slave must test if the incoming message is safe. Once a key K_i is disclosed, anyone might have access to it, including an attacker, who could use it to create a malicious message and secure it with an ICV calculated with the legitimate key K_i. Thus, a message should only be processed if it is protected with a key that is still secret (known only to the master), i.e., has not been disclosed yet. This is defined as a 'safe' key; a 'safe' message is one whose ICV was calculated using a safe key. The security parameters necessary to determine the safety of a message are: the start time T_0, the interval duration T_{int}, the disclosure delay d, and an upper bound on offset from master D_t. Given an incoming message, the slave first notes the local time T that it was received, and the interval i during which it was sent by the master (advertised in the keyID field of the AUTHENTICATION TLV). The slave can then calculate t_j, an upper bound on the time on the master's clock when the message was received:

$$t_j = T + D_t \tag{2.6}$$

Now the slave can calculate the latest possible interval x that the sender was in at the time the message arrived:

$$x = \left\lfloor \frac{t_j - T_0}{T_{int}} \right\rfloor \tag{2.7}$$

Since the message was sent in interval i (based on the keyID field of the AUTHENTICATION TLV), we want to verify that key K_i has not been disclosed yet; in other words, that the sender could not possibly be in interval $i + d$. Thus, the final check:

$$x < i + d \tag{2.8}$$

If the message is unsafe, it should be discarded; otherwise, security processing can continue.

For the second general step, the slave must deal with the disclosed key, if it is present. In a given interval i, the master may disclose K_{i-d} in several messages; thus, the slave must verify whether the currently disclosed key has been seen before, or whether it is new. If the key is new, its own authenticity must be verified by running it through the one-way function for key chain construction (Equation 2.3) until arriving at result equal to an earlier, already verified key, or the trust anchor, K_0, in the case of K_1. If the key is not new, then it has already been verified to be part of the key chain, and used to verify previously buffered messages, so processing can skip to the third step.

If the disclosed key K_{i-d} is new (has not been seen before) and verified to be legitimate, it can be used to derive the corresponding ICV-key K'_{i-d} by running it through the ICV-key construction function (Equation 2.4), which in turn can be used to verify buffered messages received in interval $i - d$. In authenticated mode, for each buffered message, if ICV verification fails, the message is discarded; otherwise, the message is 'cleared' and needs to be utilized. In unauthenticated mode, for each buffered message, if ICV verification fails, the effects of having already used the message must be undone; otherwise, processing can continue.

In the third step, attention is returned to the message at hand. The keyID field of the attached AUTHENTICATION TLV stores the time interval i in which the message was processed by the sender, which also reveals that the message is protected by an ICV calculated using the ICV-key K'_i; thus the message should be buffered accordingly so it may be easily accessed when K_i is later disclosed. In authenticated mode, the message that was just buffered should not be processed any further. In unauthenticated mode, it should be used immediately, at the risk that its effects will need to be undone later, should it fail verification.

CHAPTER 3
SECURITY MECHANISM IMPLEMENTATION

Now that we covered the relevant background topics, as well as the 1588 draft standard's security proposal, we can look at how we implemented the integrated security mechanism. We will first discuss the selection of PTPd as the PTP implementation to build on top of, and cover an overview of its structure. Next we will look at its set-up, configuration, and control flow, showing where security processing fits into the existing PTPd code, and explaining relevant design decisions. Finally, we will discuss the cryptography involved, and delayed processing considerations.

3.1 PTP Implementation Selection

In order to explore the feasibility of the proposed security mechanism, we needed to implement Prong A on top of an existing PTP implementation. PTPd [17], an open-source software implementation of IEEE 1588, was chosen as the basis for this goal. Among the few other existing implementations, PTPd was the best choice due to it being BSD-licensed (as opposed to the Linux PTP Project, which is licensed under the GNU General Public License) and due to its amenability to configuration for the power profile (as opposed to Open AVB's implementation specific to their own gPTP profile).

3.2 PTPd Structure Overview

PTPd is organized into a few major components, as shown in Figure 3.1. The diagram represents interfaces between components with connecting lines, and is color-coded to distinguish between components that security related code was added to, and those that did not need to be modified. In addition to these components, which directly correspond to source files of the same name, there are a few principal structures that are central to PTPd defined in `datatypes.h`: `RunTimeOpts` and `PtpClock`. `RunTimeOpts`, as the name suggests, holds program options that are set at run-time, read in from the command line or from a configuration file. These options are not meant to change during execution, as evidenced by consistent `const` qualification in mostly all[1] of the function prototypes requiring `RunTimeOpts`. `PtpClock` is the main program data structure, which holds, among many other variables, the PTP datasets, message buffers, temporary timestamp values and calculations, and `RunTimeOpts`. To support the security implementation, we added a `SecurityOpts` structure to `RunTimeOpts` to hold security parameters, and a `SecurityDS` structure to `PtpClock` to hold mutable security related data that needs to be accessed *and* updated during execution.

The program entry point is in `ptpd.c`, which declares various necessary structures and begins initialization with a call to the start-up component's `ptpdStartup()`, and starts the protocol engine with a call to `protocol()`.

The startup component reads in settings and options specified on the command line and in the configuration file (primarily via `parseConfig()`, a function belonging to a a subcomponent not listed in the diagram) and initializes the program state. It also includes functions for handling signals, and for shutting down. This component was modified to initialize and clean up security related datasets.

The protocol engine component implements the PTP state machine defined in IEEE 1588, and thus holds the main program logic. The principal functions are `doState()`, `toState()`, `processMessage()`, and several message handlers and issuers for each message type, `handle<`*`type`*`>()` and `issue<`*`type`*`>()`, e.g., `handleAnnounce()` or `issueSync()`. The state machine is implemented as a forever loop, in which `doState()` handles most of the con-

[1]`parseConfig()`, for instance, needs to populate `RunTimeOpts` at start-up, and so of course could not qualify it with `const`.

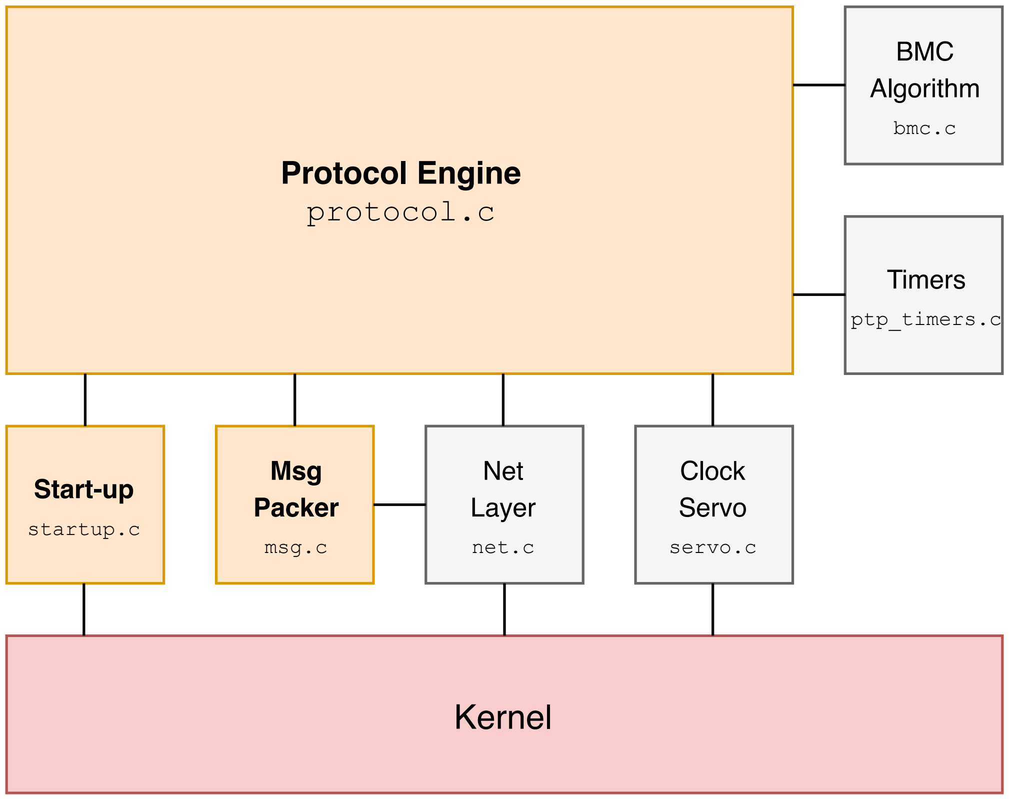


Figure 3.1: PTPd major components [3]
(bold font indicates components modified in this project)

trol flow, based on the daemon's state (master, slave, etc.), while `toState()` handles state transitions. A distinction here is made between actions and events; primary actions include sending messages (`issue<`*type*`>()`), running the BMC algorithm, setting/resetting timers, and feeding the clock servo; primary events include receiving messages (`handle<`*type*`>()` and `processMessage()`), and timer expirations. This component was modified to include inbound security processing in `processMessage()`, and outbound security processing in `issue<`*type*`>()`. Issuing messages and handling received messages require interaction with the message packer component and the network layer component, which will be discussed next.

The message packer component deals with packing and unpacking messages according to the message formats described in IEEE 1588. Important functions in this component include `msgPackHeader()`, `msgUnpackHeader()`, and packers and unpackers for each message type, `msgPack<`*`type`*`>()` and `msgUnpack<`*`type`*`>()`. This component was modified to include a function for adding an AUTHENTICATION TLV (`addAuthenticationTLV()`), packing and unpacking functions for the AUTHENTICATION TLV (`msgPackAuthenticationTLV()` and `msgUnpackAuthenticationTLV()`), and functions for packing and 'unpacking' (verifying) the AUTHENTICATION TLV's ICV (`calculateAndPackICV()` and `calculateAndVerifyICV()`).

The network component includes functions to interact with the network sockets, so as to send and receive data and collect timestamps. A layer above the system socket calls are PTPd send and receive functions split based on message class (*Event* vs. *General*); these are `netRecvGeneral()`, `netSendGeneral()`, `netRecvEvent()`, and `netSendEvent()`, and are important to note in order to understand the program control flow described later in Section 3.4.

The final three components, despite being crucial components to the functioning of PTPd, are outside the scope of this project, and therefore warrant only brief descriptions: The BMC component implements the Best Master Clock (BMC) Algorithm as defined in IEEE 1588, and as the name suggests, it is used by the protocol engine to determine the best master clock; the timers component allows for setting various timers and checking whether they have expired, and is used by the protocol engine primarily to determine when to send each type of message, according to the message interval rates specified at startup; lastly, the clock servo component interacts with both the protocol engine and the kernel, using the collected timestamps to calculate delays and offset from master, and subsequently adjusting the system clock so as to minimize the offset.

3.3 PTPd Configuration / Options

As mentioned previously, PTP is utilized in many application domains, and thus the standard was written to allow for various configurations. PTPd provides robust support in this regard. To serve this project, PTPd must be configured per the power profile specifications. This means using *transport mode* `ethernet` (and thus eliminating unicast as an option) as opposed to `ipv4`, using the *peer delay* mechanism as opposed to *delay request*, and setting the message interval rates appropriately. At startup, PTPd may be passed a configuration file containing such run time options, which are read into PTPd's `RunTimeOpts` struct. The configuration file is also leveraged to manually specify necessary security parameters for our implementation that would otherwise be obtained through SPD and SAD lookups when using an automated key management implementation. These security parameters, discussed in Section 2.5, as well as a `securityEnabled` boolean, are read into a `SecurityOpts` struct, which itself is held in `RunTimeOpts`. It is important to note that this method is used for the sake of experimentation; automated key management implementations were out of scope for this project. At the points in control flow where the draft standard specifies interactions with the SPD and SAD, the `securityEnabled` boolean — as well as a few other checks discussed later — are used to emulate an SPD lookup, and the `SecurityOpts` struct emulates the SA containing the relevant security parameters. Should an automated key management implementation become available and need to be integrated with the existing code, the SPD lookup can replace the `securityEnabled` check, the parameters in the SA can be read into `SecurityOpts`, and beyond this point, no further refactoring should be necessary.

Considering that the security implementation presented here is currently for experimentation and proof of concept purposes only, it should be easily turned on or off at compile time, so that an executable can be produced that does not contain *any* of the added security code, if desired. The model for this, used already by PTPd for other features like statistics tracking or debugging (among many others), is to wrap the optional code in preprocessor conditional statements depending on a given definition, and to have these definitions set during the build process based on options given to the configuration script. Thus, we have enclosed all security related code in `#ifdef PTPD_SECURITY` and `#endif` preprocessor commands, and added an `--enable-security` option to the configure script, which defines `PTPD_SECURITY` during the build process. This facilitates performance evaluation testing when running regression tests (Section 4.3), which require a PTPd with no security modifications, and makes the implementation acceptable for submission back to the main open-source repository.

3.4 PTPd Control Flow

In Section 3.2, we described each of the main components, including their purpose, their main functions, and their connections, and mentioned the security related functions that were added to certain components. Here, we will use several diagrams[2] to look closer at the control flow itself, and in doing so, highlight where the security related functions come in, describe what they do in more detail, and explain the design decisions involved.

3.4.1 Top-Level Control Flow

We start with a simplified view of the top-level control flow, shown in Figure 3.2. Functions are represented by a horizontally split rectangle, labeled on the top half with the component or source file it is from, and on the bottom half with the function name (parameters omitted). A dashed line coming down from a function represents the body of the function, with horizontal solid arrows to indicate a separate function call, and horizontal dashed lines to indicate a statement or multiple statements within the given function; such statements are enclosed in dashed-line boxes, and include either control structures (surrounded by braces) or simply a short explanation of the statement(s). Security related functions, source files, or statements are indicated in bold font. Figure 3.2 shows, at the first level of execution under `main()`, the startup component in `ptpdStartup()`, and the main state machine in `protocol()`. In the forever loop of `protocol()`, the program repeatedly calls `doState()` and `checkSignals()`. In `doState()`, based on the current state, the daemon checks for and deals with received messages with `handle()`, and sends *Announce*, *Sync*, and *Pdelay_Req* messages with `issue<`***type***`>()` (the *response* and *followup* message types are instead initiated as a result of a message reception). `checkSignals()` is where shutdown control flow

[2]The control flow diagrams in this section are simplified representations, in that the actual contents of the functions they show are pruned to leave only the parts relevant to explaining the security processing.

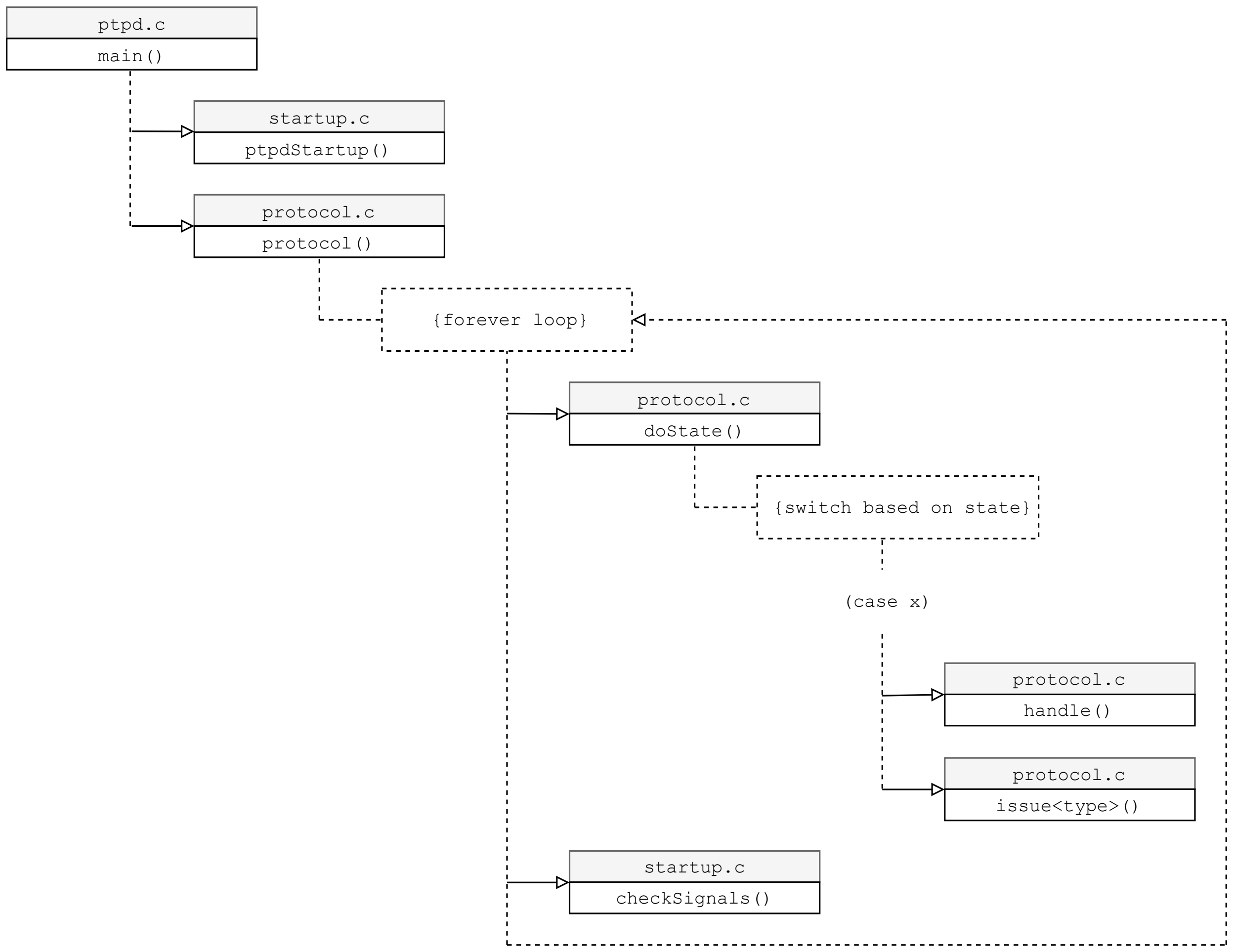


Figure 3.2: top-level control flow

originates. In order to show where security code needed to be added, we will expand on `ptpdStartup()`, `handle()`, `issue<`*`type`*`>()`, and `checkSignals()` below.

3.4.2 Startup & Shutdown Control Flow

An expanded view of `ptpdStartup()` is shown in Figure 3.3. We enter `ptpdStartup()` from the `main()` function, being passed in a pointer to the statically allocated `RunTimeOpts rtOpts`, which will need to be populated with run-time options. `ptpdStartup()` also dynamically allocates memory for the main program data structure `PtpClock ptpClock` and initializes it. `parseConfig()` is called to read in run-time options from the configuration file into `rtOpts`; this is where we added code to read in security parameters as well, into our own `SecurityOpts` structure, which was added as a member to `RunTimeOpts`. The security parameters that are read in from the configuration file that are common to both verification approaches are: a boolean `securityEnabled`, a boolean `delayed`, a `key`, an `integrityAlgTyp`, an `SPP` value, and a `keyID` value. Parameters for the delayed approach are: `chainLength`, `startTime`, `intervalDuration`, `disclosureDelay`, and `D_t`. After reading everything in, there is an 'additional logic' section in which some final checks or configuration can be done before returning back to `ptpdStartup()`. This is where we added code to derive the rest of the security parameters from the ones that were read in: based on the `integrityAlgTyp`, which is the algorithm used for calculating the ICV, we can populate `icvLength` and `secTLVLen` (since different algorithms produce ICVs of different sizes); based on the `key` we can populate `keyLen`; and if `delayed` is set, we can use the `chainLength` and the `key` (which is now interpreted as the initial random value for the first key in the key chain) to allocate memory for the `keyChain`, and to generate it. After `parseConfig()` returns, the `PtpClock` is allocated, and thus the `SecurityDS`, contained by `PtpClock`, can be initialized. This consists of allocating the buffers necessary for delayed security processing by a slave, allocating the list of `verifiedKeys`, and copying the trust anchor into the list.

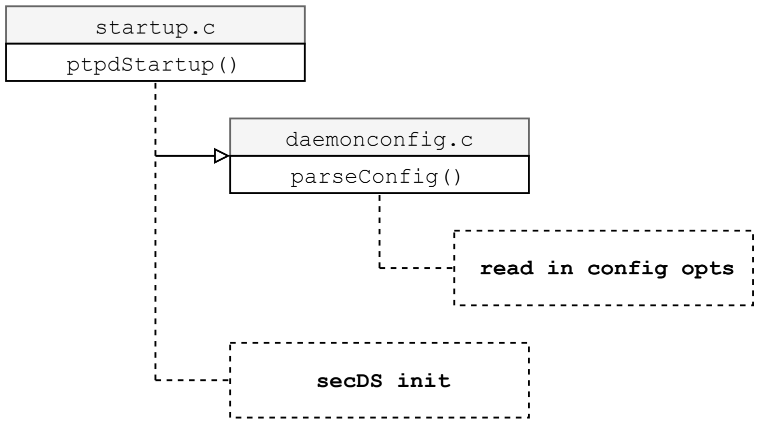


Figure 3.3: startup control flow

As explained in Section 2.5.5, the last key to be generated in the delayed processing key chain becomes the trust anchor, which is the only key from the key chain that should be distributed to the slave clocks. Here we ran into the problem of how to distribute just the trust anchor from master to slaves, in the absence of an automated key management implementation and a bootstrapping phase. The simplest solution was to not enforce a distinction between master and slave at startup, and instead have the daemon generate the key chain (as a master would) and save the trust anchor as the first key in the `verifiedKeys` member of the `securityDS` structure (emulating a slave receiving the trust anchor as part of a bootstrapping phase). This happens before the protocol engine even starts, and thus before master or slave states are entered; this means that once the daemon enters the slave state, it actually has access to the entire key chain, and similarly, once the daemon enters the master state, it has a superfluous `SecurityDS` structure containing `verifiedKeys` of which it clearly has no use. We acknowledge this workaround is not realistic; in all delayed security processing code meant to be executed by the slave alone, we follow the spec as if we did not have access to the key chain, only accessing the list of `verifiedKeys` and adding to it as more keys are disclosed. This is the clearest example of why this implementation is considered for experimentation and proof of concept only.

The shutdown process, shown in Figure 3.4, originates with `checkSignals()`, which is called every run through the forever loop in the protocol engine. If `sigint` or `sigterm` is received, then the program exits cleanly through a series of shutdown calls, beginning with `timingDomainShutdown()`, which goes through all the timing services in the timing domain — usually just one, namely, PTP — and calls their corresponding shutdown functions. In this case, that means calling `ptpServiceShutdown()`, which retries the service's controller, the `PtpClock`, and calls `ptpdShutdown()` with it. In `ptpdShutdown()`, we added code to free all dynamically allocated memory, namely, the delayed security processing data, consisting of buffers, verified keys, and the key chain.

3.4.3 Sending Control Flow

In Section 3.4.1 we saw that message transmission can originate either from `doState()`, in the case of *Announce*, *Sync*, and *Pdelay_Req* (as shown in Figure 3.2), or as a result of message

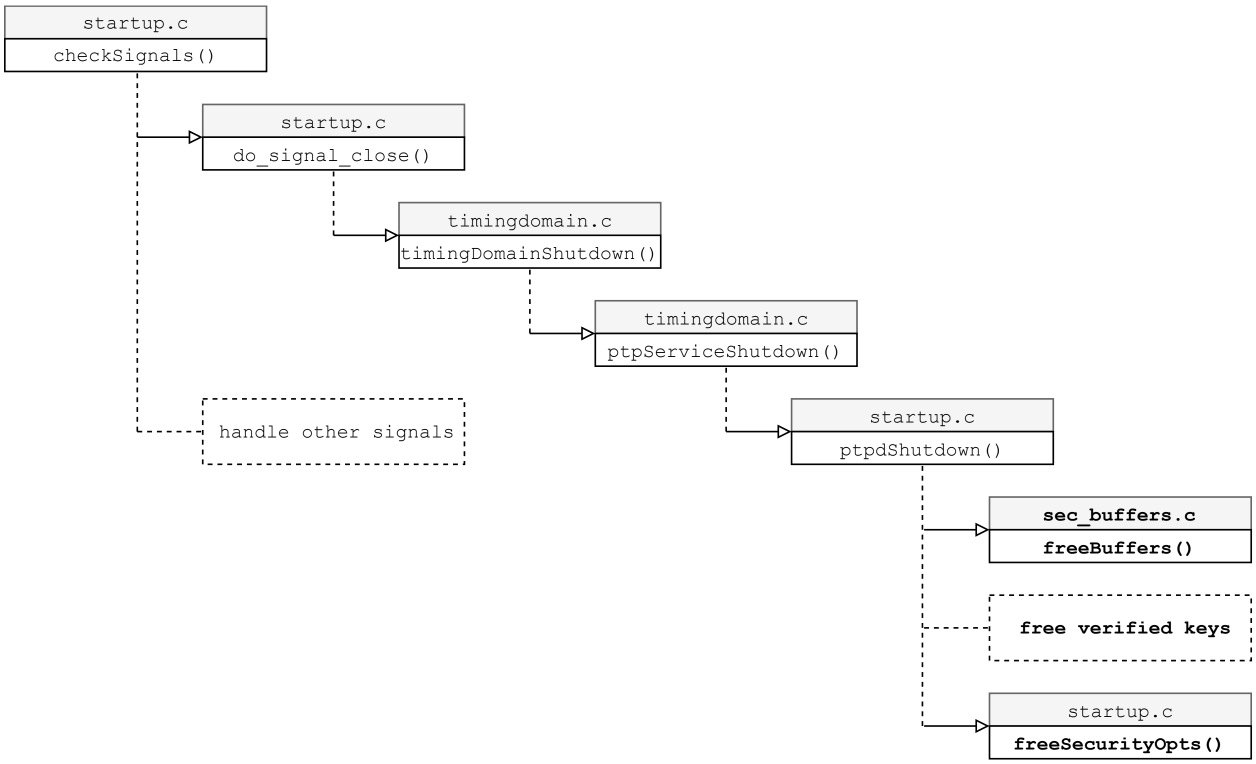


Figure 3.4: shutdown control flow

reception, in the case of *Follow_Up*, *Pdelay_Resp*, and *Pdelay_Resp_Follow_Up*. Regardless of how the transmission was triggered, all message types have a corresponding `issue<type>()` function, modeled in Figure 3.5. First, if a timestamp is needed, it is collected with the system call `getTime()`; next, the outgoing buffer is packed according to the proper message format with a call to the message packer component's `msgPack<type>()`; at this point, we inserted security related code to add the AUTHENTICATION TLV with `addAuthenticationTLV()`; finally, the network component is invoked with a call to `netSendGeneral()` or `netSendEvent()` based on the message class. As described in Section 2.5, before adding an AUTHENTICATION TLV, the policy limiting fields would be used to query the SPD to determine whether this message requires security processing or not, and if so, by what parameters. We emulate this by simply checking the `rtOpts->securityEnabled` boolean; if it is set, then the parameters will be found in `rtOpts->securityOpts`. For a slave using delayed processing, an SPD lookup would either return an indication that no security processing is to be done (since only the master has the key chain necessary to secure outbound messages), or it would return an SPP for an SA containing parameters for immediate processing. The latter case — which would involve combining delayed and security processing — was beyond the scope of this project. The former — not adding an AUTHENTICATION TLV at all — was emulated with an additional check to exclude a daemon in the slave state that also has the `rtOpts->delayed` boolean set to true from calling `addAuthenticationTLV()`.

