3-26-2020

A General Methodology to Optimize and Benchmark Edge Devices

Kyle J. Smathers

A General Methodology to Optimize and Benchmark Edge Devices

THESIS

Kyle J Smathers, B.S.C.E., CISSP, Captain, USAF

AFIT-ENG-MS-20-M-062

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-20-M-062

A GENERAL METHODOLOGY TO OPTIMIZE AND BENCHMARK EDGE DEVICES

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyberspace Operations

Kyle J Smathers, B.S.C.E., CISSP,

Captain, USAF

March 26, 2020

AFIT-ENG-MS-20-M-062

A GENERAL METHODOLOGY TO OPTIMIZE AND BENCHMARK EDGE DEVICES

THESIS

Kyle J Smathers, B.S.C.E., CISSP,
Captain, USAF

Committee Membership:

Lt Col Mark E DeYoung, Ph.D
Chair

Gilbert L Peterson, Ph.D
Member

Timothy H Lacey, Ph.D
Member

AFIT-ENG-MS-20-M-062

Abstract

The explosion of Internet Of Things (IoT), embedded and "smart" devices has also seen the addition of "general purpose" single board computers also referred to as "edge devices." Determining if one of these generic devices meets the need of a new given task however can be challenging. Software generically written to be portable or plug and play may be too bloated to work properly without significant modification due to much tighter hardware resources. Previous work in this area has been focused on *micro* or chip-level benchmarking which is mainly useful for chip designers or low level system integrators. A higher or *macro* level method is needed to not only observe the behavior of these devices under a load but ensure they are appropriately configured for the new task, especially as they begin being integrated on platforms with higher cost of failure like self driving cars or drones.

In this research we propose a macro level methodology that iteratively benchmarks and optimizes specific workloads on edge devices. With automation provided by Ansible, a multi stage 2^k full factorial experiment and robust analysis process ensures the test workload is maximizing the use of available resources before establishing a final benchmark score. By framing the validation tests with a family of network security monitoring applications an end to end scenario fully exercises and validates the developed process. This also provides an additional vector for future research in the realm of network security. The analysis of the results show the developed process met it's original design goals and intentions, with the added fact that the latest edge devices like the XAVIER, TX2 and RPi4 can easily perform as a edge network sensor.

Table of Contents

Page

List of Figures

Figure Page

List of Tables

A GENERAL METHODOLOGY TO OPTIMIZE AND BENCHMARK EDGE DEVICES

I. Introduction

1.1 Problem Background

The explosion of Internet Of Things (IoT), embedded and "smart" devices has been driven by the increased availability of minimized hardware that carries enough computing power to accomplish very specific applications. This market has also seen the addition of "general purpose" single board computers also referred to as "edge devices." This terminology is lent from the fact a portion of computation traditionally performed by a centralized point is moved down to a more tactical or perhaps disconnected level (e.g. a drone flying itself or perhaps an ATM detecting fraud on it's own based upon the users body language). Some vendors go as far as saying their 15 watt single board computer is a "supercomputer on a chip" [4].

Determining if one of these generic devices meets the need of a new given task however can be challenging. By design there is not much tolerance for hardware resource abuse either in the code running on them or from any operating system controls. Suddenly software generically written to be portable or plug and play may be too bloated to work properly without significant modification. In addition, while some of these edge devices may act independently others may be working together over a mesh network like ZigBee or 802.11s. If a comprehensive view of system performance is desired, some form of automated simultaneous testing is required.

Previous work in this area has been focused on *micro* or chip-level benchmarking

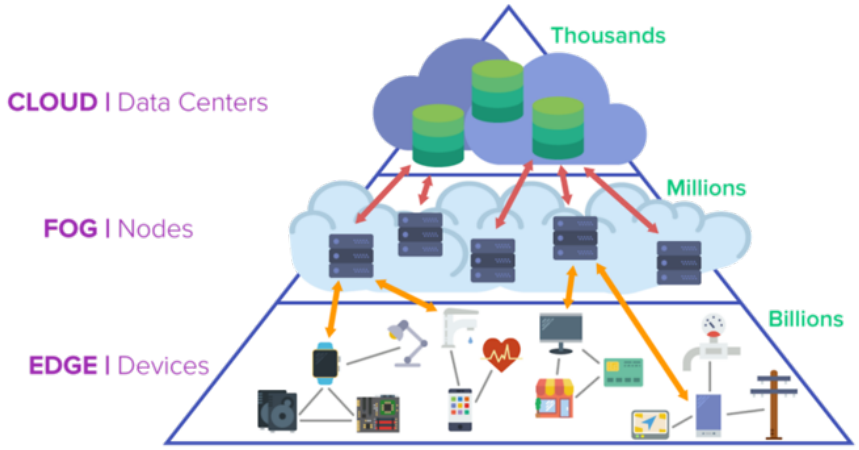


Figure 1: Scale of Edge-Like Devices [5]

[6]. These efforts are mainly useful for chip designers or low level system integrators. A higher or *macro* level method is required to not only observe the behavior of these devices under a load but ensure they are appropriately configured for the task.

1.2 Research Objectives

The objective of this research is to develop and document a macro level methodology to optimize and benchmark specific workloads on edge devices. The specific workload and thus specific performance metric needs to be interchangeable in order to best apply to all the possible scenarios an edge device may be employed (e.g. frames per second for a image tracker, power consumption for a battery powered drone, or packets dropped for a network monitor).

By framing the development with a family of network security monitoring applications an end to end scenario fully exercises and validates the developed process. This also provides an additional vector for future research in the realm of network security. Internet based security appliances which are classically done in a centralized fashion would likely stand to benefit from an edge-like deployment scenario. Decentralized sensors would greatly enhance the ability to monitor niche enclaves and devices that

historically have gone completely unprotected but carry much of the same risks (e.g. All-in-one printers or video teleconference suites) [7].

1.2.1 Design Goals

In order to apply as broadly as possible, be academically sound and actually useful, the methodology was built inline the following goals:

1. Be easy to use, modify, and scale as appropriate
    - In order to be a general methodology, it needs to easy accept different test workloads
    - Each workload will have a unique set of variable factors to consider
    - Each physical device may not have the same resources which should be accounted for
    - In cases where workloads are spread across multiple devices, the methodology must be able to test them all at the same time
2. Support a repeatable consistent baseline
    - For repeatability, it should be easy to deploy and carry no hidden system changes or tuning
3. Maximize the use of available hardware
    - Because of tight performance envelopes, ensure the result is utilizing the maximum extent of resources possible (i.e. reduce idle or blocking conditions)
4. Utilize a robust statistical analysis capability
    - In order to provide rigor to the result. In cases where statistical tests breakdown (i.e. non-normality), provide a backup test.

1.2.2 Implementation

The first two goals were met with the implementation of an automated workflow. The core of this automation was created using Ansible which is an open-source software provision, configuration, and application deployment tool [8] while further

interchangeable sub-modules perform the results analysis. The pairing of these two allow both the workload and response metric to be swappable.

Maximizing the use of the hardware is met through the operator selecting up to five potential optimization factors which are then tested in a 2^k Full Factorial experiment design. This ensures any positive or negative interactions between any factor are appropriately captured. By running the automation workflow multiple times in series, a multi-stage optimization process emerges that can find the "ideal" level for a given factor (e.g. sizing a buffer in memory without under or oversizing it).

The results once captured are analyzed one of two ways. By performing a Analysis of Variance (ANOVA) test it becomes possible to put rigor behind the result, associating the probability that any given result was just due to random luck. In cases where the ANOVA model fails to fit a simpler arithmetic comparison provides a simpler result.

1.2.3 Validation

Once the methodology was developed, it was exercised and validated with a series of network monitoring applications. The particular applications tested were based off real world Air Force network monitoring pipeline roles (Section 2.1.3.1) and focused on the performance metric of packet loss. In addition each test has a unique selection of optimization factors which are evaluated in the 2^k full factorial fashion. While a secondary effort these tests establish a worthwhile baseline for future work discussed in Section 5.2.2 and Appendix B.

1.3 Document Overview

This document is broken into 4 more chapters. First motivations and background are discussed in Chapter II while the methodology is discussed and implemented in Chapter III. Chapter IV validates the methodology through a examination of three network monitoring applications and Chapter V presents the conclusion and future work.

Select raw data used in the figures is shown in Appendix C while all the core re-usable code with a user guide is presented in Appendix D. Full code, data, and instructions will also be available electronically [9].

II. Background and Literature Review

Initial motivations for this work was framed around finding a solution to the gaps that exist in enterprise network defenses [7]. Recent modern network security suites and tactics have been very focused on high vantage points and large centralized collection points. This provides high visibility into all traffic flowing into and out of the protected network but can neglect side to side or "lateral" traffic. By decentralizing the classical threat hunting platform with smaller, external sensor suites close to the protected clients ("the edge") we reduce the hardware requirement from monitoring a whole enterprise to just one or a handful of hosts.

The solution to this problem presented another. The candidate edge devices that could perform this task come in all shapes and sizes and no method exists to examine how they perform under a given workload. This was further compounded by the edge device's tighter hardware budgets that are less tolerant to unoptimized configurations. Figure 2 below visually summarizes the relationship of the problems and direction of this work.

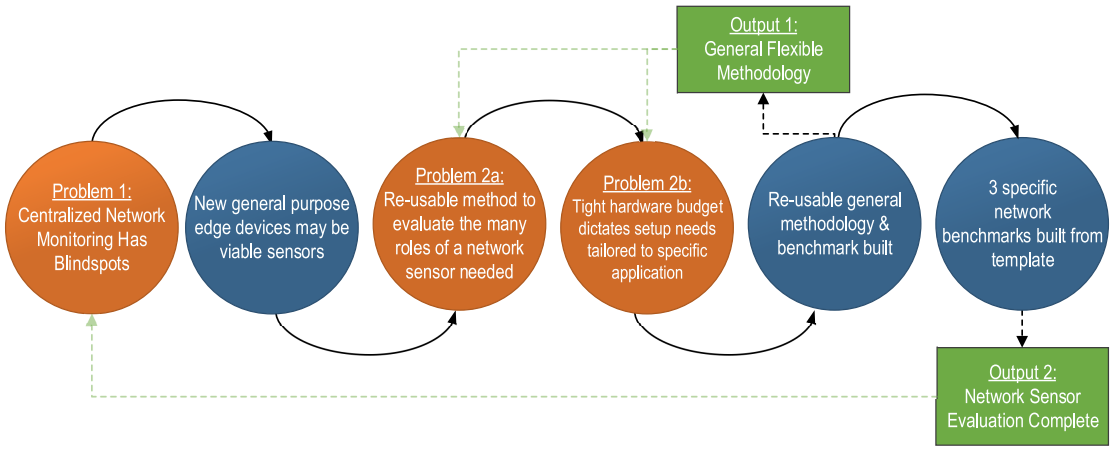


Figure 2: Visual summary of the relationship between the research problems and direction of this work

This chapter examines the background of these problems much deeper. Section 2.1 breaks down the evolution of contemporary network monitoring to examine the blind spot problem and can be skipped if only interested in the second problem, which is discussed in Section 2.2.

2.1 Problem Domain - Network Defense

Computer network defense has evolved from an afterthought to a multi-billion dollar industry [10]. Despite the attention and focus gaps still remain that are targeted by adversarial forces for personal or political gain. This section looks at the origin of contemporary tactics, modern threats to them and associated emerging counter tactics.

2.1.1 Network Defense Background

Early Days c. 1970-1989

The Internet's (and thus Cyberspace's) primordial days were less concerned about security and more about just raw functionality. The noble beginnings of ARPANET for sharing information either between universities or government agencies had little need for security if everyone who was connected was "trusted." David D. Clark, an Internet pioneer and chief protocol architect once said

> *Its not that we didn't think about security, We knew that there were untrustworthy people out there, and we thought we could exclude them [11].*

Later, Clarks' seminal 1988 paper detailing the design principles of early internet protocols included no mention of security [11] [12]. Unfortunately, in an effort to get more nodes connected the barrier to entry was quite low, and the popularity of the net grew faster then it could be contained. It was around this time that the net saw its first large scale infections and attacks with the Morris worm in 1988 [13], international

military espionage at the hands of KGB agents in 1986 [14], and malicious insiders trying to gain unauthorized access to Bell Labs [15]. As a result the naive "trust" that built the early Internet began to dissolve, meaning some form of security needed to be added to a system that didn't even consider it. Thus the emergence of "bolt-on" security began to unfold with the multi billion dollar cyber security industry (valued 104 billion in 2017 [10]). Networks began segregating themselves with devices like firewalls and proxies. System administrators and owners could now only trust their own users (which thanks to insiders and phishing we see later that's a bad idea) leading us to the era of the great filters, choke points and sensors that still remain in place today.

Passive / Reactionary Defenses c. 1990-2010+

While the early network based firewalls did their jobs blocking traffic off of simple source/destination filters this was only the first step in the game of cat-and-mouse that defenders and attackers have been playing ever since [16]. While there may have been a relative calm in the early nineties thanks to the tightening of boundaries hackers soon shifted their focus to striking the weakest link: the user. No number of security appliances or software could stop a user from giving up their password to a clever social engineering campaign or stop them from clicking a bad link.

The mid to late nineties saw the first large scale phishing campaigns with AOHell in 1995 [17], the melissa virus in 1999 [18] and the ILOVEYOU worm in 2000 [19]. By simply sending gullible users believable messages attackers were able to bypass these newly erected barriers with ease. The modus operandi for criminals quickly shifted with the rise of e-commerce and e-banking in the early to mid 2000s [20]. Figure 3 below shows a near exponential increase in reported phishing attacks towards the end of the decade. The early 2010s saw the first (caveat, reported) high profile tar-

geted phishing attacks for motivations that were likely not financial. In 2011 Chinese phishing campaigns began targeting US and South Korean government officials for political or espionage gain [21]. The same year US defense contractors were breached in a multi stage attack on themselves and their security vendor RSA Security LLC [22].

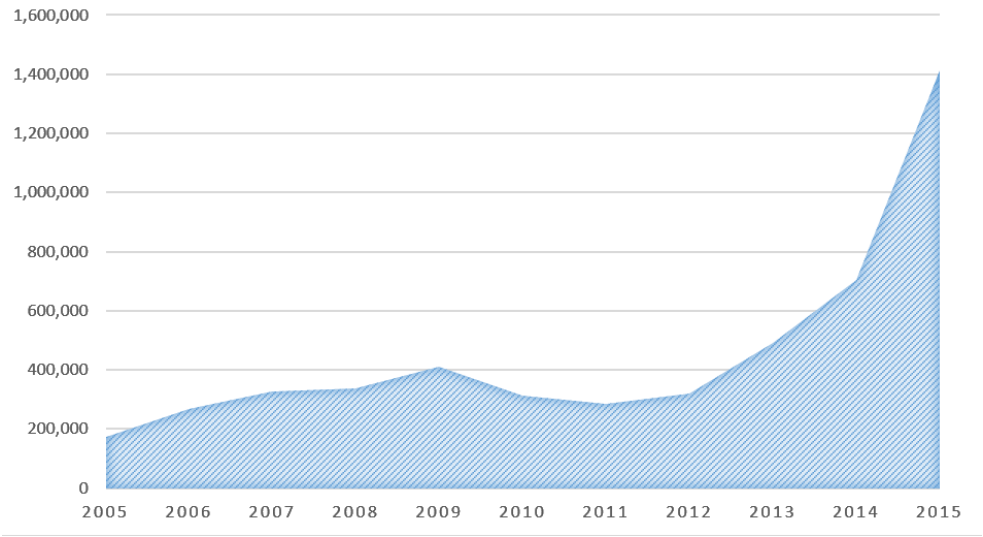


Figure 3: Phishing campaign reports by year [23]

Perhaps even more nefarious are compromised accounts that bypass all security measures without raising any alarm. From 1998-1999 one of the first ever large scale Advanced Persistent Threat (APT) hunts "MOONLIGHT MAZE" began with Wright-Patterson and Air Force Institute of Technology (AFIT) at it's epicenter thanks to a compromised account. The recently declassified report shows suspected Russian actors combing through and collecting thousands of sensitive but unclassified documents from the base, the school, and a handful of national labs. While the initial vector was a compromised account the actual beginning of the infiltration is unknown and was only discovered by accident from a careful admin reviewing logs [24]. This prompted many in the Department of Defense (DOD) to consider creating a third "public / business partners" network essentially transforming Non-classified Inter-

net Protocol Router Network (NIPRNET) into an totally disconnected unclassified version of it's cousin Secret Internet Protocol Router Network (SIPRNET). History tells us this didn't happen but instead the branches adapted the "defense in depth" strategy that is discussed further below [25].

It was in this era the realization that even internal users could not be trusted began to take hold. A variety of different strategies were born each with their own side-effects in an attempt to deal with the ever growing problem. Defense-in-depth or multi layered defenses is one such strategy re-popularized by the National Security Agency (NSA) that has been around since medieval castle times [26] [27]. Unfortunately, it's a poor analogy at best, as the original strategy involves layering defenses and purposefully ceding land in such a way to allow time for a counterattack and ultimate defeat of the aggressor. As poignantly pointed out by one author, this approach doesn't really work for cyber defense since 1.) There is typically not any "land" to cede and 2.) You typically can not counterattack and then wipe out the adversary for good. This broad "just layer more defenses" leaves too much interpretation to the operator and the free market. A false sense of security can easily result and its general failure as a strategy is clear in the eight large multi-million dollar hacks that happened in the publishing year alone [28]. What we're left with is a fragmented, compartmentalized mess that makes it very difficult for defenders to actually defend or fully understand the scale of.

Multiple standards and regulations thus came into existence to try and clearly define what "secure" means. The Defense Information Systems Agency (DISA) has been releasing Security Technical Implementation Guide (STIG) since as early as 1989. These guidelines are useful in trying to quantitatively define best practices and when combined with other standards like DoD Instruction (DODI) 8510.01 *DoD Information Assurance Certification and Accreditation Process (DIACAP)* and Na-

tional Institute of Standards and Technology (NIST) Special Publication 800-53 *Security and Privacy Controls for Federal Information Systems and Organizations* a system is likely better off then just hiding flaws behind multiple layers of firewalls. Unfortunately, the panacea of a perfect security checklist does not exist nor will it ever with the rapid pace of innovation in new software / hardware. The clash between those enforcing these checklists (like DISA's Command Cyber Readiness Inspection (CCRI)) and those trying to meet the intent of the system owner is a common source of friction. Luckily later revisions of DODI 8510.01 *Risk Management Framework* better addresses this friction and allows system owners more say on the risk they are willing to accept [29].

Despite all this, persistent adversaries continued their campaigns against both governments and corporations alike. Reactionary detection was failing as was evident in the year long loiter times of some campaigns. Chinese actors were indicated in the three year long Titan Rain campaign (2003-2005) [30] and the year long Operation Aurora (2009-2010) that targeted Google and other US based tech companies [31]. The importance of rooting out compromises increased dramatically in the 2010's as the threats shifted from espionage / stealing of documents to causing physical effects. Stuxnet (2009-2010), widely regarded as the first "cyber weapon" was destroying centrifuges for months before being discovered [32], and the Ukraine power grid shutdown of 2015 had months of preparatory work on net before the obvious effect came forward [33].

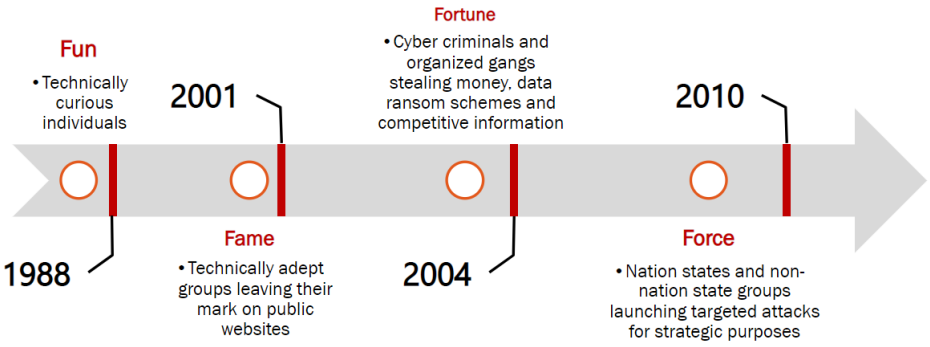


Figure 4: Evolution of cyber attacks and their motivations [34]

These events precipitated the need for a more proactive approach to cyber defense. At the end of the day these controls and measures only stop low sophistication attacks and slow down the complex ones. Zero day exploits, spear phishing, social engineering and insider threats still poised a very real danger.

Active / Proactive Hunting c. 2010+

The need to begin proactively defending was clear at the beginning of the 2010s. As seen in figure 5 below, either the in-place defenses of these institutions failed / are being manipulated, the local defenders are under staffed, trained or experienced [35], or the overall network visibility and situational awareness across their enterprise is severely lacking. (Note these are not mutually exclusive, and all three can be happening to some degree to the point of compromise)

GLOBAL MEDIAN DWELL TIME

Compromise Notification	2011	2012	2013	2014	2015	2016	2017	2018
All	416	243	229	205	146	99	101	78
External					320	107	186	184
Internal					56	80	57.5	50.5

Figure 5: Cyber Attack Dwell Times (time between initial access and detection) in days by Year [36]

With multiple month and year long dwell times being uncovered (calculated as the number of days an attacker is present on a victim network, from first evidence of compromise to detection) a not so unreasonable fear began to muster for most large enterprises and the military. "Have we been hacked and we don't even know it yet?" was probably a common phrase uttered in board meetings. While these enterprises may have had well established incident response capabilities APTs had grown just as comfortable evading their very predictable passive defenses [37]. Thus in the early 2010s many quickly turned to their forensic teams to figure this out, forming a new breed of proactive investigators that became known as "Threat Hunters "[38].

Where private companies like Mandiant and SANS began capitalizing on the new opportunities taking care of corporate and international contracts, the military was aggressively expanding their capabilities as well [38]. The Air Force created "Blue Teams" in contrast to the traditional "Red Teams" with the goal of sweeping for APT presence, scanning for better visibility while also assisting and training local defenders [39] [40]. These teams later became the mold for United States Cyber Command (USCYBERCOM)'s Cyber Protection Team (CPT)s which were coined as the joint force equivalent to assess, sweep, repair, and instruct local defenders [41] [42].

Many of these efforts contributed to the decrease in the median dwell time of attackers as seen at the end of Figure 5 above. While historical forensics were carried out looking for hard particular artifacts, the scale that hunt teams were responsible for was potentially infinite (anything and everything, "unknown unknowns"). If the signature of an exact attack was already known, its Indicator of Compromise (IOC) could just be plugged into the signature based passive defenses. Thus as the name implies, in order to be successful threat hunters typically require some information about a valid threat such as the who (what nation states or actors), what (what are

they after), when (is there a big event/anniversary coming up?), where (physical, digital, or persona), and how (tools, techniques and procedures).

Deriving and gathering the sort of information needed to properly hunt threats is still very challenging. Figure 6 below helps demonstrate this highlighting over 47% of all discovered attacks still have a dwell time over 90 days. Sharing IOC has been a hot area with a few government and private efforts like Structured Threat Information eXpression (STIX) and Trusted Automated eXchange of Indicator Information (TAXII) attempting to overcome the difficulty of sharing complex indicator data [43]. Many other organizations still struggle however with sharing potentially private or proprietary secrets when revealing their compromises.

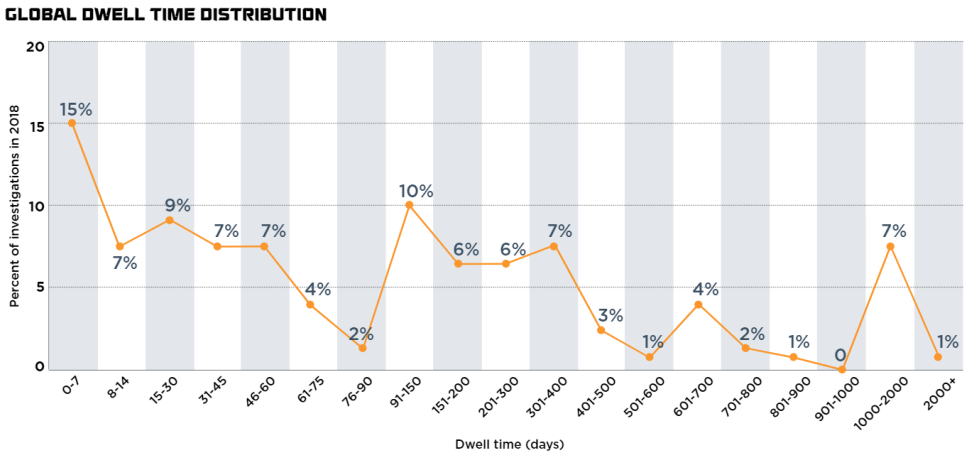


Figure 6: Global Cyber Attack Dwell Time Distribution [36]

2.1.2 Blindspots & Modern Threats

The current state of network defense can succinctly be visualized in Figure 7 below. The intersection between a motivated actor, a vulnerability and an opportunity ultimately leads to a compromise. The next few subsections examine these points further.