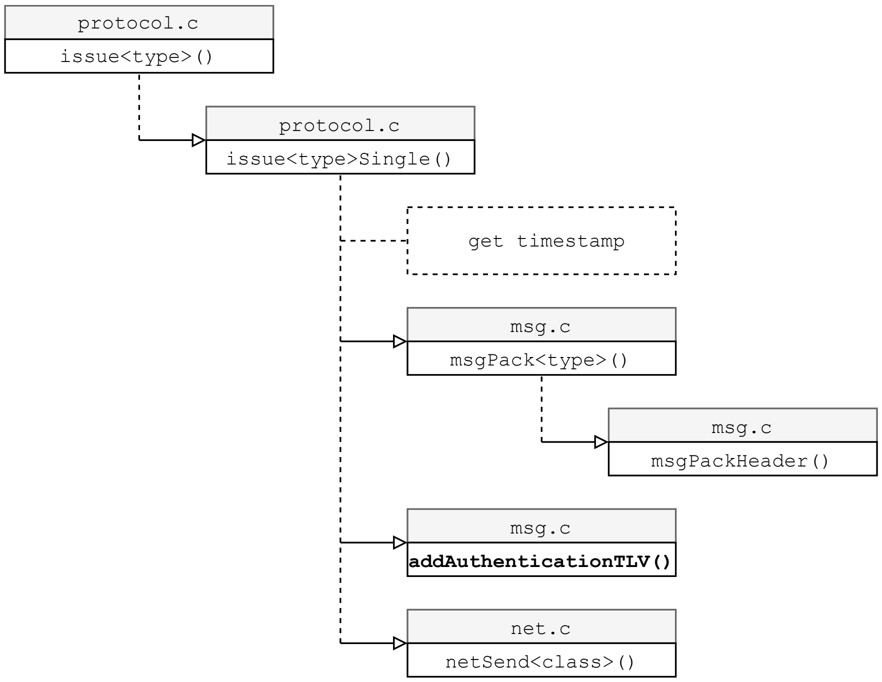


Figure 3.5: sending control flow

In `addAuthenticationTLV()`, we set the SECURE flag and adjust the length field in the PTP header header, and we pack the outgoing message buffer with an AUTHENTICATION TLV as outlined in Section 2.5. We receive as parameters the `PtpClock`, from which we can get the outgoing message buffer and informational counters that may need to be updated, the `SecurityOpts` struct containing the security parameters, and a boolean to indicate the message class, to be used in delayed processing to know whether a disclosed key should be included in the AUTHENTICATION TLV or not. The fields of the AUTHENTICATION TLV could be packed directly into the buffer here based on field sizes and offset calculations, but to make the code cleaner and more organized, we followed the existing message packing model to separate the buffer packing from other logic related to creating the AUTHENTICATION TLV. This model consists of having a struct representing the data that needs to be packed, and a corresponding packing function that takes the buffer and the said struct as parameters, and it handles calculating offsets based on the spec and actually filling the buffer. To this end, we defined a `AuthenticationTLV` struct, and added the functions `msgPackAuthenticationTLV()` and `msgUnpackAuthenticationTLV()`. Since the AUTHENTICATION TLV is variable length, we had to define our `AuthenticationTLV` struct to hold the fixed size fields only, as shown in Figure 3.6. This is still useful, as we can create the `AuthenticationTLV` struct and fill its fields without worrying about the details of buffer packing, and then simply call `msgPackAuthenticationTLV`; the remaining variable length fields are packed separately.

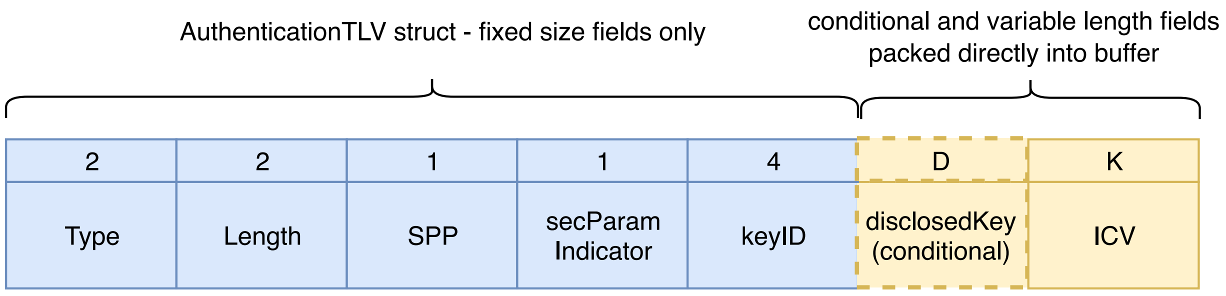

AuthenticationTLV struct - fixed size fields only
conditional and variable length fields packed directly into buffer
2
2
1
1
4
D
K
Type
Length
SPP
secParam Indicator
keyID
disclosedKey (conditional)
ICV

Figure 3.6: fixed size fields of the AUTHENTICATION TLV comprise the `AuthenticationTLV` struct

If the `delayed` security parameter is set, then throughout `addAuthenticationTLV` a few additional steps need to be taken: before calling `msgPackAuthenticationTLV()`, the current interval needs to be calculated according to Equation 2.5, and it needs to be set in the `keyID` field of the `AuthenticationTLV`; also, if the message class is *General*, we need to adjust the `lengthField` to account for a disclosed key (whose size is known from the security parameters), we need to flip the appropriate bit in `secParamIndicator`, and we need to manually pack the key for interval $currentInterval - disclosureDelay$ into the buffer at offset 10. Finally, we need to generate the ICV key from the current interval's key according to Equation 2.4 by calling `generate_icv_key()`, our implementation of the ICV key generating function.

The last step in `addAuthenticationTLV()` is to calculate the ICV and pack it into the buffer; these steps are implemented in `calculateAndPackICV()`, which takes as parameters the `SecurityOpts`, the buffer, the calculated ICV offset, and the key (which is a pointer to the single key to be used in the immediate approach, or a pointer to the recently calculated ICV key for the delayed approach). At this point, all logic related to the verification approach has been taken into account, and so `calculateAndPackICV()` stands alone as purely an ICV calculator and packer. The parameters required from `SecurityOpts` are the `integrityAlgTyp`, the `keyLen`, and the `icvLength`, which are already set accordingly based on the verification approach. Prong A of the draft standard specifies using at least HMAC-SHA256-128 for ICV calculation, meaning using HMAC with SHA256 as the internal hashing algorithm, using a key of size equal to the hash produced by SHA256 (32 bytes), and truncating output to 128 bits (16 bytes). We use the OpenSSL HMAC library to support this. Additionally, our implementation supports another ICV calculation algorithm, GMAC-AES256 [18], which will be discussed in Section 3.6. In order to support the addition of other ICV calculation algorithms, `calculateAndPackICV()` consists of a switch statement on the algorithm type, handling the ICV calculation and the packing of the buffer inside each case, as these operations are specific to the algorithm type.

3.4.4 Receiving Control Flow

Message reception starts with `handle()`, and proceeds as shown in Figure 3.7. First, `netSelect()` is called to check if any data has been received with a system call to `select()`. If data was received, it is retrieved with `netReceiveGeneral()` or `netReceiveEvent()`, which are passed the receive message buffer (contained in `PtpClock`), and a timestamp struct in the case of `netReceiveEvent()`. The received network data is stored in the buffer, and the timestamp is populated if necessary. Finally, `processMessage()` is called, passing the `PtpClock` (which contains the now populated receive message buffer), `RunTimeOpts`, the timestamp, and the message length. In `processMessage()`, the message packer component is utilized to unpack the PTP header into `PtpClock`'s `msgTmpHeader` struct with `msgUnpackHeader()`; from this, the message type is known, and each message type can be handled on its own through a switch statement.

The reasoning behind where to insert our security code for inbound processing is straightforward: as described in Section 2.5, security processing begins with using the policy limiting fields of the PTP header to query the SPD in order to determine whether security processing is required on this packet or not; therefore, as soon as the PTP header is recovered, security processing can begin.

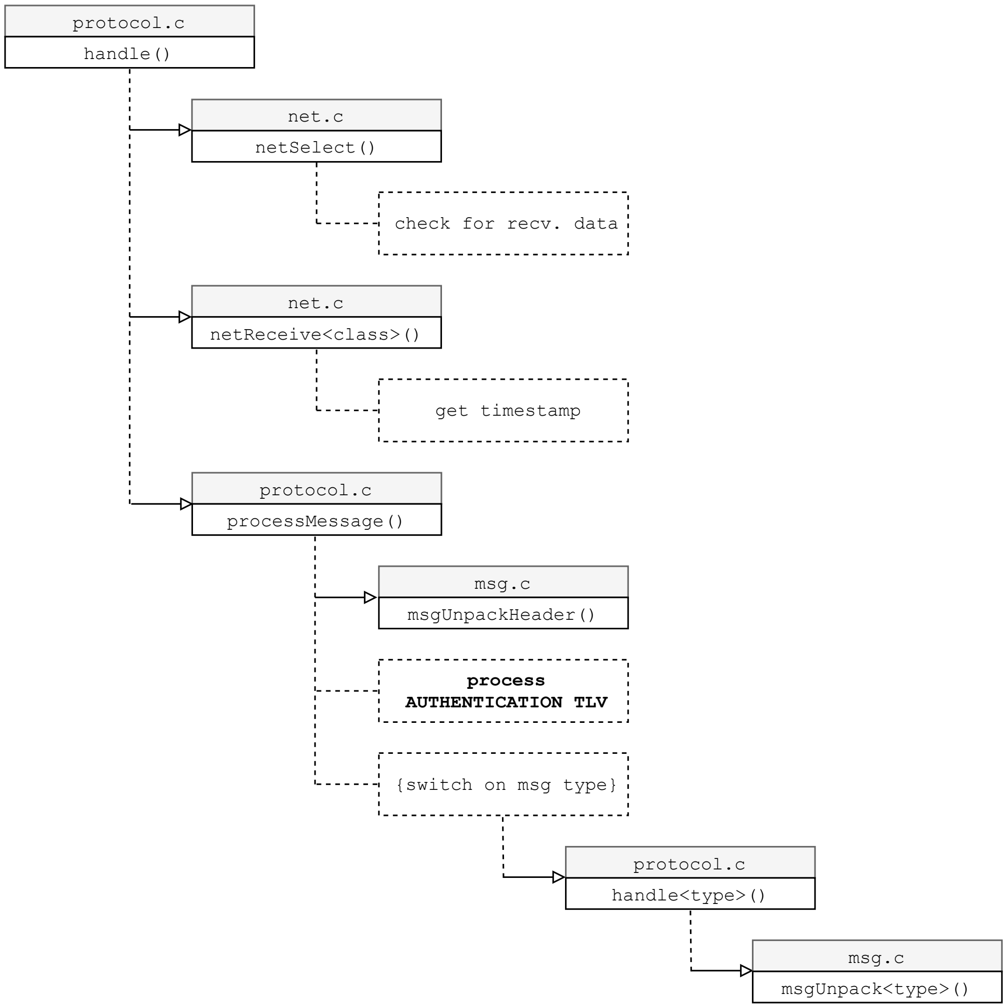


Figure 3.7: receiving control flow

We emulate the SPD query by checking the `securityEnabled` boolean. In addition, we must exclude a daemon in the master state that is using delayed processing from entering the security processing block; since our implementation does not handle combining both immediate and delayed verification approaches, the master should not be expecting any messages from the slaves to be secured and can skip to normal PTP message processing. Now, we implement the checks described in Section 2.5 for inbound processing. We first verify that the SECURE flag in the PTP header is set, indicating that an AUTHENTICATION TLV should be present. Then, based on the message length, we can unpack the fixed size fields of the AUTHENTICATION TLV into an `AuthenticationTLV` struct with `msgUnpackAuthenticationTLV()`. We can now verify that the extracted TLV is indeed an AUTHENTICATION TLV by checking the `tlvType` field, and if so, we can safely access the rest of the fields from the `AuthenticationTLV` struct. The SPP field in the received AUTHENTICATION TLV is then verified to match the expected SPP that would have been returned by an SPD query, by checking `SecurityOpts->SPP`. At this point, we have a verified SPP, which would be used to retrieve the SA containing the rest of the security parameters; in our implementation, these are stored in `SecurityOpts`. In the immediate processing case, we can proceed with the ICV check by calling `calculateAndVerifyICV()`, which takes the `SecurityOpts`, the incoming message buffer, the ICV offset, and the key to be used, and returns a boolean indicating the result of the ICV check. The design of `calculateAndVerifyICV()` mirrors that of `calculateAndPackICV()` by taking the same parameters, abstracting away any notion of verification approach, and consisting of a switch statement on the ICV calculation algorithm type. In the delayed processing case, we must do the safe packet test, deal with the disclosed key if it is present, and buffer the message, as described in 2.5. The implementation and design considerations concerning these steps will be discussed in Section 3.5.

3.5 Delayed Processing Considerations

In the delayed processing case for incoming messages, after doing the initial checks on the received AUTHENTICATION TLV, we must do the safe packet test, deal with the disclosed key if it is present, and buffer the message. These last two steps differ slightly based on the mode of operation: authenticated or unauthenticated. First, we will consider the modes of operation, and then we will address the implementation of these three steps.

3.5.1 Mode of Operation

As described in Section 2.5, there are two modes of operation when processing inbound messages via the delayed approach: authenticated and unauthenticated. Both modes require two major, complementary steps. We chose to model unauthenticated mode due to it being feasible to implement at least one of the steps, as opposed to neither of them in authenticated mode. In authenticated mode, the first step is that the contents of a buffered message are not used in any way until *after* the message is verified; the second step would require an implementation to be able to manage control flow such that a now-verified old message could be utilized during the security processing of the current message. Implementing the first step in PTPd by returning from `processMessage()` after buffering a message (in order to not use the unauthenticated message yet) may break some yet unknown aspect of the receive message handling. Possibly more complicated would be to make use of a previously buffered, now authenticated message, during the processing of the message that contained the disclosed key, after the program state had already advanced beyond what it was when the now authenticated message was initially received. It is not clear how these steps would be implemented. The simpler solution was to base our implementation on unauthenticated mode: in the first step, after received messages are buffered, they are then used anyway, in spite of their integrity and authenticity not being verified yet. In PTPd, this would mean simply letting control flow continue to the rest of the `processMessage()` code. The second step would require rolling back the effects of a message that was already used but failed the ICV check; this is an open problem in our implementation. Choosing to model unauthenticated mode allowed PTP to still function in our experiments, and leaves a clear point in the implementation where handling the 'undoing' of messages that failed ICV verification can be inserted.

3.5.2 Safe Packet Test

For the safe packet test, we call `isSafePacket()`, passing in the following values: the local time at which the packet arrived, passed in from `handle()` as `timeStamp`; the sender's advertised interval index, stored in the `AuthenticationTLV` `keyID` field; and the `disclosureDelay`, `D_t`, the `startTime`, and the `intervalDuration`, all stored in `SecurityOpts`. If the packet proves safe, we can proceed with the next steps.

3.5.3 Disclosed Key Checks

The first step in dealing with a disclosed key is to check the `secParamIndicator` field of the `AuthenticationTLV` to determine whether a disclosed key is present. If not, we proceed to buffering the message. If we have a disclosed key, we must check if it is a new key. To implement this check, we keep a `latestInterval` variable in the `SecurityDS`, which holds the latest interval for which we have a verified key. Given current interval i and disclosure delay d, a disclosed key will be for interval $i - d$; we call this the `discKeyInterval`. As a new disclosed key for interval `discKeyInterval` is verified, `latestInterval` is updated accordingly to store `discKeyInterval`. Key verification is done with a call to `verify_key()`, discussed in Section 3.6.

Once a key for interval `discKeyInterval` is verified, it must be used to derive the corresponding ICV key with `generate_icv_key()`, which in turn must be used to verify messages buffered during interval `discKeyInterval`. As will be described in Section 3.5.4, messages are stored in buffers corresponding to the interval they were received in, so this verification process is relatively straightforward; we simply retrieve the correct set of buffered messages, and call `calculateAndVerifyICV()` on each message. The special case that needs to be considered, however, is if a disclosed key happened to be missed during one or more intervals, resulting in more than one buffer of unverified messages. In this case, when a disclosed key is verified, the verification process necessarily reveals the missing key(s) as well, and all unverified buffers must be verified. To implement this, we first need to calculate the number of unverified buffers `numUnverifiedBuffers`, prior to verifying the present disclosed key for interval `discKeyInterval`, while `latestInterval` still represents the interval for the last known verified key:

$$numUnverifiedBuffers = discKeyInterval - latestInterval \tag{3.1}$$

The number of previous keys that we missed is `numUnverifiedBuffers - 1`. If this difference `numUnverifiedBuffers` is 1, then we have not missed any previous keys, and the only buffer that contains messages that will need to be verified is the one that corresponds to `discKeyInterval`; the verification of its messages can be done with the present disclosed key. However, if `numUnverifiedBuffers` is greater than 1, then it means we missed `numUnverifiedBuffers - 1` previous keys, and therefore, if the current disclosed key passes verification, then we will also discover the `numUnverifiedBuffers - 1` missing keys; these will need to be used to verify the messages in the `numUnverifiedBuffers - 1` buffers older than the buffer corresponding to `discKeyInterval`.

Since there may be more than 1 buffer containing messages that need verification, we need to go through the buffers in a loop, controlled by `numUnverifiedBuffers`; after each run through the loop, `numVerifiedBuffers` is decremented, and the termination condition is when it reaches 0. In the body of the loop, we need to derive a `targetInterval` based on the current number of unverified buffers, and the disclosed key interval:

$$targetInterval = discKeyInterval - numUnverifiedBuffers + 1 \tag{3.2}$$

In the most common case when no previous disclosed keys were missed, and there is only one unverified buffer, $targetInterval = discKeyInterval$. Otherwise, the larger the value of `numUnverifiedBuffers`, the smaller the value of `targetInterval`, and as we iterate over the loop and `numUnverifiedBuffers` decreases to approach 1, so does `targetInterval` increase to approach `discKeyInterval`. Based on this control flow, if there are multiple unverified buffers, we will start with the oldest buffer and work towards the latest one, which corresponds to `discKeyInterval`.

Since we are modeling unauthenticated mode, the ICV verification for the messages in each buffer proceeds as follows: if the current message passes the verification, we leave it as is in the buffer and proceed to the next one, since it was already used back when it was initially received and buffered. If the message fails verification, it is marked as such in the buffer, and we proceed to the next message. After finishing the verification checks on all messages, we are left with a buffer containing the messages and an indication of whether they failed verification or not. At this point, code will need to be added to undo the effects of using any messages that failed verification. We decided to leave the buffer with all messages in the order that they were received, including messages that passed verification, as all neighboring messages may be needed to implement the roll-back of the effects of having used a bad message in the past.

3.5.4 Message Buffering

After checking if a packet is safe, and dealing with a disclosed key if present (which includes verifying previously buffered messages), we need to buffer the current message. Since we are modeling unauthenticated mode, then after we buffer the message, we can simply let the rest of the PTPd code in `processMessage()` continue, and the contents of the message will be used normally according to the rules of PTP (albeit lacking proof of integrity and authenticity).

The code related to message buffering is in `sec_buffers.h` and `sec_buffers.c`. We use a `BufferedMsg` struct to hold a buffered message; this is a linked-list style node that holds a PTP message, a next pointer, and a flag to indicate whether the message passed ICV verification or not. We also use a `Buffer` struct to represent an entire buffer; this holds references to the head and tail of the linked-list of `BufferedMsg`s, and a size. We have initialization functions and freeing functions to take care of dynamically allocated memory, and `bufferMessage()`, which takes an existing Buffer and a pointer to a message, and creates a `BufferedMsg` to store the message, and adds it to the `Buffer`. We decided to use a dynamic memory allocation approach because the size of a `Buffer` is unknown, and can vary widely based on the message interval rates and interval duration.

3.6 Cryptography

Cryptographic functions and operations are bundled together in `security.h` and `security.c`. This component includes functions implementing two ICV calculation algorithms, and key-related functions used in delayed security processing.

3.6.1 ICV Algorithms

Prong A of the draft standard specifies using at least HMAC-SHA256-128 for ICV calculation, meaning using HMAC with SHA256 as the internal hashing algorithm, using a key of size equal to the hash produced by SHA256 (32 bytes), and truncating output to 128 bits (16 bytes). We use the OpenSSL HMAC library to support this, with a call to `HMAC()` from `openssl/hmac.h`, which returns a MAC. This function conveniently takes care of all the necessary calculations, requiring only the following necessary parameters: the underlying hash function, specified with a call to `EVP_sha256()`, the key, the key length, a pointer to the data to be authenticated, and its length. We truncate the output in the calling function — `calculateAndPackICV()` or `calculateAndVerifyICV()` — to produce our ICV.

Additionally, our implementation supports another ICV calculation algorithm, GMAC-AES256, specified in the NIST publication "Recommendation for Block Cipher Modes of

Operation: Galois/Counter Mode (GCM) and GMAC" [18]. GMAC is a special case of GCM (Galois/Counter Mode), which is a mode of operation for the encryption algorithm AES (Advanced Encryption Standard). To understand the use of GMAC to generate an ICV, we must first understand GCM. GCM is specified as a mode of operation for an underlying symmetric key block cipher, such as AES. With AES, we can encrypt plaintext data to provide confidentiality. With GCM-AES, we can similarly encrypt plaintext data to provide confidentiality, as well as authentication of that confidential data *and* of additional data that is not encrypted. This authentication is provided "using a universal hash function that is defined over a binary Galois field" [18]. Thus, given an approved block cipher (AES) and a key, there are three inputs to GCM: plaintext data that get encrypted and authenticated via a resulting MAC; additional authenticated data (AAD) that is not encrypted, but still authenticated through the same MAC; and an initialization vector (IV), which is essentially a nonce (a random, unique number used only once in a particular invocation of GCM). GMAC is then a special case of GCM, in which the only inputs are the AAD and the IV, and the output is a MAC that provides authenticity for the AAD. In our application, the AAD is the PTP message that needs to be protected, and the resulting MAC is the ICV.

OpenSSL provides support for AES and several of its modes of operation, including GCM, in its `openssl/evp.h` library. We implemented GMAC for calculating an ICV in `GMAC()`, with a series of OpenSSL function calls. The parameters that are passed in from `calculateAndPackICV()` or `calculateAndVerifyICV()` are a key, an IV, a pointer to the data to be authenticated, a pointer to where the ICV should be written, and corresponding value lengths. First, a cipher context `EVP_CIPHER_CTX` is declared and created with `EVP_CIPHER_CTX_new()`. The context is initialized with `EVP_EncryptInit_ex()`, passing in `EVP_aes_256_gcm()` as the underlying block cipher; this specifies to use GCM with AES256, with a key size of 256 bits, the same as what is used with HMAC. We then give the key and the IV to another call to `EVP_EncryptInit_ex()`. `EVP_EncryptUpdate()` must then be called, providing the AAD, in this case, the pointer to the PTP message data to be authenticated.

If we were using GCM, another call to `EVP_EncryptUpdate()` would be made, passing in the plaintext data to be encrypted; in the absence of such a call, we are implicitly using GMAC as a special case of GCM. The algorithm is finalized with `EVP_EncryptFinal_ex()`, and the resulting ICV is stored with a call to `EVP_CIPHER_CTX_ctrl()`, passing in `EVP_CTRL_GCM_GET_TAG`, the ICV destination pointer, and the desired ICV length. Special care must be taken with the IV in the calling functions `calculateAndPackICV()` and `calculateAndVerifyICV()`. First, our implementation defines the IV size as 12 bytes in `constants.h` with `GMAC_IV_LEN`, per the recommendation in the NIST GCM specification [18]. In `calculateAndPackICV()`, the IV is created by reading `GMAC_IV_LEN` bytes in from `/dev/urandom`, and passed to `HMAC()` along with the other necessary parameters to create the ICV. Since the receiver will need to use the same IV, it is packed into the AUTHENTICATION TLV as the first `GMAC_IV_LEN` bytes of the ICV field, followed by the ICV itself. In `calculateAndVerifyICV()`, the IV from the received AUTHENTICATION TLV must be passed to `HMAC()` along with the other necessary security parameters to ensure a correct ICV calculation is made.

3.6.2 Key Functions

In addition to the ICV calculation algorithms, the cryptography component contains key-related functions used in delayed security processing. These include the key chain generating function `generate_chain()`, the key verification function `verify_key()`, and the ICV-key generating function `generate_icv_key()`. As described in Section 2.5.5, the functions for generating the key chain and for ICV-key construction are $F(k) = f_k(0)$ and $F'(k) = f'_k(1)$ (Equations 2.3 and 2.4 respectively), where f and f' are pseudo-random functions, and where $f_k(x)$ denotes hashing over x using key k. Any well established keyed hash function would qualify as a pseudo-random function, and so for the sake of simplicity, we chose HMAC-SHA256 for f and f'.

Out of these three functions, the simplest implementation was for `generate_icv_key()`. We pass in a pointer to where the resulting key (i.e., the output of the internal hashing function) should be stored, a key, and a key length, and simply call `HMAC()` using `EVP_sha256()`, hashing over the message 1. The calling functions would pass in a key from the key chain, and retrieve the corresponding ICV key.

The key chain generating function `generate_chain()` is similar to `generate_icv_key()` in that we also call `HMAC()` using `EVP_sha256()`, hashing over the message 0 instead, per Equation 2.3. This is a recursive function, however, taking a pointer to a list of keys `base` (initially the `SecurityOpts->keyChain`), the key length, and a number `n`, which represents how many more keys will be generated based on the key currently pointed to by `base`. The base case, of course, is when `n == 0`, and each successive call uses the newly generated key (stored in the next open slot in the key chain) as the new `base`, and `n - 1` as the new number of keys that still need to be generated.

The key verification function `verify_key()` is used by a slave to verify that a newly disclosed key is indeed authentic. As mentioned in 3.4.2, the slave keeps a list of `verifiedKeys` in its `SecurityDS` struct, starting out with just the trust anchor in the last slot, and a `latestInterval` variable representing the latest interval for which a verified key is known. Since the trust anchor does not correspond to an actual interval, `latestInterval` is initialized to -1. To verify a newly disclosed key K_i for interval i, we call `verify_key()`, passing in `verifiedKeys`, new key K_i, new interval i, `latestInterval`, the chain length, and the key length (to be used in subsequent calls to `HMAC()`). We copy K_i into the `verifiedKeys` list in the slot corresponding to i, and call `generate_chain()`, using the new key as a base, and stopping just before $K_{latestInterval}$, the key corresponding to `latestInterval`. Then, we hash the last key to be generated by `generate_chain()` with a call to `HMAC()` into a temporary buffer and compare it with $K_{latestInterval}$, which we know is good. If the comparison succeeds, we can return 1 for success, and the calling function can update `latestInterval` accordingly; the `verifiedKeys` list is now updated to include K_i and any intermediate keys between it and $K_{latestInterval}$. If the comparison fails, `verify_key()` returns 0; the caller's `verifiedKeys` list will still have the bad keys stored, but they are effectively garbage since `latestInterval` will not be updated by the caller.

CHAPTER 4
EVALUATION

To evaluate whether implementing the proposed security mechanism in the context of the power profile is feasible, we must verify the following:

- *Feasibility*: the standard is specific enough to allow for implementation
- *Functionality*: the implementation does what it is supposed to do
- *Performance*: the implementation performs at an acceptable level

Each of these will be discussed in turn below, for both the immediate and delayed verification approaches. In the functionality and performance experiments, three systems were used to run our modified PTPd:

- *Linux Desktop* — processor: 2.70GHz Intel Core i5-6400; memory: 8.192 GB 3200 MHz DDR4; operating system: Ubuntu 16.04.4 LTS (Xenial Xerus); network interface card/controller: Intel I210
- *MacBook Pro* — processor: 2.9 GHz Intel Core i7; memory: 16 GB 1600 MHz DDR3; operating system: macOS 10.12.6 (Sierra)
- *Raspberry Pi 3 model B* — processor: 1.2 GHz ARM Cortex-A53; memory: 1 GB 900MHz LP-DDR2; operating system: Raspbian 8 (Jessie)

4.1 Feasibility

As a result of this work, we have found that the security proposal in the IEEE 1588 draft standard is specific enough to enable the implementation of a functioning prototype of the security mechanism in the context of the power profile, albeit with a few limitations to be described in this section.

4.1.1 Power Profile Implications

Regarding the power profile [7], none of its specifications were found to prevent the adoption of the current implementation; however, employing the security mechanism would preclude the use of one-step mode, since ICV calculation done on the fly — after a timestamp is taken and inserted in the outbound message — is unfeasible and would significantly hinder timestamp accuracy. The required message interval rates may affect performance, specifically in the case of *Announce* messages, as they are more frequent under the power profile compared to the default profile, but this in and of itself does not prevent the integrated security mechanism from functioning. Similarly, the specification of Ethernet as the transport mode does not preclude security functioning. The fact that outgoing messages are broadcasted and thus received by the sender as well required a work-around in the case of the delayed verification approach, given the underlying PTPd implementation, in order to avoid a master processing messages from itself; this issue should be taken care of, however, when the implementation is integrated with an automated key management scheme that can provide SPD and SAD lookups. The requirement of using the peer delay mechanism as opposed to the delay response mechanism similarly has no bearing on the security implementation.

4.1.2 Lack of Key Management

The biggest limitation to the our implementation is the lack of an automated key distribution and management scheme; the related security parameters instead need to be specified in the configuration file passed to the daemon at startup, and maintained as part of PTPd's program state. This does not significantly hinder the immediate verification approach, but it does pose some unique issues for the delayed verification approach, which will be discussed below. The draft standard maintains that key management is outside its scope, and thus this is not considered a limitation of the IEEE 1588 draft standard's security proposal. It is expected that the details of integration with a key management scheme will be specified by individual profiles as they adopt the new security mechanism recommendations [2].

4.1.3 Delayed Verification Issues

There are a few issues to consider in our implementation when using the delayed verification approach: trust anchor distribution, key chain expiration, and authentication modes. The first two stem from the lack of an automated key management scheme, while the lattermost relates to the underlying PTP implementation.

As described in Section 3.4.2, after the master generates the key chain, the last key to be generated — serving as the trust anchor — must be distributed to all participating slaves, so that they may verify disclosed keys as they receive them. This should be done securely as part of a bootstrapping phase of the key management scheme. To circumvent this problem, we did not enforce a distinction between master and slave at startup, and instead had the daemon generate the key chain (as a master would) and save the trust anchor in a `securityDS` structure (emulating a slave receiving the trust anchor as part of a bootstrapping phase). Since this happens prior to the protocol engine starting and thus before master or slave states are entered, once the daemon enters the slave state, it actually has access to the entire key chain. This is an unrealistic state, and it prevents the implementation from being used in a real application. However, in order to continue with the proof of concept implementation, in all delayed security processing code meant to be executed by the slave alone, we follow the spec as if we did not have access to the key chain, instead only accessing the trust anchor and the list of derived verified keys.

Another similar issue arises at the end of the last time interval, when the key chain expires. In the last time interval i, given a disclosure delay d, slaves will receive the key for interval $i - d$ — using it to verify messages buffered in interval $i - d$ — and the messages corresponding to intervals $i-d+1$ ... i will still be buffered. In our implementation, the last d keys corresponding to these buffers are never disclosed, as we have no way of generating a new key chain, distributing a new trust anchor, and continuing the delayed verification protocol. When integrated with an automated key management scheme, a possible solution would be as follows: prior to the start of a new set of intervals, the master would generate a new key chain, and distribute the corresponding trust anchor. When entering the new set of intervals, slaves would need to keep the old trust anchor for the first d intervals, and the master would similarly need to keep the last d keys from the old key chain, disclosing them during the first d intervals of the set of new intervals. After this, slaves should switch to the new trust anchor, as they will be receiving disclosed keys from the new key chain. These details are left for a key management scheme to specify, but should be carefully considered when integrating with a delayed security solution.

In Section 2.5, we introduced the two modes of operation involved in delayed processing message reception: authenticated mode and unauthenticated mode. In both modes, a received packet that is safe is first buffered; beyond this, they can both can be split into two steps. In the first step of authenticated mode, the buffered message is not used yet as part of the PTP protocol. We consider this a distinct step because it would require modification to PTPd, by returning from the current function, and possibly by saving some program state. In the second step, a previously buffered message that has passed ICV verification must be used; this would also require modifications beyond the scope of this project. In the first step of unauthenticated mode, the buffered message is used in the PTP protocol; this is in contrast with the first step of authenticated mode, and is trivial to implement, as it requires letting regular PTPd control flow continue to the message handling component. In the second step of unauthenticated mode, if a previously buffered message failed ICV verification, then the effect it had on clock synchronization must be undone; this would likely require an interaction with or possibly the modification of the PTPd servo component. As described in Section 3.5.1, we chose to model unauthenticated mode, as it would allow PTP synchronization to continue underneath our security mechanism by way of step one. However, in lacking a complete implementation of step two, there is no guarantee of authenticity and integrity; the only thing gained is knowledge that these properties were compromised.

The trust anchor distribution and key chain expiration issues would both be resolved when integrating an automated key management scheme such as TESLA, while implementing the modes of operation would likely require modifications relating to the PTPd servo component. Currently, however, each one of these issues makes our implementation unsafe to use outside of a research setting.

4.1.4 Immediate Verification

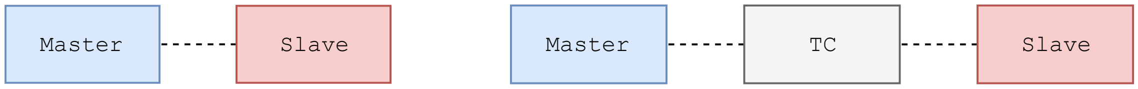
Due to the issues outlined above, the immediate verification approach is the most feasible option in the absence of a key management scheme, although it would require manual configuration and maintenance of the required security parameters. Even if an automated key management scheme becomes available, the delayed approach would still require more work in order to implement the details relating to the authentication modes.

4.2 Functionality

In this section, we describe the process and results of evaluating the functionality of our security implementation. The goals of the functionality testing were to ensure that the modified PTPd functions correctly with regards to the security mechanism, and that it still functions correctly with regards to synchronizing time. First, we will describe the experimental set up, and the organizational structure of the tests used to verify the functioning of the security mechanism. Next, we will describe the test cases themselves, split into sections based on their organizational structure. Finally, we will look at additional means we used to examine whether time synchronization functionality was compromised by the security mechanism implementation.

4.2.1 Experimental Setup

To test the functionality of our security implementation, three devices were used: a modified PTPd instance running on a Raspberry Pi, a modified PTPd instance running on a MacBook Pro, and a PTP stack running on Microchip's EVB KSZ9477 acting as a *Transparent Clock*. For each test case, we first set up the conditions, then ran the PTP instances, and finally verified expected behavior. Setting up the conditions entailed setting up a topology (physically connecting the involved devices), and setting the appropriate parameters in the configuration file. All configuration files were set up to adhere to the power profile as a base configuration, and security parameters were set according to the requirements of each test case. The two topologies that were tested were a master directly connected to a slave, and a master running through a TC to a slave (Figure 4.1). Verifying expected behavior was done by observing the status logs produced by PTPd, and when necessary, inspecting the packets via Wireshark.



(a) topology 1: direct (b) topology 2: multi-hop

Figure 4.1: topologies for testing security functionality

The organizational structure of all the functionality tests is shown in Figure 4.2. At the top level, we categorized the tests by the security verification approach: regression (where security is disabled on all instances), immediate, and delayed. The next level is split based on the topology used: direct, as shown in Figure 4.1a, and multi-hop, as shown in Figure 4.1b. For the direct topology, the PTPd instance to be run as master was run on macOS, and the slave instance was run on Raspberry Pi. For the multi-hop topology, the master instance was run on Raspberry Pi, and the slave instance was run on macOS, to facilitate Wireshark observations. At the topology level, we begin numbering of the test cases as shown in Figure 4.2, in order to better organize the configuration file names corresponding to each individual test case. The next level is split based on test type, which includes basic functionality scenarios and attack scenarios. The final level holds the test cases themselves; at this level, Figure 4.2 simply shows the number of test cases that were used for the given permutation of parent categories. The full list of test cases, organized according to this structure and including a description of the setup and the steps used to verify expected behavior is listed in Appendix B. The naming scheme for the configuration files that correspond to each test case is `<security approach>_<test case number>_<state>.conf`. As an example, `imm_1.2.4_m.conf` is the configuration file for the fourth test case under the attack scenarios for a master using the immediate verification scheme, in the direct connection topology.

4.2.2 Regression Tests

The purpose of the regression tests was to show that our modified PTPd still functioned correctly when built without security enabled, by running the configuration script without the `--enable-security` switch. Building PTPd this way should result in an executable that is no different than one produced by an unmodified PTPd. We verified normal functioning of PTPd by observing the status logs in real time, and by inspecting the messages in Wireshark. In the status logs, we checked the following: the mean path delay reaches a stable value; the offset from master in the slave's log approaches zero and reaches a stable value; the message counters show sent *Announce*, *Sync*, and *Follow_Up* counts increasing for the master, received *Announce*, *Sync*, and *Follow_Up* counts increasing for the slave, and all *Pdelay* message counters increasing for both master and slave. In Wireshark, we verified that no messages have an AUTHENTICATION TLV attached. These same means of verification were used in subsequent test cases; as such, the descriptions of subsequent tests will only mention deviations from these means of verification or additional means of verification.

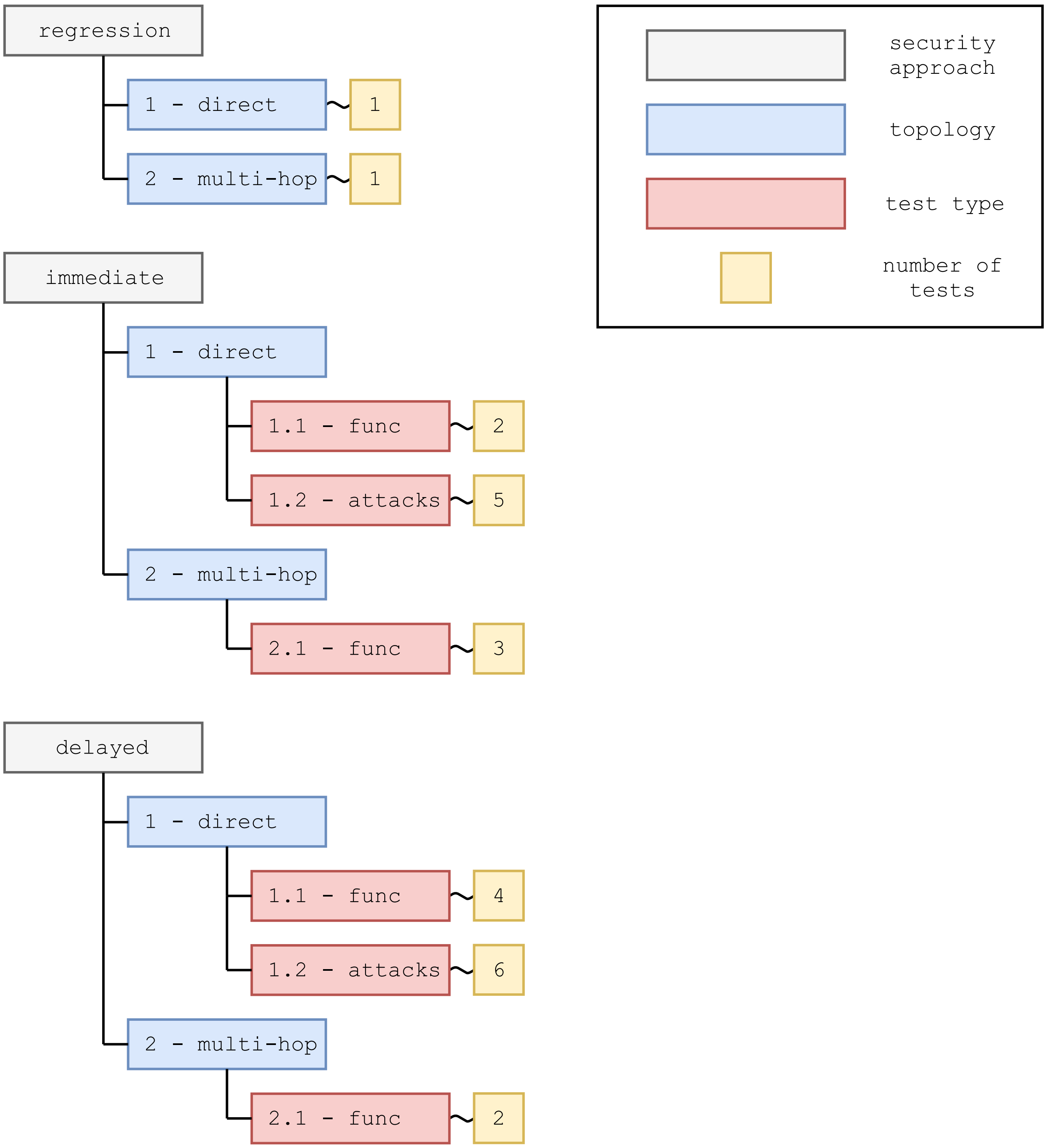


Figure 4.2: functionality tests structure

4.2.3 Immediate Verification Tests

As described in Section 4.2.1, the third level of the organizational structure makes a distinction between basic functionality scenarios, and attack scenarios. We will look at each of these test type categories in turn below.

The functionality scenarios we tested were a fully secured network, emulated by enabling security on both PTPd instances, and a mixed network, emulated by enabling security on the master PTPd instance, and building the slave without security. In the setup of the fully secured scenario, both PTPd instances had the same security options set, notably, the key and the ICV calculation algorithm. Verifying expected behavior was done as described previously, but also included checking that all messages had an AUTHENTICATION TLV appended via Wireshark. In the mixed network scenario, everything should function normally, with the slave ignoring the AUTHENTICATION TLVs present on incoming messages. To accommodate such a network, the master needed to be configured to accept unsecured *Pdelay* messages; this option is available as a parameter in the configuration script.

The attack scenarios we tested included three cases emulating a rogue master, and two cases emulating a man-in-the-middle (MITM) attack. In all of these cases, the slave was built with security enabled, and configured for immediate verification processing using HMAC as the ICV calculation algorithm. The rogue master cases included: the master PTPd instance being built without security enabled; the master PTPd instance being configured to use a different key from the one used by the slave; and the master PTPd instance being configured to use GMAC for ICV calculation (as opposed to HMAC for the slave). In the first case, the slave was confirmed to reject all incoming messages as the SECURE flag in the message headers was not set (confirmed via Wireshark), and the `authenticationTLVExpected` error counter in the status log was ever increasing. In the other cases, the slave similarly rejected all incoming messages: when the master used a different key, the slave reported `icvMismatch` errors; when using different ICV calculation algorithms, the slave reported `lengthMismatch` errors, as the slave was expecting messages secured with HMAC, which results in an AUTHENTICATION TLV length of different size compared to messages secured with GMAC.

For the MITM attack scenarios, we emulated a slave receiving a bogus message hypothetically altered by an attacker. To implement this, we created a separate testing branch in which we modified our implementation to support a master altering a given message by rewriting either its payload or its ICV. The message to be altered is specified by its sequence ID (by setting a `targetMessageSeqId` variable), and the master either sets the target message's PTP payload or its ICV to all `0xff` bytes, based on the attack type (`ICV_ALTERED` or `PAYLOAD_ALTERED`). The verification process was the same for both cases: we checked the slave status log to find the `icvMismatch` error counter at 1; in the slave event log, where more descriptive error messages are reported, we found the security informational message reporting an ICV mismatch on the message with the given sequence ID number.

For the multi-hop topology, we did not test any of the attack scenarios, as we would not expect to see any different behavior based on there being a TC in-between the master and slave instances. However, we did need to create two test case variants for the fully secured scenarios. Since the PTP stack running on the Microchip device does not implement the security mechanism, from the point of view of a slave running the immediate security approach, the TC is effectively performing a MITM payload attack by updating the correction field. To support such a case of a network with TCs that do not implement the security mechanism, we added parameters to allow masters and slaves to ignore the correction field in their ICV calculations. To test the functionality of this option, we created two variants of the fully secured scenario: one with the master and slave PTPd instances both configured to ignore the correction field in ICV calculation, and another with this option turned off. In the former case, PTP functioned normally, with received *Sync* messages containing non-zero values in the correctionField; in the latter case, the slave's status log showed no received *Sync* messages (since the slave was dropping them) and reported increasing `icvMismatch` error counts, while *Sync* messages were indeed observed in Wireshark.

The cases described in this section were all replicated in the delayed verification tests as well, and as such they will only be mentioned again to highlight the differences in set up and verification of expected behavior.

4.2.4 Delayed Verification Tests

For the delayed verification test cases, setup involved setting the additional security parameters that are unique to delayed verification: a start time as a Unix epoch time value, a chain length (number of intervals), interval duration in seconds, a disclosure delay, and *d_t* in seconds. The start time needed to be set so that by the time a given test case is run, the PTPd instances were within a given interval; the chain length was set to 500 so as to not worry about chain length expiration; the interval duration was set to 5 and the disclosure delay to 2 in order to be better able to observe expected behavior, and *d_t* was set to 0.0002.

In the functionality scenarios in which PTP was expected to function normally, the verification of expected behavior included a few additional checks on top of those described in the immediate case: in the slave's status log, the `safePacket` counter increased and the `keyVerificationSuccesses` counter increased once every 5 seconds (one new verified key per time interval); in Wireshark, outgoing messages from the slave had no AUTHENTICATION TLV attached; all messages from the master had an AUTHENTICATION TLV attached; and *General* messages from the master included a disclosed key, while *Event* messages did not.

As mentioned previously, all test cases described for the immediate approach were replicated in the delayed approach. In addition to these, there were three more test cases designed to verify the functionality specific to delayed security processing. For functionality scenarios, we tested a slave correctly identifying an unsafe packet, and a slave missing a disclosed key for an entire interval. For attack scenarios, we added a MITM attack in which a disclosed key was altered. All of these tests required modifications to the test branch implementation.

For the unsafe packet test, we modified the test branch implementation to allow a master to doctor the `keyId` field of one packet — the first one — in a specified interval. The desired interval was set in a `unsafePacketInterval` variable, and a boolean `unsafePacketSent` was maintained to enforce the modification of only the first packet of the interval. During delayed processing, when the master populates the `keyId` field, it uses `unsafePacketInterval` and `unsafePacketSent` to check whether to doctor an unsafe packet; if yes, it sets the `keyId` field to `currentInterval - disclosureDelay`. As described in Section 2.5, when a slave receives such a message, it would consider it to be unsafe, as the key corresponding to the advertised interval would have already been disclosed. We verified expected behavior by monitoring the slave status log and observing the `unsafePacket` counter having been incremented during the specified interval.

To test the key verification process in the case when one or more previous keys are missed, we modified the test branch implementation to allow a master to skip packing a disclosed key for an entire interval, specified by `intervalToSkipDisclosure`. We verified expected behavior in Wireshark, by observing that no messages during the appropriate interval contained a disclosed key. We also modified the slave to dump buffered messages to the event log once they get verified; this enabled us to see two consecutive buffer dumps, in the correct order, corresponding to when the next disclosed key was received, confirming proper key verification behavior.

For the MITM attack in which a disclosed key is altered, we used a similar approach as with the other MITM attacks, modifying the test branch implementation to allow a master to set the disclosedKey field of the AUTHENTICATION TLV to all `0xff` bytes for a given interval. We observed expected behavior through messages reported by the slave in the event log indicating a key verification failure.

The test cases described so far, outlined in Figure 4.2, were designed primarily for the purpose of verifying the security implementation functionality, although through the verification of expected behavior, we also see that the security mechanism did not negatively impact the regular functioning of PTP. In order to more robustly verify that the time synchronization functionality is indeed in tact for each security approach, we ran a series of additional tests, which will be described next.

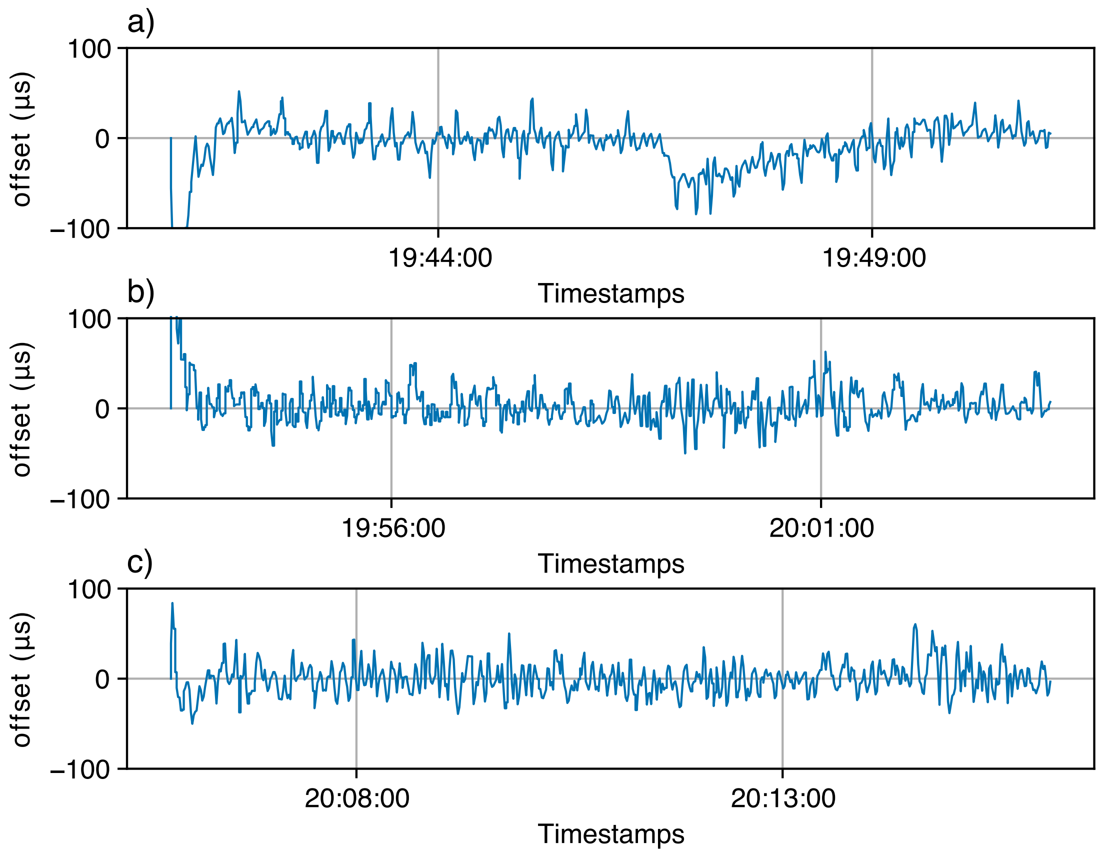


Figure 4.3: offset from master (Linux):
a) disabled b) immediate c) delayed

4.2.5 Time Synchronization Functionality

To get a better idea of the impact of the security mechanism on time synchronization, we needed to compare the offset from master values reported by a slave PTPd instance running with security disabled against the the values reported by a slave running with security enabled, across both verification approaches. Using a Linux desktop machine, we did three separate runs of the daemon as a slave for 10 minutes each: one with security disabled, one with immediate security processing enabled, and one with delayed security processing enabled. For the best comparison, we needed the slave clock to start from a similar initial offset from master for each case. To accomplish this, we did an initialization run prior to starting the experiments to get the slave clock synchronized to the master to within tens of microseconds. Then, we waited 2 minutes with PTPd shut down prior to starting the first test, allowing the slave clock to drift such that its starting offset from master on the next run would be close to 100 μs. We then ran the next two cases one after the other, waiting 2 minutes in between each test with PTPd shut down. The results are shown in Figure 4.3.

First, we see that with security disabled, offset from master reaches an equilibrium within the first minute, at which point it oscillates between approximately +/-40 μs, staying in a range of 80 μs. With both security approaches too, within the first minutes the offset from master reaches an equilibrium, oscillating between approximately +/-40 μs. This shows that for both security approaches, PTP synchronization as measured by offset from master is achieved to a comparable degree as with security disabled.

We ran the same set of tests using different platforms, and observed similar results; Figure 4.4 shows the results from macOS, and Figure 4.5 shows the results from Raspberry Pi.