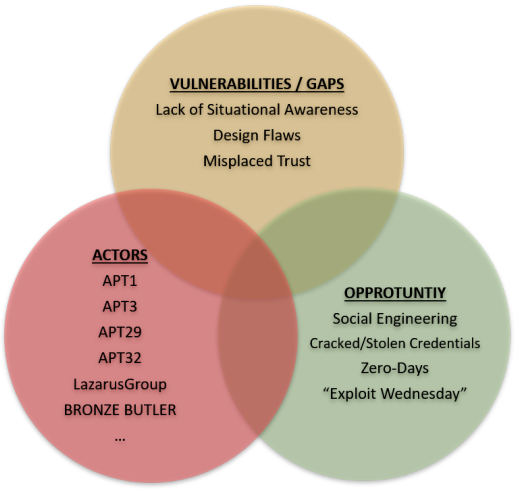


Figure 7: Threat Model depicting the current state of the art [44] [45]

2.1.2.1 Vulnerabilities / Gaps

The hardware requirement to maintain situational awareness of an entire enterprise can be substantial. Capturing traffic at line rates requires storage that can keep up for the desired retention period, typically multiple terabytes (e.g. 152 TB is required to store two weeks at 1 Gbps sustained) [46]. Meanwhile processing thousands of intrusion signatures on the same traffic requires dozens of CPU cores and a few hundred gigabytes of RAM as detailed in the Suricata High Performance documentation [47].

The best cost efficiency has historically come from tapping central egress/ingress points as shown with the Air Force's 16 gateways which was later driven to even further cost sharing with the Army and larger DoD's Joint Regional Security Stack (JRSS) [48]. Threat hunters have leveraged this enhanced visibility to shorten dwell times but gaps still remain in uninspected trusted insider zones as evidenced by the latest Maindiant M-Trends Report [36] and even recent "Hack The Air Force" events [49]. As seen below in section 2.1.2.2, many threat actors actively leverage these blind spots and enjoy free reign as "trusted" lateral traffic [7].

As an example, a critical vulnerability in HP all in one printers was recently discovered by security researchers. By sending a specially crafted fax, it was possible to establish a foothold on the printer which was then pivoted into an internal network. This is thanks to the printer having a full blown Operating System (OS) shoved into the device [50]. All command, control and data exfiltration was done over the traditional phone line which means the only possible way this would have been spotted was via the network traffic generating from the printer. While in this scenario this enterprise could have had very robust boundary protections and inspections, it is unlikely due to cost or bandwidth limitations that the traffic from this printer or the random office "last mile" switch would be spanned or cloned to a sensor. Nor would there be a host based agent riding on top the limited firmware.

Log shipping agents like Beats can be deployed on supported hosts / end-points to help cover part of this gap though the number of unsupported appliances like printers and video teleconferce suites as seen still contain enough of an operating system to become a target. Even the supported hosts are susceptible to log manipulation, [51] with many actors taking advantage of this and hiding their tracks [52] [53]. Not to say it's not worth collecting logs, as sometimes the sudden absence of certain events may be an indicator in itself.

Unfortunately, logs only go far especially if valid credentials as being used as any log entries will only show successes. Especially damning are third party based authentication system compromises (like Kerberos and Active Directory) as in some situations no log is generated good or bad. Introduced in 1988 by MIT the Kerberos authentication protocol is widely used as the backbone to how Active Directory performs authentication. Microsoft's implementation has interesting design choices however that allow stateless ticket based attacks (and thus lateral movement) to work [54].

In many cases in place cyber defenses are not adequate to the task of proactive searching and could become a target themselves [37]. As an example, the passive/reactionary suite of tools in Host Based Security System (HBSS) were not designed for active threat hunting as any reactionary measures they perform require signatures and other parts only report issues like patch compliance. In addition, variants of these agents were attacked and disabled by malware in the Lazarus Group's 2014 hack of Sony Pictures Entertainment [55].

2.1.2.2 Motivated Actors

Many open source databases have begun tracking APT and their specific tactics and tools. The MITRE Corporation publishes the MITRE ATT&CK Matrix which conglomerates sources from companies like Symantec, Mandiant, Microsoft, McAfee and others. It has become a reference point for Air Force hunt operations and even some endpoint protection suites. Table 1 below highlights a few actors who exploit many of the items discussed above.

Table 1: Sampling of threat actors observed exploiting factors discussed previously.

Actor	Tactics	Targets	Source
APT1	Pass the Hash psexec lateral movement	US Canada South Korea	[56]
APT3	Windows share abuse Native lateral movement	US	[57]
BRONZE BUTLER	Stolen credentials Forged Kerberos	Japan	[58]
APT29	Pass the Ticket Powershell evasion	US	[53]
APT32	Remote Pass the Ticket Native lateral movement	Southeast Asia	[59]
Lazarus Group	Disables security tools Windows share abuse Native lateral movement	US Central America	[52]
WannaCry	Ransomware "Zero day" lateral movement	Worldwide	[60][61]

2.1.2.3 Opportunities

Just because a motivated actor exists and its target is vulnerable doesn't mean a compromise is imminent. The proper opportunity must still present itself, perhaps as an improperly exposed port, a gullible user clicking a link, or a previous successful attack that can be leveraged as a pivot. Some opportunities appear inline with a vulnerability like a zero-day vulnerability or "Exploit Wednesday" and must be addressed as quickly as possible since it is safe to assume a motivated actor is just waiting for their chance [45].

2.1.3 Emerging Defense Tactics

In the absence of any concrete indicators a threat hunter must begin to think like a potential adversary. By becoming more aware of the state of the targets defenses like gaps in their oversight or other trust relationships the hunter can narrow their search. This brings us close to the present approach that has quickly gained popularity: "Big Data or Data Analytics". By aggregating as many data sources as possible together (e.g. logs, packet captures, compliance scans, etc.) to form a comprehensive view of an enterprise, anomalies quickly jump out to a trained eye or in even more recent cases, a trained algorithm. This shift enables detection of "unknown unknowns" and is especially interesting as it theoretically solves both the visibility and the experience shortage, reducing the number of "trained/experienced" operators needed.

2.1.3.1 Big Data Pipelines

While many open source and commercial products exist that can implement a data processing pipeline like this, the tools used/examined in this thesis will be based off the open source Elastic stack (Beats, Elasticsearch, Logstash, Kibana). We select the Elastic stack because a free-tier licence is available, it is used in Air Force systems, and the stack's popularity [62]. Below is an example of one such pipeline as implemented in another open source project, the Response Operations Collection Kit Network Security Monitor (RockNSM) suite.

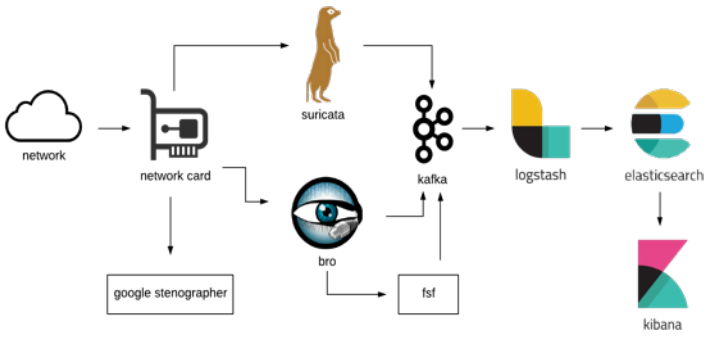


Figure 8: Example of the RockNSM sensor pipeline, a popular open source security sensor suite [63]

Suricata is the signature based Intrusion Detection System (IDS), Google Stenographer captures raw packets and bro handles netflow and metadata. Logstatsh and Kafka aggregate all types of logs and alerts to be stored in the Elasticsearch database where it is reliably stored and indexed. Lastly Kibana provides rapid retrieval and analysis of the data with user friendly dashboards and queries [63].

For some, a pipeline like this is achievable at an enterprise level depending on budget and size. Unfortunately, for many the scale involved in aggregating hundreds of thousands of workstation and server logs in the is no simple feat. Especially if not planned for / budgeted in the initial build-out thanks to the bandwidth needed alone.

2.1.3.2 Mobile Hunt Platforms

While in large in place implementations of a data pipeline can be effective, hunt teams commonly maintain their own suite of tools and sensors for either a unmanipulated third party perspective or to gain situational awareness on a enterprise that lacks it. They are architected in a way to allow expandability and portability. Traditionally this has meant airline friendly (weighing less then 100 pounds) transit cases full of multiple high end server chassis that can run any variety of roles from packet capture and Network Intrusion Detection System (NIDS) to Security Informa-

tion and Event Management (SIEM) aggregators and compliance scanners. In early 2010 the Air Force led the way with their initial creation of a mobile "interceptor" hunting platform, which later in 2013 became one of the first declared defensive cyber weapons Cyberspace Vulnerability Assessment / Hunter (CVA/H) [64].

Having flexibility in these roles is crucial as no two landscapes are the same. Rack space, available power, available cooling, and logical topology can all vary between deployment locations. By allowing each role to be platform agnostic a robust and scalable pipeline can be built on anything from a single physical machine to a half rack full of equipment or even entirely virtual in a cloud provider.

Figure 9: Typical high performance network sensor server with an "airline approved" hard sided transit case. Typical airline restrictions limit special checked items like this to 100 lbs [65]

Unfortunately, these data pipeline implementations can suffer from the same problems as their large stationary counterparts. The cost and scale of their hardware can limit their deployment scope which as seen previously in Section 2.1.2.1, leaves gaps that adversaries have and will continue to exploit to achieve their goals.

2.2 Solution Domain - Edge Devices

Where before the software involved in a contemporary data pipeline required substantial hardware that made edge deployment cost prohibitive, certain families of embedded hardware have emerged that may be able to provide the functional capabilities necessary to function as an edge sensor at a fraction of the cost.

2.2.1 Survey of Edge Devices

Edge devices come in an array of form factors, cost, and capability. Many come in a Single Board Computer (SBC) configuration which includes things like standard Input / Output (I/O) ports effectively making it an analog to a regular computer tower. An example of this is the very low cost and educationally focused Raspberry Pi boards. Another common configuration is a specialized "compute module" which contains the same core as its SBC sibling but has only raw pins instead of I/O ports. These are intended for industrial applications and allow systems designers to build their own carrier boards and just outsource the actual core component of it.

(a) Raspberry Pi 3B+ (b) Pi Compute Module 3+

Figure 10: Example of the same edge device on two significantly different form factors

Regardless of form factor the actual cores of these devices generally come in three capability groups. The low cost Raspberry Pi is mainly targeted at educational

and low risk uses [66]. Some devices come with a slightly more focused application and specialization like the Google Coral which features specialized machine learning accelerating hardware [67]. The higher tail of these devices are specialized workhorses that feature very robust computing capability like the NVIDIA Jetson XAVIER and are employed in higher risk environments like drones and automobiles [68].

For this research two lower cost SBC Raspberry Pi and three NVIDIA Jetson compute modules on a carrier board were selected as the devices to evaluate. This range of high and low should represent a good gradient of the commercially available devices as of mid-2019. Table 2 below details more on these device specific components.

Table 2: Selected readily available commercial off the shelf single board computers, as of mid-2019

Device	**Component**	**Value**
NVIDIA TX1	Architecture	aarch64
Compute Module	Cores	4
Default Carrier	CPU MHz	1734
2015	RAM	4GB
	NIC & Driver	Intel e1000e (PCIe x4)
	Distribution	Ubuntu 18.04.2
	Kernel	linux 4.9.140-tegra
	Power Modes	6.5w / 15w
NVIDIA TX2	Architecture	aarch64
Compute Module	Cores	6* (technically 2/4, two die)
Default Carrier	CPU MHz	2035
2017	RAM	8GB
	NIC & Driver	Intel e1000e (PCIe x4)
	Distribution	Ubuntu 18.04.3
	Kernel	linux 4.9.140-tegra
	Power Modes	7.5W / 15W
NVIDIA AGX Xavier	Architecture	aarch64
Compute Module	Cores	8
Default Carrier	CPU MHz	2265
2018	RAM	8GB
	NIC & Driver	Marvel eqos (RGMII)
	Distribution	Ubuntu 18.04.2
	Kernel	linux 4.9.140-tegra
	Power Modes	10w / 15w / 30w
Raspberry Pi 3B+	Architecture	armv7l
SBC	Cores	4
2018	CPU MHz	1400
	RAM	1GB
	NIC & Driver	Microchip Tech lan78xx (usb2)
	Distribution	Raspbian 9.11 (stretch)
	Kernel	linux 4.19.66-v7+
Raspberry Pi 4	Architecture	armv7l
SBC	Cores	4
2019	CPU MHz	1500
	RAM	4GB
	NIC & Driver	Broadcom bcmgenet (usb3)
	Distribution	Raspbian 10 (buster)
	Kernel	linux 4.19.75-v7l+
	Form Factor	Single Board Computer

2.2.2 Edge Benchmarking Challenges

Properly evaluating if these edge devices would function as a network sensor is not as straight forward as it sounds. With vastly more restrictive hardware budgets compared to contemporary computer hardware there is no room for resource abuse either in the applications running or even the operating system itself. These boards are designed for low power consumption, thermal generation and cost. This typically translates less silicon for things like RAM as seen in the Raspberry Pi offerings [69] to slower Central Processing Unit (CPU) clocks or perhaps more aggressive power saving features like entire chip disabling as seen in NVIDIA design guides [68] and pre-installed power profiles [70].

This can be compounded by the very "generic" kernels that vendors build for these devices as they can not predict the ultimate use case. As an example if power consumption was a crucial performance factor and a workload was placed on a headless Raspberry Pi running Raspian with no modification, the "score" would be skewed by the fact the the HDMI port is powered up by default utilizing 30mA despite no monitor being plugged in [71]. Likewise the default NVIDIA JetPack Software Development Kit (SDK) utilizes an Ubuntu image which includes a full blown GNOME Graphical User Interface (GUI), which consumes resources like memory even if no display is connected.

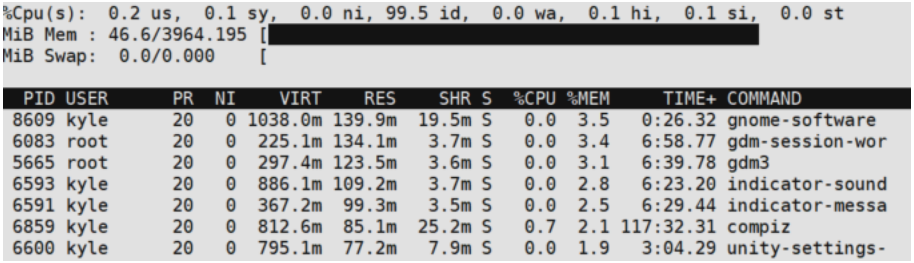


Figure 11: Screenshot of the running processes on a headless TX1. The GUI and related sub-processes are shown using approx 21% of the available memory, despite no monitor or remote desktop services being enabled

Likewise for a network monitoring application like Suricata or Snort the default engine settings will work out of the box but quickly can become overwhelmed by too much traffic even on modest hardware [47] [72]. Both of these tools have a large array of tunable controls that requires consideration of hardware variables like number of cores, interrupt balance and available memory in addition to expected traffic patterns like a lot of small Transmission Control Protocol (TCP) sessions ("flows") or a few large flow sessions. Many cases are not black and white either with some variables acting like a sliding scale. For instance, the Network Interface Card (NIC) rx_ring buffers are set very small by default with low per packet latency in mind which may not really matter when raw throughput is the only critical statistic. Too large of a rx_ring however like one larger then the CPU cache size may have a detrimental effect.

Previous authors have contributed IDS benchmarking research which provides a foundation to test from. A survey by Khalil [72] examines the three big common IDS tools (Bro, Snort, Suricata). Bu [73] examines the impact of virtualization and containerization on Suricata while another group compared and contrasted the single threaded performance of Snort versus the multi threaded Suricata in both live and offline (pcap) scenarios [74]. For both previous works they utilized traffic replay utilities to simulate live traffic. While possible to test the engine functionality by reading directly from a traffic capture file this excludes about 20-30% of the actual engine pipeline from test and there is no real-time urgency to get through the traffic before it gets dropped out of a buffer [75].

Other previous efforts in benchmarking edge devices have primarily been focused on the micro, or chip and architecture scale. MiBench (2001) was an open source suite of tests built targeting embedded like systems [76]. They featured 35 small scale tests based on different categories like basic math and image recognition for

"auto/industrial" or AES, SHA and Dijkstra for "security/network." A more recent update ParMiBench (2010) updated the suite for multi core embedded devices [6]. These tests mainly satisfy chip designers and integrators however which is much too focused for purposes of this research. A broader approach is to take a macro, or end to end, look at the system to test it. One of the only such papers found to discuss such a benchmark on single board systems was still niche in it's findings and implementation, observing only how one application performed across a few different vendors [77]. A variation of this approach however is likely the best course of action to evaluate an application as a whole.

As can be seen simply dropping any particular tool and running it in a pure default environment will not provide a significant indicator of peak performance. To ensure a fair assessment of how an edge device can handle a particular workload, a macro benchmark framework is needed that would allow a wide array of applications to be tested each with their own configuration, optimization, and desired result considerations.

2.3 Summary

This chapter discussed how the state of the art in cyber defense has arrived at large centralized detection platforms. These platforms while successful still allow blind spots that enable unwanted adversarial presence and action. The rapid advancement of versatile single board computers presents an opportunity to fill these coverage gaps. In order to proceed further with answering if these small devices can perform network monitoring, a macro benchmarking methodology is developed and introduced in the first half of Chapter III. The second half implements and validates the overall process with network monitoring applications.

III. Methodology

As highlighted in section 2.2.1 the embedded machine market has recently seen the addition of "edge devices" that are capable of running traditional operating systems and applications. While plenty of tools are available that can benchmark individual components like the Central Processing Unit (CPU) or Random Access Memory (RAM), no higher or *macro* level method exists to establish how an application performs end to end on a given device. This is further complicated by the trap that the limited hardware of such a device may not play well with "plug-and-play" software. This chapter proposes a methodology and automation framework to not only observe the peak performance of a task on an edge device but ensure they are appropriately configured for the task.

Section 3.1 discusses the overall design goals and their motivations while Section 3.2.2 details a specific framework that implements them. Section 3.3 then prepares the automation framework with a particular series of network monitoring tests to fully exercise and validate the developed process.

3.1 Design Goals

The following design goals drove the overall development and each is detailed below.

1. Be easy to use, modify, and scale as appropriate
2. Support a repeatable consistent baseline
3. Maximize the use of available hardware
4. Utilize a robust statistical analysis capability

Ease of Use Since edge devices move computation to the tactical or consumer level and not a central point the scale of devices can quickly grow out of hand. Man-

ually connecting and configuring each one would be a daunting task so naturally automation becomes important. Even on non edge devices enterprises have struggled to keep up with the scale demand of either large physical networks or rapidly ephemeral virtual ones [78]. A plethora of "IT Automation" tools like Chef, Ansible, and Jenkins have emerged to help deal with these problems [79]. By adopting one of these tools the scale of testing multiple devices at once becomes much more manageable. This also reduces custom or esoteric code that becomes hard to transition between efforts or projects.

Consistent Baseline An important aspect of any experiment is accounting for any nuisance or noise factors. This is especially true for testing complex systems that have multiple layers of abstraction. One small change on the top could have large unpredictable consequences on the layers and ultimate performance below. Taking a smart phone or even laptop from a few years back provides a great example. Upon release, the device performs well up to the users expectations. A year later a new Operating System (OS) version arrives with new "features" and performance of the device drops significantly. The increased load of adding a few screen transitions, maybe a transparent menu or the overhead of allowing videos to play in the background become too much for the underlying hardware which was abstracted away from the user and perhaps even the developer.

For this reason a key part of the methodology needs to ensure any modifications from a clean vendor baseline are applied consistently across devices and are either traceable (written down, scripted) or non-persistent (undone with a reboot). Some factors are unable to be accounted for and they are discussed in the Limiting Factors (Section 3.2.4).

Maximizing Hardware Utilization As seen in Section 2.2.2 simply dropping any particular tool and running it in a pure default environment will not provide a

significant indicator of peak performance. It is for that reason that this methodology needs to support applying pre-selected optimizations at varying levels to find the "best fit" for the device under test before giving a final score judgement. This can be accomplished via a multi-stage optimization loop which iteratively increases or decreases the selected optimization factors to observe at what level the system performs the best. In order to fully examine the potential optimization solution space a 2^k full factorial experiment design is needed. This allows factors to be both individually tested and tested together to observe any possible constructive or destructive interference [80].

Table 3: Example of optimization factor combinations where each letter is any possible configuration or system state. The presence of the letter indicating that particular option is "high" while absence means "low."

Factor Count	**Combinations**
2^1	A
2^2	A, B, AB
2^3	A, B, AB, C, AC, BC, ABC
2^4	A, B, AB, C, AC, BC, ABC D, AD, BD, ABD, CD, ACD, BCD, ABCD
2^5	A, B, AB, C, AC, BC, ABC D, AD ,BD, ABD, CD, ACD, BCD, ABCD E, AE, BE, ABE, CE, ACE, BCE, ABCE, DE, ADE, BDE, ABDE, CDE, ACDE, BCDE, ABCDE

Robust Analysis Analyzing the results from the multi stage optimization loop can be performed by any array of tests from simple mean comparisons to full blown statistical tests. Ideally a statistical test like Analysis of Variance (ANOVA) can precisely determine if the optimizations chosen are actually having a desired impact and not just due to random noise. The usefulness of the ANOVA test depends heavily

on assumptions of normality however. In cases where normality appears violated a secondary test should be supported that is able to continue testing until no further improvement is observed. In all cases the definition of what improvement is should be user definable.

3.2 General Test Methodology

Considering the goals, Figure 12 below presents a visual of the proposed methodology. Working from the inside most process out (or bottom up according to the figure, this section lays out a specific end to end implementation. This specific implementation is also visually presented later in Figure 14.

3.2.1 Analysis Design

Two separate tests have been implemented to satisfy the robust analysis requirement. First, a ANOVA test is implemented that can analyze up to $k = 5$ factors of a 2^k Factorial Factorial design. A backup test is then implemented around the same lines that performs a much simpler sample mean comparison. The results of either one of these tests is used to inform the feedback loop of the multi-stage optimization process.

3.2.1.1 ANOVA

While a multitude of commercial software like JMP, Minitab, and Design-Expert exist that performs advanced statistical tests they can be expensive and/or difficult to integrate [80]. Following the process outlined in Chapters 3,5, and 6 in Montgomery's *Design and Analysis of Experiments (9th Edition)* [80] we can re-create the process needed to perform an ANOVA test on a multi-factor experiment like this. The Python code listed below in Snippet 1 implements these steps to create a table like Table 4.