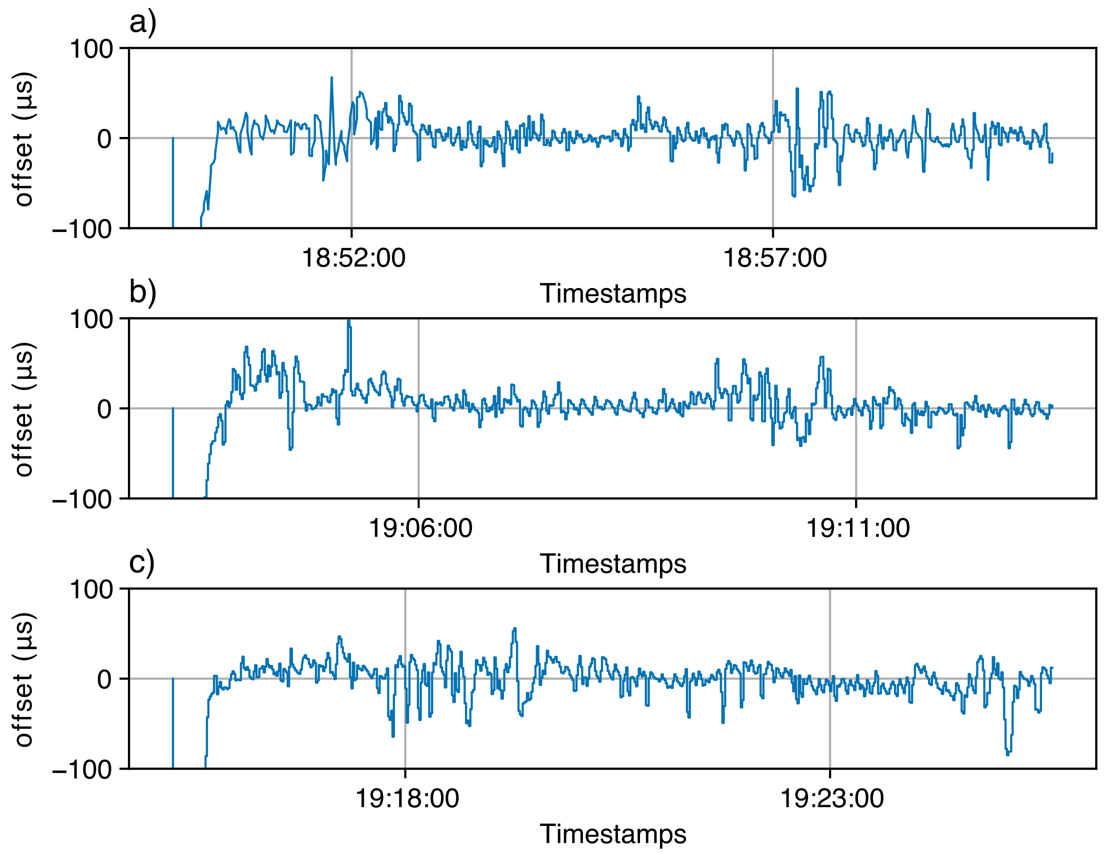


Figure 4.4: offset from master (macOS):
a) disabled b) immediate c) delayed

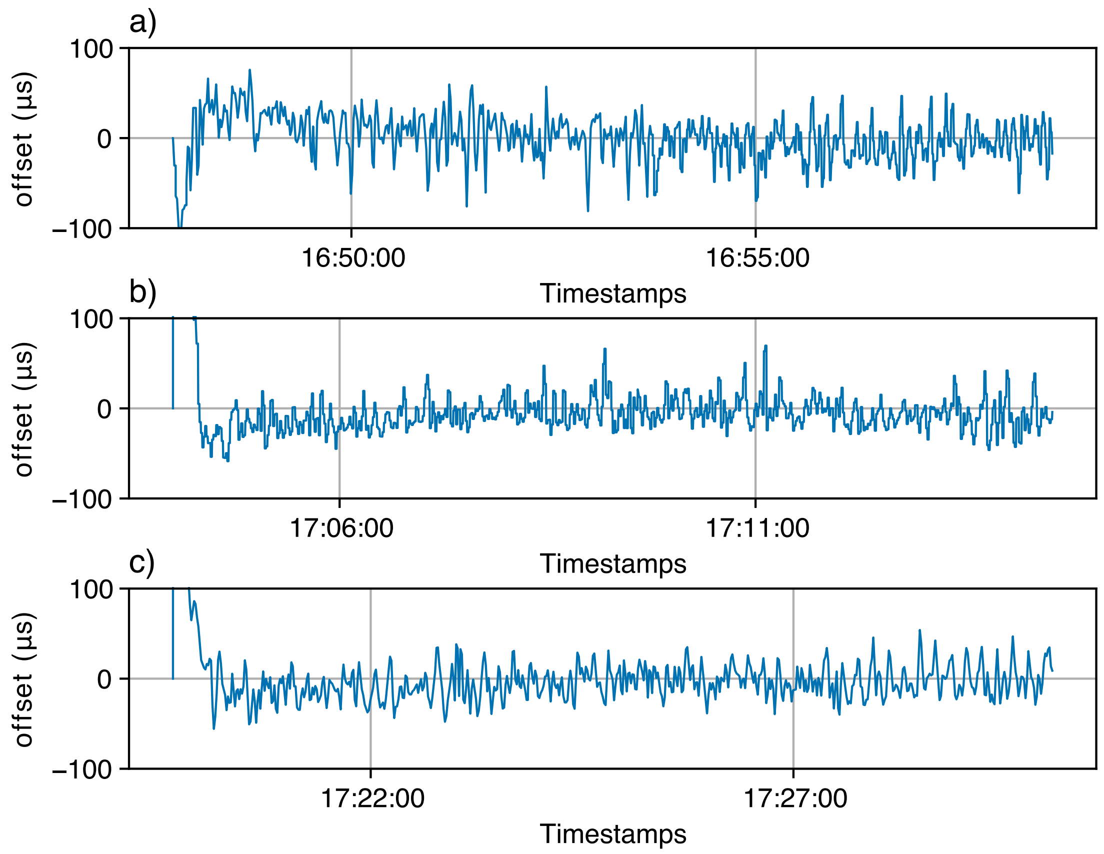


Figure 4.5: offset from master (Raspberry Pi):
a) disabled b) immediate c) delayed

4.3 Performance

The goal of the performance testing was to measure how much time security processing adds to the normal functioning of PTPd. This was done by measuring elapsed time from the entry point of the security code to the exit point, with security disabled to get a baseline (effectively timing just a conditional statement that evaluates to false), and then for both the immediate and delayed verification approaches.

4.3.1 Experimental Setup

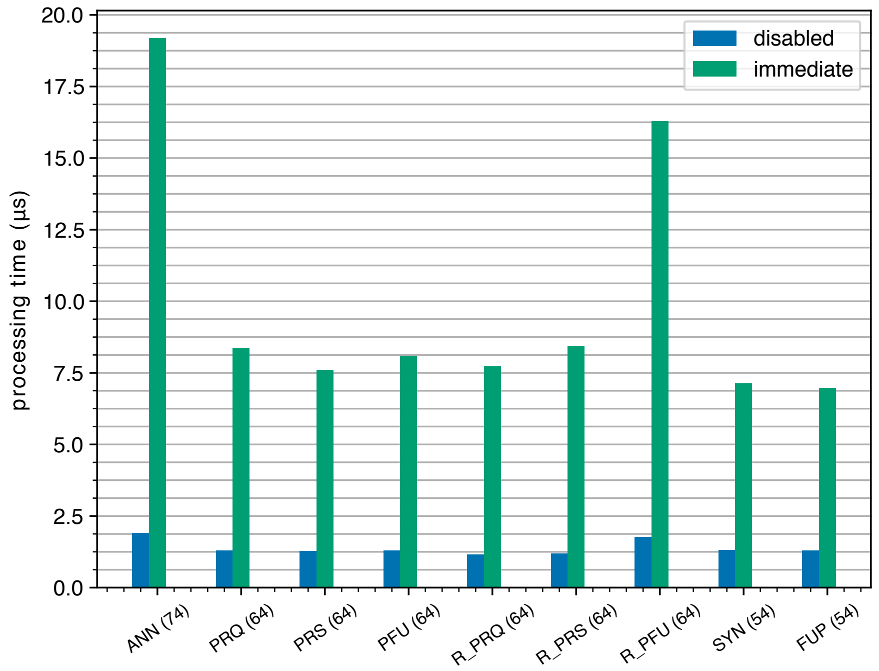
In order to get measurements on all message types for both master and slave, we inserted a timing probe to collect start times and end times around added security processing code. The timing probe consisted of a call to `clock_gettime(CLOCK_MONOTONIC_RAW, &ts)`, which uses a clock that increments monotonically from an arbitrary point, and is unaffected by frequency or time adjustments. To time security processing on outgoing messages, we inserted the probe around every call to `addAuthenticationTLV()`; to time security processing on incoming messages, we inserted the probe around the security block in `processMessage()`. After the end time gets collected, the difference between the end time and start time is calculated and stored in a `securityTiming` struct with a call to `recordTimingMeasurement()`. Only the first 1000 measurements for each message type get recorded; once all 1000 measurements are recorded for each relevant message type (a master does not wait on received *Announce*, *Sync*, and *Follow_Up* messages, for example), the timing measurements get dumped to a security timing file. Measurements were taken for both master and slave states, for all message types, and for three different platforms — Linux desktop, Linux Raspberry Pi, and macOS laptop. In all charts and graphs in this section, message type names were abbreviated as follows: *Announce* (ANN), *Sync* (SYN), *Follow_Up* (FUP), *Pdelay_Request* (PRQ), *Pdelay_Response* (PRS), *Pdelay_Response_Follow_Up* (PFU), with receive variants being preceded by an *R* as in R_ANN to indicate received *Announce* messages.

4.3.2 Immediate Processing

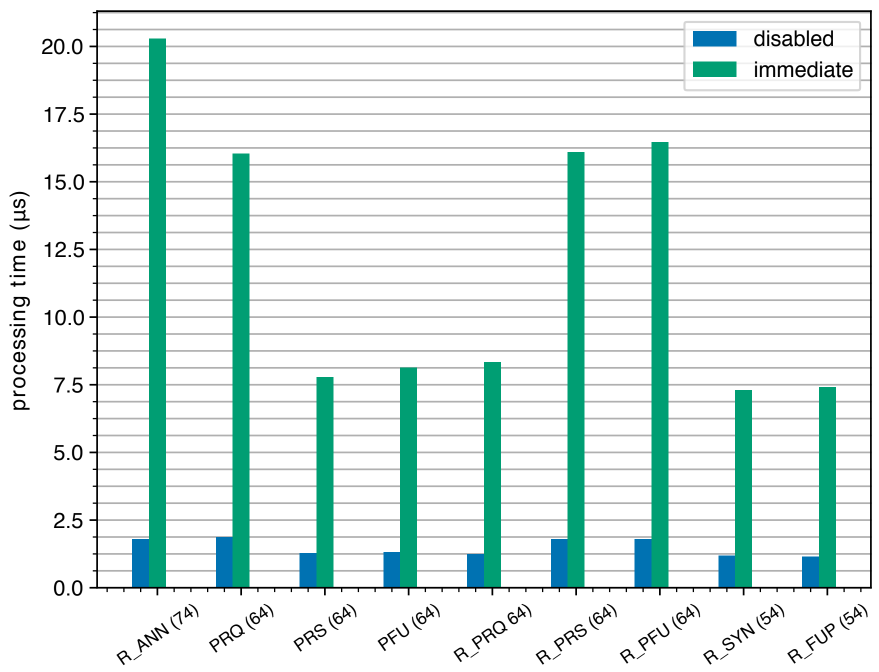
Figure 4.6 shows the median values for performance of the immediate verification approach for every message type, next to the values from the disabled run to use for a baseline comparison. The immediate security processing measurements range from 7-20 μs; these values include the time it takes to simply evaluate the conditional statement to determine whether to enter security code, which is shown by the *disabled* bars to be just under 2 μs. We would expect that message size would correlate with performance cost, considering the costliest operation in security processing is the calculation of the ICV; as such, we arranged the data by message size in bytes, which includes the PTP header, payload, and AUTHENTICATION TLV (when using HMAC as the ICV calculation algorithm). The data do not show this correlation: for example, in Figure 4.6a R_PFU messages show about double the cost as other messages of the same size, and in Figure 4.6b, this unexpected result holds for PRQ and R_PRS messages as well. We also see for both the master and slave that the smallest messages at 54 bytes, SYN and FUP (R_SYN and R_FUP for the slave), show a similar performance cost as other larger messages of 64 bytes; this is also contrary to the hypothesis that a correlation would exist between message size and performance cost. The only support for this hypothesis is the consistently higher cost of ANN messages, which are the biggest at 74 bytes. It is possible that if a correlation exists, it is not linear, and rather exists across certain size boundaries.

4.3.3 Delayed Processing

Figure 4.7 shows the median values for performance of the delayed verification approach for every message type, next to the values from the disabled run to use for a baseline comparison. The delayed processing costs shown here are on the same order of magnitude as the immediate processing costs, ranging from 12-26 μs (including a baseline cost of just under 2 μs). As we did for the immediate analysis, we arranged the data by message size in bytes; this results in a different message ordering compared to Figure 4.6, since ANN, FUP and PFU messages from



(a) master immediate



(b) slave immediate

Figure 4.6: medians for immediate security by message size

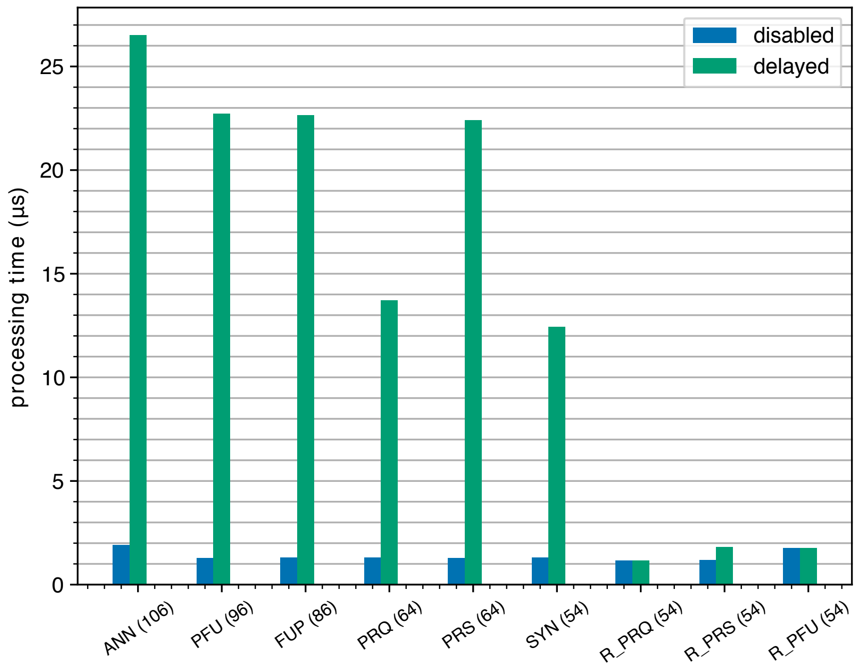
the master contain disclosed keys, adding another 32 bytes to the message size. Also, since all *Pdelay* messages sent by the slave are without an AUTHENTICATION TLV, R_PRQ, R_PRS and R_PFU from Figure 4.7a and PRQ, PRS, and PFU from Figure 4.7b retain their original size of 54 bytes. The correlation between message size and performance cost is once again shown to be spurious in Figure 4.7a, in which we see PFU and PRS messages having nearly equal performance cost despite a size difference of 32 bytes.

Using data gathered from the slave measurements, we can confirm several expected results. Since the slave does not add an AUTHENTICATION TLV to outbound messages when using the delayed verification approach, we would expect the processing costs for PRQ, PRS, and PFU messages to be comparable to those from the disabled run; this is confirmed in Figure 4.7b. A similar observation can be made for the master in Figure 4.7a for R_PRQ, R_PRS and R_PFU messages, since security processing is skipped for those message types. A slave does enter security processing code for all receiving messages, but due to the delayed verification approach, ICV calculation is never performed during *Event* message processing, and thus we would expect processing costs for R_SYN, R_PRQ, and R_PRS to be comparable to those from the disabled run; this too is confirmed in Figure 4.7b. We would expect *General* messages to incur additional processing costs due to key verification, and we do see this reflected in higher values for R_ANN and R_FUP, but not in R_PFU. This may be due to the possibility that by the time a slave receives the first R_PFU message in a given time interval, it may have already received and verified the disclosed key for that interval in a R_ANN or R_FUP message. Finally, associated with key verification is ICV verification on all buffered messages corresponding to the newly verified key; this should incur a huge performance cost, but only for the subset of received messages that disclose a new key. We do not see this performance cost accurately reflected in Figure 4.7b, as the plotted values are medians, and a new key is only verified once per time interval. However, if we plot all 1000 measurements, we should be able to confirm this effect. Figure 4.8 shows all 1000 measurements for R_ANN from a slave using delayed processing; in Figure 4.8a each measurement is arranged along the x-axis in the order the messages were received, and in Figure 4.8b each measurement is sorted by value (referred to as CDF for cumulative distribution function). The spikes up to 350 μs in Figure 4.8a indicate the verification of a key and the subsequent ICV verification of all corresponding buffered messages; the periodicity of these spikes reflects the fact that this event happens only once per time interval. In Figure 4.8b we see that 800 of the received messages incurred a minimal cost of approximately 5.82 μs, while 200 of the received messages incurred a much higher cost in the range of 350-400 μs. Considering the time interval duration was set to 5 seconds, and the *Announce* message interval rate is 1/s, this is exactly what we would expect to see.

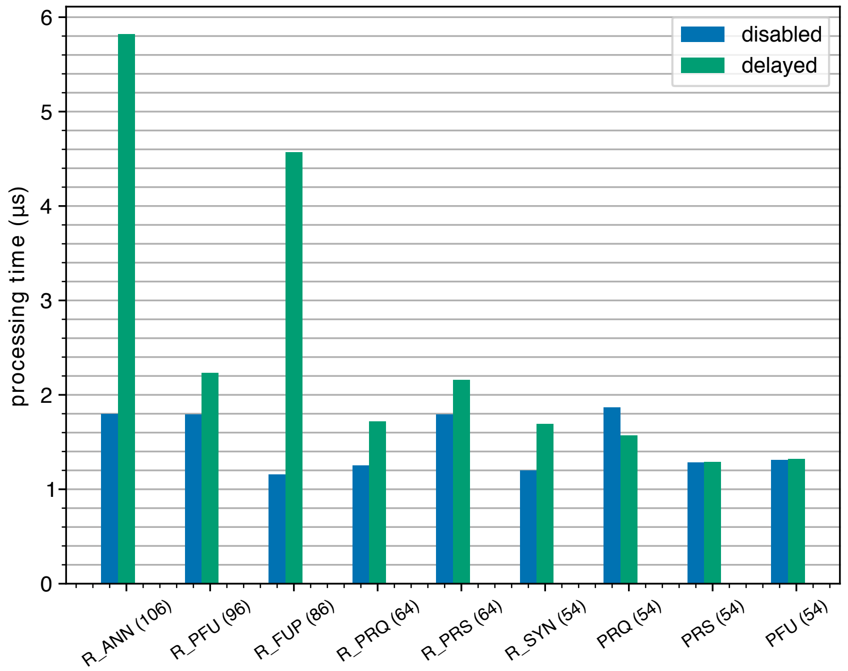
4.3.4 Immediate vs Delayed

To compare the two security approaches, we plotted the median values for each message type for disabled, immediate, and delayed in the same chart, as shown in Figure 4.9. The first point to note is that both security approaches incur performance costs on the same order of magnitude, in the tens of microseconds. Second, we note that that the performance costs in the immediate verification approach are shared across all message types, and between both master and slave; in the delayed verification approach, however, only the master incurs costs across all outgoing messages (Figure 4.9a), presumably due to the fact that an ICV calculation must be done on each message, while the costs for the slave are minimal for all incoming messages, with the exception of whichever received *General* messages happen to include the first disclosed key of the given time interval (mostly *Announce* messages in this data).

In Figures 4.10 through 4.27, each figure shows all 1000 data points from each security approach (disabled, immediate, and delayed) for a given message type; each figure consists of two graphs plotting the same data, one according to the order in which messages were received (time), and the other with the values sorted (CDF). The first half — Figures 4.10 through 4.18 — show data for master messages ANN, SYN, FUP, PRQ, PRS, PFU, R_PRQ,



(a) master delayed



(b) slave delayed

Figure 4.7: medians for delayed security by message size

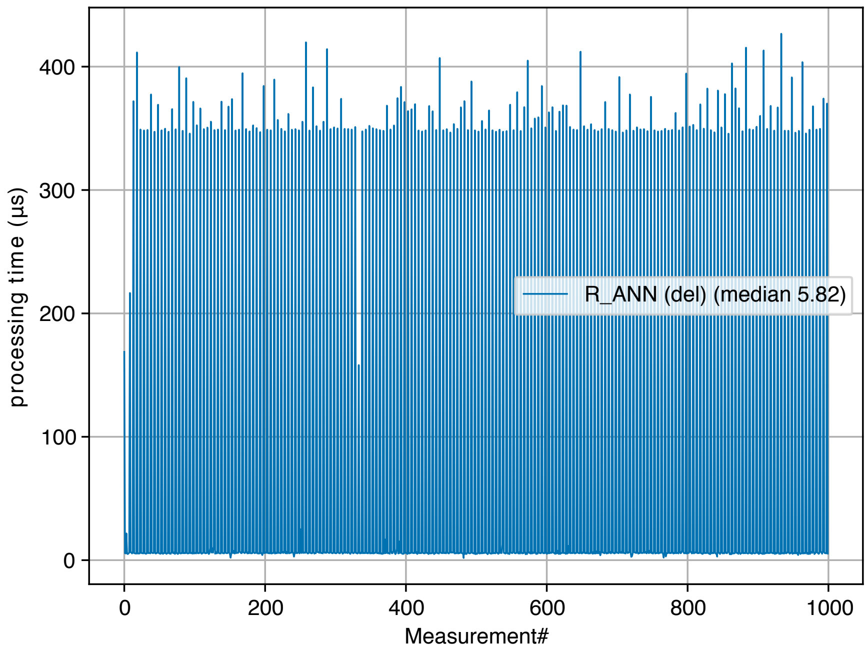
R_PRS, and R_PFU while the second half — Figures 4.19 through 4.27 — show data for slave messages R_ANN, R_SYN, R_FUP, PRQ, PRS, PFU, R_PRQ, R_PRS, and R_PFU. All of these figures are included for the sake of completeness, and to serve as an additional means of interpreting the median charts, which may not always paint a complete picture (as seen in the case of received *Announce* messages from a slave using delayed verification). Interesting cases to note are ones which show clear periodicity, indicated by oscillating values in the time graphs (often seen as a thick block due to the resolution), and by a bimodal distribution in the CDF graphs. For the master data, these include Figures 4.13 (PRQ), 4.17 (R_PRS), and 4.18 (R_PFU); for the slave data, these include Figures 4.19 (R_ANN), 4.21 (R_FUP), 4.22 (PRQ), 4.25 (R_PRQ), 4.26 (R_PRS), and 4.27 (R_PFU). In the cases of received *General* messages for a slave using delayed security processing, there is a clear explanation for this periodicity. It is not clear what causes these oscillations in the other cases, however. The jumps in performance cost are so small (within 10 μs) that we may just be seeing noise.

4.3.5 Comparison Across Platform

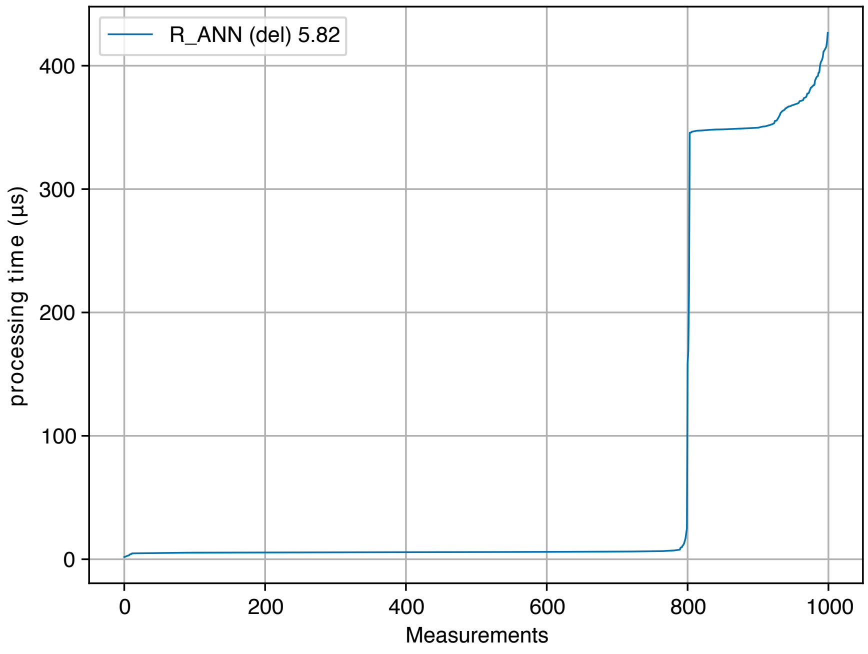
As a final point of comparison, we ran the same set of tests across two more platforms — macOS laptop and Raspberry Pi — in addition to the Linux desktop machine, and plotted the results to show median performance cost for each message type across each platform. Figure 4.28 compares the performance costs across platforms when using PTPd with security disabled (simply timing the evaluation of the conditional statement that evaluates to false), Figure 4.29 compares the performance costs across platforms when using the immediate verification approach, and Figure 4.30 compares the performance costs across platforms when using the delayed verification approach.

Although there are many variables to consider in cross-platform comparisons, we predicted that most important one that would impact security processing would be processor speed: in this regard, the Mac and Linux desktop machines were comparable, at 2.9 GHz and 2.7 GHz respectively, while the Raspberry Pi is noticeably lower, at 1.2 GHz. Another factor that may carry some influence is the OpenSSL version used for each platform: the Mac and the Linux desktop machines were both using version 1.0.2, with the macOS version being a slightly newer revision (December 2017 vs. March 2016), while the Raspberry Pi was using version 1.0.1 (May 2016). The release notes for each version did not indicate any obvious changes or updates to the OpenSSL HMAC algorithm we used in these tests, but it is possible that other changes may have an impact, which is why we bring the versions into consideration.

With security disabled, we see in Figure 4.28 a clear ordering in terms of performance, with the Mac being slightly better than the Raspberry Pi, while the Linux desktop machine lags behind both. However, considering the scale (under 2 μs), it is possible that these differences may have to do with the timing probes or influenced by other factors. In Figures 4.29 and 4.30, we see that the Linux machine and the Raspberry Pi generally outperform the Mac on the majority of messages, although there are several exceptions. Importantly, the performance costs incurred by the Raspberry Pi are comparable to those of the Linux desktop machine, despite the difference in processors and despite using the slightly older OpenSSL version.