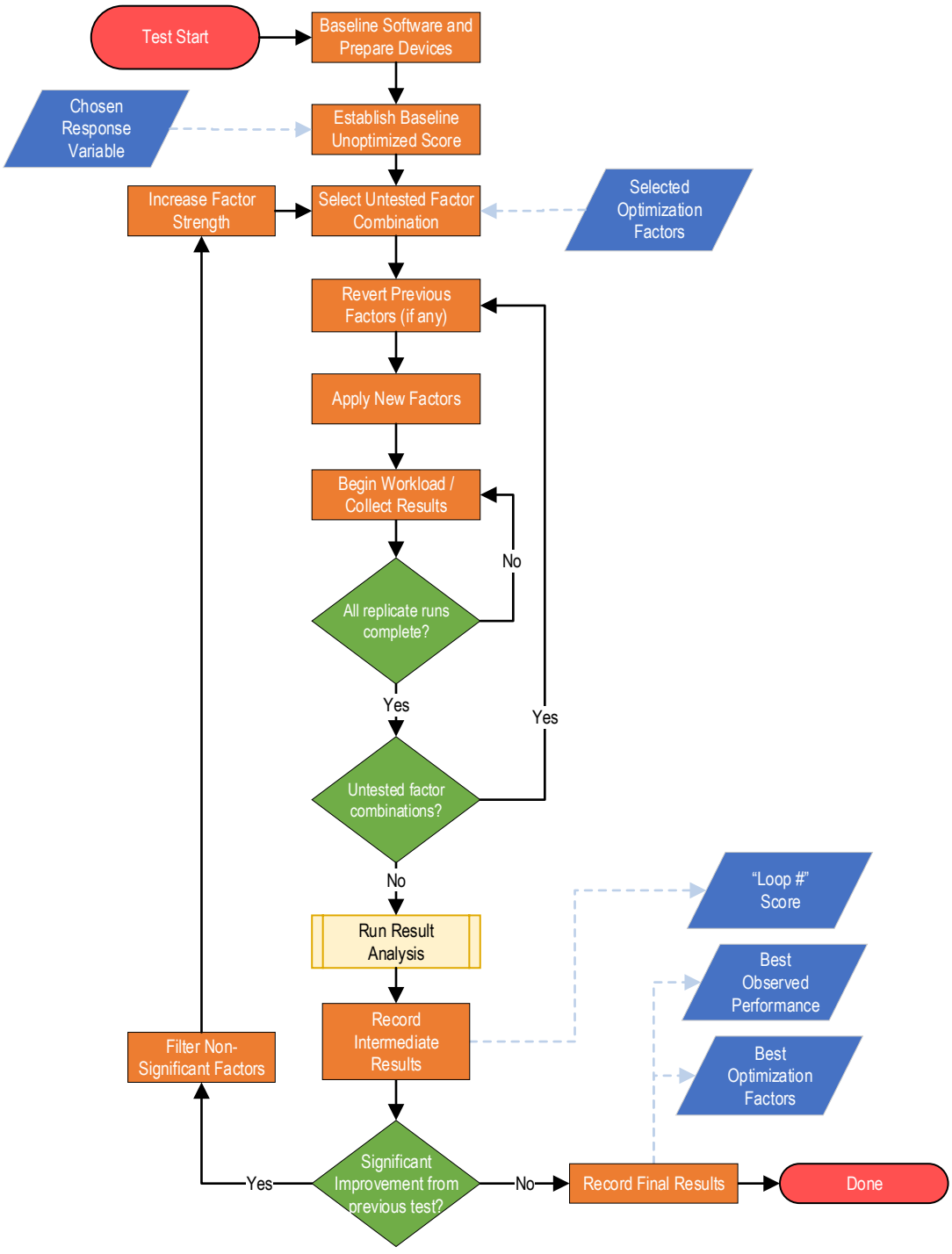


Figure 12: High level flow diagram of the proposed testing methodology

Table 4: Example ANOVA Table for 3-Factor Test [80]

Factor Variation	**Sum of Square**	**Degrees of Freedom**	**Mean Squares**	F_0
A	SS_A	$a-1$	MS_A	$F_0 = \frac{MS_A}{MS_{Error}}$
B	SS_B	$b-1$	MS_B	$F_0 = \frac{MS_B}{MS_{Error}}$
C	SS_C	$c-1$	MS_C	$F_0 = \frac{MS_C}{MS_{Error}}$
AB	SS_{AB}	$(a-1)(b-1)$	MS_{AB}	$F_0 = \frac{MS_{AB}}{MS_{Error}}$
AC	SS_{AC}	$(a-1)(c-1)$	MS_{AC}	$F_0 = \frac{MS_{AC}}{MS_{Error}}$
BC	SS_{BC}	$(b-1)(c-1)$	MS_{BC}	$F_0 = \frac{MS_{BC}}{MS_{Error}}$
ABC	SS_{ABC}	$(a-1)(b-1)(c-1)$	MS_{ABC}	$F_0 = \frac{MS_{ABC}}{MS_{Error}}$
Error	SS_{Error}	$abc(n-1)$	MS_{Error}	
Total	SS_{Total}	$abcn-1$		

Where

$$a, b, c = \text{number of levels for each main factor} \tag{1}$$

$$n = \text{the number of samples} \tag{2}$$

$$SS_X = \frac{(Contrast_X)^2}{n * 2^k} \tag{3}$$

$$Contrast_{AB\cdots K} = (a \pm 1)(a \pm 1)\cdots(k \pm 1) \tag{4}$$

$$MS_X = \frac{SS_X}{x-1} \tag{5}$$

$$SS_{Total} = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{c}\sum_{l=1}^{n} y_{ijkl}^2 - \frac{y_{....}^2}{abcn} \tag{6}$$

$$SS_{Error} = SS_{Total} - SS_{Factors} \tag{7}$$

Snippet 1 Sample of the 3 factor code from *anova.py* that builds an ANOVA table. Full code is available in Appendix D

```
...
if(k >= 3):
 df_index=['A', 'B', 'AB', 'C', 'AC', 'BC', 'ABC', 'Error', 'Total']
 c  = response_var[response_var['code'] == 'C'].loc[:,rv[1]].to_numpy()
 ac  = response_var[response_var['code'] == 'AC'].loc[:,rv[1]].to_numpy()
 bc  = response_var[response_var['code'] == 'BC'].loc[:,rv[1]].to_numpy()
 abc = response_var[response_var['code'] == 'ABC'].loc[:,rv[1]].to_numpy()
 means_all = np.array([np.mean(a), np.mean(b), np.mean(ab), np.mean(c),
  ↪  np.mean(ac), np.mean(bc), np.mean(abc)])
 total = np.array([one, a, b, ab, c, ac, bc, abc])
 contrast_A = np.sum(-one + a - b + ab - c + ac - bc + abc)
 contrast_B = np.sum(-one - a + b + ab - c - ac + bc + abc)
 contrast_AB = np.sum(one - a - b + ab + c - ac - bc + abc)
 contrast_C = np.sum(-one - a - b - ab + c + ac + bc + abc)
 contrast_AC = np.sum(one - a + b - ab - c + ac - bc + abc)
 contrast_BC = np.sum(one + a - b - ab - c - ac + bc + abc)
 contrast_ABC= np.sum(-one+ a + b - ab + c - ac - bc + abc)
 contrasts_all = np.array([contrast_A, contrast_B, contrast_AB, contrast_C,
  ↪  contrast_AC, contrast_BC, contrast_ABC])
...
# Sum Squares
num_effects = np.power(2,k)-1
num_elements = num_effects+2
sum_squares = np.ones(num_elements) #All effects plus error and total
for i in range(num_effects):
 sum_squares[i] = np.square(contrasts_all[i])/(n*np.power(2,k))
total_mean = np.mean(total)
SST = np.sum(np.square(total - total_mean))
SSE = SST - np.sum(sum_squares[0:num_effects])
sum_squares[num_effects] = SSE
sum_squares[num_effects+1] = SST

#Build datafile
for i in range(num_effects):
    F0 = mean_squares[i]/MSE
    f_vals[i] = F0
    f_crits[i] = stats.f.ppf(1-alpha,DF[i],DF[num_effects])
    p_vals[i] = 1 - stats.f.cdf(F0, DF[i],DF[num_effects])
    effects[i] = contrasts_all[i]/(n*np.power(2,k-1))
    means[i] = means_all[i]
anova_df_numpy = np.array([means, effects, sum_squares, DF, mean_squares, f_vals,
 ↪  f_crits, p_vals])
```

This script calculates contrasts, estimated effects, f-statistics and p-values for up to five factors. Using this information it filters only the factors that had the desired impact and returns them in an array to be re-evaluated at even stronger levels. Should there not be any "statistically" significant effects (defined as a p-value less then the chosen confidence level) but perhaps some strong performers with a "good" sample mean the secondary mode is triggered and they are returned instead.

3.2.1.2 Secondary Test

The secondary results analysis test is simply a "trial and error" mode that compares sample means and continues testing until no further improvement is observed. Improvement again is definable by the researcher as either a higher or lower mean compared to the previous one. In addition due to development time, this "best mean" heuristic mode is the default analysis used beyond the first iteration of the multi stage optimization loop. Future work could easily modify the Python script to perform a different kind of test or handle shrinking factor combinations to re-apply the ANOVA test as appropriate.

Snippet 2 Sample from *best-mean.py* that determines if the re-run factors of a middle loop beat the previous best. This function is only called on loops beyond the first one

```
print("Re-run factor means")
print(response_var.groupby('code')[rv[1]].mean())

print("Lowest observed sample mean (target to beat)")
print(response_var.groupby('code')[rv[1]].mean().min())

#print factors still remaining as viable
candidiate_factors_index =
 ↪  response_var.groupby('code')[rv[1]].mean().index.array.to_numpy() #all
 ↪  factors from csv
improved_factors_bools = (response_var.groupby('code')[rv[1]].mean() <
 ↪  target_to_beat).to_numpy() #boolean series
all = ""
i=0
for y in candidiate_factors_index:
 if improved_factors_bools[i]:
  all = all + y + ","
 i=i+1
print("Effects")
if len(all) == 0:
 print("NONE") #No more improvements...
 exit()
print(all.rstrip(','))
```

3.2.2 Automation Design

Of the many possible automation frameworks available, Ansible was chosen mainly because of its use in Air Force cyber weapon systems [81], low overhead for deployment

(no databases, agents, servers, or daemons) and ability to meet all of the design goals mentioned previously [8]. The only thing required on the edge device is an operating system, some form of transit like Secure Shell (SSH) and Python.

A single control workstation with the Ansible package installed is used to launch all the commands, which are documented in human readable Yet Another Markup Language (YAML) based "playbook" files. Inside the playbook file is a series of serial tasks that run on either the remote devices or the control machine itself (shown in Figure 13 below). A task is essentially a single call to one of the many possible community supported modules which can be anything from a shell command, git request, docker pull, cloud provider batch job, or firewall modification. While the tasks are run in serial order from the file, each machine is issued the tasks in parallel. This is particularly useful if trying to test a swarm of edge devices that must cooperate together or when trying to reduce confounding factors like room ambient temperate when a test started.

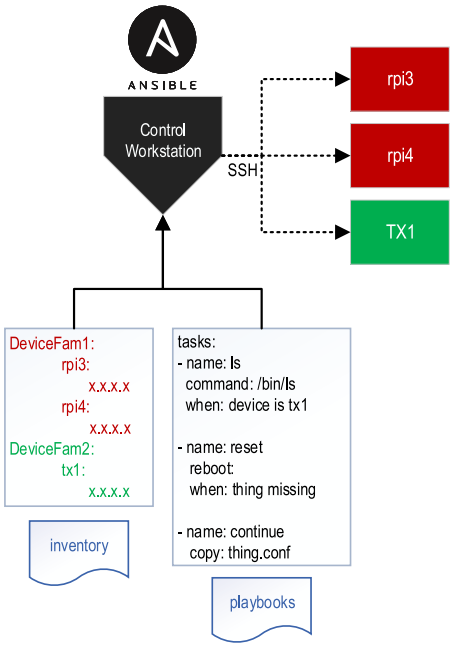


Figure 13: Example of core Ansible files (the inventory and a YAML playbook) and how it communicates with external devices

As with many of its class of tools, it allows an administrator or in this case a researcher to carefully define every variable and step taken in a repeatable step by step process. It also supports dynamic input and conditionals which can be captured to provide a repeatable but evolving use case as seen later in Figure 14. Code snippet 3 demonstrates how conditional statements can apply to either certain devices or broader environment conditions. The first two tasks make changes that only apply to Raspberry Pi boards and the latter checks how much free memory each board has so it can build an appropriate buffer size for libpcap.

Snippet 3 Example of conditonal tasks in a playbook. The first two tasks make changes that only apply to Raspberry Pi boards and the latter checks how much free memory each board has so it can build an appropriate buffer size for libpcap

```
  #When Disk I/O is very important. Also lifetime of flash...
- name: Disable swap on RPis
  shell: swapoff -a
  become: yes
  when: "'nvidia' not in group_names"

- name: Bump RPi Throttling Temp (3B+ only)
  lineinfile:
    path: /boot/config.txt
    regex: "temp_soft_limit="
    line: temp_soft_limit=70.0
  when: "'nvidia' not in group_names"
  become: yes

  #If device has less then 2GB free,
  #Take 70\% of it, mb to KiB
- name: Limit buffer_size for libpcap
  set_fact:
    buffer_size: "{{ 700 * ansible_facts['memory_mb']['nocache']['free']|int }}"
  when:  " ansible_facts['memory_mb']['nocache']['free'] < 2048"
  changed_when: false
```

The next few subsections discuss Ansible's input file structure and how it ties to the larger picture.

3.2.2.1 Device Specific Variable Storage

The inventory file stores each device under test's key variables and potential hardware selections. While the example shown below is pre-built with variables that matter for a network test, just about anything can be stored in the key : value pair. This file closely relates to *vars.yml* in section 3.2.2.2 but contains device specific entries. Each machine is identified by its hostname which must either resolve via a Domain Name Service (DNS) or a local hosts file. Following best practice, credentials for connecting to these devices are not stored here although they could be. This is discussed later in the baselining section 3.2.2.5. The flexibility provided here allows devices of different shapes and sizes to still be tested at the same time, or perhaps the same type of device to be tested in varying ways.

Snippet 4 An example of the inventory.yml file which contains the hostnames and specific variables that each device or group of devices may have

```
sensors:
        children:
            rpi:
                hosts:
                    rpi3bp:
                        send_interface: eth7
                        capture_interface: eth0
                        interface_pps_limit: 70000
                        rps_mask: 0
                        NAPI_budget_best: 300
                    rpi4:
                        ...
                vars:
                    ansible_user: pi
                    ansible_become_method: sudo
                    sensor_dir: /sensor
            nvidia:
                hosts:
                    tx2:
                        capture_interface: eth0
                        send_interface: eth3
                        rps_mask: 3E #0011 1110
                        NAPI_budget_best: 300
                        backlog_best: 1000
                        backlog_weight_best: 300

                vars:
                    ansible_user: nvidia
                    ...
```

3.2.2.2 Global Variable Storage

The vars file is the primary "one stop shop" file to tweak any of the global experiment controls. As an important distinction, this file does not contain any tasks but just key:value pairs that are referenced later in the *static_controls.yml* and *variable_controls.yml* playbooks. Below in snippet 5 three main examples are highlighted. Lines 2 to 5 are the primary setup variables used in the ANOVA portion of the experiment. Lines 12, 13 are two global toggles for these particular system options and lines 17 and 20 are two of the potential optimization factors chosen for the experiment. These global variables provide the backbone for experiment control and baselining discussed in section 3.1.

Snippet 5 An example of the vars.yml file which contains general, non device specific variables for testing

```
###Master variable file for all playbooks
total_factors: 5
total_combinations: 32 #2^5
replicates: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] #must be an array
factor_combos: [N,A,B,AB,C,AC,BC,ABC,
                D,AD,BD,ABD,CD,ACD,BCD,ABCD,
                E,AE,BE,ABE,CE,ACE,BCE,ABCE,
                DE,ADE,BDE,ABDE,CDE,ACDE,BCDE,ABCDE]

#Large Receive / Generic Receive offload on/off.
#Off for suricata
lro_status: 'off'
gro_status: 'off'

#Factor D
#Receive Flow Steering (RFS) hash table size off / large
rfs_table: [ 0, 262144 ]
#Factor B
#Max Kernel Backlog default / large
backlog: [ 1000 , 65536 ]
```

3.2.2.3 Static Test Controls

As the name implies this playbook contains all actual tasks that ensure certain options are set every single run of the test. The two examples shown in snippet 6 setup the appropriate capture_interface (device unique, pulled from inventory.yml) offload options (global, pulled from vars.yml). Generically speaking, any system level configuration that can be influenced over a command line can be placed into this playbook like disabling a service or tweaking a CPU fan response profile.

Snippet 6 Example of tasks from the static_controls.yml playbook that are run every new iteration of a test, ensuring a consistent baseline

```
- name: Set Receive Offloads
  command: "ethtool -K {{ capture_interface }} lro {{ lro_status }} gro {{ gro_status }}"
  become: yes

- name: Enable Capture Interface and Set Promiscuous
  shell: |
    ifconfig {{ capture_interface }} promisc
    ifconfig {{ capture_interface }} up
  become: yes
```

3.2.2.4 Variable / Factor Test Controls

The variable control playbook is similar to the static playbook with the exception that not every task runs every time. By utilizing the conditional task and loop mechanism of Ansible the tasks will run only when their letter is detected in the current loop/array variable "factor_combos" (from vars.yml). Further exploiting this loop functionality allows us to incrementally increase the value passed in subsequent runs. A key aspect of this design is that these configuration changes do not persist upon a reboot. This allows each new factor combination run a "clean slate" to work from.

Snippet 7 Example of tasks from the variable_controls.yml playbook that are only run based on the current factors under test

```
  ###FACTOR B###
- name: (Factor B) Set Kernel Max Backlog to {{backlog[1]|int*test_counter|int}}
  shell: sysctl -w net.core.netdev_max_backlog={{backlog[1]|int*test_counter|int}}
  become: yes
  when: "'B' in current_factor_list"
  become: yes

  ###FACTOR D###
- name: (Factor D) Set Receive Flow Steering (RFS) Table Size to
 ↪  {{rfs_table[1]|int*test_counter|int}}
  shell: |
    sysctl -w net.core.rps_sock_flow_entries="{{ rfs_table[1]|int*test_counter|int
     ↪  }}"
    echo "{{ rfs_flow_cnt }}" >
     ↪  /sys/class/net/{{capture_interface}}/queues/rx-0/rps_flow_cnt
  become: yes
  ignore_errors: yes
  when: "'D' in current_factor_list"
```

3.2.2.5 New Device Preparation and Baseline

As shown static and variable control playbooks is it trivial to apply changes repeatedly however for sake of time some of these only need applied or checked once. These steps may include setting SSH keys, installing certain dependencies, copying large datasets, compiling tests or creating directories. For repeatability sake it is im-

portant to track these one-time changes, which is accomplished in *prep-playbook.yml*. Examples of these tasks are provided in snippet 8.

Snippet 8 Example of tasks from the prep_playbook.yml playbook that are only run once per device. Ensures dependencies and other important settings like SSH keys are set

```
  - name: Set SSH Keys
    authorized_key:
        user: "{{ansible_user}}"
        state: present
        key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
    tags: auth

  - name: Create Sensor Directory
    file:
      path: "{{ sensor_dir }}"
      state: directory
      owner: "{{ ansible_user }}"
      group: "{{ ansible_user }}"
      mode: '0777'
    become: yes

  - name: Install Prerequisites (may take awhile)
    when: ansible_facts['os_family'] == "Debian"
    apt:
      name: "{{ packages }}"
      force_apt_get: yes
    vars:
        packages:
        - build-essential
        - libpcap-dev
        - dnet-common
        - libdumbnet-dev
        - libdnet
        - libevent1-dev
        - libdnet-dev
        - libdumbnet1
        - nano
        - locate
        - docker.io
    ignore_errors: yes
    become: yes
```

3.2.3 Multi-Stage Optimization Design

As highlighted in section 3.2.2 through the use of a few basic variables, loops and conditionals Ansible provides an excellent way to implement a multi-stage 2^k full factorial optimization loop. Figure 14 visually represents this in the context of Ansible. The output of this process is twofold, it not only provides a view of

the performance of a given task at particular factor levels but also extrapolates the most significant among them. The latter is important as it can become arduous and subjective if the changing of a few settings is indeed helping or hindering performance.

A small quirk of Ansible is that the loop: keyword can only apply to one "task." Luckily there is a built in task that allows entire other playbooks to be inserted into the current one, and the loop: keyword still applies. This workaround is shown in snippet 9. As a consequence however a total of 4 YAML files are needed due to the workflow containing essentially three loops. A traditional playbook makes up the first file and calls the outer loop. The outermost loop is technically a recursive function that performs the ANOVA/heuristic testing while the middle loop is for the current factor combination (i.e. ABC) and the inner most loop for repeat or tests of the same factor (i.e. ABC x5). Each one of these playbooks is laid on in the subsections that follow.

Snippet 9 Example of introducing looping mechanic in Ansible. Used multiple times as elaborated below

```
- name: Begin Primary Test Control
  include_tasks: general-control.yml
  loop: "{{ factor_combos }}" #This will run 2^(#factors) times
  loop_control:
   loop_var: current_factor_list
   index_var: factor_idx
   extended: yes
  when: "test_counter == 1"
  tags:
    - discover
    - initial
```

3.2.3.1 Main Playbook

This playbook is akin to the "main" function of any other program. It initially calls the recursive outer loop *general-benchmark-outerloop.yml*. Upon its return, it finalizes the results to file and displays them.

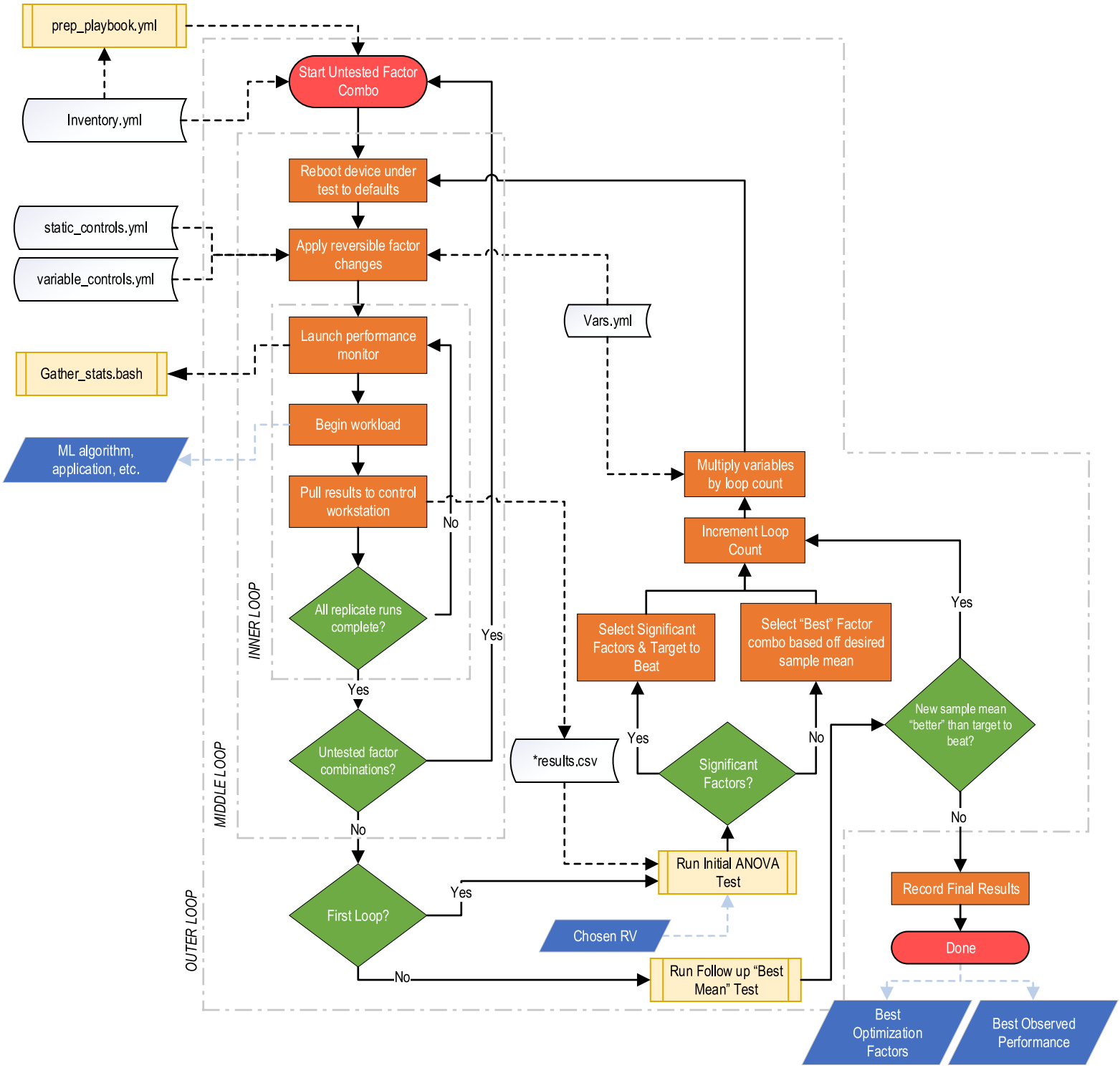


Figure 14: Automation workflow diagram for the multi stage optimization loop, with context of Ansible input files and output files

3.2.3.2 Outer Loop Playbook

The outermost loop begins by calling the middle loop with all possible 2^k factors after which the initial ANOVA test is performed for factor significance. It then continuities to call the middle loop with a reduced sample space of either statistically significant factors or best observed factors. The recursive base case checks if the response variable has continued to improve and if not, terminates calling itself and returns to main. Another quirk of Ansible (though likely by design) is that there is no built in way to perform this sort of recursive call. By slightly abusing the exception handler however we can "infinitely" recursively re-call the same playbook while still maintaining the overall test counter. This is demonstrated in snippet 10.

Snippet 10 Recursive call of *general-benchmark-outerloop.yml* keeps calling itself to perform tests if the previous middle loop's test performed even better then the last

```
- name: Last Loop Results
  debug:
    msg: "Last Loop best was {{last_loop_best}}. Needs to beat {{target_to_beat}}. Was Iteration {{

- name: Check Last Loop Results
  block:
    - name: Check Recursive Base Case
      fail:
        #Maybe run a set number of times....
        msg: "Continuing {{test_counter}} < 4"
      when: "test_counter | int < 4"

       #Or have a target
        #msg: "Still room to improve {{last_loop_best}} < {{target_to_beat}}"
      #when: "last_loop_best < target_to_beat"

  rescue:
    - name: Update Target to Beat
      set_fact:
         target_to_beat: "{{ last_loop_best }}"
         loop_multiplier: "{{ test_counter|int**test_counter|int }}"

    #Recursively call self to keep going
    - name: Begin New Round
      include_tasks: general-benchmark-outerloop.yml
```

3.2.3.3 Middle Loop Playbook

This playbook acts as the "middle" loop of the overall process. It begins by rebooting each device under test to revert the temporary changes of the last iteration. Although not implemented in this version, reverting non-temporary changes could be performed right after the reboot as well. After waiting for the reboot it applies the next set of controls and begins the inner loop *general-benchmark-innerloop.yml* n repeat times. The repeat tests increase the power of the 2^k full factorial test. Upon completion of all the repeat tests, a call to the analysis subprocess finds the latest response variable's best arithmetic mean and stores it for later comparison in the recursive playbook.

3.2.3.4 Inner Loop Playbook

Lastly the "inner" loop is responsible for the actual workload task in question. Prior to beginning the actual workload, an asynchronous call to the metric gathering subprocess described in Section 3.2.3.5 begins collecting data on the device under test. The actual workload itself can be anything from training a machine learning model to a robotic control simulation. Upon completion of each test, the performance monitor records it's data which is copied back to the control workstation in an organized fashion. The inclusion of all the *failed_when* and *ignore_errors* keys in this and other files is to prevent the entire chain from collapsing should one process fail to run to completion for whatever reason.

Snippet 11 The middle loop playbook is implemented in *general-benchmark-middleloop.yml* and is responsible to reset, reconfigure, and prepare for each individual factor combination test and it's repeats performed in the inner loop

```
#This playbook is the "middle" loop
- name: Reboot to Defaults. Beginning Factor {{current_factor_list}} ({{ansible_loop.index}} of {{ansible_loop.length}})
  reboot:
  become: yes
  tags: skippable

  #If any configuration changes are not undone with a reboot,
  #add a playbook here to manually "revert" them

- name: Set Static Controls
  include_tasks: general-static-controls.yml

- name: Set Variable Factor Controls
  include_tasks: general-variable-controls.yml

  #Run repeats
- name: Begin Inner Loop
  include_tasks: general-benchmark-innerloop.yml
  loop: "{{ replicates }}"
  loop_control:
    extended: yes
    loop_var: inner_counter
    index_var: inner_idx
  tags: workload

- name: End of Run Best Mean Test {{test_counter}}
  local_action:
    module: shell
    _raw_params: |
        python best-mean.py 'results/{{ inventory_hostname }}-results-run{{test_counter}}.csv' <<**RESPONSE VARIABLE**>> "{{target_to_beat}}"
  register: anova
  tags: anova
  #if the last item in loop and not inital run
  when: "ansible_loop.revindex == 1 and test_counter > 1"
  changed_when: false
  ignore_errors: yes

- name: Update Last Middle Loop Best
  set_fact:
   last_loop_best: "{{anova.stdout_lines[anova.stdout_lines|length -3]|int}}"
   significant_factors_array: "{{anova.stdout_lines[anova.stdout_lines|length -1].split(\",\")}}"
   significant_factors_string: "{{anova.stdout_lines[anova.stdout_lines|length -1]}}"
   significant_factors_history: "{{significant_factors_history}} + [ '{{anova.stdout_lines[anova.stdout_lines|length -1]}}' ]"
   last_loop_best_history: "{{last_loop_best_history}} + [ '{{anova.stdout_lines[anova.stdout_lines|length -3]|int}}' ]"
  when: "ansible_loop.revindex == 1 and test_counter > 1"
```

Snippet 12 The inner loop playbook implemented in *general-benchmark-innerloop.yml* launches the actual desired workload and performance monitoring subsystem. This is repeated however many times are desired to repeat tests

```
- name: Launch Performance Monitor (Factors {{ current_factor_list }})
    shell: "./gather_stats.bash <<**PID**>> <<**SAMPLE RATE**>> {{ current_factor_list }}"
    args:
      chdir: "{{ experiment_dir }}/"
    register: results_async
    poll: 0
    async: 3600
    become: yes
    changed_when: false

   # <<**YOUR WORKLOAD TASK(S) GOES HERE.....**>>
   #       SEE RATELIMIT TEST FOR EXAMPLE

  - name: Stop Everything
    shell: kill "$(cat gather.pid)"
    args:
      chdir: "{{ experiment_dir }}"
    become: yes
    ignore_errors: yes
    failed_when: false

  - name: Wait for Results
    async_status: jid="{{ results_async.ansible_job_id }}"
    become: yes
    register: results
    until: results.finished
    retries: 30
    failed_when: false
    ignore_errors: yes

  - name: Copy Results
    fetch:
      src: "{{experiment_dir}}/{{ inventory_hostname }}-results-verbose.csv"
      dest: "results/{{ inventory_hostname }}-results-run{{test_counter}}-verbose.csv"
      flat: yes
    changed_when: false
```

3.2.3.5 Metric Gathering Subprocess

In order to collect the necessary performance metrics to perform any sort of analysis a method was needed to poll the necessary sensors and software counters. Since many of the desired software counters were stored in the kernel's *proc* file system reading them straight from the command line was simple and efficient enough. Since bash is the default shell for all the devices tested in Table 2 a shell script was written to perform these collections. This comprehensive script was written to gather the desired statistics per a set interval then perform some basic calculations like the

arithmetic mean before cleanly terminating it's results to a file. Table 5 displays the currently supported collections which could easily be expanded in future work to include things like total runtime, input/output wait time, disk utilization or Graphics Processing Unit (GPU) usage.

CPU	$PID\%_{\mu}$	$PID\%_{max}$	$system\%_{\mu}$	$system\%_{max}$	
Memory (MB)	$PID\%_{\mu}$	$PID\%_{max}$	PID_{max}	$system\%_{min}$	$system_{min}$
Temperature (C)	CPU_{μ}	CPU_{max}			
Power (W)	CPU_{μ}	CPU_{max}			
Network (HW)	$rxpps_{\mu}$	$rxpps_{max}$	$nicdrop_{\sum}$	$nicdrop_{\mu}$	nicdrop%
Network (SW)	$rxbps_{\mu}$	$rxbps_{max}$	$kerndrop_{\sum}$	$kerndrop_{\mu}$	

Table 5: Supported Response Variables

Process ID (PID) is the identifier of the main workload processes (all threads) where "system" is a conglomerate of the *user* (regular processes), *system* (kernel threads), *niced* (high/low priority process), *wa* (IO waiting), *hi* (servicing hardware interrupts) and *si* (servicing software interrupts) CPU timers. The power variable currently depends on the board family. The Raspberry Pi boards do not have a built in power senor so a power state is returned instead. The NVIDIA boards tested support on-die power for the CPU, GPU and carrier board though the latter two were omitted for development time. Since the script is launched in parallel from Ansible it doesn't necessarily know which board it landed on which matters for certain sensors. Therefore as part of the startup sequence (snippet 13) it detects the proper device family and adjusts accordingly.

Snippet 13 The *gather_stats.bash* subprocess automatically detects which device it is running on to reference the correct sensors. This simplifies the Ansible task to a single unified one instead of N devices unique ones

```
1  if [ $(sudo lshw -short -c system | awk 'FNR == 3 {print $2}') == 'Raspberry' ];
    ↪  then DEVICE_FAM=pi;
2  elif [ $(sudo lshw -short -c system | awk 'FNR == 3 {print $2}') == 'Jetson-TX1'
    ↪  ]; then DEVICE_FAM=nvidia-tx1;
3  ...
4  if [ "$DEVICE_FAM" == 'pi' ]; then
5    TEMPERATURE_CPU[$LOOP_COUNT]=$(vcgencmd measure_temp | grep -ow
      ↪  "[0-9][0-9].[0-9]")
6    POWER_CPU[$LOOP_COUNT]=\$(expr $(vcgencmd measure_clock arm | grep -oP
      ↪  "([0-9]+)" | tail -1)/1000000)
7
8  elif [ "$DEVICE_FAM" == 'nvidia-tx1' ]; then
9    TEMPERATURE_CPU[$LOOP_COUNT]=$(bc <<< 'scale=1; '$(cat
      ↪  /sys/devices/virtual/thermal/thermal_zone1/temp)'/1000')
10   POWER_CPU[$LOOP_COUNT]=$(bc <<< 'scale=3; '$(cat
      ↪  /sys/devices/7000c400.i2c/i2c-1/1-0040/iio_device/in_power0_input)'/1000')
11
12 elif [ "$DEVICE_FAM" == 'nvidia-tx2' ]; then
13   TEMPERATURE_CPU[$LOOP_COUNT]=$(bc <<< 'scale=1; '$(cat
      ↪  /sys/devices/virtual/thermal/thermal_zone1/temp)' / 1000')
14   POWER_CPU[$LOOP_COUNT]=$(bc <<< 'scale=3; '$(cat
      ↪  /sys/bus/i2c/drivers/ina3221x/0-0041/iio_device/in_power0_input)'/1000')
```

Another key component of this script is to adapt to a heavily loaded system. Since any of the "per second" samples are heavily reliant on a set interval the script will keep track of the wall clock time between passes and factors it in when doing "per second" calculations. This is demonstrated on line 16 in code block 14 below.

The bash `time` keyword returns the "wall clock" or "real" time spent in the sub-shell contained within the brackets. Inside the brackets are a call to `sleep` and a asynchronous call to `captureLap` which actually polls all the sensors. Meanwhile the builtin `wait` holds execution of the main thread until the sleep has finished. The resulting real time is returned and shuffled around with file descriptors before being saved to a file for later reference.

Snippet 14 Some samples obtained in *gather_stats.bash* are timing dependant ("per second") so it uses some bash built in mechanisms to adjust for heavy load drift

```
captureLap{
    ...
    #Time sensitive samples
    RX_PKTS_NOW=$(cat /sys/class/net/$IFACE/statistics/rx_packets)
    RXPPS[$LOOP_COUNT]=$(bc<<<"scale=0;($RX_PKTS_NOW-$RX_PKTS_LAST)/($LOOP_TIME_REAL)")
    RX_PKTS_LAST=$RX_PKTS_NOW
    ...
}
...
exec 3>&1 4>&2 #Preserve the original stdout/stderr
while [[ -d /proc/$PID  ]]
do
   #This needs to be as close as possible to SAMPLE_RATE sec for
   #"per second" calculations to be accurate
   #As system load nears 100% the loop will likely drift, so try to account for it.
   { time { sleep $SAMPLE_RATE & captureLap 1>&3 2>&4; wait $!; } } 2>"$tmp/lastloop"
   LOOP_TIME_REAL=$(cat $tmp/lastloop)
done
```

By default the sample interval is set to one second though it is adjustable. This is based off a recommendation from NVIDIA as reading their internal sensors too frequently will incur excessive amount of power consumption as it still utilizes internal CPU resources [68].

In order to save development time some metrics like CPU usage and memory were obtained from other processes like *ps* and *top*. Since these pre-built applications already query the same built-in kernel data structures and counters, creating anything custom would just be re-implementing an existing product. CPU utilization however proved a little more elusive as the definition seems to flex based upon on the tool used. the common *ps* command for example calculates things a little bit differently then *top* (from ps man page):

> *CPU usage is currently expressed as the percentage of time spent running during the entire lifetime of a process. This is not ideal, and it does not conform to the standards that ps otherwise conforms to. CPU usage is unlikely to add up to exactly 100%.*

Whereas commands like *top* provide a more expected behavior with one caveat that it must be running to begin calculating:

%CPU – CPU Usage
The task's share of the elapsed CPU time since the last screen update, expressed as a percentage of total CPU time.

While future work could likely remove this subprocess altogether by manually calculating the metric from */proc/pid/stat*, a background top process provides the core functionality needed.

At the end of each main loop, the results generated from the *gather_stats.bash* script are compiled and fed into the analysis process discussed earlier in Section 3.2.1. Using this information it selects the factors that had the best desired impact and returns them in an array for Ansible to re-evaluate at even stronger levels. In addition the rolling "target to beat" is established here. This best performing factor result is fed into the next loop as the base case to continue or end the recursion. If the next loop performs better then this score, increasing the factor levels even more had a good impact. If the next loop performs worse, increasing the factor levels had a detrimental affect and the entire workflow will end. Likewise If all the attempted factors had a "bad" sample mean after the first test compared to the intersection or "unoptimized" case, nothing is returned ending the entire process. Again the desire or "good" and "bad" in this case is definable by the researcher by switching the greater or less than operators.

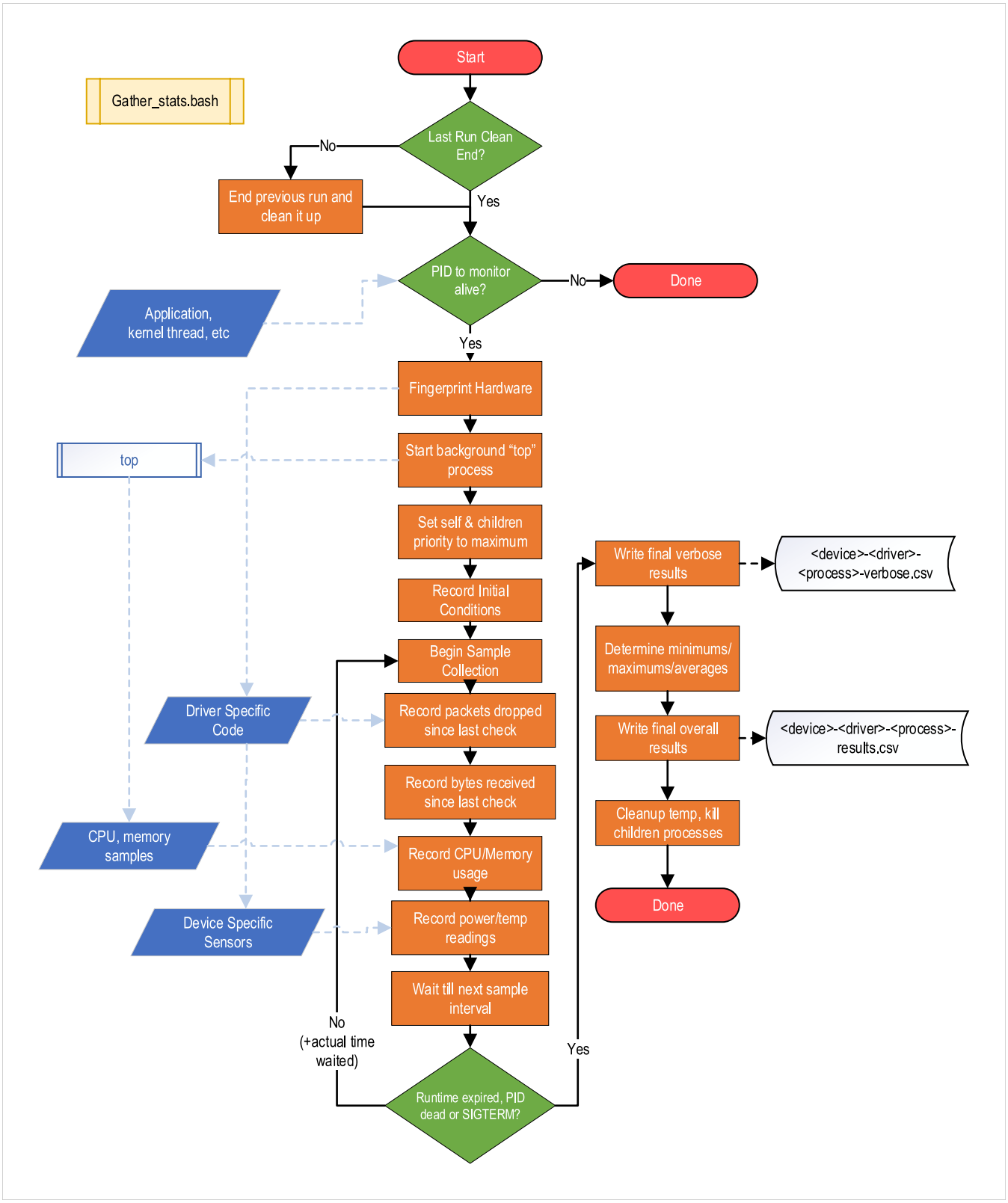


Figure 15: High level metric gathering script (gather-stats.bash) workflow diagram

3.2.4 Limiting Factors

Unfortunately, some confounding factors such as power quality, ambient environmental conditions, or engineering quality will be not controlled. The Raspberry Pi family of boards are much more susceptible to these nuisance factors and begin throttling rather early in some cases [69].

In addition the ANOVA test does not automatically implement any procedural normality tests like Kolmogroov-Smirnov or Shapiro-Wilk. The normality was instead validated with graphical methods like normplots and histograms as demonstrated in Section 4.1.3.

3.3 Specific Application: Edge Network Sensors

We use the automated test framework described in Section 3.2 to execute three end-to-end experiments. Each test consists of a unique network related workload with a combination of unique and shared input variables across five different devices from two hardware vendors. These selected applications not only show the validity of the methodology but also help answer the trade-offs from miniaturizing traditional intrusion sensor roles to edge devices. The first test is a hardware only test that establishes the hardware limit of the Network Interface Card (NIC), a prerequisite of sorts for the next two tests. If the NIC itself is failing to keep up with real-time demand, then the application cannot be expected to perform any better.

The sensor roles selected for examination are raw traffic capturing with *tcpdump* and signature based Intrusion Detection System (IDS) detection. *Suricata* is the particular IDS of choice for this test, chiefly due to its inclusion into the Air Force's Cyberspace Vulnerability Assessment / Hunter (CVA/H) network monitoring platform but also for its ability to scale across CPU cores. The specifics for these tests along with reasoning behind certain decisions and the final results are detailed in

Chapter IV.

3.3.1 Experiment Architecture

Figure 16 below outlines the overall end-to-end test architecture using devices from Table 2. In order to properly simulate external network traffic being passed to these devices an external generation source was needed. Luckily multiple open source projects exist that can turn a moderately powered desktop workstation into a 10 gigabit traffic generator given the proper network interface. For purposes of this test however a multi port 1 gigabit network adapter suited the need. Section 3.3.2 next discusses this further.

An "out-of-band" ad-hoc 802.11 network allows all of the single board computers to create a mesh to communicate among themselves and the operator workstation. Wireless chips based on this standard are ubiquitous in single board computers and are usually part of the System on a Chip (SOC) die itself. While not specifically tested in this research some interesting scenarios for leveraging a mesh network of sensors like this is discussed in chapter V (Future Work). This out-of-band connection is used to relay commands and reporting back to either an operator workstation or larger collection server. This also allows the sensors to appear transparently on the target network as only passive listeners, an important fact when hunting a Advanced Persistent Threat (APT) or perhaps even when trying to get approval just to connect it to a enterprise.

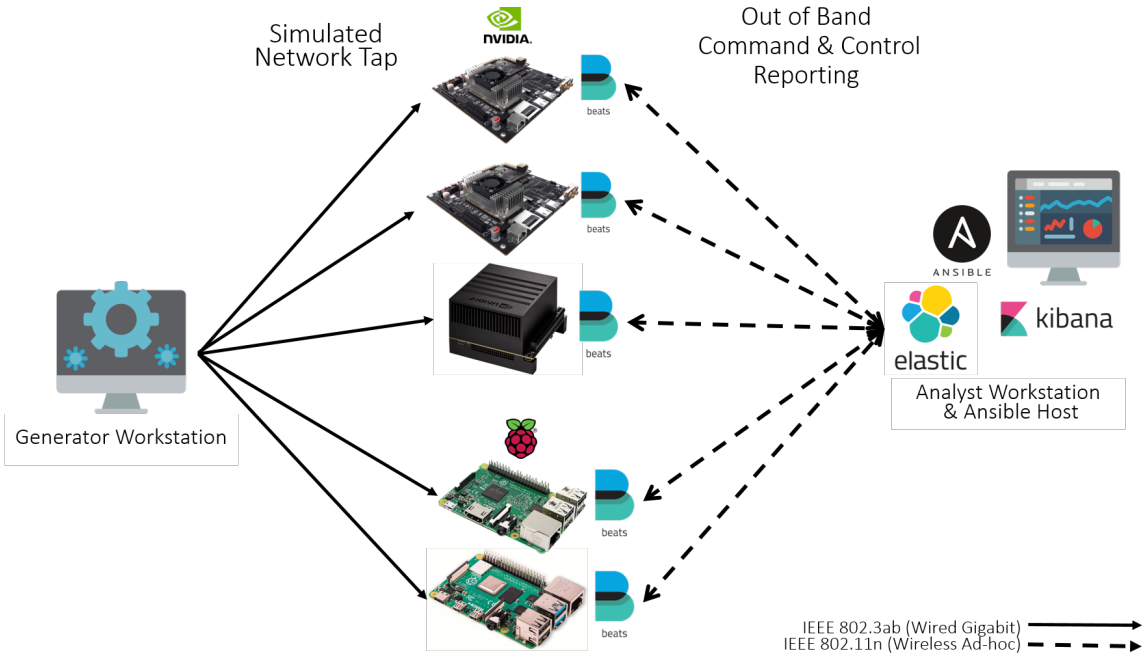


Figure 16: Network monitoring test layout and architecture [82]

Component	Version
Ansible	2.8.5
Netmap	commit cb68f3db
Tcpreplay	4.3.1
Suricata	5.0.0
libpcap	1.8.1
Tcpdump	4.9.2
Elasticsearch	7.5.0
Kibana	7.5.0
Beats	7.5.0

Table 6: Software Versions Utilized in Testing

3.3.2 Traffic Generation

Two open source projects provide the needed functionality to drive the traffic generator (specs in table 7), *netmap* and *tcpreplay*. Netmap is a framework for very fast packet Input / Output (I/O) from userspace applications. This is accomplished via a a few steps, mainly intelligent I/O batching, a custom kernel module with modified network interface drivers, and pre-allocated memory mapped buffers [83]. These techniques allow a researcher or developer to fully utilize modern links up to 40 gigbits per second. Driver support is a bit limited however, with only one non Intel based chip supported (r8169 out of ixgbe, igb, i40e, e1000, e1000e). Regardless this project has enabled commodity hardware to achieve something that had been traditionally reserved for expensive commercial appliances.

Component	**Value**
Architecture	x86_64
Cores	24
CPU MHz	2776
RAM	24GB
NIC & Driver	4x Intel igb (PCIe x16)
Distribution	Ubuntu 18.04.3
Kernel	linux 5.0.0.37

Table 7: Traffic Generator Hardware Specifications

tcpreplay is a project that is used for replaying and editing previously obtained packet captures. It was originally designed to replay malicious traffic for tuning of IDS but has evolved to providing background noise and simulating requests to things like web servers. [84] It fully supports the netmap API as well which provides the much

needed optimizations to get packets on the wire with as little meddling as possible from the host kernel.

Multiple preexisting packet capture datasets exist so there was no need to generate one. The Canadian Institute for Cybersecurity (CIC) has published a survey and dataset of their own which contains a good blend of modern attacks on a fairly large virtualized environment. These datasets also come with attack truth data that can be used to validate results seen after replaying. [85]

3.3.3 Dataset Preparation

This particular dataset required some conditioning however before it could be properly replayed over a live wire. When it was first captured, their sensor machine likely had Large Receive Offload (LRO) and/orGeneric Receive Offload (GRO) enabled. These receive side optimizations buffer and automatically recombine segmented packets on the network card instead of interrupting the CPU to handle each one, thereby reducing overhead. These mega packets then get sent to the application layer and saved in the packet capture as-is with packet sizes that are larger then the Maximum Transmission Unit (MTU). For purposes of all testing in this work, the MTU is set the Ethernet default of 1500 bytes.

The problem arises when attempting to replay captures with these gigantic packets which can be up to the size of the "IP total length" field ($2^{16} = 64k\ bytes$). The transmit side counterpart optimizations TCP Segmentation Offload (TSO) and Generic Segmentation Offload (GSO) do the same thing in reverse for normal user applications and could potentially handle the large payloads. The netmap driver bypasses these optimizations however for finer tuned speed control meaning any packet larger then the MTU gets dropped before transmission as it's too big for the buffer [86].

The *tcpreplay* project includes other tools that can modify packet captures temporarily or permanently . By using the *tcprewrite* tool in combination with the *fragroute* engine it is possible to break any frames larger then the MTU into IP fragments. The fragroute engine handles creating the appropriate headers and the result is saved as a new ready to replay packet capture [87]

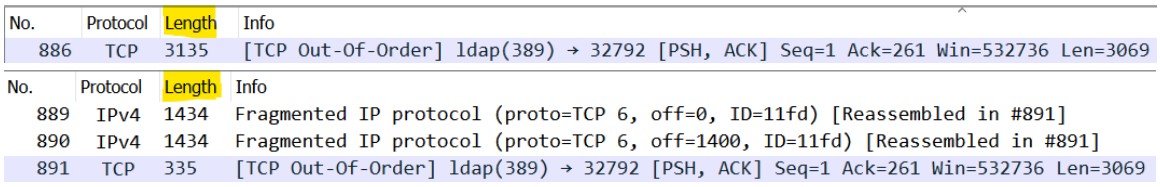


Figure 17: Example showing the same dataset packet before and after artificial segmentation. The new capture now works in other tools properly

Pre-testing also verifies that this segmentation of the dataset has no impact on signature based alerts, as shown below in Table 8

CIC 2017 IDS Dataset File	**Alerts Generated (Original)**	**Alerts Generated (Fixed)**
Monday-WorkingHours	16	17
Tuesday-WorkingHours	2,430	2,429
Wednesday-WorkingHours	13	12
Thursday-WorkingHours	2,475	2,702
Friday-WorkingHours	235	240

Table 8: CIC 2017 IDS Dataset alert rates before and after segmenting super jumbo packets to properly fit in Ethernet MTU

3.3.4 Optimization Factor Selection

Since the goal of the proposed methodology is twofold; one to obtain a desired metric of performance but also ensure fair use of the hardware it is important to select appropriate optimization factors to try. By understanding the exact path a packet takes upon arriving to a linux based device it becomes possible to hone in on possible

bottlenecks. Below is a breakdown of the workflow from when a new packet arrives to final delivery to the proper application. This is built from the perspective of a single receive queue interface, which applies to all single board computers tested in Table 2. Multiple queue interfaces exist mainly on enterprise servers and can essentially implement the workflow (# of queue) parallel times. New API (NAPI) was added to the kernel around version 2.6 as new method of reducing expensive hardware interrupts by combining their asynchronous requests with a period of scheduling friendly polling.