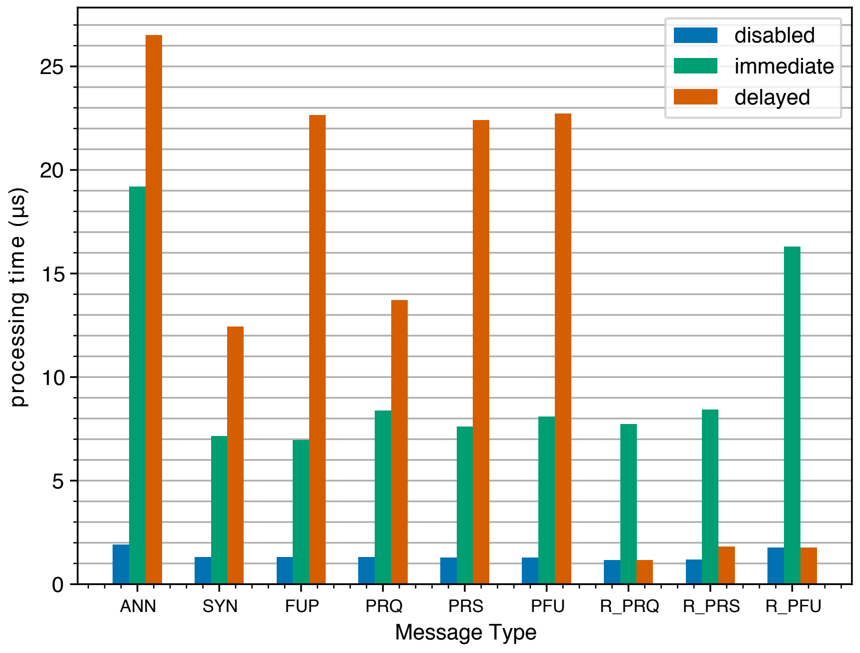


(a) time

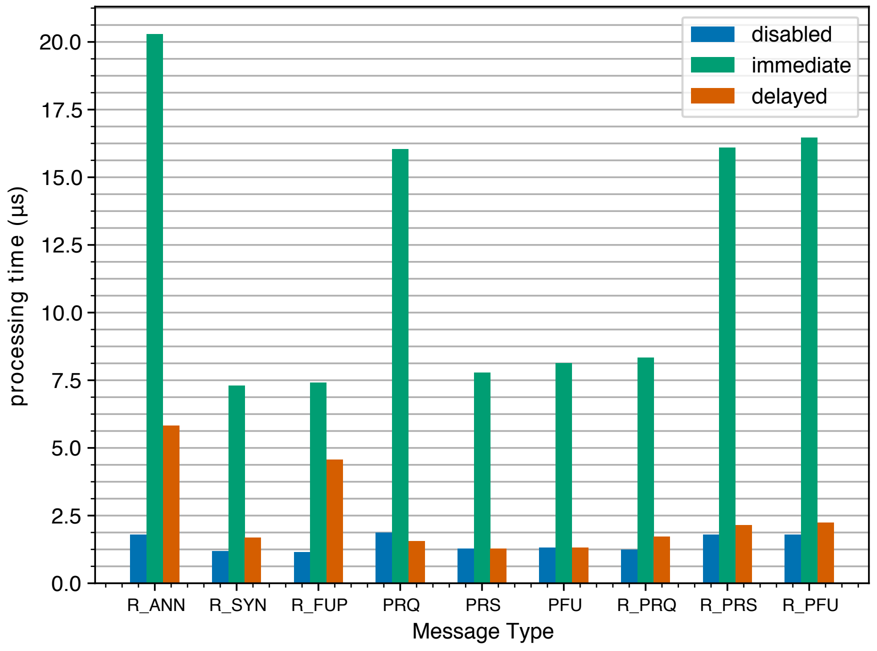


(b) CDF

Figure 4.8: receive *Announce* measurements for delayed processing (slave)

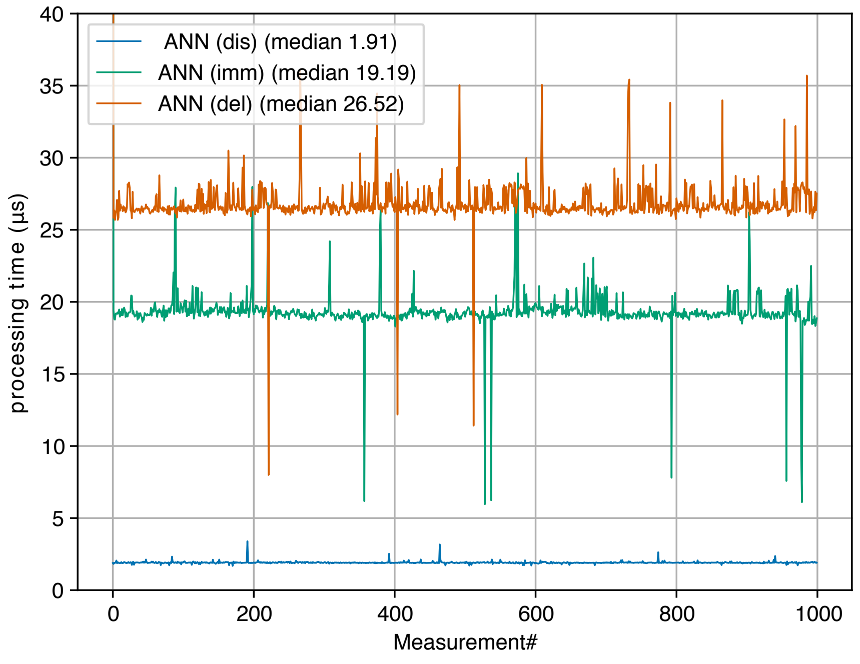


(a) master

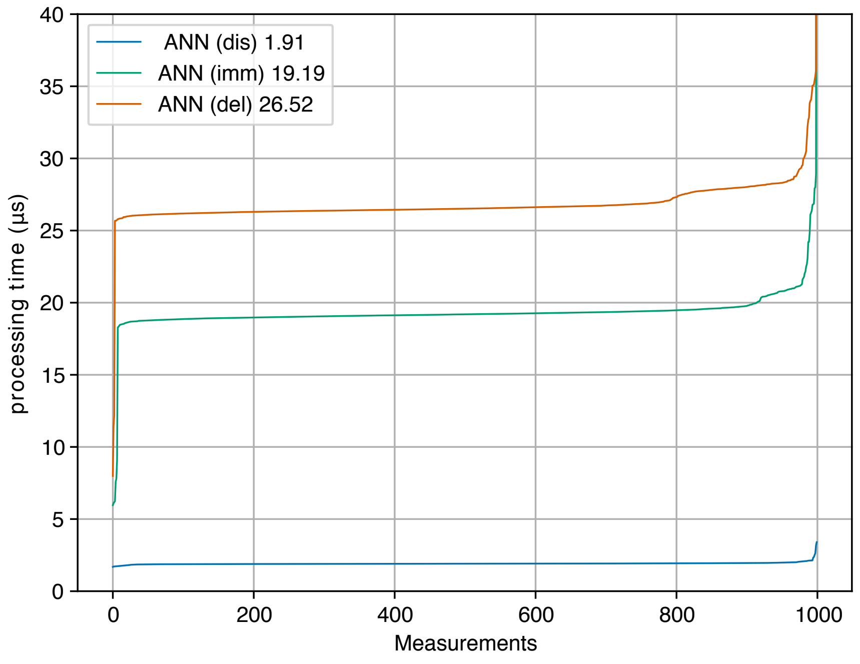


(b) slave

Figure 4.9: medians across security approach (Linux)

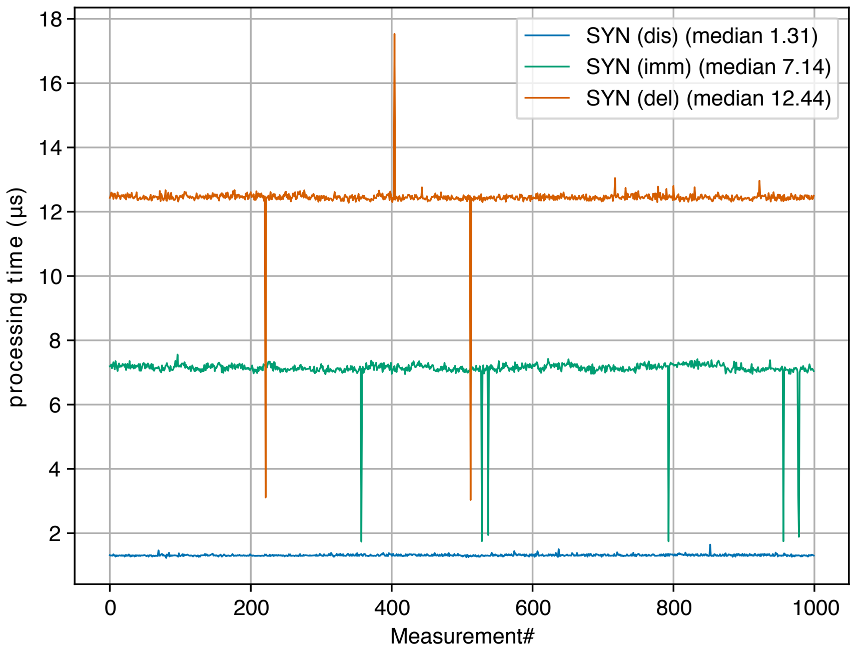


(a) time

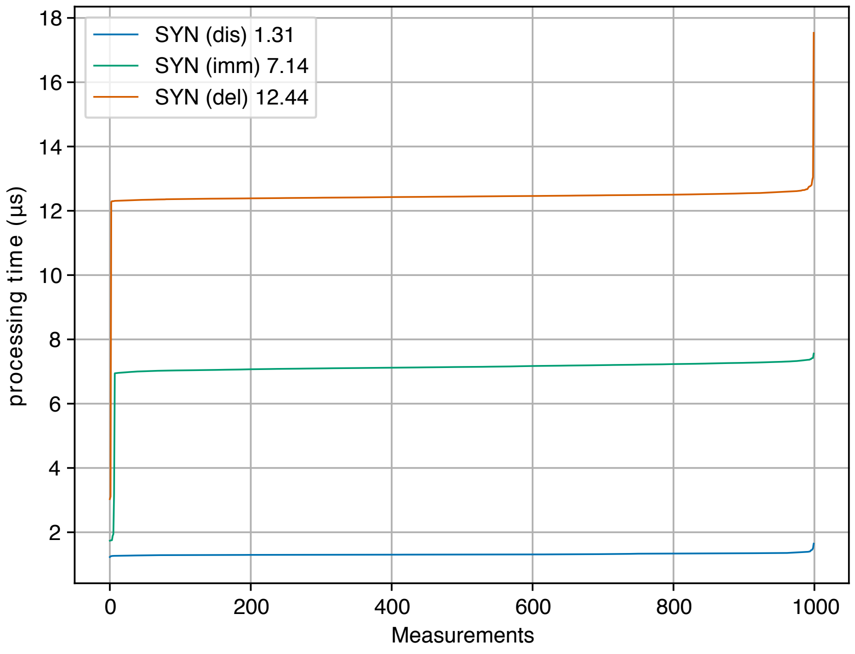


(b) CDF

Figure 4.10: send *Announce* measurements across security approaches (master)

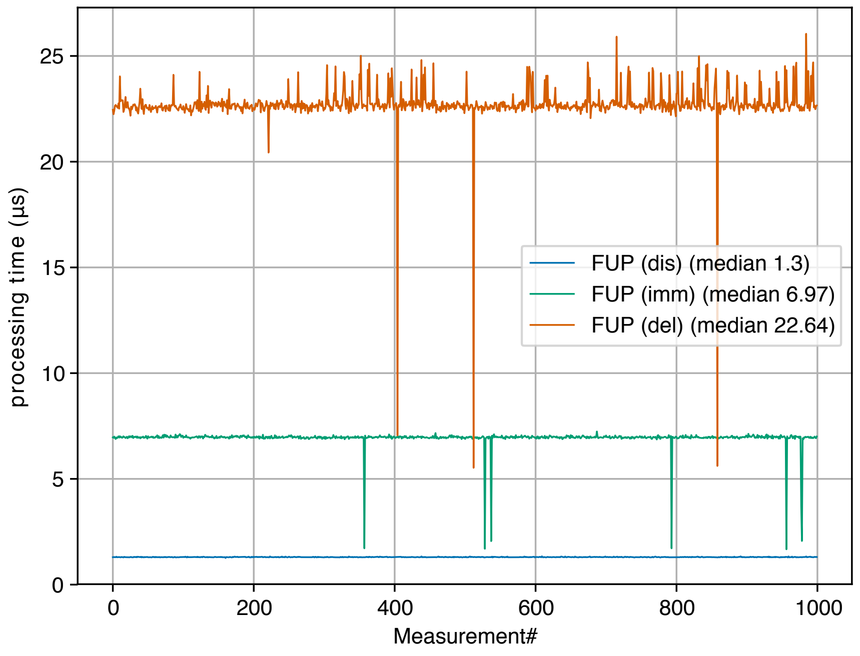


(a) time

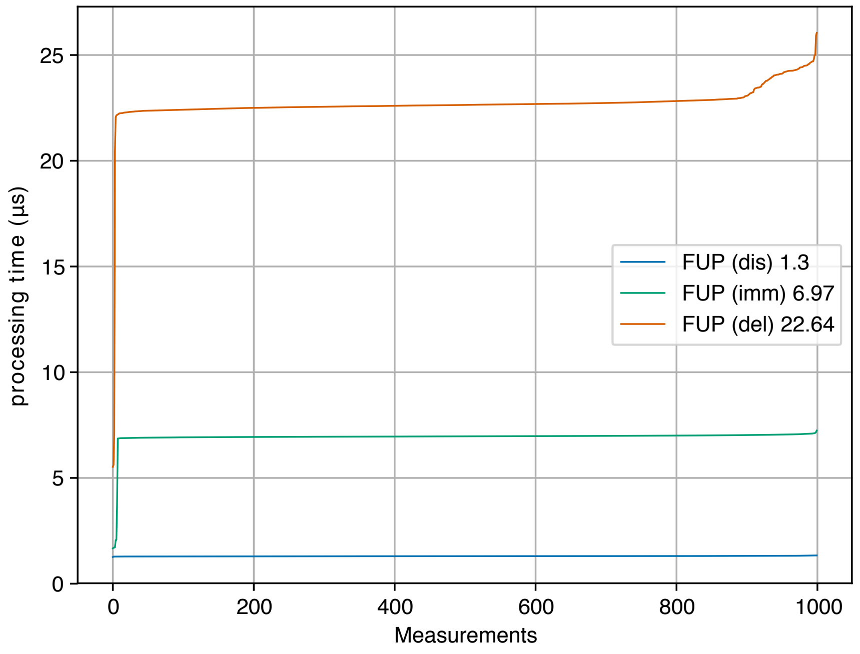


(b) CDF

Figure 4.11: send *Sync* measurements across security approaches (master)

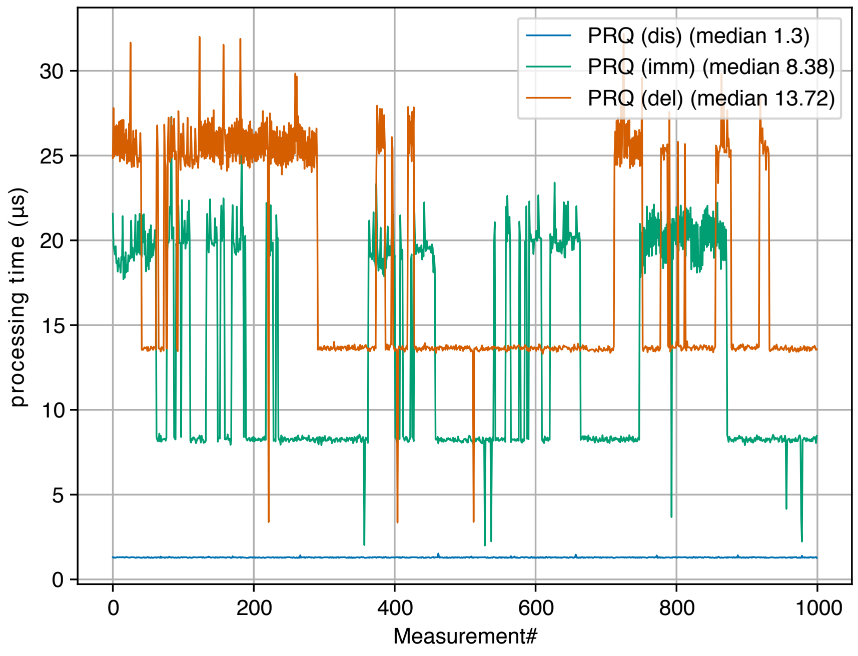


(a) time

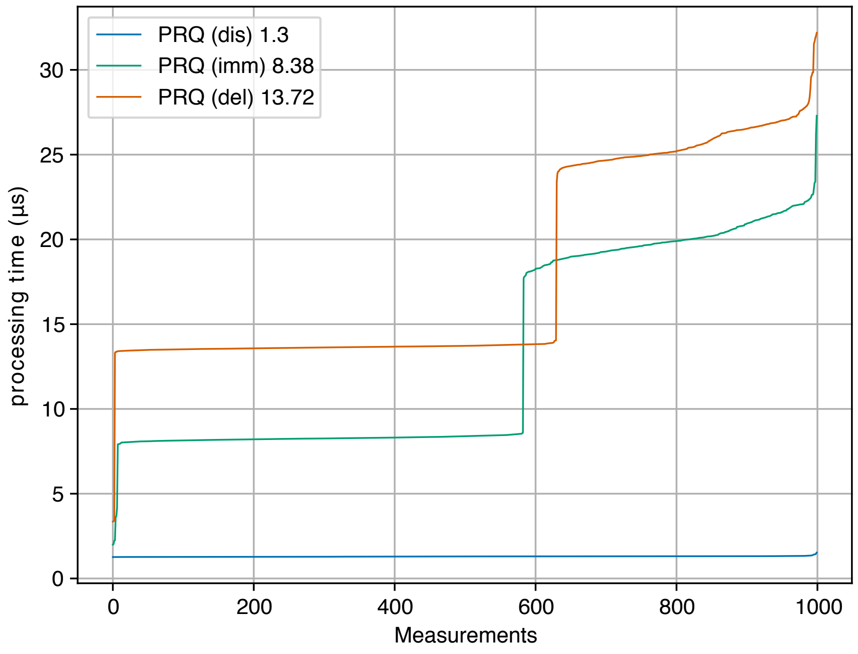


(b) CDF

Figure 4.12: send *Follow_Up* measurements across security approaches (master)

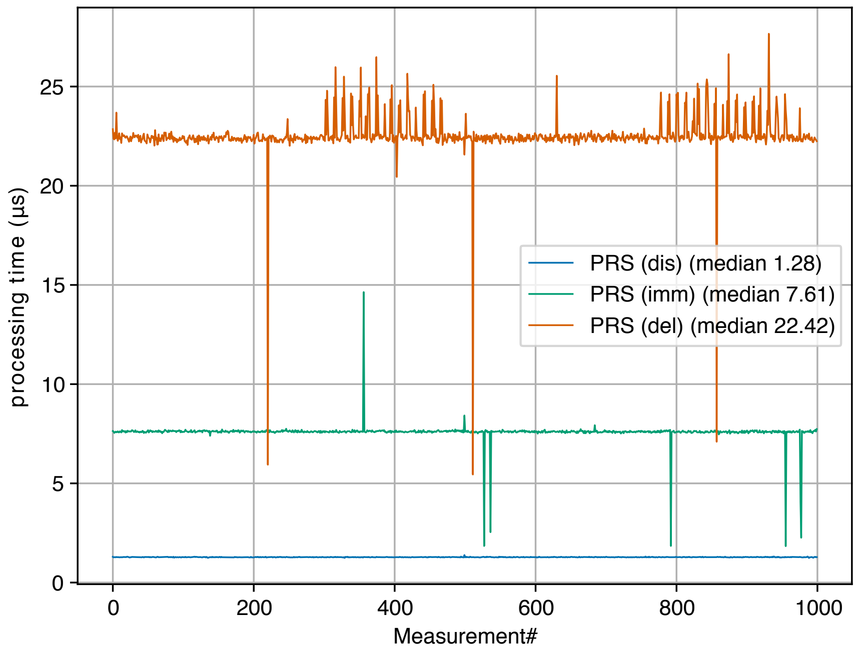


(a) time

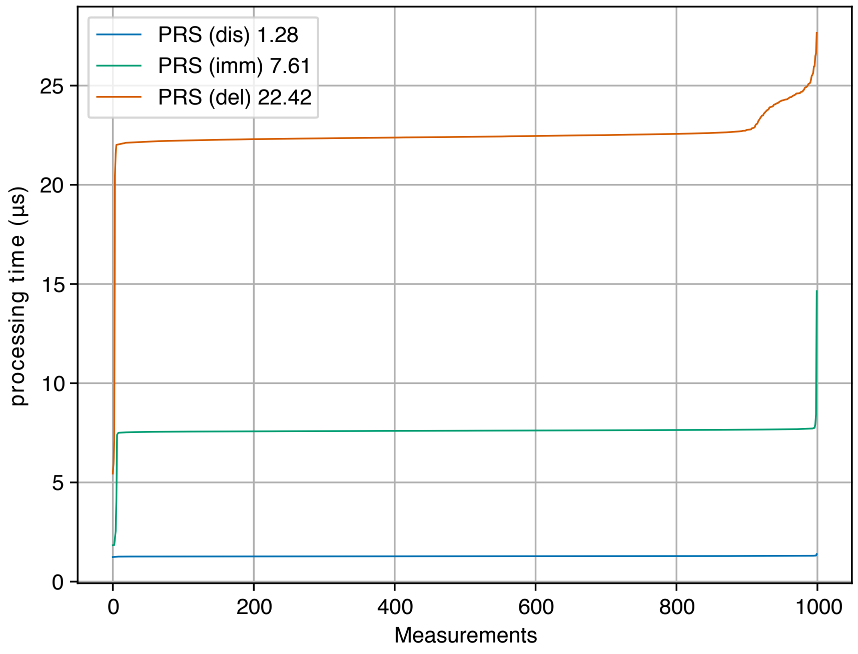


(b) CDF

Figure 4.13: send *Pdelay_Request* measurements across security approaches (master)

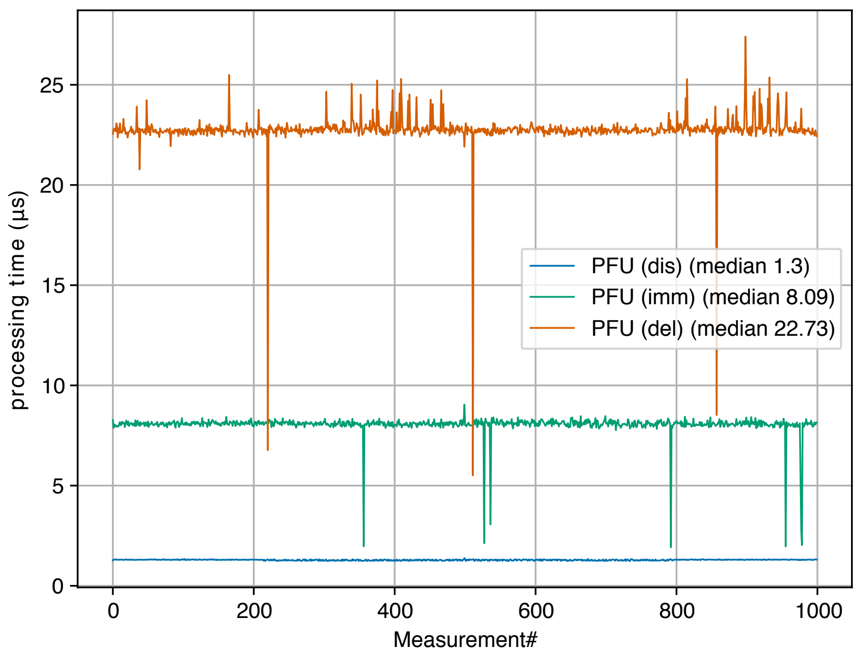


(a) time

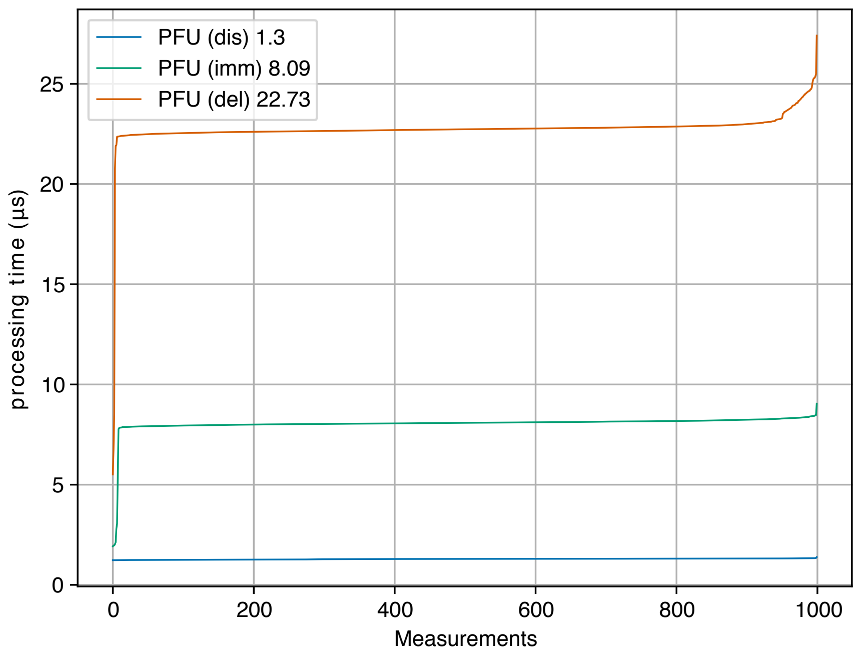


(b) CDF

Figure 4.14: send *Pdelay_Response* measurements across security approaches (master)

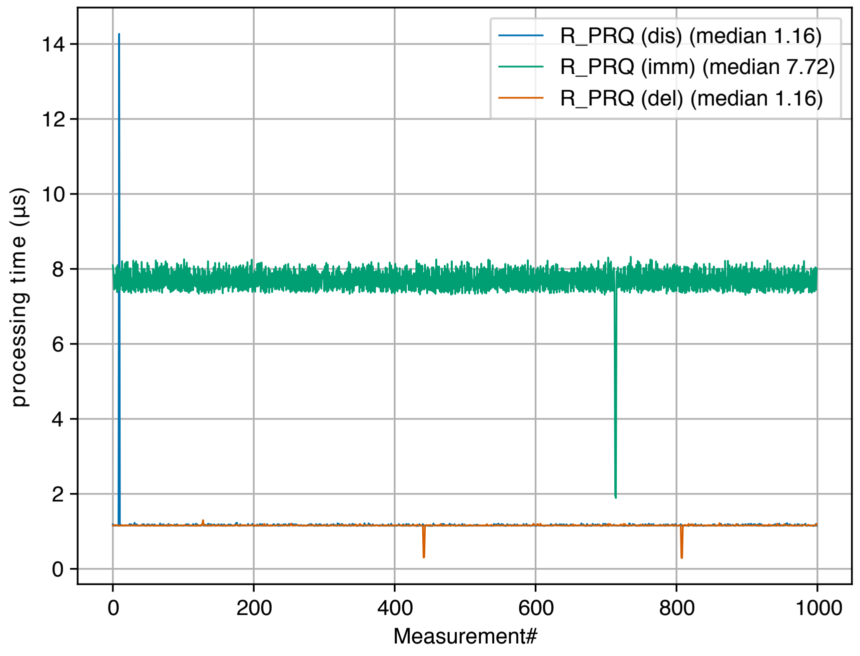


(a) time

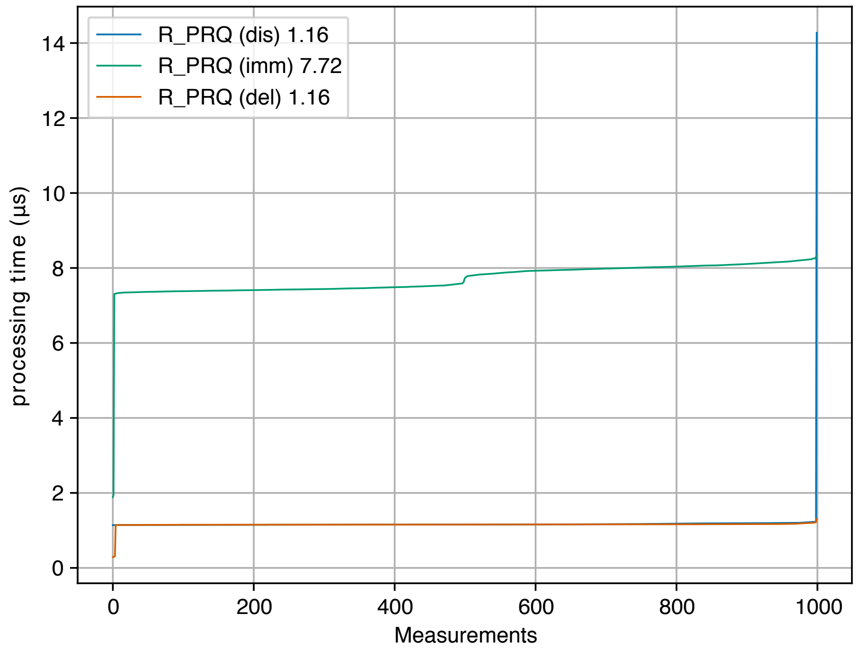


(b) CDF

Figure 4.15: send *Pdelay_Response_Follow_Up* measurements across security approaches (master)

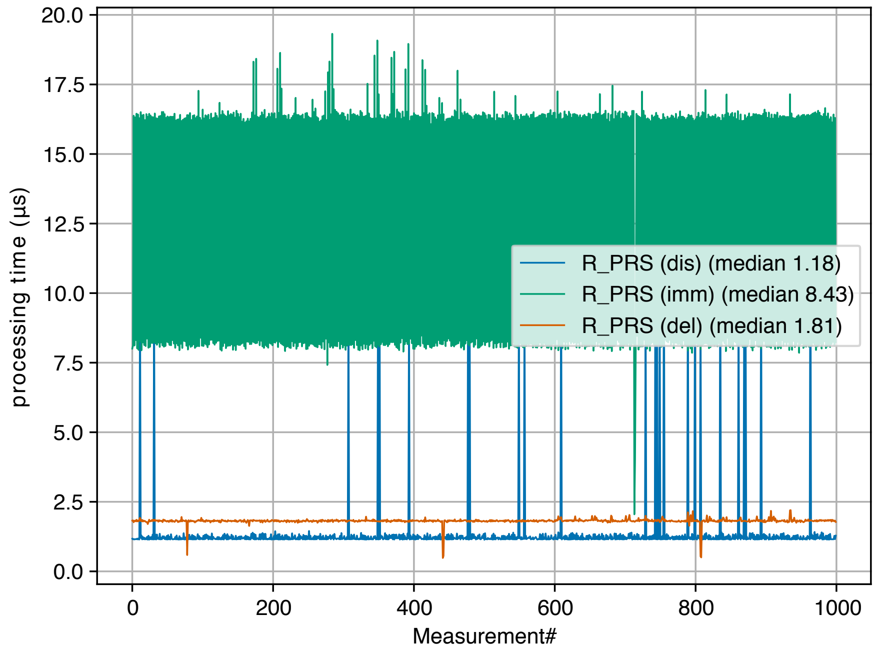


(a) time

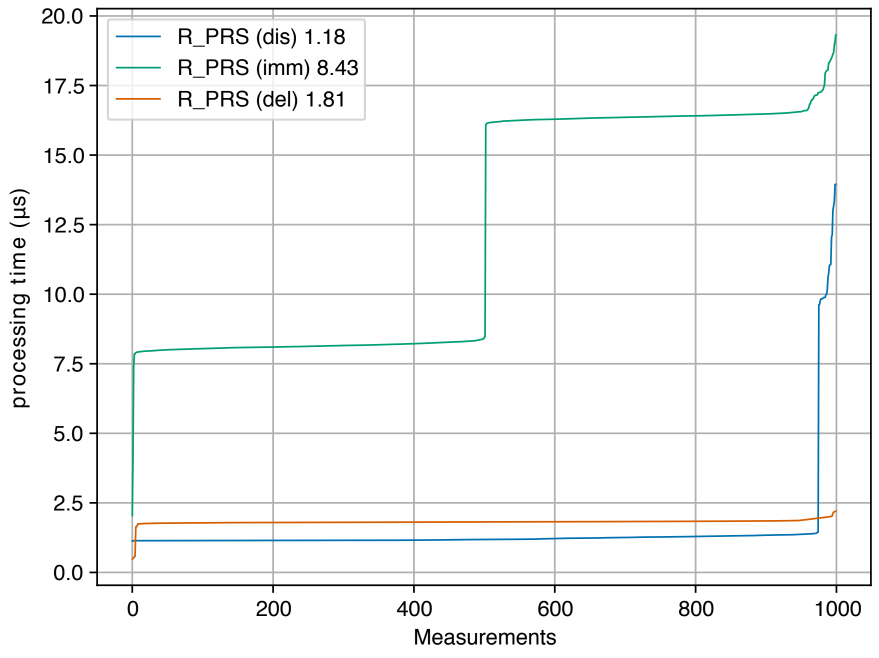


(b) CDF

Figure 4.16: receive *Pdelay_Request* measurements across security approaches (master)

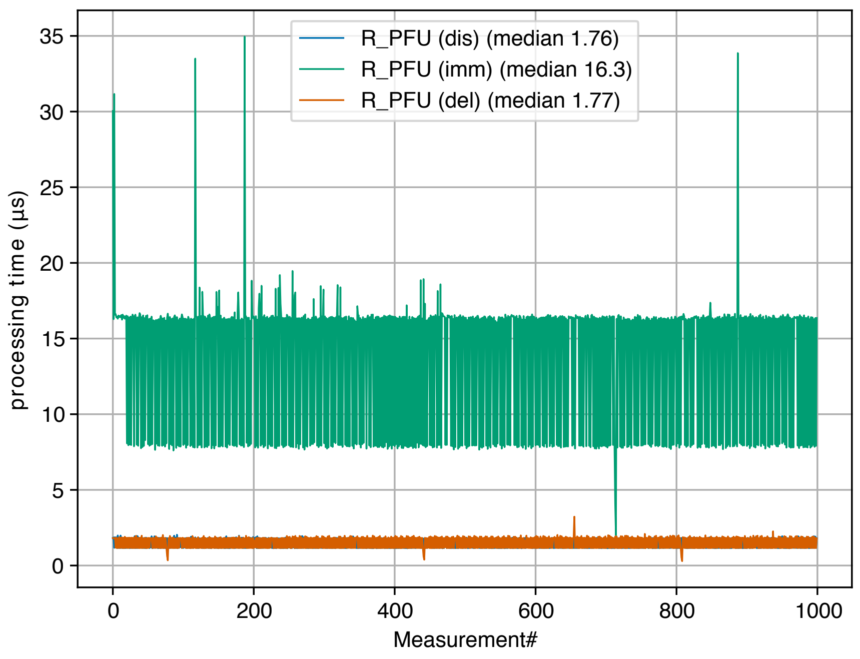


(a) time

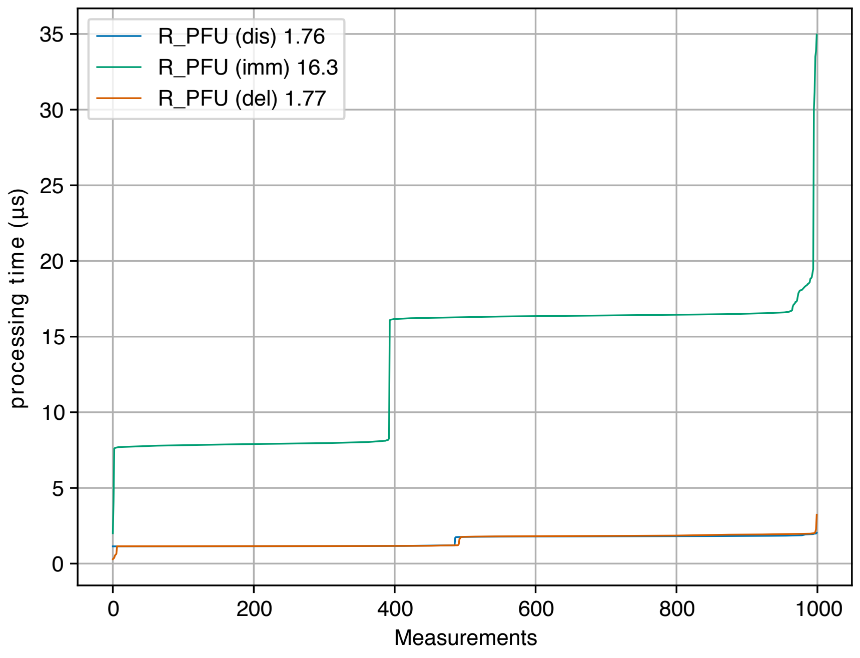


(b) CDF

Figure 4.17: receive *Pdelay_Response* measurements across security approaches (master)

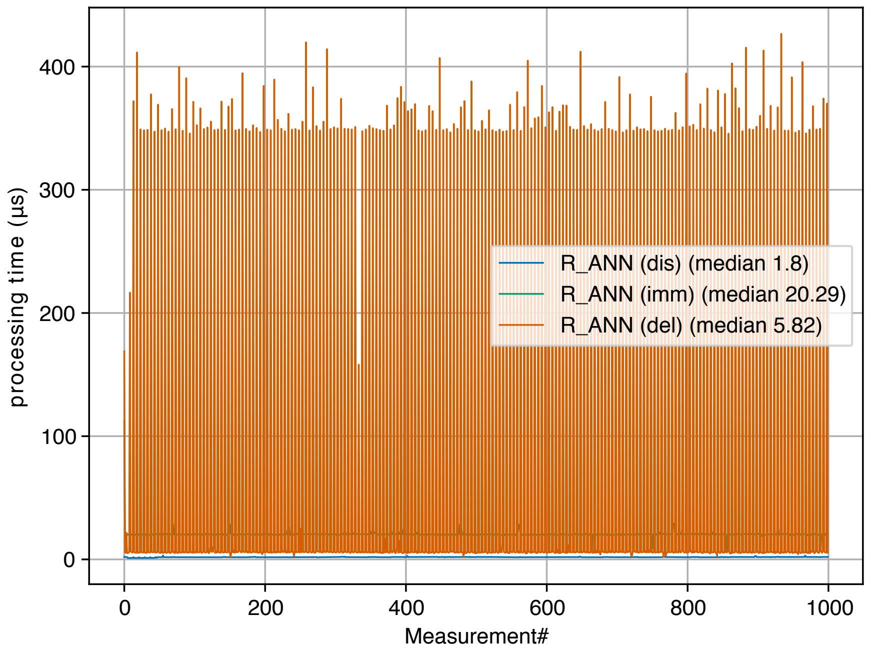


(a) time

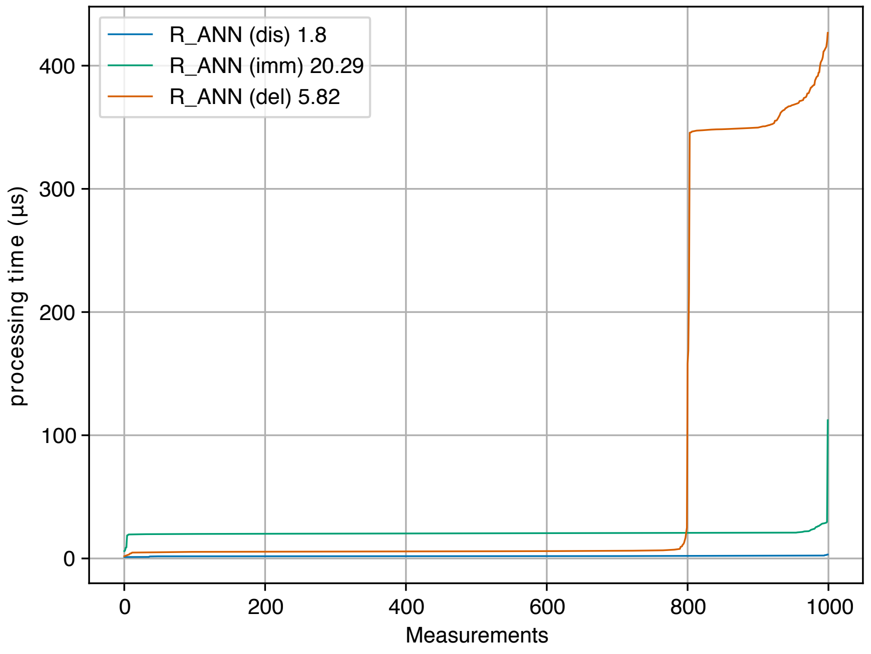


(b) CDF

Figure 4.18: receive *Pdelay_Response_Follow_Up* measurements across security approaches (master)

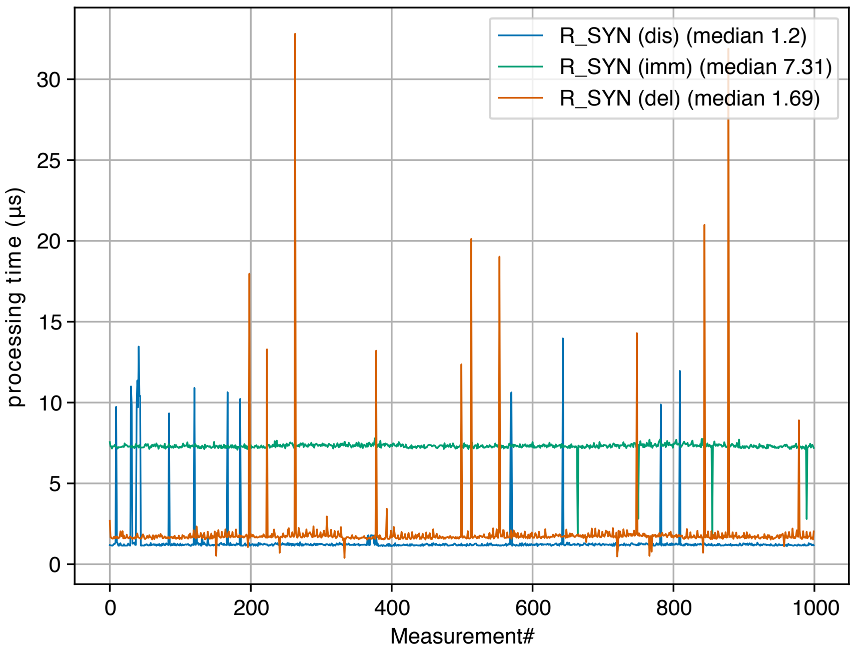


(a) time

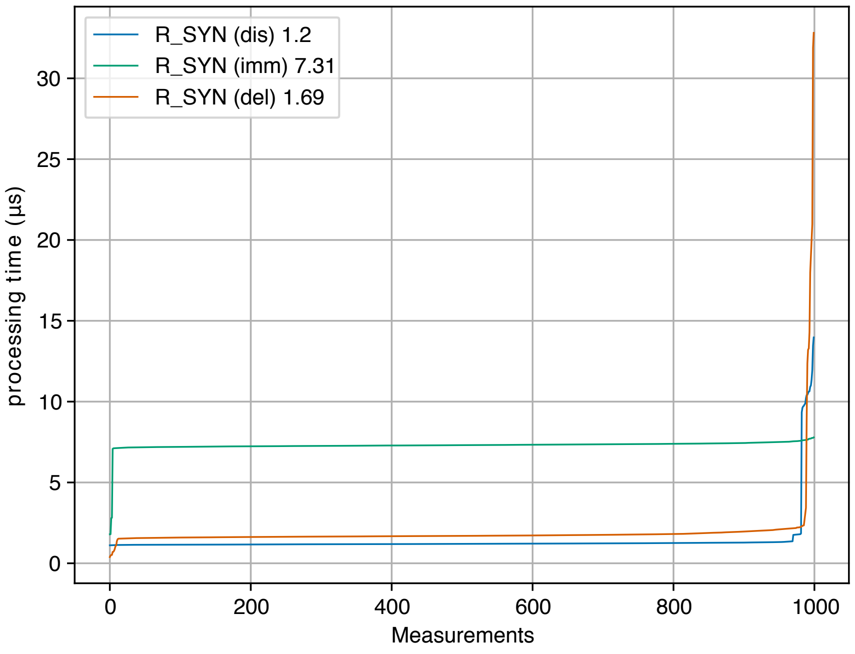


(b) CDF

Figure 4.19: receive *Announce* measurements across security approaches (slave)

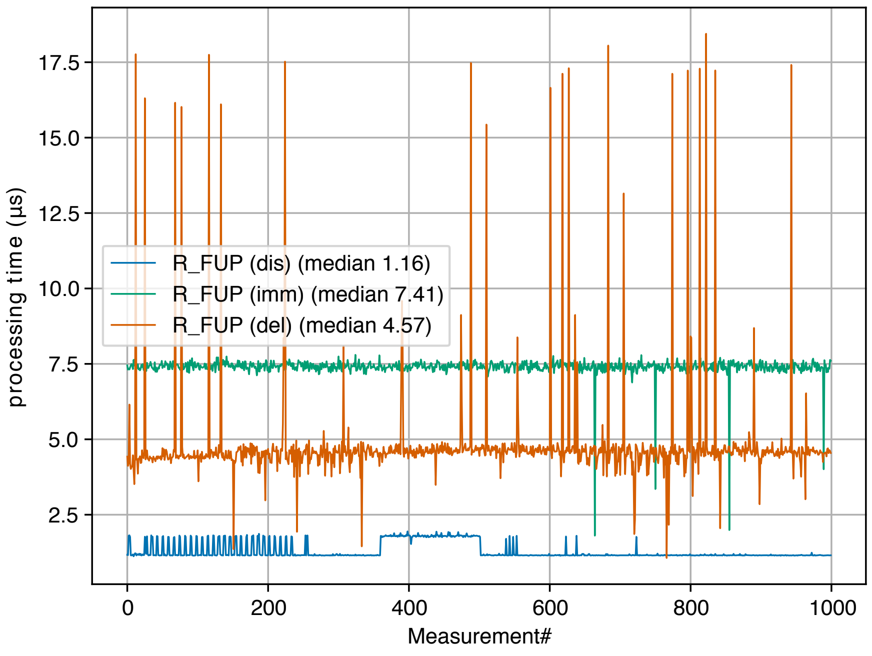


(a) time

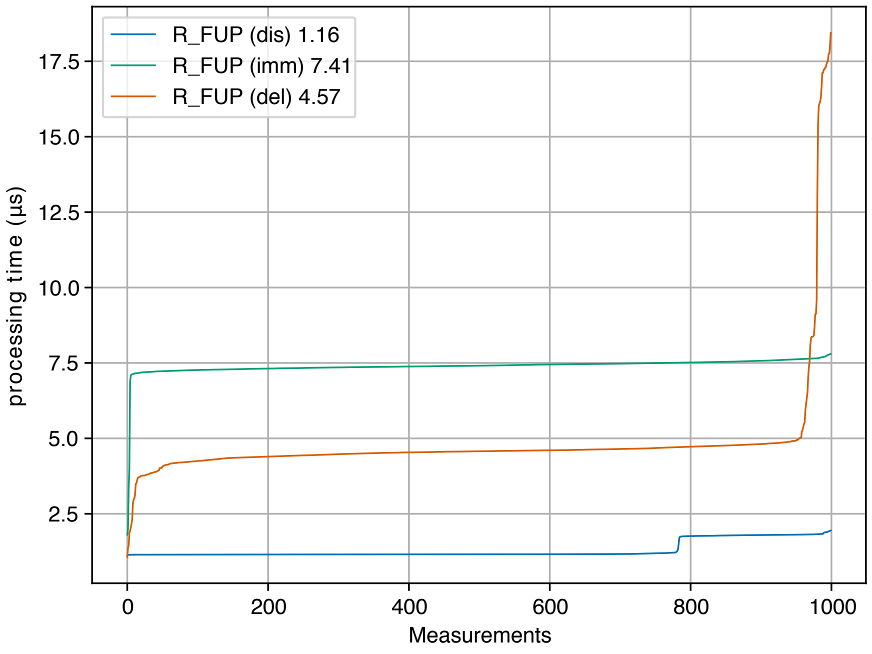


(b) CDF

Figure 4.20: receive *Sync* measurements across security approaches (slave)

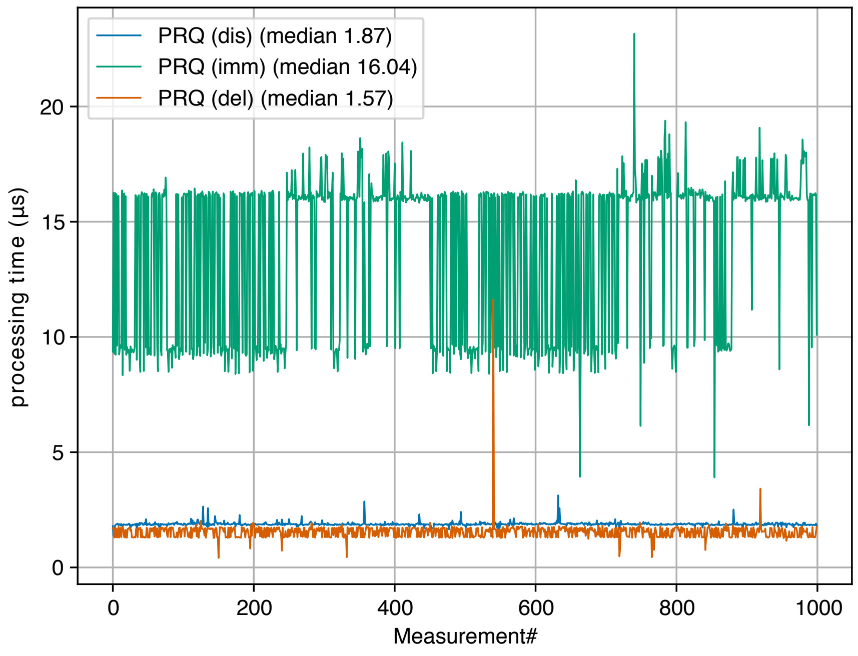


(a) time

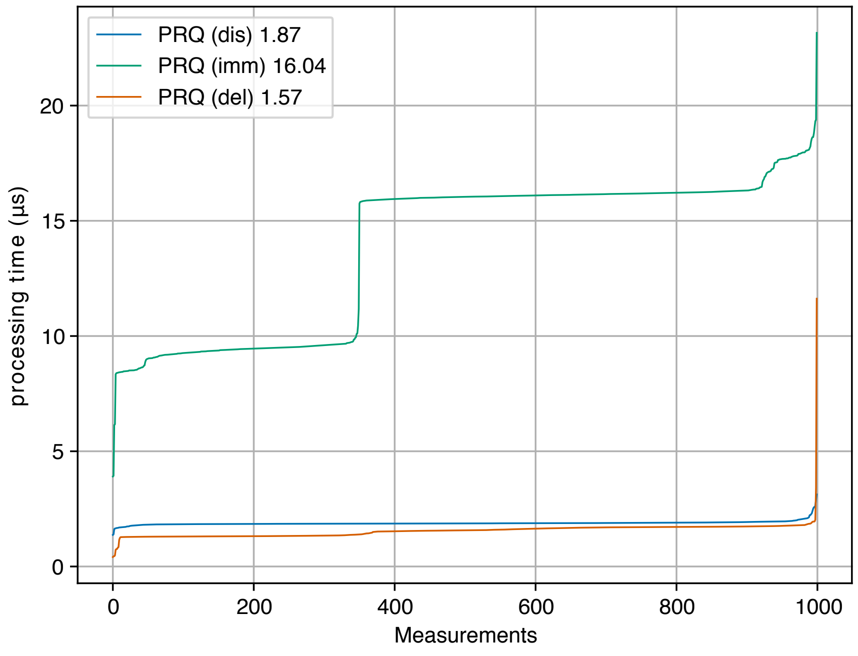


(b) CDF

Figure 4.21: receive *Follow_Up* measurements across security approaches (slave)