1. Frame is received by the network adapter
2. Frame is moved (via Direct Memory Access (DMA)) to a RX ring buffer in kernel memory
3. The NIC notifies the system of that there is a new frame ready for processing by raising a hardware Interrupt Request (IRQ)
4. The hard IRQ is cleared and the pending data is scheduled to be moved with NAPI software based IRQ
5. The kernel executes NAPI driver code which drains the RX ring via "software IRQ"
6. The software IRQ place the frames into a kernel data structure called a "skb" (socket buffer)
7. (if enabled) Receive Flow Steering (RFS) hashes the incoming traffic to keep similar flows together for cache coherency
8. (if enabled) Receive Packet Steering (RPS) selects and bins traffic to a CPU for further processing.
9. (if enabled) cores keep each other active via Inter-Processor Interrupt (IPI) preventing backlogs from piling up
10. Each backlog queue decodes new frames into appropriate protocol buffer. Optionally delivers raw bytes to packet tapping software
11. Applications expecting traffic are notified and they read packets from kernel memory via syscall

Figure 18 below is a workflow diagram showing each step just mentioned. This figure will be important and referenced many times in Chapter IV specific test factor selections.

Since the fundamental unit to the kernel is a "packet", regardless of it's payload size the primary network generation control measure is chosen as Packets Per Second (PPS). For purposes of this experiment, all links considered will be gigabit Ethernet (1000BASE-T) based. This is due to it being the de-facto supported link in all the boards tested and availability of external hardware. Considering this and the minimum overhead required to send a packet (shown in figure 19) the maximum packet rate and thus ultimately the interrupt rate is:

$$\frac{1,000,000,000 \text{ bits/sec}}{(84 \text{ bytes} * 8 \text{ bits/byte})} = 1,488,096 \text{ Packets Per Second (PPS)} \tag{8}$$

Frame Part	Minimum Frame Size	Maximum Frame Size
Inter Frame Gap (9.6 ms)	12 bytes	12 bytes
MAC Preamble (+ SFD)	8 bytes	8 bytes
MAC Destination Address	6 bytes	6 bytes
MAC Source Address	6 bytes	6 bytes
MAC Type (or length)	2 bytes	2 bytes
Payload (Network PDU)	46 bytes	1,500 bytes
Check Sequence (CRC)	4 bytes	4 bytes
Total Frame Physical Size	**84 bytes**	**1, 538 bytes**

Figure 19: Range of possible Ethernet frame sizes and associated overhead [90]

While limiting the packet size to the absolute minimum is possible via Netmap this does not simulate realistic traffic, except in perhaps flood or denial of service type attacks which are outside the scope of this research. The average packet size seen by The Center for Applied Internet Data Analysis (CAIDA), an organization dedicated

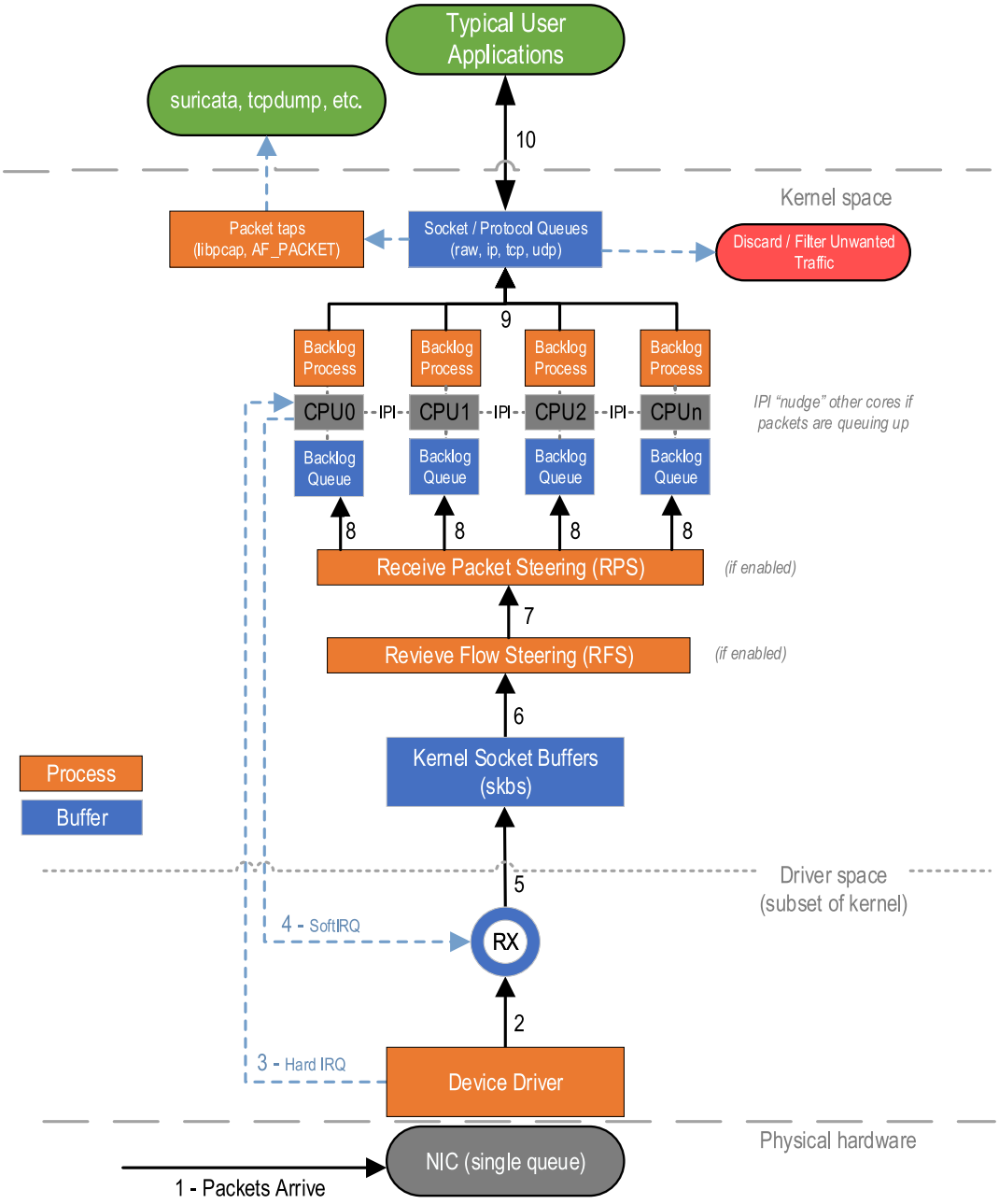


Figure 18: Recieve packet flow through the linux kernel, post NAPI implementation (2.6+) [88] [89]

to conducting network and infrastructure research, is anywhere from 750-900 bytes. They obtain their data via multiple passive taps on large Internet backbone sites [91]. One such observation event shown in Figure 20 in early 2019 saw 2.5 billion packets in an hour with a mean IPv4 packet size of 891. According to CAIDA, IPv6 still only accounts for about 11% of the traffic and will not be considered at this time [91].

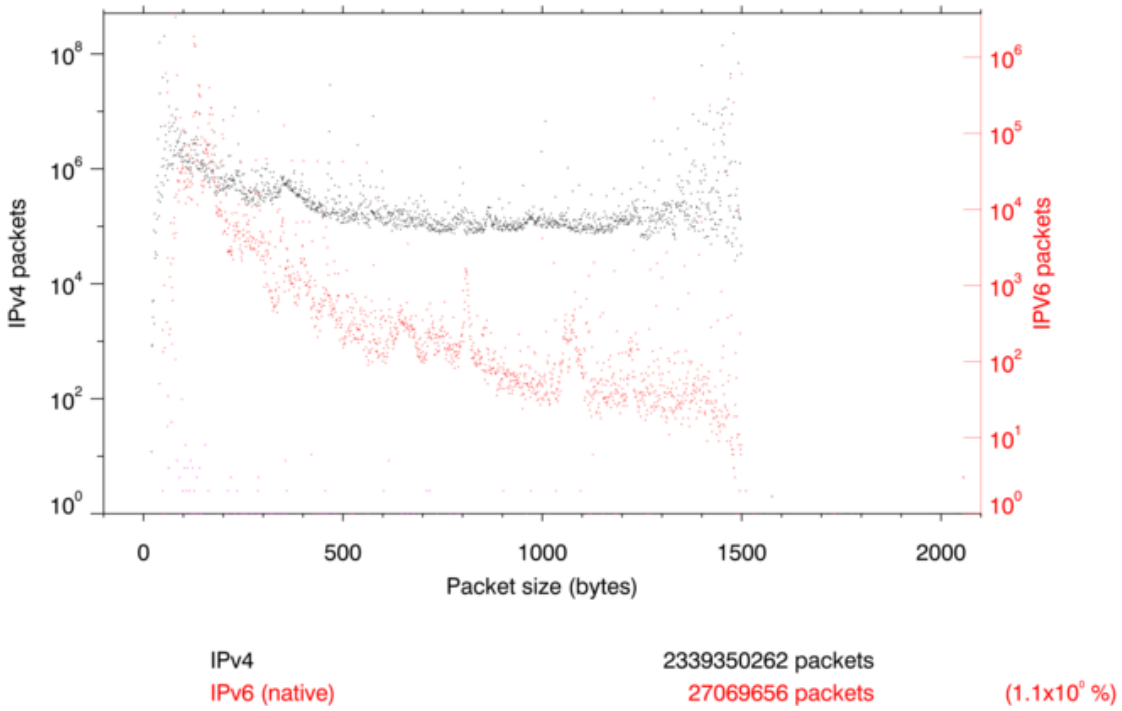


Figure 20: NYC Internet Backbone on 2019-01-17. 1 Hour of observed packet lengths [91]

Using netmap to generate a sample of 10 million random packets, the average packet length ends up around 783 bytes. While its distribution of sizes is less then desirable as it heavily favors packets sized between 640 - 1279 (figure 21), the CIC dataset suffers from a similar problem with an average of 890 bytes per packet favoring sizes 40-79 (mostly TCP ACKs) and 1280+ (mostly fragmented continuations) [85]. Regardless, all of these observations are inline with previous research showing real world interarrival times and burst lengths for Internet / Ethernet [92] and application

traffic (e.g. WWw and FTP) [93] behave as fractal and self-similar, failing to fit traditional "normal-like" distributions like Poisson. Regardless the robust analysis method described in Section 3.2.1 should be able to handle this potential adversity.

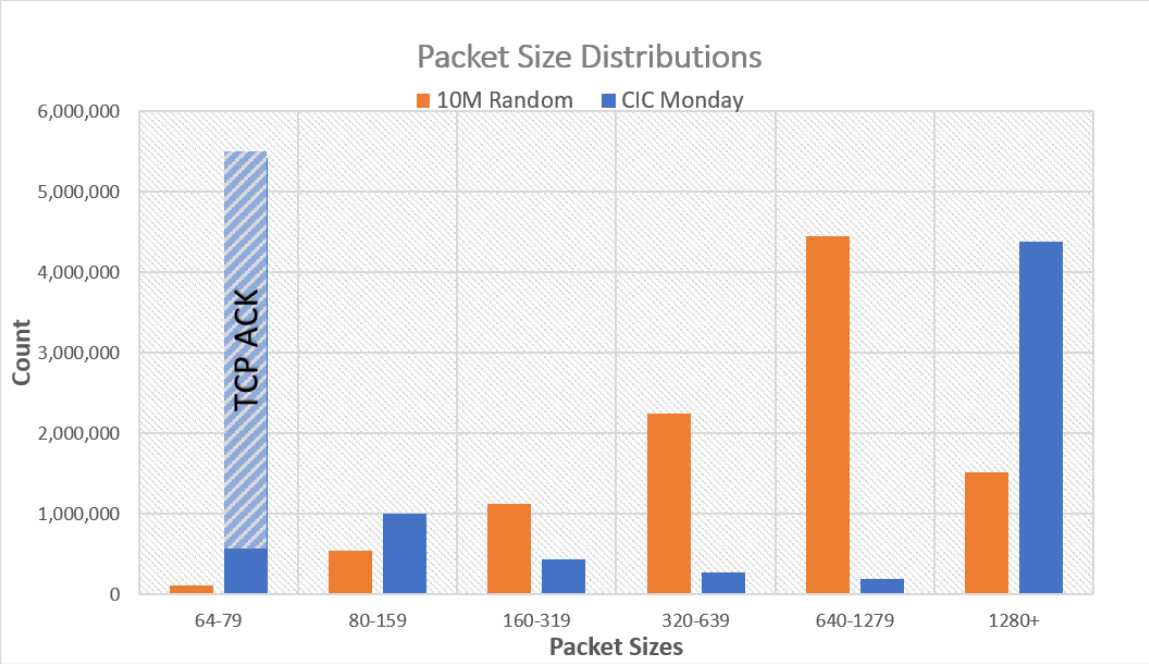


Figure 21: Test traffic packet length distribution, randomly generated versus CIC dataset

Considering all the above the goal is to fully saturate the gigabit based interface with reasonable packets, the following "perfect transmission" calculation yields an upper limit of the physical medium itself that will be used in further testing [90].

$$\frac{1,000,000,000 \text{ bits/sec}}{(783 \text{ bytes} * 8 \text{ bits/byte})} = 159,624\text{PPS} \tag{9}$$

The traffic generation machine has a few other input variables of its own. Since the modified netmap driver is interfacing directly with the hardware to send packets at line speeds, it does not interface well with typical kernel optimizations. This means disabling GSO, TSO, and TX checksum offloading as leaving them on may lead to

"dropped" TX packets. [86] In addition, speed throttling mechanisms like Ethernet pause frames (IEEE 802.3x) must be disabled to prevent the sending side from slowing down despite the congestion. Lastly as discussed earlier in section 3.3.4 the initial PPS is also set at 160,000.

Snippet 15 The traffic generator machine is shared with the Ansible host, which can perform actions on itself. These "Local actions" are re-performed for good measure at the begging of each middle loop in tandem with the other static re-configurations in *general-static-controls.yml*

```
- name: Disable Flow Control On Send Interface
  local_action:
      module: shell
      _raw_params: sudo ethtool -A {{ send_interface }} autoneg off tx off rx off
      args:
          warn: false
  register: local_result
  failed_when: "'Cannot' in local_result.stderr"

- name: Disable Offloads on Send Interface
  local_action:
      module: shell
      _raw_params: "sudo ethtool -K {{ send_interface }} gso off tso off tx off"
      args:
          warn: false
```

3.4 Summary

In summary, this chapter introduced and discussed a general methodology that utilizes a robust 2^k Full Factorial design to optimize and benchmark various workloads for various edge type single board computers. Furthermore a specific use case is prepared to evaluate five different commodity device's performance with respect to three network monitoring applications. Next in Chapter IV the specific results are presented and discussed in detail.

IV. Results and Analysis

4.1 Methodology Validation

Examining select results from the three network monitoring tests it is possible to validate the three main design points detailed in Chapter III. The full network monitoring optimization factor selection, results, and analysis is provided in Section 4.2. An annotated guide on the layout of all the results figures in this chapter is available in Appendix A.

4.1.1 Automation Validation

The automation design was driven by the fact edge devices come in a wide array of capability and cost. In other words, it needed to be easy to use, modify, and scale as appropriate. It also needed to be repeatable and support a consistent baseline. While proving something is "easy" is not very straightforward, a few qualitative observations were taken. Figure 22 below shows approximately the amount of unique lines (including white space) each test performed contains. In all three cases the amount of reusable code is 90% or greater. It only took 142 lines on average to swap the workload, all the variables, and a few housekeeping items. This does not even factor in the very generous use of newlines in the human readable Yet Another Markup Language (YAML) playbooks (i.e. a lot of lines only have two words on them like `become: yes` or `ignore_errors: yes`)

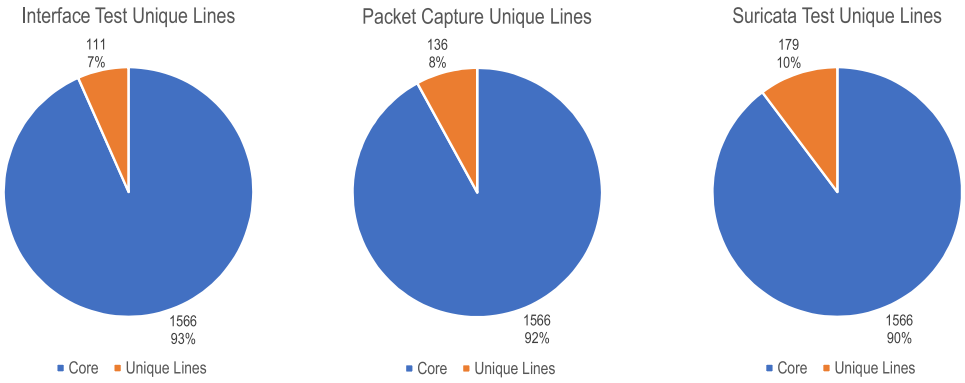


Figure 22: Breakdown of unique lines of code per test. Highlights the ease of swapping in different test workloads

The nature of how Ansible works handles the scale and repeatability dilemma, since the tool is typically used in large data centers or cloud environments where the entire enterprise could almost be ephemeral [8]. Likewise with every single step taken documented in a playbook the baseline is predictable. Figure 23 below highlights how many individual tests were run when building all the results seen later in this chapter. Each "replicate" in this included fully starting, stopping and recording the result of whatever workload was being tested and each device ran in parallel. While this particular number of tests (almost 9,000) may or may not be impressive, a scenario with say 10 devices testing in parallel would increase the number of tests completed to almost 18,000 in the same time frame.

4.1.2 Multi-Stage Optimization Validation

The Suricata test showed the most clear evidence of the optimization process working since in all test cases each device dropped a substantial amount of traffic initially. Using one such result as an example, in Figure 24 below the TX2 was observed initially dropping 7.3 million packets out of the 14.1 million (51%) it was sent. After the first optimization pass this was reduced to only 4.4M, a 41% reduction.

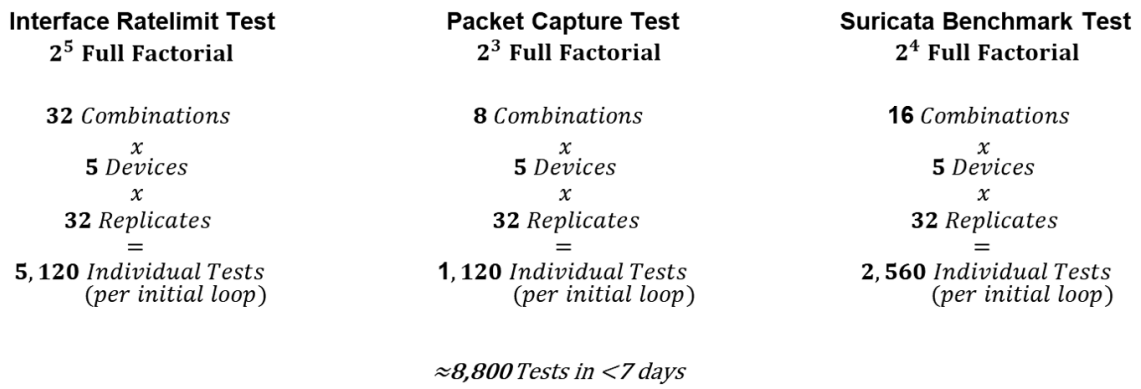


Figure 23: Demonstrates how many individual tests were run when building all the results in Chapter IV. Highlights the automation portion of the implementation works well

A second and third pass both saw another 10% reduction to 4M and 3.6M respectively. The process came to an end after the fourth attempt saw no further improvement.

In the end this test saw almost a 50% improvement of the devices ability to handle the dataset traffic, going from 7.3M packets dropped to only 3.6M. A similar result was seen in the other devices tested as well, full details of which are available in Section 4.2.3.2 with factor discussion and further test details in Section 4.2.3.

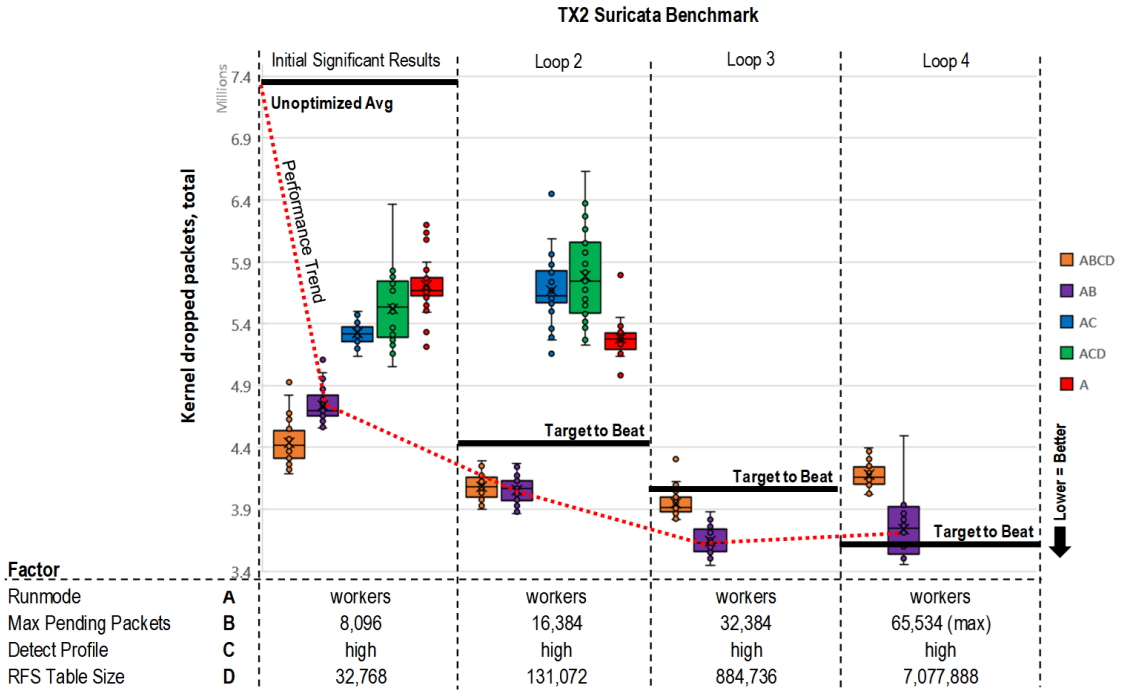


Figure 24: The optimization portion of the Suricata test is observed selecting the best runmode (A) and Max Pending Packets (B) after multiple iterations. The performance trend line shows the progression of optimization from start to finish

Similarly the packet capture initial unoptimized scores had a large sum of dropped packets. Figure 25 shows the TX1 dropped 1.7 million packets out of 3.3 million (51%) and the XAVIER dropped 41,910 out of 4.8 million (1%). After the first iteration of optimization the amount of dropped packets reduced to zero for the XAVIER but remained at 42% for the TX1. Since the TX1 still had remaining memory not being used, a second and third iteration of the loop continued on this device and found the optimal buffer size to reduce drops to zero. The final result stored for each individual device was its maximum sustained Packets Per Second (PPS) at this optimal buffer configuration. The full discussion of this test is in Section 4.2.2 with results available in Section 4.2.2.2.