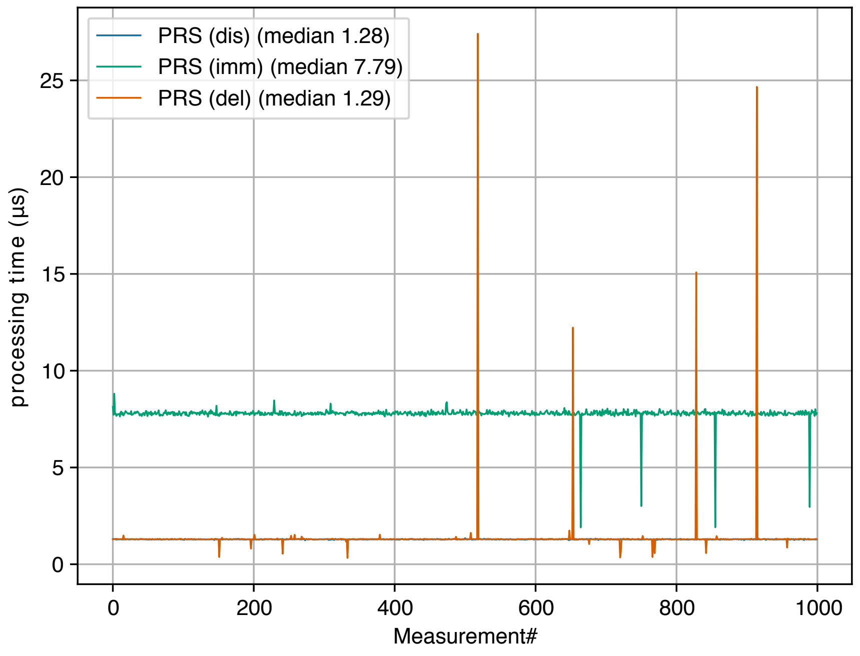


(a) time

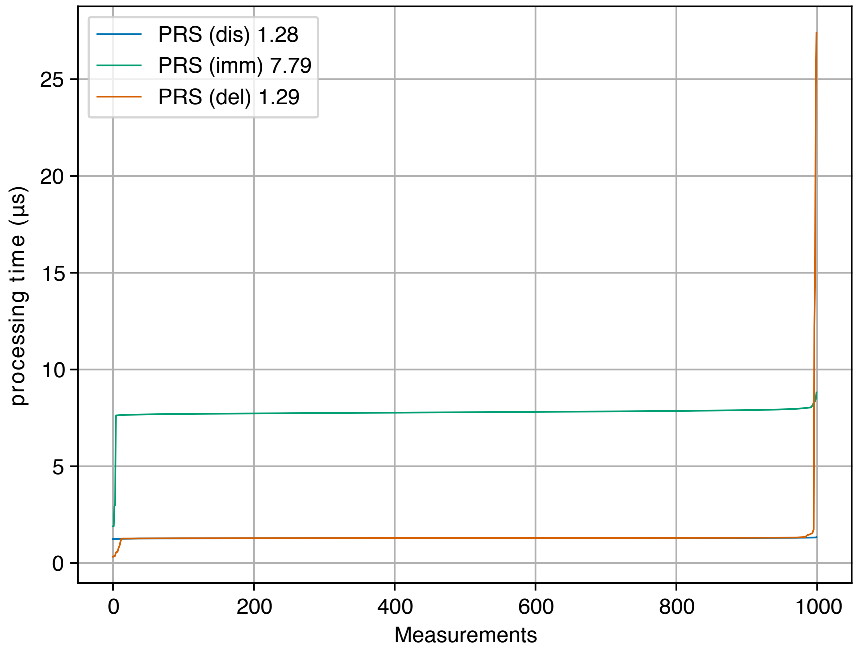


(b) CDF

Figure 4.22: send *Pdelay_Request* measurements across security approaches (slave)

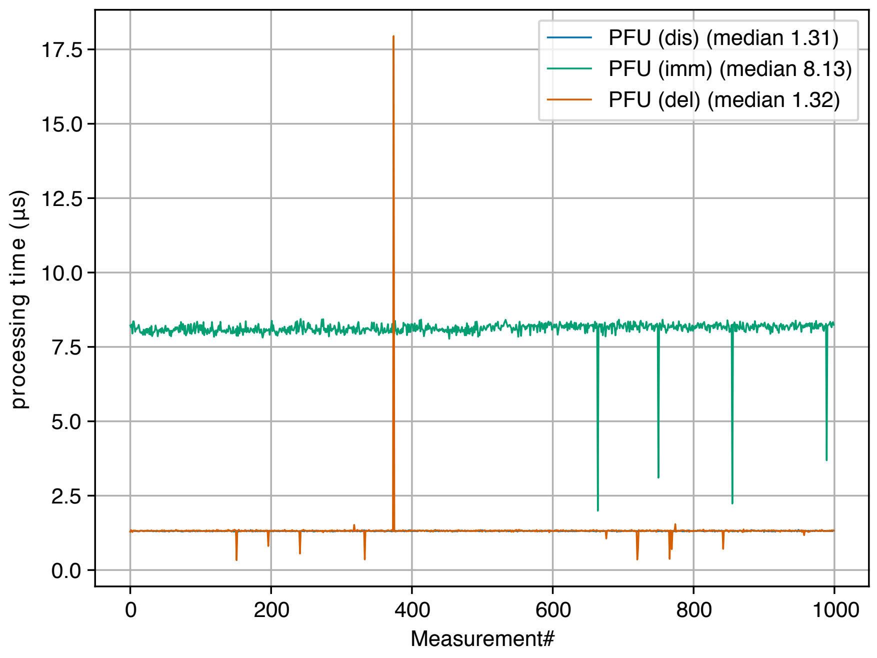


(a) time

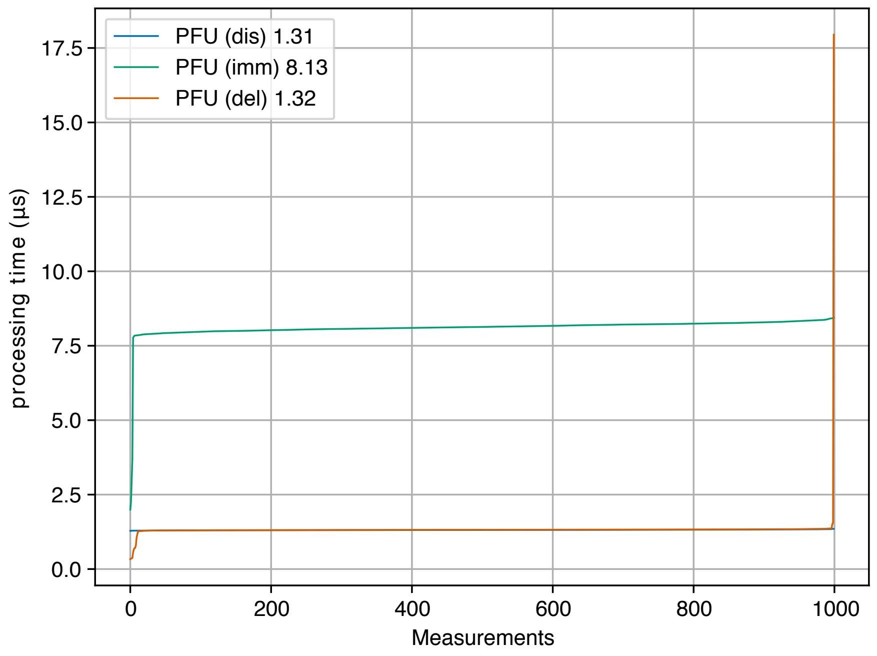


(b) CDF

Figure 4.23: send *Pdelay_Response* measurements across security approaches (slave)

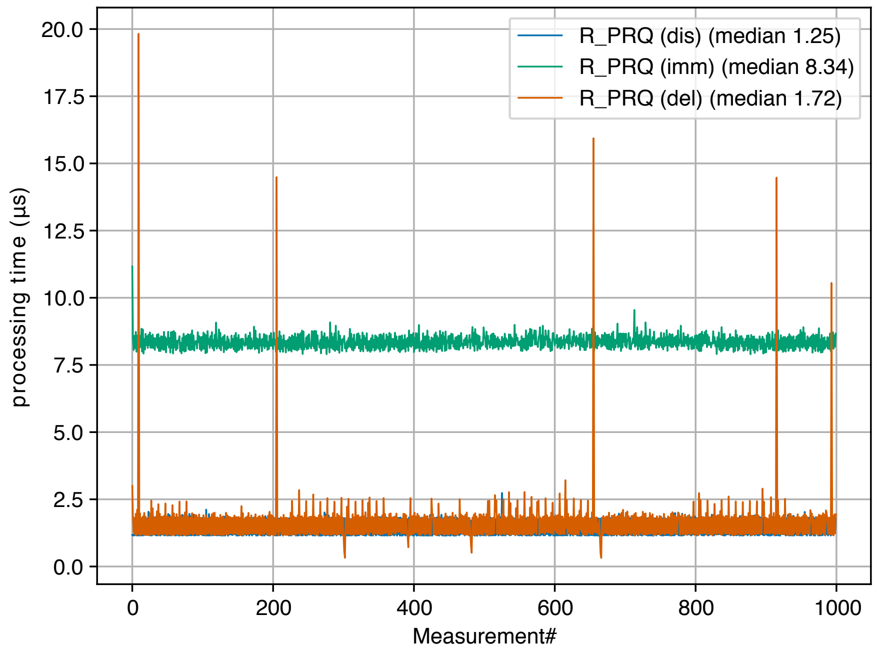


(a) time

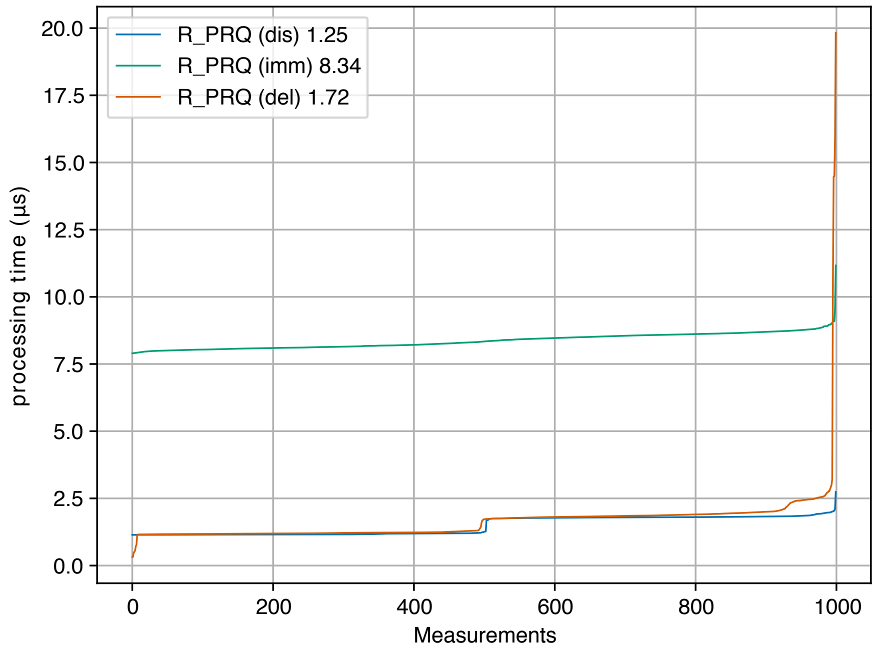


(b) CDF

Figure 4.24: send *Pdelay_Response_Follow_Up* measurements across security approaches (slave)

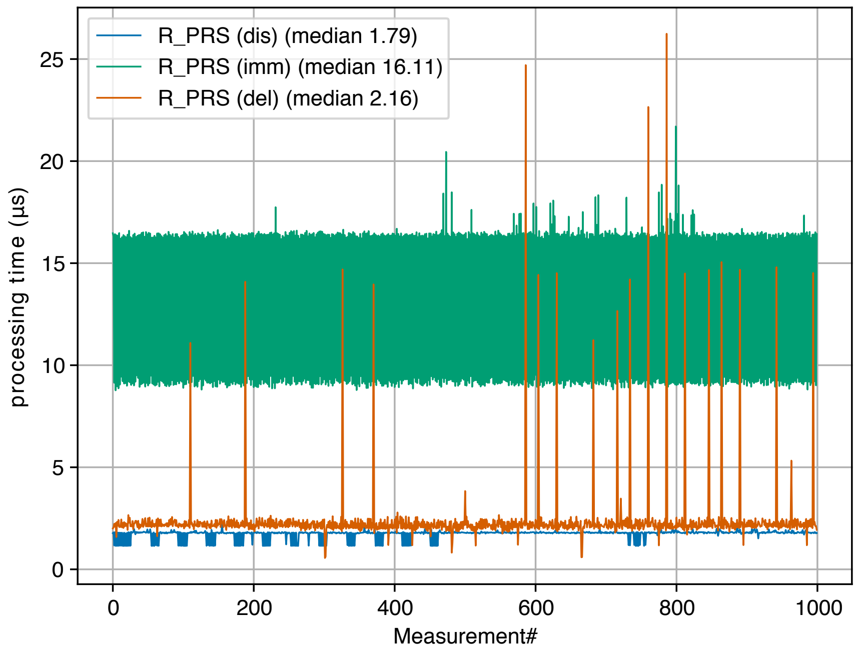


(a) time

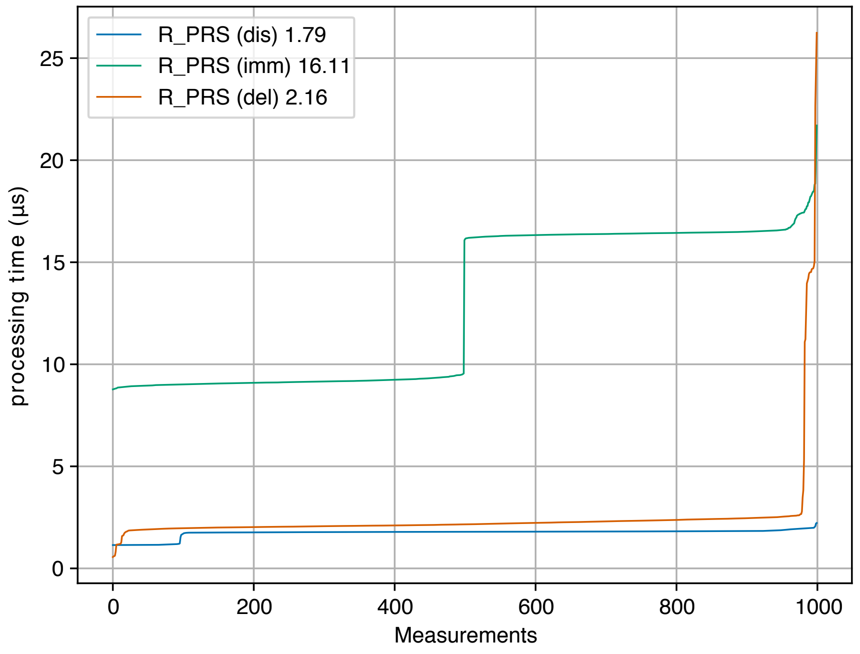


(b) CDF

Figure 4.25: receive *Pdelay_Request* measurements across security approaches (slave)

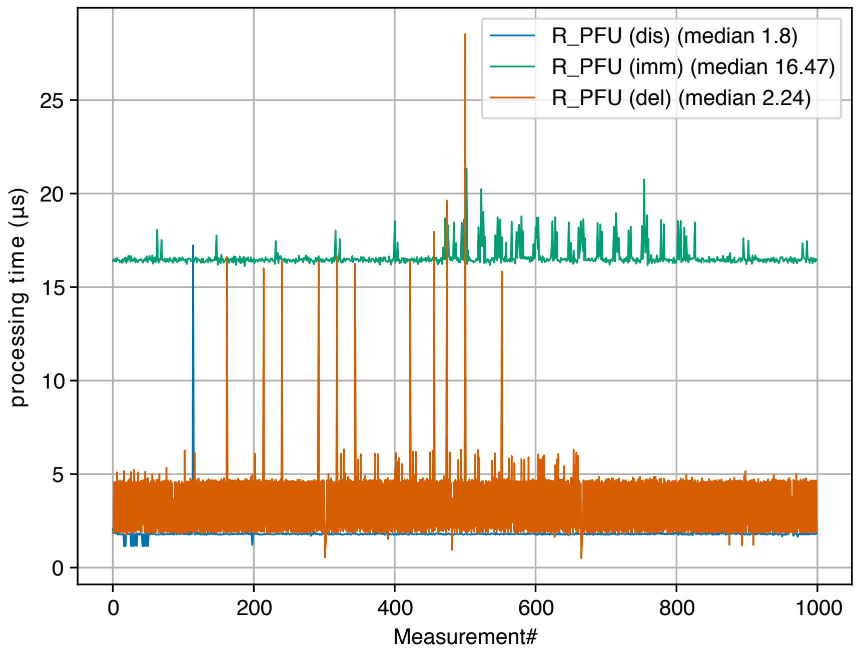


(a) time

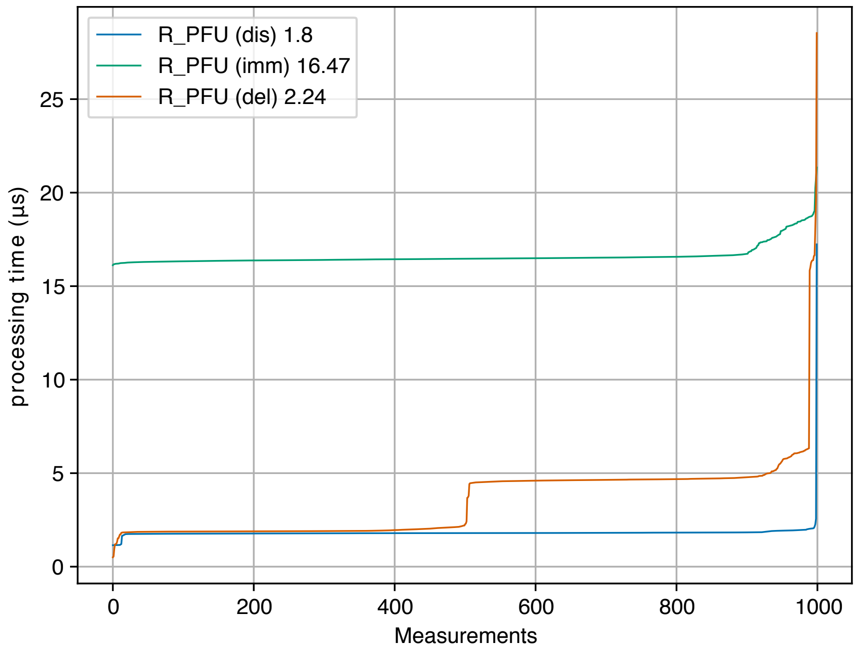


(b) CDF

Figure 4.26: receive *Pdelay_Response* measurements across security approaches (slave)

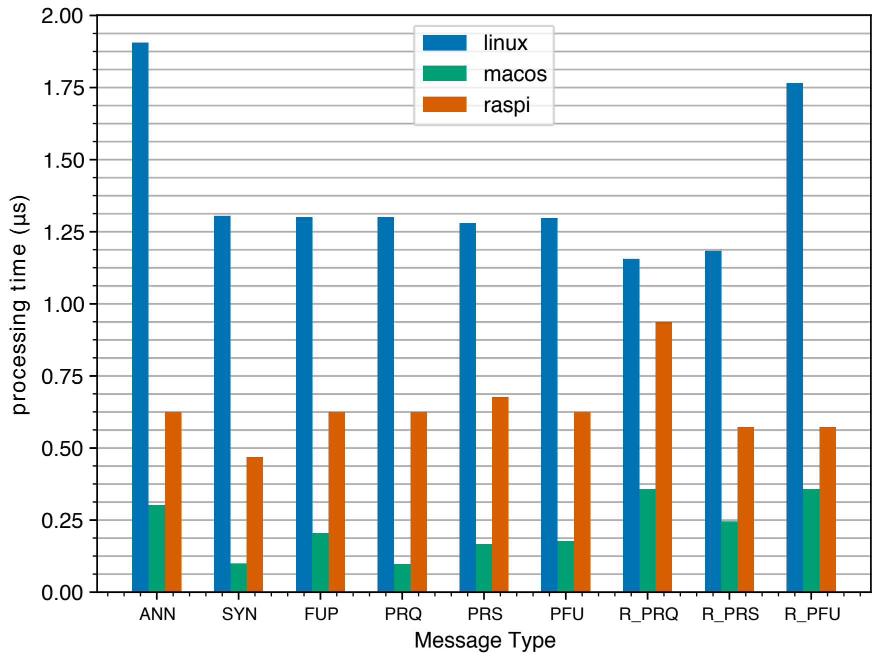


(a) time

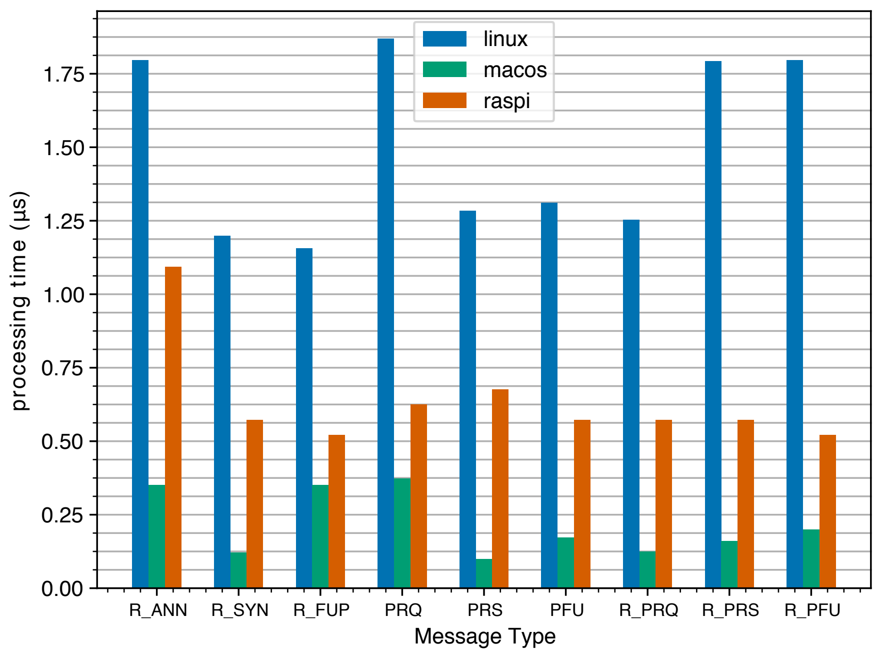


(b) CDF

Figure 4.27: receive *Pdelay_Response_Follow_Up* measurements across security approaches (slave)

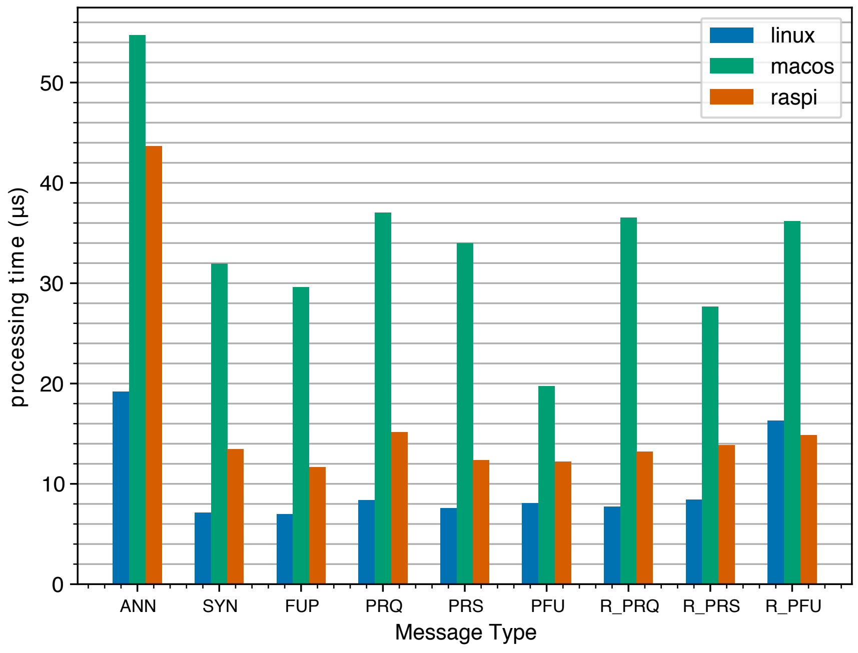


(a) master

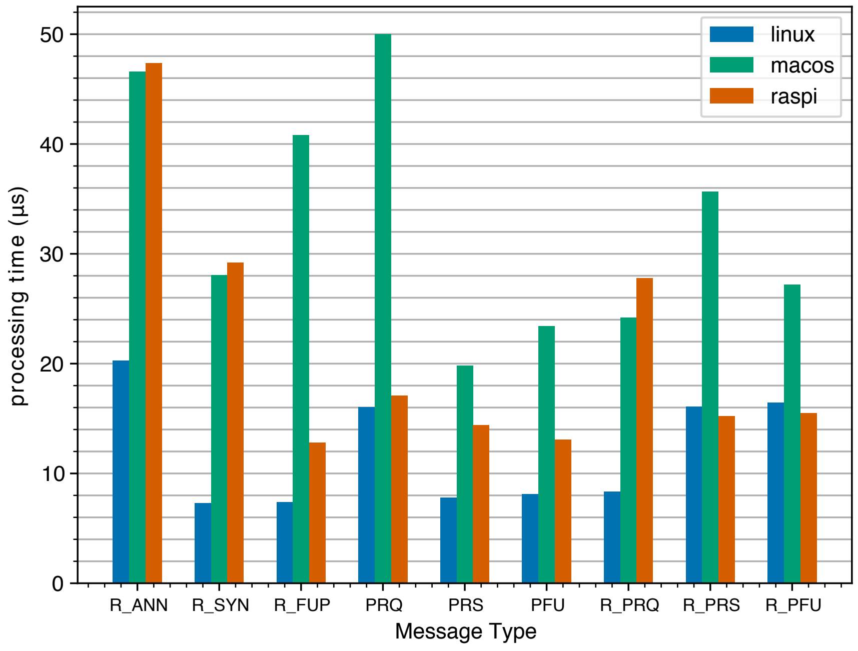


(b) slave

Figure 4.28: medians across platform (no security)

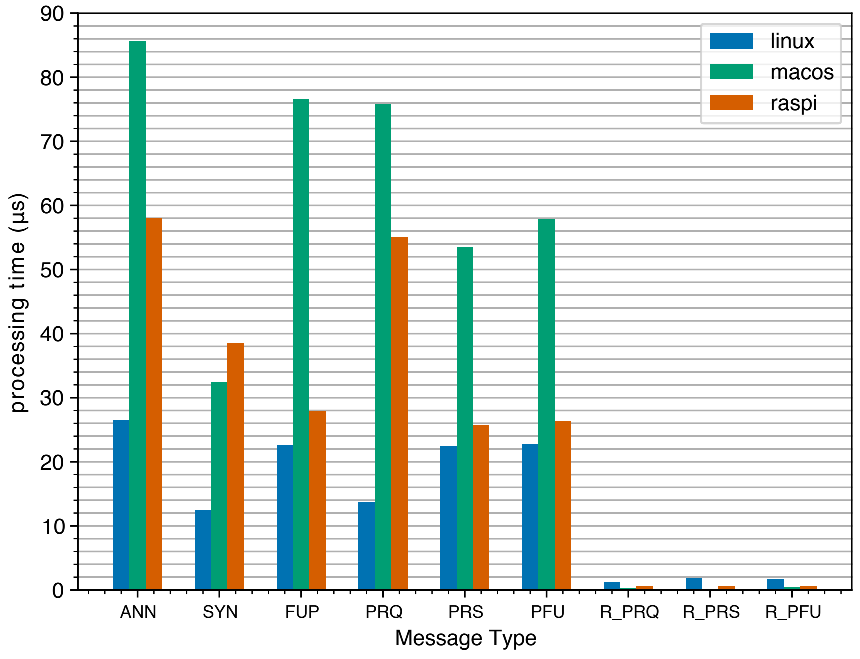


(a) master

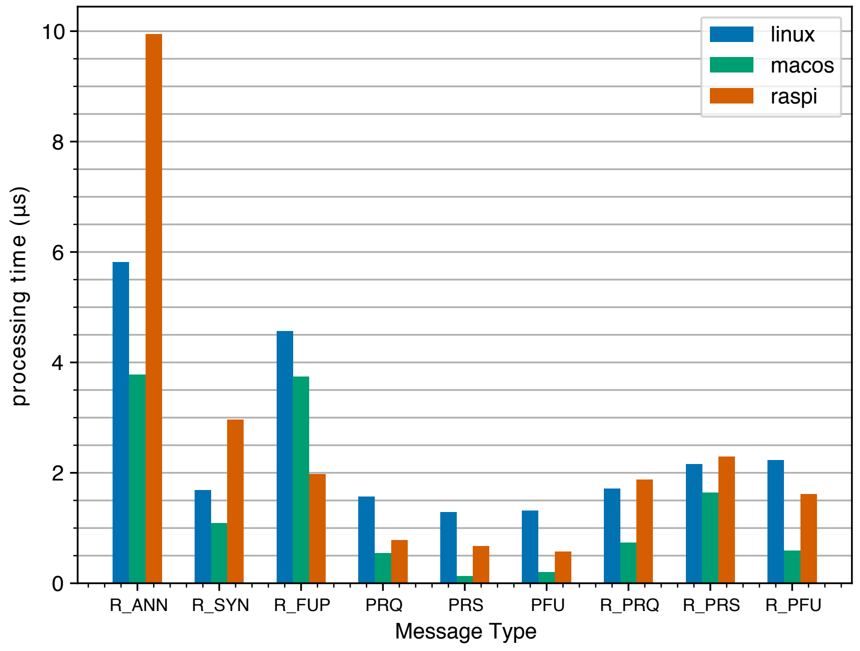


(b) slave

Figure 4.29: medians across platform (immediate)



(a) master



(b) slave

Figure 4.30: medians across platform (delayed)

CHAPTER 5
CONCLUSION

As the IEEE 1588 Precision Time Protocol usage continues to grow, so does the need for a security mechanism that can provide authenticity and integrity. The IEEE 1588 standard is currently undergoing work for a third revision, which will include a much-needed update to the current security mechanism specified in the informative Annex K. The purpose of this thesis was to establish the feasibility of the newly proposed security mechanism of the draft standard [2], specifically, its integrated security mechanism, in the context of usage in the power grid, as specified by the IEEE C37.238 Power Profile [7].