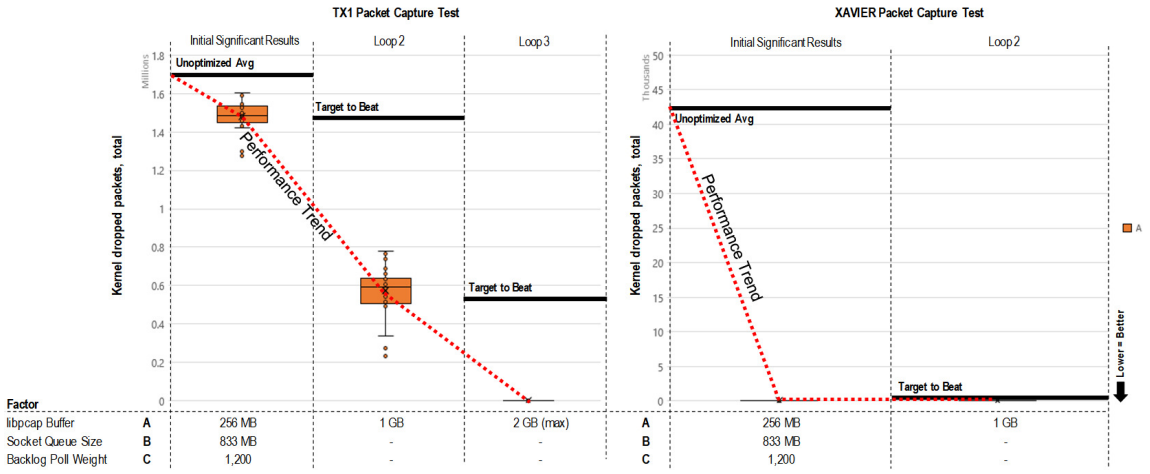
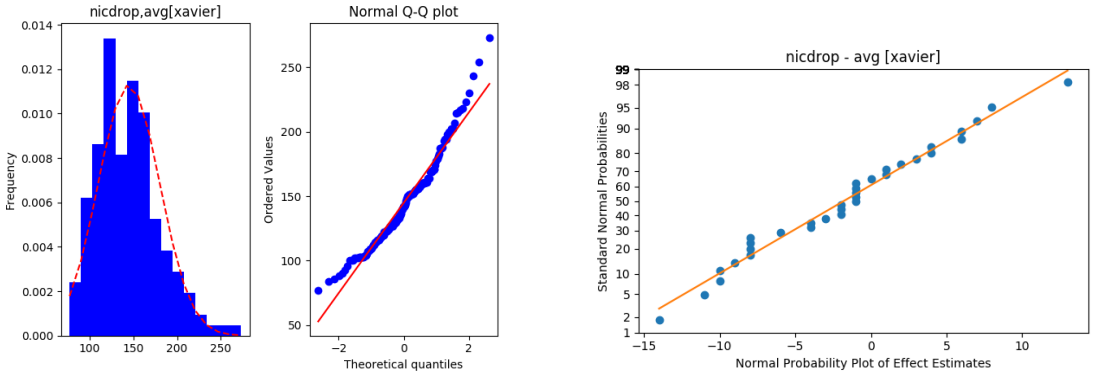


Figure 25: The optimization portion of the packet capture test found the optimal *libpcap* buffer size after multiple iterations without over-sizing. The performance trend line shows the progression of optimization from start to finish

4.1.3 Analysis Validation

While the analysis process working is almost implied when discussing the optimization results, a deeper look reinforces the confidence that the process as implemented will be able to handle diverse situations. Most of the gray area resides when he assumption of normality is violated and the backup analysis is triggered. As discussed in Section 3.2.1.2 this test can be reduced to a simple "trial and error" approach to optimization.

For example, with the introduction of the Canadian Institute for Cybersecurity (CIC) traffic dataset the lack of randomness seen from the traffic generator begins to influence and break down the normality assumptions of the network based tests. Figure 26 below shows the graphical normality for the XAVIER's interface rate test. While not perfect, the random packet generation of *netmap* as discussed in 3.3.2 supports an argument for normality.

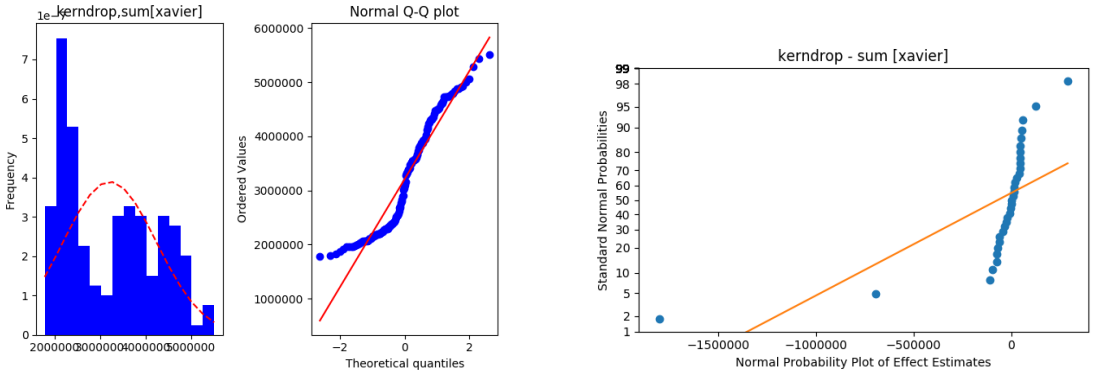


(a) Histogram, Q-Q Plot

(b) Normplot of Effects

Figure 26: XAVIER Interface Normality Tests. The network traffic was randomly generated, and part of this is shown through in the response variable nearly fitting the model lines

Figure 27 afterward shows the graphical normality tests for the XAVIER's Suricata test. The response variable fails to fit the assumption model, likely due to a break down when switching from randomly generated traffic to repeatable dataset traffic. This will ultimately manifest itself as either a Type I (false positive) or Type II (false negative) error.



(a) Histogram, Q-Q Plot

(b) Normplot of Effects

Figure 27: XAVIER Suricata Normality Tests. The Dataset packet captures were replayed, and thus the response variable poorly fit the normality model

While the Analysis of Variance (ANOVA) test is still ran after the end of the

first optimization loop, it's accuracy is questionable when observing the estimated effects seen in the XAVIER Suricata result (Figure 28). Factor combination AB had the lowest sample mean but had an erroneous positive estimated effect. This type of behavior is evident on the other replayed traffic test results as well. Despite this, the result of the ANOVA was still recorded and the backup mode took over identifying the good impact AB had regardless.

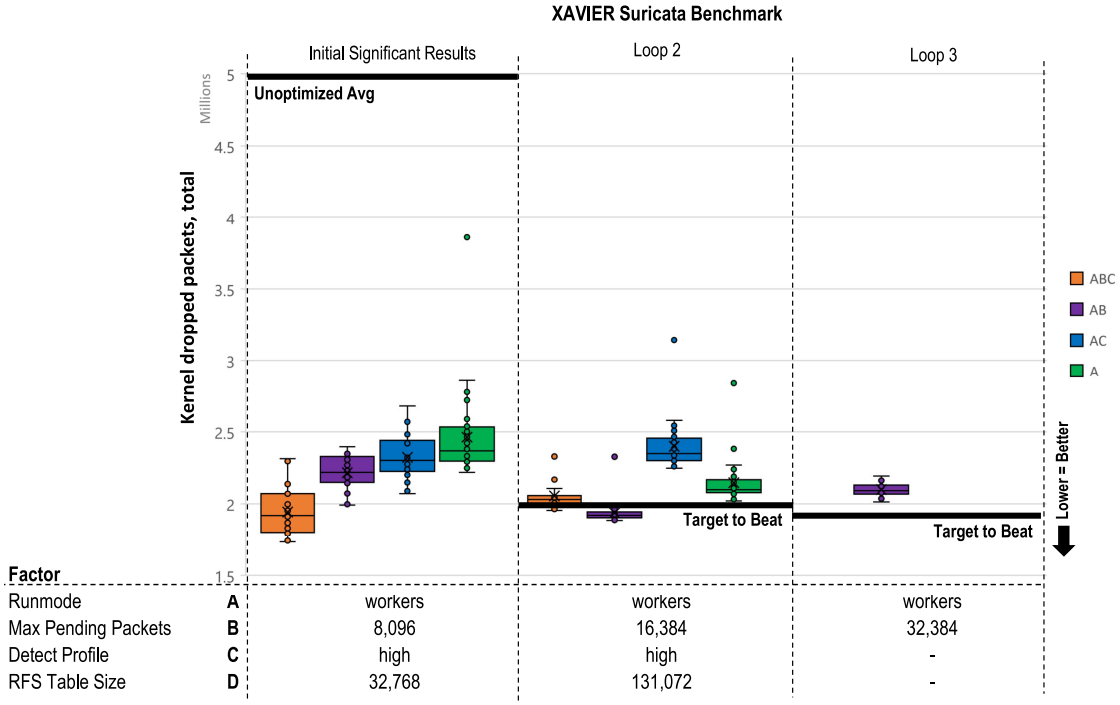


Figure 28: Caption

Validating the analysis process itself is more straightforward. The actual Python code implemented in *anova.py* was validated as accurate by comparing the output between it and some professional software. Additionally we can observe it working as intended in the result of the TX1 packet capture test shown in Figure 29. Pretesting

had shown that only increasing the *libpcap* buffer size (Factor A) would greatly reduce the amount of traffic dropped by all the devices. The ANOVA process successfully identified this lone single factor as significant and did not erroneously also select the combinations that had it (e.g AB, AC) even though they too showed a significant boost. The full results of this and similar tests are in Section 4.2.2.2.

First Run ANOVA

n = 32		Sample Mean	Effect Est.	F0	p-value
k = 3	**A**	**1,481,130**	**-276,769**	**399.797**	0.00000
α = 0.5	B	1,739,562	-21,826	2.486	0.11612
RV= Kernel packet drops, total	C	1,714,068	-19,970	2.081	0.15038
Input=109k packets per sec.	ABC	1,434,471	-19,938	2.075	0.15103
	AB	1,455,380	-16,108	1.354	0.24568
	AC	1,484,588	11,243	0.660	0.41739
	BC	1,736,040	7,753	0.314	0.57590
	(N)	1,772,971	-	-	-

Figure 29: The ANOVA process successfully identified Factor A as the only significant factor of the TX1 packet capture test and did not erroneously also select the combinations that had it (e.g AB, AC) even though they too showed a significant boost

4.2 Network Monitoring Results

4.2.1 Interface Ratelimit Test

The first and perhaps most important test of the hardware is the interface ratelimit test. Generally speaking, this test establishes how well the Network Interface Card (NIC) and Central Processing Unit (CPU) packages balance the load of oncoming network traffic. Specifically speaking, this test establishes a macro benchmark of how well the CPU handles interrupts from the NIC and further processes incoming bytes to their destination.

Since the rate of incoming traffic to a network sensor is out of its control, it is not unusual for more packets to arrive then can be processed. If at any point the network interface driver is filling its circular RX buffer faster then it can be emptied,

information will be lost as an "interface drop." The goal of this test is to establish the maximum rate packets can arrive (*PPS limit*) and highlight which parts of the processing pipeline can be optimized to reduce interface drops.

4.2.1.1 Selected Factors

Reexamining the workflow shown in figure 18, there was an assortment of possible variables to tune. Some of them were simple on/off toggles (like enabling Receive Packet Steering (RPS) and others were a scaling window (like backlog queue size) at various points in the path. Based upon suggested tuning in the Suricata official documentation [47], High speed vendor recommendations [94] [95], the Linux kernel documentation [96], and other privately funded research [89] the following factors were chosen initially.

Static Controls

Disabling Large Receive Offload (LRO) and Generic Receive Offload (GRO) on the sensors not only prevents the capture issues seen in section 3.3.2, but is recommended by the official Suricata documentation. The auto combination of similar traffic while fine for typical end clients strips off too much metadata for certain signatures to work properly [47]. While these changes are not necessarily required for the ratelimit test, making them here will provide a better foundation for future tests.

In addition, early tests had shown that multi receive queue interfaces like the Broadcom tg3 performed poorly compared to their single queue counterparts like the Intel e1000e. This is likely due to a hardware limitation of the interrupt handler on ARM boards [97]. Since each queue has its own IRQ handler, most modern x86 multi processor systems have an advanced programmable interrupt controllers like IO-APIC that allow each one to be "re-mapped" to different cores. For the tested ARM based boards, the more limited interrupt handlers used like GICv2 lack any method

of remapping to different cores, meaning all interrupts will be bound to the CPU core that booted the system, usually CPU0 [98] [97]. Thus a multi queue interface ends up with multiple hard IRQs all mapped to the same core setting the stage for an interrupt storm. By forcing the number of active queues in the driver to 1 this scenario can be avoided.

Another option is enabling RPS which attempts to load balance incoming traffic processing on a multi core system once the driver has fully delivered it to a Socket Buffer (skb) (item 7 in figure). This is accomplished by a hashing and sorting algorithm that determines which CPU should process the data. Each core then keeps the other informed via a Inter-Processor Interrupt (IPI) that new bytes have arrived for processing. [96] RPS is disabled by default and can be enabled on a per CPU basis with a bitmask. The official recommendation is to enable it on every core not handling hardware interrupts [96] [94] which as seen above is CPU0. Three of the devices under test have four cores, which leads to a bit mask of 0xE (1110) where CPU0 is the least significant bit. The TX2 has six and the Xavier has eight cores however requiring two more unique masks (0x3E & 0xFE respectively). A simple entry in the inventory can account for each devices unique definition of what "high" means for this factor.

```
1 tx2:
2     capture_interface: eth1
3     send_interface: eth6
4     line_pps_limit: 160000
5     rps_mask: 3E #0011 1110 (6 cpu total, CPU0 handling IRQ)
```

Surprisingly, initial tests had shown (in Figure 30 below) RPS very positively affected the NVIDIA boards while it quite negatively affected the RPi boards. Upon further investigation this is likely due to the overhead associated with calculating the hash needed to load balance traffic across multiple cores. This overhead was seen

early on as a problem after RPS was introduced, so some vendors began to offload this calculation to the hardware. [99] The RPi boards built-in Ethernet port comes with rather limited drivers and lack this capability where the Intel external cards on the TX1 and TX2 do not. The Xavier is using it's built in port but is fast enough the effect is not seen. With these stark contrasts RPS was selectively enabled as a static control where it made sense.

As another important initial test, the Rx Ring buffer is also scalable on some devices. Despite recommendations from enterprise vendors like Red Hat [100] and Mellanox [95] another initial test (also shown in Figure 30 below) showed increasing the Rx Ring size made a very large negative impact on the two supported devices. It is speculated this is related to a mismatch created between the hardware buffer and the CPU cache sizes. While this memory resides in the kernel itself the ability to control it is limited by the NIC and its driver [89]. The two Intel drivers tested (igb, e1000e) support it but the integrated drivers (eqos, lan78xx and bcmgenet) did not. Future work could retry this test to seek out a potential "sweet spot" for these buffers but it was ultimately removed due to only 2 of 5 devices supporting it.

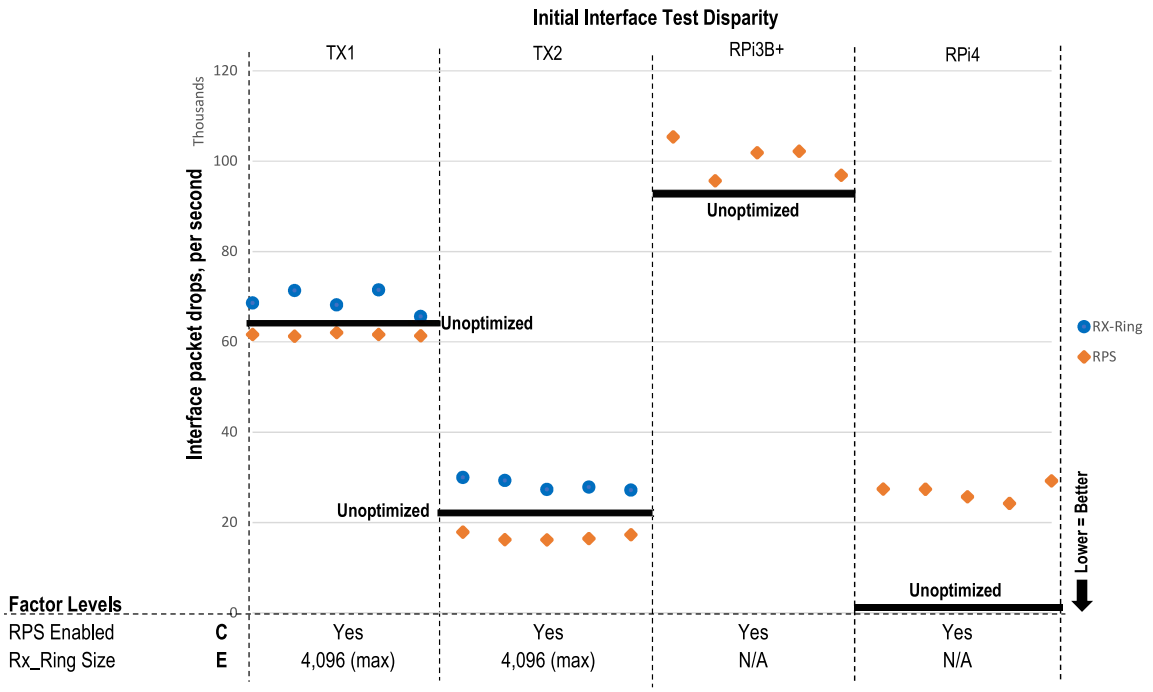


Figure 30: An initial interface test showed disparity the effect enabling RPS had, in addition to the negative impact of RX-Ring resizing

Lastly in order to observe the peak performance of these devices a few unique toggles to ensure they are in "full power" mode. As an important distinction, these are not overclocking anything beyond factory specifications but rather removing factory power saving defaults. For the Raspberry Pi3B+, increasing the *temp_soft_limit* increases the time it's CPU runs at full clock before dialing back to prevent overheating. The hard limit is 85 C and this value could likely be tested even higher by a bold researcher [69]. (The RPi4 boards currently do not implement any similar system) The NVIDIA boards have a more rigid power profile definition that tightly controls the power consumption. It does this by toggling entire cores and subsystems on and off while limiting clocks substantially. The "MAXN" mode ensures all cores are online and at maximum frequency [70]. Other primary modes include a 10W, 15W, and 30W power cap with various sub modes that enable or disable things like Vision Accelerators and Deep Learning Accelerators.

```
- name: Set Receive Offloads
  command: "ethtool -K {{ capture_interface }} lro {{ lro_status }} gro {{ gro_status }}"
  become: yes

  #Multiple rx queues dont make much sense on small boards where the IRQs cant be remapped
  #the SMP affinity for all of them hits the same core, making it worse
- name: Limit Number of Hardware Queues
  shell: ethtool -L {{capture_interface}} rx 1
  become: yes
  ignore_errors: yes
  register: queues_result
  failed_when: "'Invalid argument' in queues_result.stderr"

- name: Bump RPi Throttling Temp (3B+ only)
  lineinfile:
    path: /boot/config.txt
    regex: "temp_soft_limit="
    line: temp_soft_limit=70.0
  when: "'nvidia' not in group_names"
  become: yes

- name: Set MAXN Power Profile on NVIDIA Boards
  shell: |
    nvpmodel -m 0
    jetson_clocks
  become: yes
  when: "'nvidia' in group_names"
```

Variable Controls

Five unique variable factors were chosen for this test and are detailed below.

Factor A: New API (NAPI) Budget

The NAPI budget determines how much processing time can be spent among all the device driver polling structures, where each interface and each queue (rx/tx) is considered a structure. The driver poll is typically responsible for draining the RX ring and moving the received bytes further into kernel memory, into a struct called a skb. (Item 5 on figure) Increasing this may only prevent drops on a multi-queue system but may also prevent the sensing interface from hogging all the available "network related" CPU time from other non-sensing interfaces. [89]

```
###FACTOR A###
#Default 300 / Test Level 1200
- name: (Factor A) Increase NAPI Budget to {{NAPI_budget[1]|int * loop_multiplier}}
  shell: sysctl -w net.core.netdev_budget={{NAPI_budget[1]|int * loop_multiplier}}
  become: yes
  when: "'A' in current_factor_list"
```

Factor B: Max Kernel Backlog (Pending Packets)

The Max Kernel Backlog (item 8 in figure) is simply how many packets can be backed up waiting on a CPU. The larger the number, the more tolerance for latency is implied in exchange for throughput as some packets may be waiting awhile [89] [95].

```
###FACTOR B###
#Default 1000, test level 262144
- name: (Factor B) Set Kernel Max Backlog to {{backlog[1]|int * loop_multiplier|int}}
  shell: sysctl -w net.core.netdev_max_backlog={{backlog[1]|int * loop_multiplier|int}}
  become: yes
  when: "'B' in current_factor_list"
```

Factor C: Socket Receive Buffer Size

The last stop in the kernel processing of packets is the appropriate application layer protocol queue. (Item 9 in Figure 18) Here packets await an application to consume them. For purposes of the interface rate test these packets have no consuming application, however if they fit into a queue at this point they are no longer considered a "drop."

```
  ###FACTOR C###
- name: (Factor C) Set Socket Recieve Max Buffer Size to {{rmem_max[1]|int *
 ↪  loop_multiplier|int}}
  shell: sysctl -w net.core.rmem_max={{rmem_max[1]|int * loop_multiplier|int}}
  become: yes
  ignore_errors: yes
  when: "'C' in current_factor_list"
```

Factor D: Receive Flow Steering

Receive Flow Steering (RFS) (item 6) is used in conjunction with RPS and helps

solve some data locality issues by steering kernel processing of packets to the CPU where the application thread consuming the packet is running, thus increasing cache hit rates [89] [96]. This process has two main control variables, the size of the hash table that tracks all the individual socket flows and how many flows are tracked per receive queue. For a single queue system, it's recommended they be the same value.

In this test, since there is no process actually consuming packets it is unlikely to make an impact. The socket flow table is informed by *recvmsg* and *sendmsg* system calls from a userspace application and in the absence of any of these, RFS will fall back to plain RPS [96].

```
  ###FACTOR D###
  Default off / Test level 32768 (for both)
- name: (Factor D) Set Receive Flow Steering (RFS) Table Size to
 ↪  {{rfs_table[1]|int * loop_multiplier|int}}
  shell: |
    sysctl -w net.core.rps_sock_flow_entries="{{ rfs_table[1]|int *
     ↪  loop_multiplier|int }}"
    echo "{{ rfs_flow_cnt[1]|int * loop_multiplier|int }}" >
     ↪  /sys/class/net/{{capture_interface}}/queues/rx-0/rps_flow_cnt
  become: yes
  ignore_errors: yes
  when: "'D' in current_factor_list"
```

Factor E: NAPI Weight

The NAPI weight adjusts how much time (out of budget set in Factor A above) can be spent in the "backlog" phase of packet processing (item 9 in Figure 18). Not to be confused with the driver poll, which happens earlier and should have a hard coded weight of 64. This poll takes packets out of a per CPU queue and determines which protocol handler it belongs in. If RPS is enabled each CPU has it's own queue to work on, otherwise the same CPU that handled the initial IRQ sees it through to the end [89].

```
  ###FACTOR E###
- name: (Factor E) Set Backlog Loop Weight to
  ↪  {{backlog_weight|int*loop_multiplier|int}}
  shell: sysctl -w net.core.dev_weight={{backlog_weight|int*loop_multiplier|int}}
  become: yes
  ignore_errors: yes
  when: "'E' in current_factor_list"
```

4.2.1.2 Test Results

The TX1 (Figure 32) saw negligible optimization gains (4%). This is likely due to saturation of the single CPU core responsible for servicing network interrupts. The TX2 (Figure 33) saw a 13% decrease in dropped packets by increasing backlog poll weight (E) to 1,200, but this was a diminishing return and setting any higher caused this boost to regress. Like the TX1, the CPU core handling the NIC interrupt was completely saturated. This is shown in Figure 31

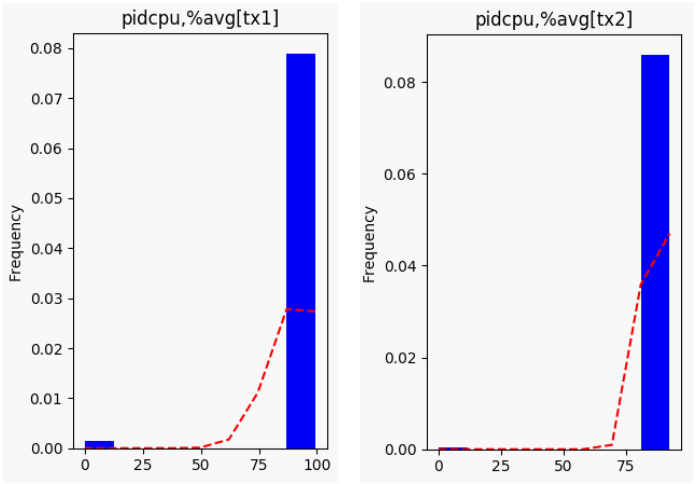


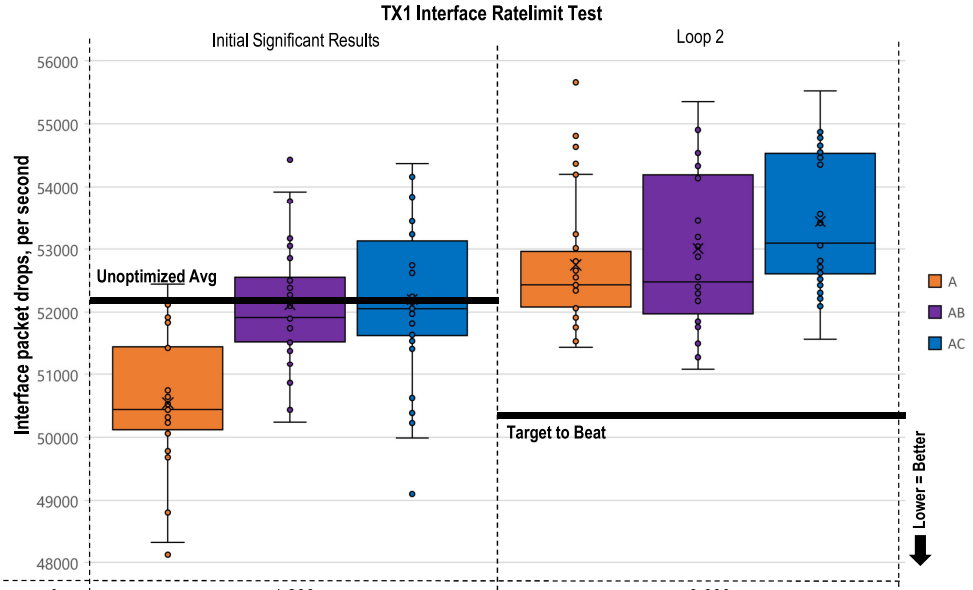
Figure 31: The TX1 and TX2 CPU responsible for NIC interrupts was near 100% for the interface test

The XAVIER (Figure 34) and RPi4 (Figure 36) devices appear to lack any significant bottleneck as their unoptimized sample means were 194 and 33 packets dropped per second respectively. At the line rate of 160k PPS this equates to a negligible 0.0012% and 0.0002% of the traffic being missed. The Xavier did further improve this

to 127 packets dropped by increasing the kernel backlog (B) and RFS table size(D). The RPi3b+ saw a small but statistically significant 2.2% reduction in dropped packets with factor combo BC (Figure 35). Further improvements beyond this are unlikely due to the Ethernet riding over the 480 mbps USB 2.0 bus which has a theoretical max of 76k PPS at the generated traffic avg size:

$$\frac{480,000,000 \text{ bits/sec}}{(783 \text{ bytes} * 8 \text{ bits/byte})} = 76,628 \text{ pps} \tag{10}$$

Overall, for the TX1 and TX2 the bottleneck seems to remain at bottom half of softIRQ polling loop (item 4 in Figure 18). Only the top half is load balanced across the cores with RPS and RFS. Adding more receive queues will not help either as the ARM IRQ handler will still only map to one core [97]. Overclocking the CPU would likely further reduce these drops but was outside the scope of research.