5.1 Review

In order to establish the feasibility of the proposed security mechanism for usage in the power grid, we implemented it on top of an existing PTP implementation, PTPd, configured according to the power profile. The draft standard outlines two verification approaches as part of the integrated security mechanism: immediate verification and delayed verification. Both of these approaches are closely linked to suggested key management schemes [19] [16], which are considered out of scope of the draft standard. Through implementing both of these verification approaches in our work and running functionality and performance experiments, we are better equipped to asses the pros and cons of each approach, and to asses the readiness of the security mechanism as a whole to be adopted for use in the power grid.

5.2 Findings

In this work we have achieved several important results with regards to implementation: we support the immediate verification security approach using manual key management at startup; we support a variable length AUTHENTICATION TLV; we support two ICV-calculation algorithms, HMAC-SHA256 and GMAC, with a design that is easily extensible to allow for other algorithm types to be added in the future; we emulated interaction with an SPD and SAD, with the aim of facilitating future integration with automated key management; and we support the delayed verification security approach, emulating automated key management for one set of security parameters corresponding to one manually configured time period. The delayed verification implementation includes functionality for generating a one-way key chain; deriving ICV-keys from key chain keys; verifying the safety of received packets; verifying authenticity of disclosed keys, including cases where one or more previously disclosed keys were missed; buffering received messages for later verification; and ICV verification of buffered messages, for potentially multiple unverified buffers in the case of previously missed disclosed keys.

In the absence of an automated key management scheme, the immediate verification approach is the only feasible option, as the necessary security parameters could be configured manually at start up. Since all nodes share the same set of security parameters, a dedicated master is not needed. This is a benefit in terms of allowing for dynamic reconfiguration of the PTP hierarchy (in case of failures, for instance); however, this also weakens the security of the network as a whole, as if any node gets compromised, it would have access to the secret key and could potentially act as a rogue master and send bogus synchronization information to the rest of the network.

In the delayed verification approach, a dedicated master is needed, as only one node must have access to the key chain to be used in securing messages. This strengthens the security of the network as a whole, as any slave node that may become compromised would not have enough information to be able to act as a rogue master. The delayed verification approach requires, however, two aspects related to key management that are crucial for proper functioning and security: first, a bootstrapping phase in which a trust anchor key is securely distributed to all participating nodes; our implementation only emulates this process. Second, there must be an automated means of generating a new key chain and properly handling the transition to a new set of time intervals. These are not limitations of the security approach itself; rather, they illustrate the strong dependence of the delayed verification approach upon an automated key management scheme. Another issue that may make the delayed verification approach more difficult to adopt is the authentication mode; when a slave receives a safe packet, it may either wait to use it until it is authenticated (authenticated mode), or it may use the message anyway, at the risk of having to undo its effects later, should the message fail verification (unauthenticated mode). The details of how to implement these behaviors are left up to the PTP implementation.

In terms of performance, the added time needed to do security processing for both approaches was within 30 μs. In the immediate approach, there is a more uniform cost across all messages; in the delayed approach, emulating the unauthenticated mode, there is minimal cost for most messages as ICV calculation is not done immediately, and a significant cost an order of magnitude higher on the messages that contain a new disclosed key, as they trigger bulk ICV verification on previously buffered messages. We have not found evidence that these performance costs hinder clock synchronization.

5.3 Future Work

There are several future work items as a result of this work. First, a closer study needs to be done on the impact of the security processing on clock synchronization quality. Second, automated key management schemes specifically designed for use with immediate verification and delayed verification should be implemented and integrated with the security mechanism. Third, the issue of authentication modes in delayed processing needs to be addressed;

is it possible for a PTP instance to implement each of the modes (authenticated and unauthenticated), and if so, how should this be done? Finally, the draft standard mentions the possibility of using both verification approaches together in the same network; this would require the integration of multiple key management schemes, and the support of processing multiple AUTHENTICATION TLVs.

LIST OF REFERENCES

[1] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, July 2008.

[2] "Draft Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," 2017, Draft Standard.

[3] "PTPd source code documentation." [Online]. Available: http://ptpd.sourceforge.net/doc.html

[4] A. Stevenson, Ed., *Oxford Dictionary of English.* Oxford University Press, 2015.

[5] D. L. Mills, "Network Time Protocol (NTP)," Internet Requests for Comments, RFC Editor, RFC 958, September 1985.

[6] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," *IEEE Std 1588-2002*, 2002.

[7] "IEEE Standard Profile for use of IEEE 1588 Precision Time Protocol in Power System Applications," *IEEE Std C37.238-2017 (Revision of IEEE Std C37.238-2011)*, June 2017.

[8] K. O'Donoghue, "Emerging solutions for time protocol security," in *2016 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, Sept 2016, pp. 1–6.

[9] "Precision Time Protocol Telecom Profile for Frequency Synchronization," *ITU-T G.8265.1*, 2010.

[10] "Precision Time Protocol Telecom Profile for Phase/Time Synchronization," *ITU-T G.8275.1*, 2014.

[11] "IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks," *IEEE Std 802.1AS-2011*, March 2011.

[12] T. Mizrahi, "Security requirements of time protocols in packet switched networks," Internet Requests for Comments, RFC Editor, RFC 7384, October 2014.

[13] A. Treytl and B. Hirschler, “Security flaws and workarounds for IEEE 1588 (transparent) clocks,” in *2009 International Symposium on Precision Clock Synchronization*, Oct 2009, pp. 1–6.

[14] C. Önal and H. Kirrmann, “Security improvements for IEEE 1588 Annex K: Implementation and comparison of authentication codes,” in *2012 IEEE International Symposium on Precision Clock Synchronization*, Sept 2012, pp. 1–6.

[15] N. Moreira, J. Lázaro, J. Jimenez, M. Idirin, and A. Astarloa, “Security mechanisms to protect IEEE 1588 synchronization: State of the art and trends,” in *2015 IEEE International Symposium on Precision Clock Synchronization*, Oct 2015, pp. 115–120.

[16] A. Perrig, D. Song, R. Canetti, J. Tygar, and B. Briscoe, “Timed efficient stream loss-tolerant authentication (TESLA): Multicast source authentication transform introduction,” RFC Editor, RFC 4082, 2005.

[17] [Online]. Available: https://github.com/ptpd/ptpd.

[18] M. Dworkin, *Recommendation for block cipher modes of operation. [electronic resource] : Galois/Counter mode (GCM) and GMAC.*, ser. NIST special publication Computer security: 800-38D. Gaithersburg, MD : U.S. Dept. of Commerce, National Institute of Standards and Technology, [2007], 2007.

[19] B. Weis, S. Rowles, and T. Hardjono, “The Group Domain of Interpretation,” RFC Editor, RFC 6407, 2011.

APPENDIX A
ACRONYMS

- AAD: Additional Authenticated Data
- AES: Advanced Encryption Standard
- BC: Boundary Clock
- GCM: Galois/Counter Mode
- GDOI: Group Domain of Interpretation
- GMAC: Galois Message Authentication Code
- HMAC: Hash-based Message Authentication Code
- ICV: Integrity Check Value
- IEEE: Institute of Electrical and Electronics Engineers
- IETF: Internet Engineering Task Force
- IV: Initialization Vector
- MAC: Message Authentication Code
- MITM: Man-In-The-Middle
- NIST: National Institute of Standards and Technology
- NTP: Network Time Protocol
- OC: Ordinary Clock
- PRF: Pseudorandom Function
- PTP: Precision Time Protocol
- PTPd: Precision Time Protocol daemon
- RFC: Request For Comments

- SA: Security Association
- SAD: Security Association Database
- SHA: Secure Hash Algorithm
- SP: Security Policy
- SPD: Security Policy Database
- SPP: Security Parameter Pointer
- TC: Transparent Clock
- TESLA: Timed Efficient Stream Loss-Tolerant Authentication
- TLV: Type Length Value

APPENDIX B
FUNCTIONALITY TEST CASES

The full list of functionality test cases outlined in Section 4.2 is included below. The devices used were a modified PTPd instance running on a Raspberry Pi, a modified PTPd instance running on a MacBook Pro, and a PTP stack running on Microchip's EVB KSZ9477 acting as a *Transparent Clock*. These devices are referred to hereafter as *Pi*, *Mac*, and *Microchip*, respectively.

For each test case, we first set up the conditions, then ran the PTP instances, and finally verified expected behavior. These three steps are labeled in each test case as *Setup*, *Run*, and *Verify*, respectively. Setting up the conditions entailed setting up a topology (physically connecting the involved devices), and setting the appropriate parameters in the configuration file. All configuration files were set up to adhere to the power profile as a base configuration, and security parameters were set according to the requirements of each test case. In the list below, the *Run* step for each test case names the configuration files used for the test; these are in a testing directory in the project source code. The naming scheme for the configuration files that correspond to each test case is `<security approach>_<test case number>_<state>.conf`. As an example, `imm_1.2.4_m.conf` is the configuration file for the fourth test case under the attack scenarios for a master using the immediate verification scheme, in the direct connection topology.

B.1 Regression tests

1. Direct topology: master (Mac) directly connected to slave (Pi)

   1.1. Functionality:

   1.1.1. Unsecured network:

   A. Setup: Both nodes built without `./configure --enable-security`

   B. Run: master started with `reg_m.conf`, slave started with `reg_s.conf`

   C. Verify: in the slave's status log, mean path delay should reach a stable value, and offset from master should decrease to a stable value; sent Announce, Sync, and Follow_Up message sent counters should increase for master, received variants should increase for slave; sent and received Pdelay messages counters should increase for both master and slave; in Wireshark, all messages should be observed with no AUTHENTICATION TLV attached

2. Multi-hop topology: master (Pi) connected to TC (Microchip port 2), and slave (Mac) connected to TC (Microchip port 1)

   2.1. Functionality:

   2.1.1. Unsecured network:

   A. Setup: Both nodes built without `./configure --enable-security`, TC using release version 1.1.2 (August 2017)

   B. Run: master started with `reg_m.conf`, slave started with `reg_s.conf`, TC started with `killall ptp4l; killall phc2sys; cd /ptp/p2p/tc; ./linuxptp.sh`

   C. Verify: status log results same as for case `reg_1.1.1` above, but no Pdelay_Response_Followup messages should be received on either PTPd node, since TC is using one-step mode; in Wireshark should observe received Sync messages with non-zero correctionField, as they are adjusted for residence time per IEEE 1588-2008 11.5.2.1; Pdelay_Response messages from the TC should have non-zero correctionField per IEEE 1588-2008 11.4.3.b

B.2 Immediate Verification Tests

1. Direct topology: master (Mac) directly connected to slave (Pi)

   1.1. Functionality:

   1.1.1. Fully secured network

   A. Setup: security enabled on both nodes, all security options are the same (notably, the keys and integrity alg type)

   B. Run: master started with `imm_1.1.1_m.conf`, slave started with `imm_1.1.1_s.conf`

C. Verify: status log results should be same as `reg_1.1.1`; in Wireshark, an AUTHENTICATION TLV should be observed on all messages

1.1.2. Mixed network

A. Setup: security enabled on master, master accept insecure Pdelay messages; slave built without security

B. Run: master started with `imm_1.1.2_m.conf`, slave started with `imm_1.1.2_s.conf`

C. Verify: status log results should be same as `imm_1.1.1`, as the slave will ignore AUTHENTICATION TLV on all master messages and continue to calculate its offset accordingly; in Wireshark, incoming messages to the slave should show AUTHENTICATION TLV present, while outgoing messages from slave should not have AUTHENTICATION TLV attached

1.2. Attack scenarios:

1.2.1. Rogue master: master unsecure

A. Setup: master built without security; security enabled on slave

B. Run: master started with `imm_1.2.1_m.conf`, slave started with `imm_1.2.1_s.conf`

C. Verify: slave status log should show `authenticationTLVExpected` error counters increasing, and should show state as `PTP_LISTENING`, since slave should reject all messages; in Wireshark, Announce, Sync, Follow_Up and Pdelay_Request messages from master to slave should not have AUTHENTICATION TLV attached

1.2.2. Rogue master: different key

A. Setup: master first byte of key is `0xff` instead of `0xed`

B. Run: master started with `imm_1.2.2_m.conf`, slave started with `imm_1.2.2_s.conf`

C. Verify: slave status log should show `icvMismatch` error counters increasing, and should show state as `PTP_LISTENING`, since slave should reject all messages; in Wireshark, Announce, Sync, Follow_Up and Pdelay_Request messages from master to slave should have AUTHENTICATION TLV attached

1.2.3. Rogue master: different ICV algorithm

A. Setup: slave configured to use HMAC-SHA256; master configured to use GMAC

B. Run: master started with `imm_1.2.3_m.conf`, slave started with `imm_1.2.3_s.conf`

C. Verify: slave status log should show `lengthMismatchError` error counters increasing, since AUTHENTICATION TLV lengths differ based on algorithm type, and should show state as `PTP_LISTENING`, since slave should reject all messages; in Wireshark, Announce, Sync, Follow_Up and Pdelay_Request messages from master to slave should have AUTHENTICATION TLV attached

1.2.4. MITM: PTP payload altered

A. Setup: master compiled from test branch with test values `MITM=true`, `MITM_attack=PAYLOAD_ALTERED`, `targetMessageSeqId=15`

B. Run: master started with `imm_1.2.4_m.conf`, slave started with `imm_1.2.4_s.conf`

C. Verify: slave status log should show `icvMismatchError` counter at 1; event log should show INFO message "SEC: ICVs didn't match on SeqId 0x000f"; in Wireshark, Announce message with sequenceId 15 should have PTP payload set to all `0xff`

1.2.5. MITM: ICV altered

A. Setup: master compiled from test branch with test values `MITM=true`, `MITM_attack=ICV_ALTERED`, `targetMessageSeqId=15`

B. Run: master started with `imm_1.2.5_m.conf`, slave started with `imm_1.2.5_s.conf`

C. Verify: slave status log should show `icvMismatchError` counter at 1; event log should show INFO message "SEC: ICVs didn't match on SeqId 0x000f"; in Wireshark, Announce message with sequenceId 15 should have ICV field set to all `0xff`

2. Multi-hop topology: master (Pi) connected to TC (Microchip port 2), and slave (Mac) connected to TC (Microchip port 1)

2.1. Functionality:

2.1.1. Fully secured network: ignore *correctionField* ON

A. Setup: like `imm_1.1.1`, but `imm_ignore_correction` must be on, and both PTPd nodes must be set to accept unsecured Pdelay messages; TC using release version 1.1.2 (August 2017)

B. Run: master started with `_m.conf`, slave started with `_s.conf`, TC started with `killall ptp4l; killall phc2sys; cd /ptp/p2p/tc; ./linuxptp.sh`

C. Verify: no Pdelay_Response_Followup messages should be received on either PTPd node, since TC is using one-step mode; in Wireshark, should observe received Sync messages with non-zero correctionField, but since both PTPd nodes should ignore the correctionField in ICV calculation, ICV verification should succeed and offset from master should be observed to approach zero and become stable; Announce, Sync, and Follow_Up messages received by slave should have AUTHENTICATION TLV attached

2.1.2. Fully secured network: ignore *correctionField* OFF

A. Setup: as `imm_2.1.1` above, but with `imm_ignore_correction` OFF; TC using release version 1.1.2 (August 2017)

B. Run: master started with `_m.conf`, slave started with `_s.conf`, TC started with `killall ptp4l; killall phc2sys; cd /ptp/p2p/tc; ./linuxptp.sh`

C. Verify: the slave should drop all received Sync messages, as they will have been modified by the TC, resulting in a different ICV; this should be observed in slave status log with received Sync message counter at 0, and with `icvMismatchError` counters increasing; master status log should show that it is indeed sending Sync messages; in Wireshark, should observe Sync messages received by slave with non-zero correctionField, and with a AUTHENTICATION TLV attached

2.1.3. Mixed network:

A. Setup: master must accept unsecured Pdelay messages; slave built without security; TC using release version 1.1.2 (August 2017)

B. Run: master started with `_m.conf`, slave started with `_s.conf`, TC started with `killall ptp4l; killall phc2sys; cd /ptp/p2p/tc; ./linuxptp.sh`

C. Verify: same as `reg_2.1.1`, since slave should ignore attached AUTHENTICATION TLVs on Announce, Sync, and Follow_Up messages

B.3 Delayed Verification Tests

1. Direct topology: master (Mac) directly connected to slave (Pi)

1.1. Functionality:

1.1.1. Fully secured network

A. Setup: master and slave must have same delayed options: `chain_length=500`, `interval_duration=5`, `disclosure_delay=2`, `d_t=0.0002`, `start_time` set based on Unix epoch time

B. Run: master started with `del_1.1.1_m.conf`, slave started with `del_1.1.1_s.conf`

C. Verify: slave status log should show increase in `safePackets` counter, and increase in `keyVerificationSuccesses` every 5 seconds (once per interval); in Wireshark, all messages from master should have AUTHENTICATION TLV attached; General messages from master should include a disclosed key, while Event messages should not; outgoing messages from slave (Pdelay mesages) should not have AUTHENTICATION TLV attached

1.1.2. Mixed network

A. Setup: master as `del_1.1.1`, slave built without security

B. Run: master started with `del_1.1.2_m.conf`, slave started with `del_1.1.2_s.conf`

C. Verify: same as `imm_1.1.2`; in Wireshark, AUTHENTICATION TLV will be present on all messages from master until the key chain is exhausted (when `chain_length` * `interval_duration` seconds have passed since the start time

1.1.3. Unsafe packet (RFC 4082 3.5.1)

A. Setup: master compiled from test branch with test values `testUnsafePacket=TRUE`, `unsafePacketInterval=10`
B. Run: master started with `del_1.1.3_m.conf`, slave started with `del_1.1.3_s.conf`
C. Verify: in the slave status log, in interval 10, the `unsafePackets` counter will increment to 1; in the slave event log, before the INFO messages reporting the buffering of messages into buffer 10, should see "unsafe packet seqID x (interval 8)"; this sequenceID x can be used to identify the message in the Wireshark capture; in Wireshark, the doctored message with seqID x should have 8 in the keyId field, while the rest of the messages from interval 10 should have 10 in their keyId fields

1.1.4. Disclosed key missing for entire interval

A. Setup: master compiled from test branch with test values `discKeySkipInterval=TRUE`, `intervalToSkip=10`
B. Run: master started with `del_1.1.4_m.conf`, slave started with `del_1.1.4_s.conf`
C. Verify: since `disclosure_delay` is 2, then in interval 12, the master should not disclose key 10, and thus the master event log should show INFO messages "currentInterval 11, disclosing key 9" followed by "currentInterval 13, disclosing key 11"; on the slave side, messages from interval 10 will have already been buffered, but will not be ready for verification; in interval 13, master will disclose key 11, which through the verification process, will reveal missing key 10; thus, in interval 13, both buffers 10 and 11 should be verified; this should be observed in the slave event log as two consecutive buffer dumps for buffers 10 and 11; in Wireshark, no General messages from the master from interval 12 should contain a disclosed key; this is verified by using the slave event log to find the buffer dump of messages from buffer 12, and using the seqIDs from these messages to find them in the Wireshark capture

1.2. Attack scenarios:

1.2.1. Rogue master: master unsecure

A. Setup: master built without security; security enabled on slave
B. Run: master started with `del_1.2.1_m.conf`, slave started with `del_1.2.1_s.conf`
C. Verify: same as `imm_1.2.1`

1.2.2. Rogue master: different keychain

A. Setup: master first byte of key is `0xff` instead of `0xed`
B. Run: master started with `del_1.2.2_m.conf`, slave started with `del_1.2.2_s.conf`
C. Verify: all Event messages with no disclosed key should still get buffered, thus in the slave status log, the `safePackets` counter should increase; since General messages containing disclosed keys should get rejected as

soon as the disclosed key fails verification, the `keyVerificationFails` counter in the slave status log should increase; in the slave event log, should see some Event messages being buffered, we should see reported key verification failures, and we should see no buffer dumps

1.2.3. Rogue master: different ICV algorithm
   A. Setup: slave configured to use HMAC-SHA256; master configured to use GMAC
   B. Run: master started with `del_1.2.3_m.conf`, slave started with `del_1.2.3_s.conf`
   C. Verify: same as `imm_1.2.3`

1.2.4. MITM: disclosed key altered (RFC 4082 3.5.3)
   A. Setup: master compiled from test branch with test values `MITM=true`, `MITM_attack=DISCKEY_ALTERED`, `IntervalToAlterDiscKey=10`
   B. Run: master started with `del_1.2.4_m.conf`, slave started with `del_1.2.4_s.conf`
   C. Verify: in interval 10, the first message to disclose a key (key 8) will have an altered key; thus, in the slave event log, just before the first INFO message "buffering msg into buff 10", should see an INFO message reporting key verification failure "key verification failed on key ff (interval 8) from message w/ SeqId x"; after this, should see a key verification message "verified key 77 (interval: 8)"; this is because the next General message after the one carrying the altered key should carry a valid disclosed key, which will be verified; later in the event log, the buffer dump for buffer 10 should not include the message with seqId x that contained the altered key, as that message should have been discarded; in Wireshark, the message with seqId x should have all `0xff` in its disclosedKey field

1.2.5. MITM: PTP payload altered (RFC 4082 3.5.4)
   A. Setup: master compiled from test branch with test values `MITM=true`, `MITM_attack=PAYLOAD_ALTERED`, `targetMessageSeqId=15`
   B. Run: master started with `del_1.2.5_m.conf`, slave started with `del_1.2.5_s.conf`
   C. Verify: the slave status log should show the `icvMismatchError` counter set to 1; the slave event log should show INFO message "ICVs didn't match on SeqId 0x000f" followed by "message type: announce (seqId 000f) w/ ICV: *xx...xx* verified? 0" to indicate ICV verification failure in the buffer dump containing the Announce message with seqID 15 (`0x0f`); in Wireshark, the Announce message with seqId 15 should have payload set to all `0xff`

1.2.6. MITM: ICV altered (RFC 4082 3.5.4)
   A. Setup: master compiled from test branch with test values `MITM=true`, `MITM_attack=ICV_ALTERED`, `targetMessageSeqId=15`
   B. Run: master started with `del_1.2.6_m.conf`, slave started with `del_1.2.6_s.conf`

C. Verify: the slave status log should show the `icvMismatchError` counter set to 1; the slave event log should show INFO message "ICVs didn't match on SeqId 0x000f" followed by "message type: announce (seqId 000f) w/ ICV: ff..ff verified? 0" to indicate ICV verification failure in the buffer dump containing the Announce message with seqID 15 (`0x0f`); in Wireshark, the Announce message with seqId 15 should have ICV set to all `0xff`

2. Multi-hop topology: master (Pi) connected to TC (Microchip port 2), and slave (Mac) connected to TC (Microchip port 1)

   2.1. Functionality:

   2.1.1. Fully secured network

   A. Setup: security enabled on both master and slave as `del_1.1.1`; TC using release version 1.1.2 (August 2017)

   B. Run: master started with `_m.conf`, slave started with `_s.conf`,
   TC started with `killall ptp4l; killall phc2sys; cd /ptp/p2p/tc; ./linuxptp.sh`

   C. Verify: in the delayed verification approach, the correctionField is not considered in ICV calculation by neither master nor slave; thus, the same verification tests apply here as in `del_1.1.1`; to verify proper handling of the correctionField, we need to observe slave behavior when it verifies messages from a given buffer as follows: in Wireshark, Sync messages incoming to slave should have non-zero correctionField; if ICV verification passes, it means slave correctly ignored correctionField prior to calculating ICV; finally, buffered messages should remain in their buffer after ICV verification with the values in their correctionField unmodified - this should be verified in the buffer dump in the slave event log, which should include the value of the correctionField

   2.1.2. Mixed network

   A. Setup: security enabled on master as `del_1.1.1`; slave built without security; TC using release version 1.1.2 (August 2017)

   B. Run: master started with `_m.conf`, slave started with `_s.conf`,
   TC started with `killall ptp4l; killall phc2sys; cd /ptp/p2p/tc; ./linuxptp.sh`

   C. Verify: same as `reg_2.1.1`, since slave should ignore attached AUTHENTICATION TLVs on Announce, Sync, and Follow_Up messages; in Wireshark, AUTHENTICATION TLV will be present on all messages from master until the key chain is exhausted
   (when `chain_length` * `interval_duration` seconds have passed since the start time)