Factor			
NAPI Budget	A	1,200	3,600
Kernel Backlog	B	32 MB	128 MB
Socket Queue Size	C	83 MB	332 MB
RFS Table Size	D	32,768	131,072
Backlog poll Weight	E	1,200	3,600

First Run Result

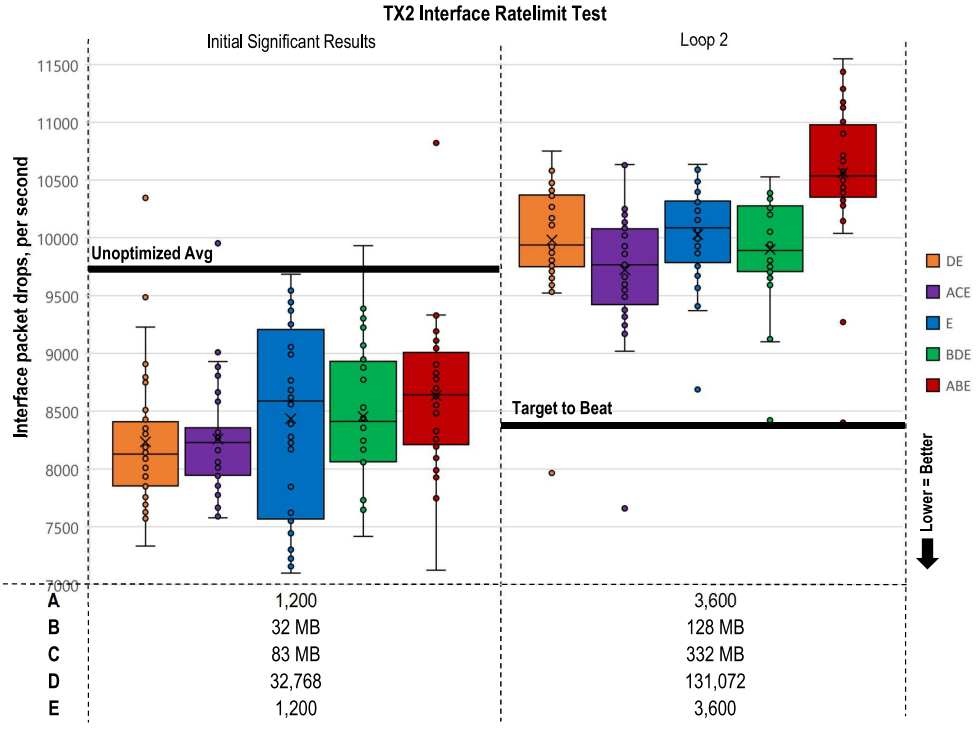
n = 32		Sample Mean	Effect Est.	F0	p-value
k = 5	A	50,543	-82	1.284	0.25752*
α = 0.5	AB	52,117	59	0.689	0.406611*
RV= Interface packet drops, per second	AC	52,198	41	0.334	0.56371*
	(N)	52,298	-	-	-

Input=160k packets per sec.

Optimization Loop 2

	(Target)	50,543	Final Score	109,248	Maximum
	A	52,748			Interface Packets per second
	AB	53,006			
	AC	53,434			

Figure 32: The TX1 saw negligible optimization gains (4%). This is likely due to saturation of the single CPU core responsible for servicing network interrupts



Factor			
NAPI Budget	A	1,200	3,600
Kernel Backlog	B	32 MB	128 MB
Socket Queue Size	C	83 MB	332 MB
RFS Table Size	D	32,768	131,072
Backlog poll Weight	E	1,200	3,600

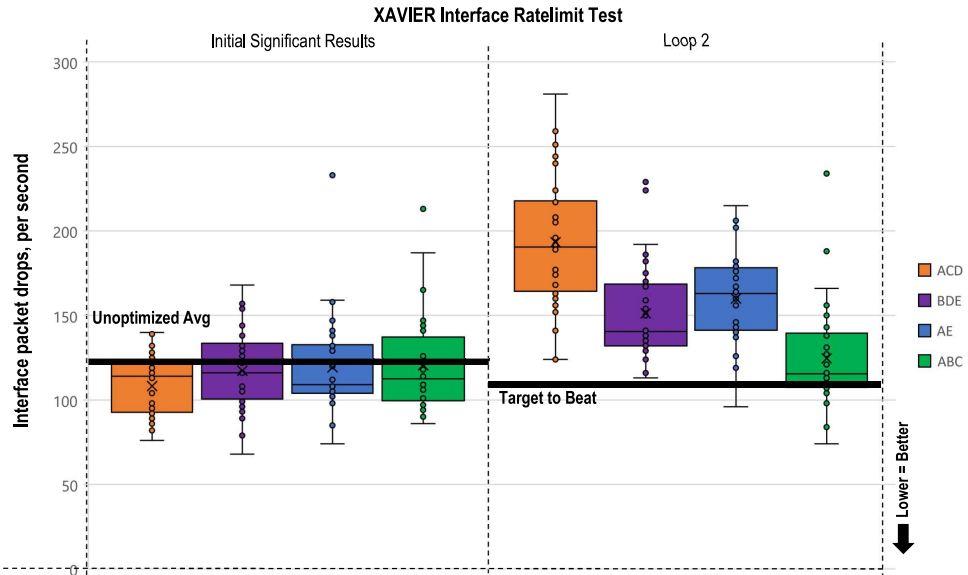
First Run Results (Top 5)

n = 32		Sample Mean	Effect Est.	F0	p-value
k = 5	DE	8,235	200	27.356	0.00000
α = 0.5	ACE	8,259	-140	13.167	0.00030
RV= Interface packet	E	8,435	-491	163.978	0.00000
drops, per second	BDE	8,454	89	5.450	0.01977
Input=160k packets per sec.	ABE	8,635	-223	33.581	0.00000
	(N)	9,413	-	-	-

Optimization Loop 2

(Target)	9,334	Final Score	151,765	Maximum
DE	9,981			Interface Packets per second
ACE	9,721			
E	10,028			
BDE	9,899			
ABE	10,564			

Figure 33: The TX2 saw a 13% decrease in dropped packets by increasing backlog poll weight (E) to 1,200, but this was a diminishing return and setting any higher caused this boost to regress. Like the TX1, the CPU core handling the NIC interrupt was completely saturated



Factor			
NAPI Budget	A	1,200	3,600
Kernel Backlog	B	32 MB	128 MB
Socket Queue Size	C	83 MB	332 MB
RFS Table Size	D	32,768	131,072
Backlog poll Weight	E	1,200	3,600

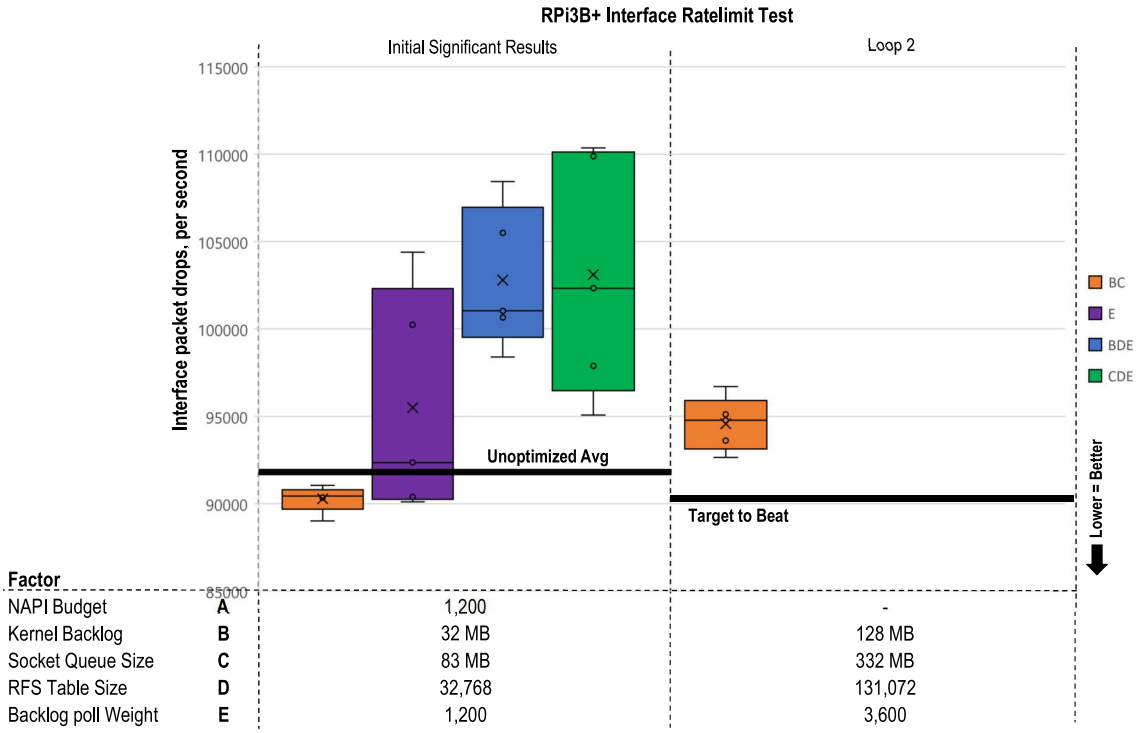
First Run Results

n = 32		Sample Mean	Effect Est.	F0	p-value
k = 5	ACD	108	-10	28.107	0.00000
α = 0.5	BDE	117	-11	30.060	0.00000
RV= Interface packet	AE	119	-8	15.388	0.00009
drops, per second	ABC	121	-10	24.171	0.00000
Input=160k packets per sec.	(N)	122	-	-	-

Optimization Loop 2

(Target)	108	Final Score	151,765	Maximum
ABC	125			Interface Packets per second
BDE	151			
AE	160			
ACD	193			

Figure 34: The Xaiver appears to lack any significant bottleneck as its unoptimized sample mean was 194. At the line rate of 160k PPS this equates to a negligible 0.0012% of the traffic being missed. The Xavier did further improve to 127 packets dropped by increasing the kernel backlog (B) and RFS table size(D)



Factor			
NAPI Budget	A	1,200	-
Kernel Backlog	B	32 MB	128 MB
Socket Queue Size	C	83 MB	332 MB
RFS Table Size	D	32,768	131,072
Backlog poll Weight	E	1,200	3,600

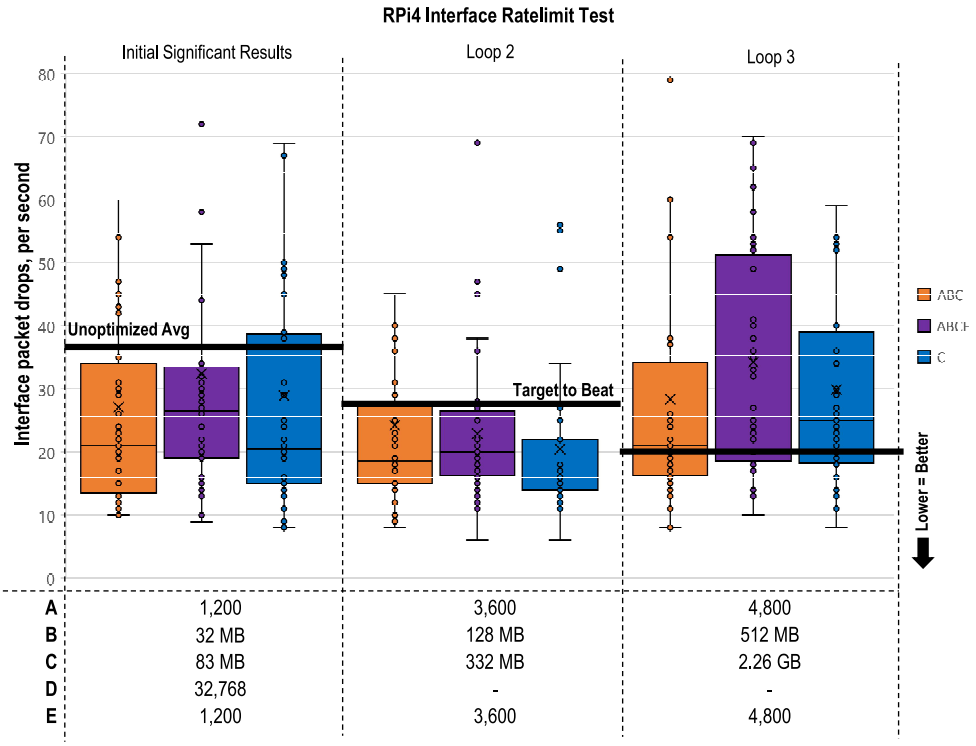
First Run Results

n = 5		Sample Mean	Effect Est.	F0	p-value
k = 5	BC	90,276	2,784	6.645	0.01107
α = 0.5	(N)	92,065	-	-	-
RV= Interface packet drops, per second	E	95,489	6,297	34.001	0.00000
	BDE	102,801	-2,152	3.968	0.04850
Input=160k packets per sec.	CDE	103,102	-3,913	13.126	0.00042

Optimization Loop 2

	(Target)	90,275	Final Score	69,725	Maximum
	BC	94,566			Interface Packets per second

Figure 35: The RPi3b+ saw a small but statistically significant 2.2% reduction in dropped packets with factor combo BC (Figure 35). Further improvements beyond this are unlikely due to the Ethernet riding over the 480 mbps USB 2.0 bus



Factor		Initial Significant Results	Loop 2	Loop 3
NAPI Budget	A	1,200	3,600	4,800
Kernel Backlog	B	32 MB	128 MB	512 MB
Socket Queue Size	C	83 MB	332 MB	2.26 GB
RFS Table Size	D	32,768	-	-
Backlog poll Weight	E	1,200	3,600	4,800

First Run Result (Top 3)

n = 32		Sample Mean	Effect Est.	F0	p-value
k = 5	ABC	27	-3	1.701	0.1924*
α = 0.5	C	29	-3	1.206	0.27239*
RV= Interface packet drops, per second	ABCE	32	2	1.193	0.2749*
	(N)	36	-	-	-

Input=160k packets per sec.

Optimization Loop 2

	C	20
	ABCE	22
	ABC	24
	(Target)	27

3 Factors Continue

Optimization Loop 3

	(Target)	20	**Final Score** 159,980
	ABC	28	
	C	29	
	ABCE	34	

Maximum Interface Packets per second

Figure 36: The RPi4 appears to lack any significant bottleneck as its unoptimized sample means was 33 packets dropped per second. At the line rate of 160k PPS this equates to a negligible 0.0002% of the traffic being missed.

4.2.2 Traffic Capture Ratelimit Test

Traffic capture is in a sense the most raw and fundamental tool when it comes to network monitoring. As discussed in Chapter 2, other more sophisticated tools come with a larger overhead and can easily fail to keep up at network burst speeds [101]. With a machine capable of capturing 100% of the traffic passed to it, those bursts can be queued and replayed or ingested at a later, quieter time. If a packet is missed in the initial capture, there are no second chances and any potential proof or indicator held in that instantaneous moment is gone.

4.2.2.1 Selected Factors

Despite the lower overhead of simply writing packets to disk, keeping up with line speeds largely depends on the medium to which it is being written or stored [46]. For purposes of this test, only the "on board" storage will be considered as a destination since external drives carry too many variables and may be impractical in large scale deployments. As an example writing to a traditional spinning HDD can be influenced by things like spinning speed, cache size, platter density, fragmentation, controller load and bus connection type. The NVIDIA boards tested all utilize a built in eMMC 5.1 based flash storage [102] while the RPi leverage similar but considerably slower SDHC flash storage [103]. Since the bottleneck of this test is predictably going to be storage, a few simple tests will help better inform expectations. A "raw" maximum write speed test can be performed with the command:

```
dd if=/dev/zero of=disktest.img bs=1G count=1 oflag=dsync
```

Where 1GB of zeros is forced to be physically written (no caching) to the current directory. The result of this preliminary test is listed in Table 9 below which is then used in the consideration of further factors.

	TX1	TX2	XAVIER	RP3B+	RPi4
Technology	eMMC	eMMC	eMMC	SDHC	SDHC
Class	5.1	5.1	5.1	UHS-I	UHS-III
Theoretical Write Speeds	125 MB/s	125 MB/s	125 MB/s	10 MB/s	30 MB/s
Observed (3 sample avg)	60 MB/s	99 MB/s	110 MB/s	15 MB/s	23 MB/s

Table 9: Storage Speed Observations

Static Controls

Since disk I/O is very precious for this test, swap was disabled to help prevent unexpected writes. This is especially important for the devices with lower RAM like the RPis, since buffer sizes are likely to balloon to near RAM capacity the swap would likely become active. In addition, since the interface PPS rate was inherited from the first test, carrying over any positive configurations is important. These values are now implemented statically with varying levels as defined in the *inventory.yml* file.

```
  #When Disk I/O is very important. Also lifetime of flash...
- name: Disable Swap
  shell: swapoff -a
  become: yes

  ### VALUES INHERITED FROM PREVIOUS TEST###
  #Former Factor A from interface test
- name: Increase NAPI Budget to {{NAPI_budget_best}}
  shell: sysctl -w net.core.netdev_budget={{NAPI_budget_best}}
  become: yes

  #Former Factor B from interface test
- name:  Set Kernel Max Backlog to {{backlog_best}}
  shell: sysctl -w net.core.netdev_max_backlog={{backlog_best}}
  become: yes

- name: Enable / Set Receive Packet Steering Affinity with Mask {{ rps_mask }}
  shell: "echo {{ rps_mask }} >
   ↪  /sys/class/net/{{capture_interface}}/queues/rx-0/rps_cpus"
  become: yes
```

Variable Controls

Utilizing the interface PPS results from the previous test it is possible to observe the maximum data rate that would be expected of the underlying storage. As shown in the last row of Table 10 below, each device is projected to develop a deficit of packets waiting to be written to disk. At this point the packet will either get dropped because of a full CPU backlog or a full socket queue (items 8 and 9 in Figure 18 respectively). Packets waiting in the socket queue for consumption (in this case by *tcpdump* via *libpcap*) must move to one last buffer in userspace memory after which point they are "guaranteed" to be handled eventually. This leads to three variable factors which are detailed below.

	TX1	**TX2**	**XAVIER**	**RP3B+**	**RPi4**
Interface PPS	109k	149k	160k	70k	160k
Packet Size Avg			780 bytes		
Expected bitrate (Mbps)	680	920	998	430	998
Expected Datarate (MB/s)	85	115	125	54	125
Expected Datarate vs. Observed Storage Speed	-25 MB/s	-16 MB/s	-15 MB/s	-39 MB/s	-102 MB/s

Table 10: Storage Demand Observations

Factor A: Libpcap Buffer Size

Thus expanding the libpcap userspace buffer size in Factor A may buy enough time for larger disk I/O requests to complete during periods of calmer, smaller packet sizes in the generated sample. Factors B (Protocol / Socket Buffer), C (Backlog Poll Weight), and D (RFS Table Size) are repeats from the interface test. They may play a larger role in this test now that an application is actually consuming packets. Line 8-11 in the snippet below leverages the "facts" subsystem in Ansible to ensure the desired buffer isn't more then the available memory. Line 16-19 likewise ensures the desired buffer size is not larger then what the library supports.

```
  ###FACTOR A###
- name: (Factor A) Set libpcap Buffer Size to
  ↪  {{libpcap_buffer|int*loop_multiplier|int}}
  set_fact: #Stored as KiB
      libpcap_buffer_size: "{{ libpcap_buffer|int * loop_multiplier|int }}"
  when: "'A' in current_factor_list"

  #Limit based on available
  #1000 MB = 976563 KiB
- name: (Factor A) Cap Oversized Buffer for Hardware
  set_fact:
      libpcap_buffer_size: "{{ 900 *
       ↪  ansible_facts['memory_mb']['nocache']['free']|int }}"
  when: "libpcap_buffer_size | int >  977 *
   ↪  ansible_facts['memory_mb']['nocache']['free']|int"

  #libpcap uses a 32 bit (signed?) int
  #https://github.com/the-tcpdump-group/libpcap/issues/651
  #2048 MB = 2000000 KiB
- name: (Factor A) Cap Oversized Buffer for Software
  set_fact:
      libpcap_buffer_size: "2000000"
  when: "libpcap_buffer_size | int > 2000000"
```

Factor B: Socket Queue Size

This factor is a repeat form the interface ratelimit test. Now that there is an application actually consuming packets, it may make a larger appearance.

Factor C: Backlog Poll Weight

This factor is a repeat form the Interface Ratelimit test. Likewise, this factor didn't have as big of an impact as expected in the interface test. Repeating it here may show otherwise.

Note: Originally this test had a fourth factor (RFS Table Size) which was inherited form the previous test. Later re-evaluations removed this factor with the exception of the RPi3b+ which was performed with all 4.

Factor D: RFS Table Size

This factor is a repeat form the interface ratelimit test. Now that there is an application actually consuming packets and informing the hashing algorithm which cache is hot or not for a given flow, a larger presence should be felt then previous.

4.2.2.2 Test Results

Somewhat predictably, increasing the *libpcap* buffer size (A) made a significant difference in all five tests, dropping the packets lost to zero on all three NVIDIA boards (Figures 37,38,39). The TX1 required a buffer size of 2GB before zero packets were lost, while the faster disk speed of the TX2 and XAVIER were able to keep up with only a 256 MB buffer.

The RPi3B+ (Figure 40) saw a 51% reduction in lost packets with a 512MB buffer. Despite having 1GB of ram, increasing the buffer any larger then this caused kernel instability. The RPi4's (Figure 41) most significant factor was also A, however when attempting to use any buffer larger then 256MB caused a kernel panic. It is unclear if the driver or other part of the software was responsible as no log is generated and the device becomes completely unresponsive (black screen).

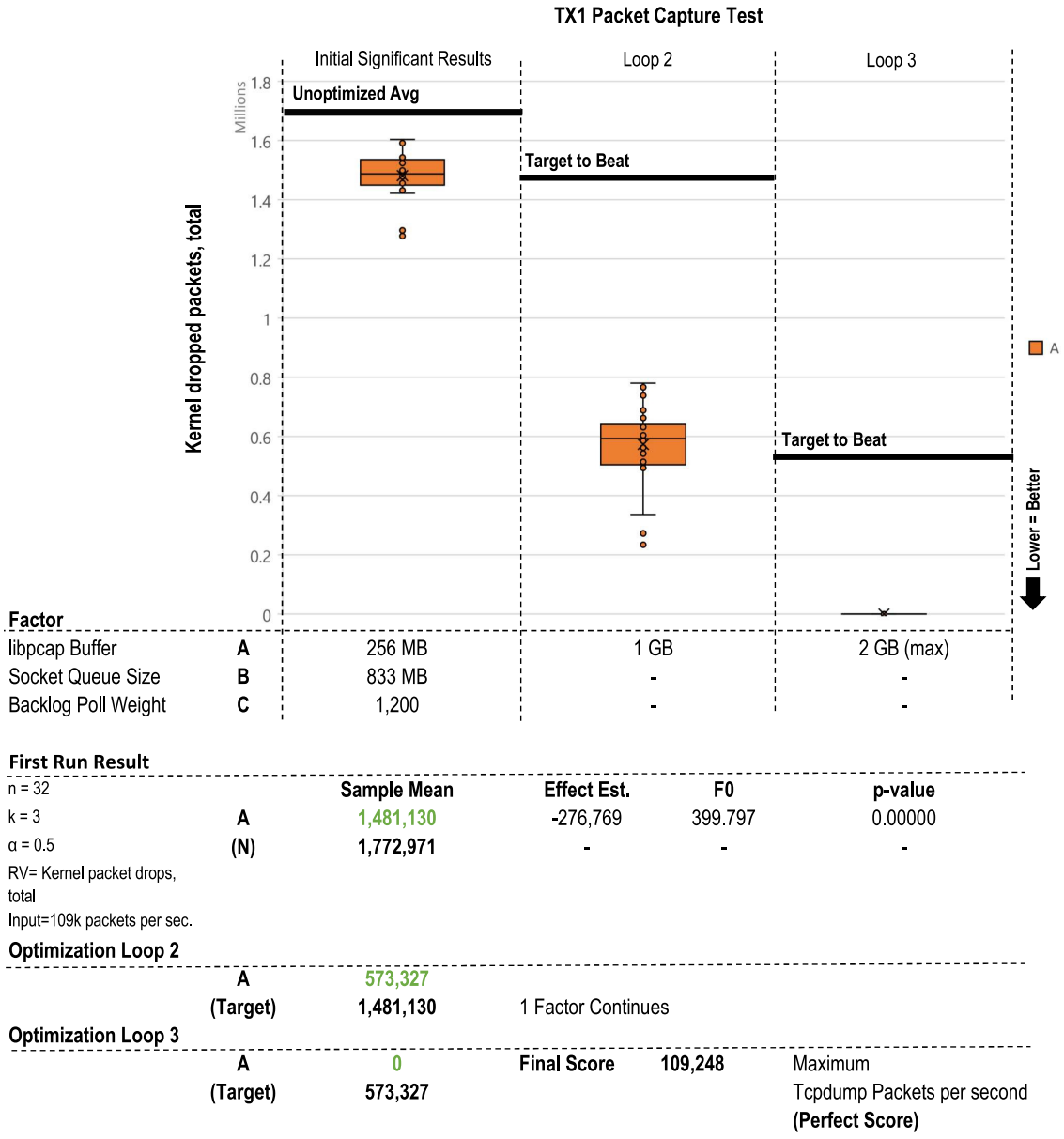
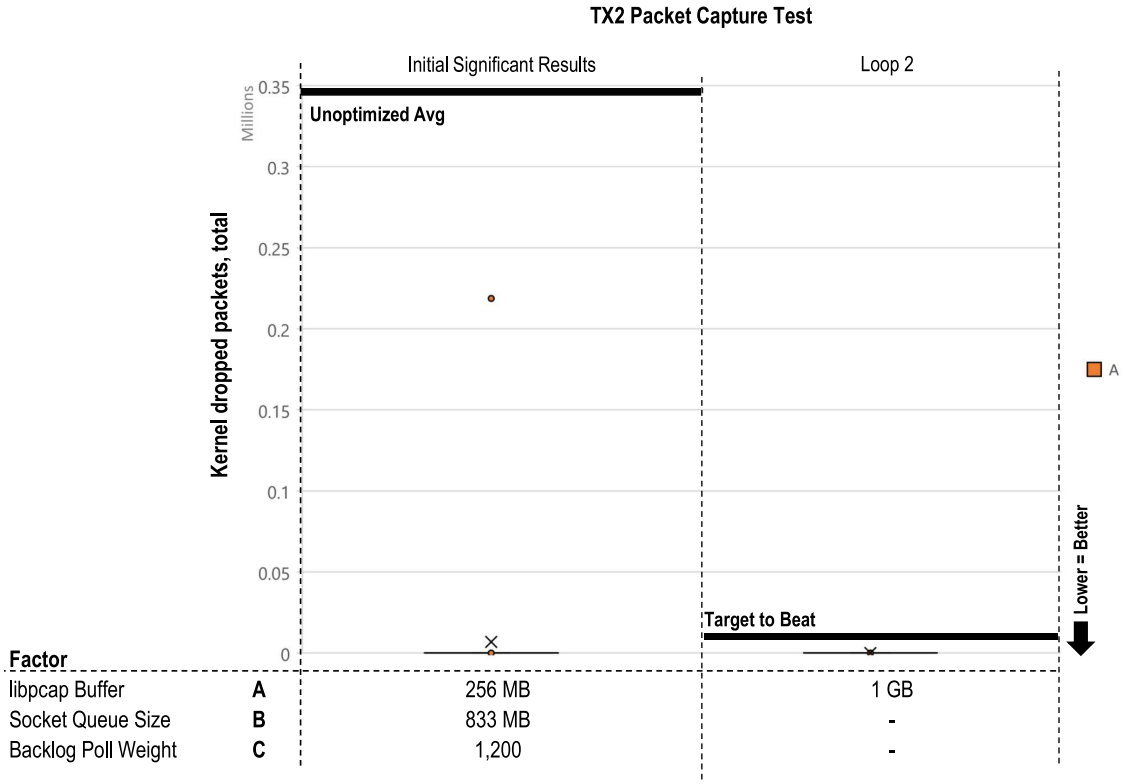


Figure 37: The TX1 required a buffer size of 2GB before zero packets were lost



Factor		Initial Significant Results	Loop 2
libpcap Buffer	A	256 MB	1 GB
Socket Queue Size	B	833 MB	-
Backlog Poll Weight	C	1,200	-

First Run Result

		Sample Mean	Effect Est.	F0	p-value
n = 32					
k = 3	A	6,835	-276,769	399.797	0.00000
α = 0.5	(N)	347,300	-	-	-

RV= Kernel packet drops, total
Input=148k packets per sec.

Optimization Loop 2

A	0	Final Score	148,000	Maximum
(Target)	6,835			Tcpdump Packets per second
				(Perfect Score)

Figure 38: The TX2 was able to keep up with only a 256 MB buffer

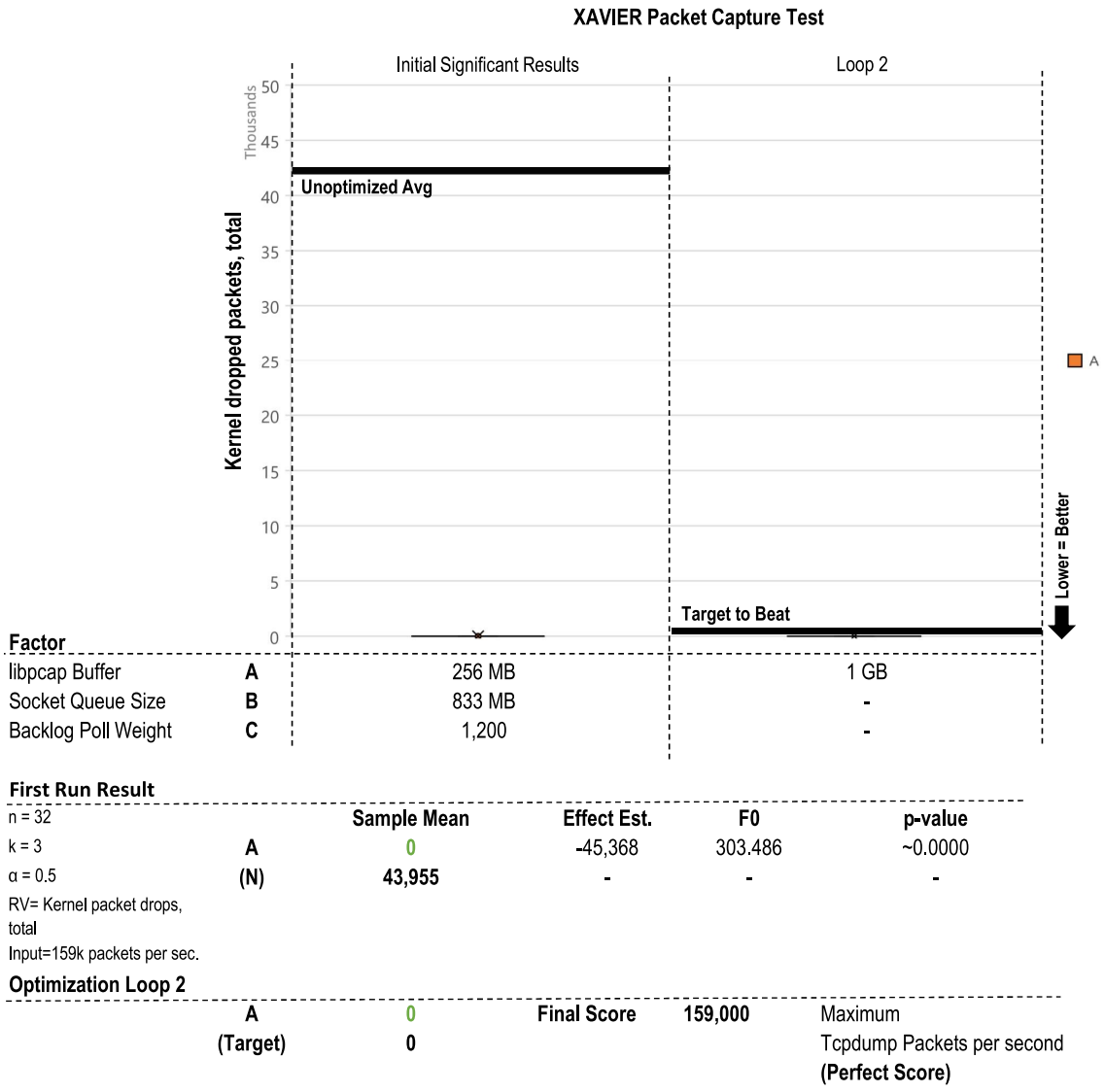
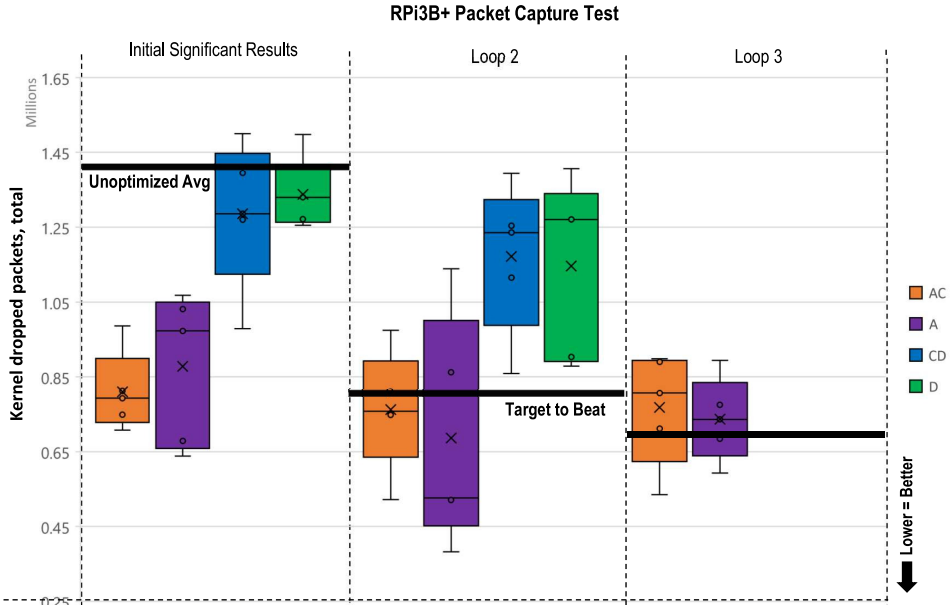


Figure 39: The XAVIER was able to keep up with only a 256 MB buffer



Factor		Initial Significant Results	Loop 2	Loop 3
libpcap Buffer	A	256 MB	512 MB (max)	512 MB (max)
Socket Queue Size	B	833 MB	833 MB	-
Backlog Poll Weight	C	1,200	4,800	32,400
RFS Table Size	D	32,768	131,072	884,736

First Run Result

n = 5		Sample Mean	Effect Est.	F0	p-value
k = 4	AC	809,556	79,526	6.893	0.01081
α = 0.5	A	877,845	-399,357	173.831	0.00000
RV= kernel packet drops,	CD	1,285,907	75,158	6.157	0.01573
Input=69k packets per sec.	D	1,338,059	66,403	4.806	0.03200
	(N)	1,342,479	-	-	-

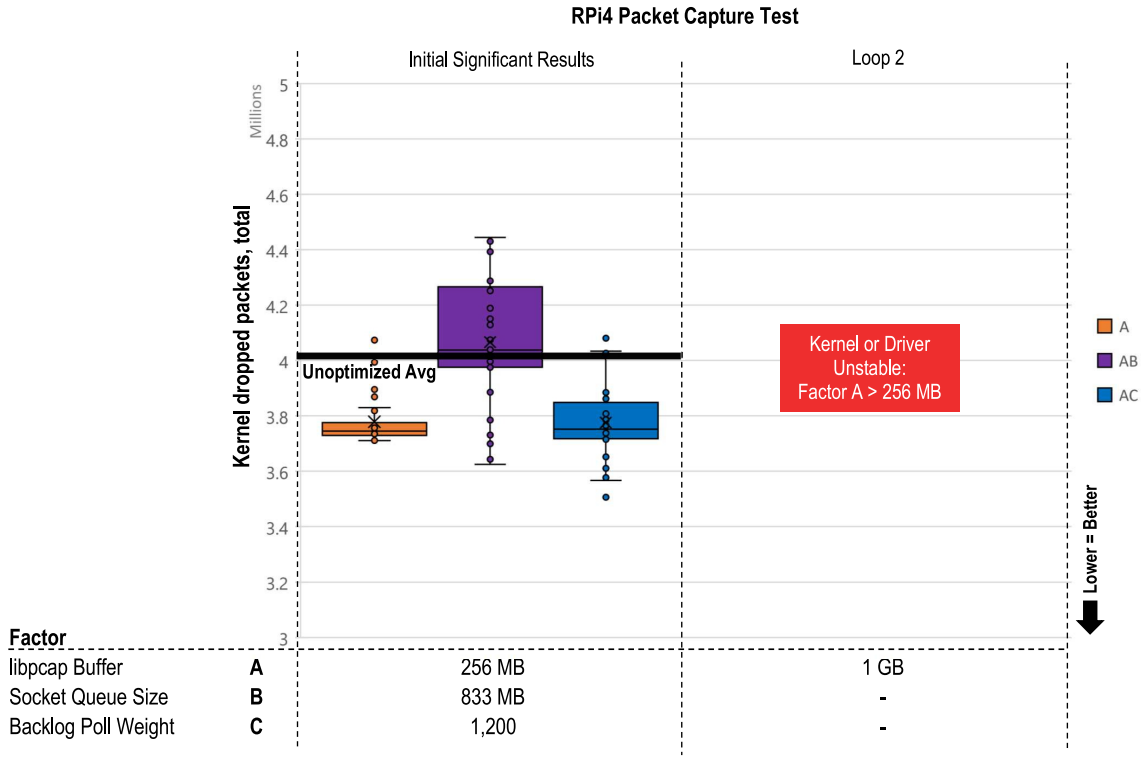
Optimization Loop 2

	AC	686,089	
A Maxed	A	762,537	
	(Target)	809,556	2 Factors Continue
	D	1,146,408	
	CD	1,171,579	

Optimization Loop 3

	(Target)	686,089	Final Score	45,173	Maximum
A Maxed	A	736,551			Tcpdump Packets per second
	AC	768,281			

Figure 40: The RPi3B+ saw a 51% reduction in lost packets with a 512MB buffer. Despite having 1GB of ram, increasing the buffer any larger then this caused kernel instability



First Run Result

n = 32		Sample Mean	Effect Est.	F0	p-value
k = 3	A	3,778,500	-198,557	113.979	0.00000
α = 0.5	BC	4,065,733	99,949	28.881	0.00000
RV= Kernel packet drops, total	ABC	3,775,759	-75,483	16.472	0.00007
Input=159k packets per sec.	(N)	4,036,603	-	-	-

Figure 41: The RPi4's most significant factor was also A, however when attempting to use any buffer larger then 256MB caused a kernel panic. It is unclear if the driver or other part of the software was responsible as no log is generated and the device becomes completely unresponsive (black screen)

4.2.3 Suricata Ratelimit Test

Live signature based detection is another of the fundamental tools in a network security sensor. While they mostly reactionary and typically wont prevent novel intrusions, they nonetheless present a hurdle to an attacker. (i.e. Re-using certain tools or exploits are likely to cause an alert) They are also typically one of the first stops on a network monitoring pipeline, after raw traffic mirroring [63] [64]. Suricata is the particular Intrusion Detection System (IDS) of choice for this test, chiefly due to its inclusion into the Air Force's Cyberspace Vulnerability Assessment / Hunter (CVA/H) network monitoring platform but also for its ability to scale across CPU cores.

4.2.3.1 Selected Factors

As discussed in Section 2.2.2, building a fair benchmark for a IDS can be quite nebulous. These tools have a large array of tunable controls that requires consideration of hardware available, with the most significant and intractable factor being the signatures enabled [104]. In Suricata the Detect phase takes up to 80% of the engines overall CPU processing effort and therefore should be limited to only packets worth inspecting [1]. The default rule set itself is plagued by poorly written community rules [105]. Future work is proposed in Chapter V that would address this large variable but for purposes of this test is outside the scope.

Static Controls

The "default" Emerging Threats ruleset is used in all Suricata tests. This ruleset contains 25,806 signatures with 20,714 enabled by default [106]. In addition by pre-processing the benign dataset control file (CIC-Monday-WorkingHours-Fixed.pcap) in pcap offline mode, 103 noisy rules were manually suppressed which greatly reduces false positive alerts in later testing. The full exact configuration files used are available

on the code repository [9]. The particular dataset packet capture replayed for this test was the *CIC-Thursday-WorkingHours-Fixed.pcap* as it contained the most alerts and best blend of attacks (Brute force, cross site scripting, SQL injection, Insider infiltration, port scanning) [85]

CIC 2017 IDS Dataset File	**Alerts Generated (No supression)**	**Alerts Generated (Suppressed)**
Monday-WorkingHours-Fixed.pcap	11,903	17
Tuesday-WorkingHours-Fixed.pcap	7,055	2,429
Wednesday-WorkingHours-Fixed.pcap	15,532	12
Thursday-WorkingHours-Fixed.pcap	29,711	2,472
Friday-WorkingHours-Fixed.pcap	8,379	240

Table 11: CIC 2017 IDS Dataset alert rates before and after suppressing noisy rules in threshold.config

The default packet ingestion engine is left as AF_PACKET. Suricata also supports other highly specialized methods of ingestion with special hardware like the Endace DAG network capture card and software like PFRING and Netmap. Typically these options would be seen in very high throughput environments where the hardware would otherwise struggle to keep up before ever making it to the application for decoding. In addition Factors A,B,E (NAPI Budget, Max Kernel Backlog, Backlog Weight) have been inherited from the previous tests as static inputs to this test.

Variable Controls

The Suricata test has four variable factors which are detailed below.

Factor A: Suricata Runmode

The Suricata engine supports two main multi-threaded operation modes. “Autofp” (auto flow pinning) mode is the default and is where the engine works in combination

with the hardware to load balance traffic. "Workers" assume all load balancing is done in the kernel or driver. The official documentation [1] recommends workers mode typically performs the best due to cache and thread coherency but other research has shown this seems to only apply to either specific hardware or precisely tuned setups [107].

Factor B: Max Pending Packets per Thread

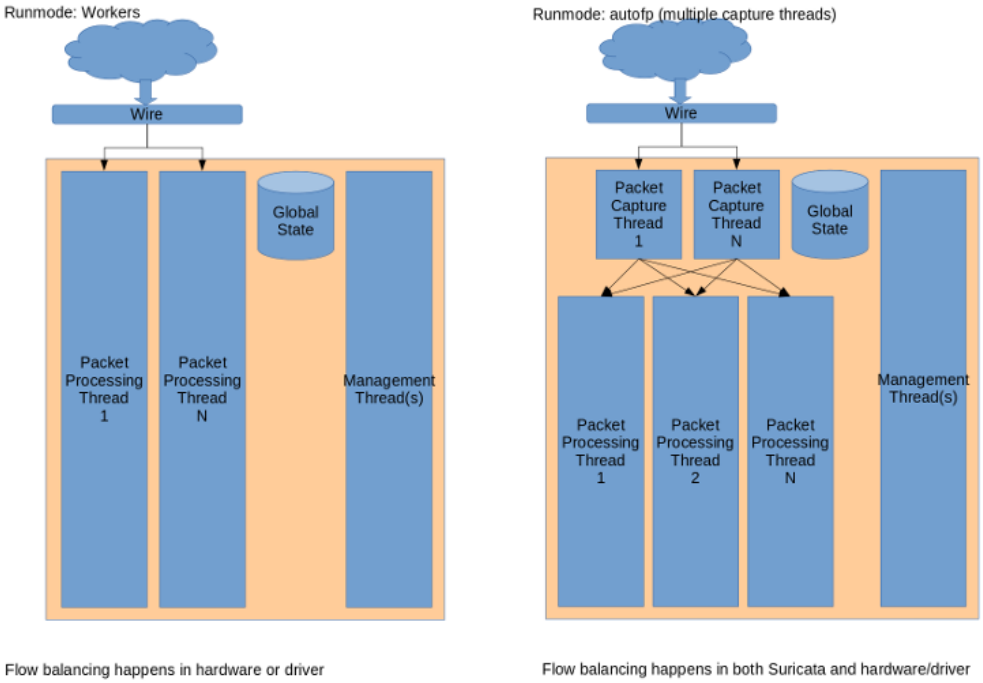
Despite the slightly misleading name the max pending packets tunable actually sets how many packets can be processed simultaneously. The higher the number the more busy the system is likely to remain at the cost of more memory. Official documentation suggests setting it no more then 60,000 as setting too high will result in cache issues [47]. The default is 1024.

Factor C: Detection Profile

By default, the Suricata engine will group similar rules for decoding based on certain aspects of the rule. (i.e. UDP vs TCP header based rules). The more groups the smaller they become and the higher the granularity of their differences, which in turn positively impacts performance at the cost of more memory. The default is "medium" or "balanced" [108].

Factor D: RFS Table Size

Once again this factor is a repeat form the interface ratelimit test and packet capture test. Sort of surprisingly this factor had little to no observable impact in the packet capture test. It is left in this test to see if it has any influence in the load balancing done in factor A (runmode).



(a) Workers Runmode (b) Auto Flow-Pin Runmode

Figure 42: Suricata Runmodes [1]

```
  ###FACTOR A###
- name: (Factor A) Set Suricata Runmode to {{ suricata_runmode }}
  lineinfile:
      path: "{{sensor_dir}}/suricata.yml"
      state: present
      regexp: '^#runmode: '
      line: "runmode: {{ suricata_runmode }}"
  when: "'A' in current_factor_list"

  ###FACTOR B###
- name: (Factor B) Set Max-Pending-Packets to {{ suricata_max_pending|int *
 ↪  loop_multiplier|int }}
  lineinfile:
    path: "{{sensor_dir}}/suricata.yml"
    state: present
    regexp: 'max-pending-packets: 1024'
    line: "max-pending-packets: {{ suricata_max_pending|int * loop_multiplier|int
     ↪  }}"
  when: "'B' in current_factor_list"

  ###FACTOR C###
- name: (Factor C) Set Detect Profile to {{ suricata_detect_profile }}
  lineinfile:
    path: "{{sensor_dir}}/suricata.yml"
    state: present
    regexp: '  profile: medium'
    line: "  profile: {{ suricata_detect_profile }}"
  when: "'C' in current_factor_list"
```

4.2.3.2 Test Results

In all test cases each device dropped a substantial amount of traffic but saw significant gains applying optimizations. The unoptimized TX1 (Figure 43) dropped 8.4 million out of the 14.1 million packets sent (60%) where after switching to workers mode (A) and increasing Max Pending Packets (B) showed a 47% reduction in dropped packets to 4 million. Likewise the TX2 (Figure 44) initially dropped 7.3 million packets out of 14.1 (51%) which after optimizations lowered to 3.8 million (48% reduction). At face value this looks like both devices performed almost identically however the time scale was not the same for both tests as it was operating at the device's interface max PPS limit. For the TX1 this meant it was sent the entire dataset packet capture in 130 seconds (14.1M / 109k) and the TX2 did the same job in 93 seconds (14.1M / 151k).

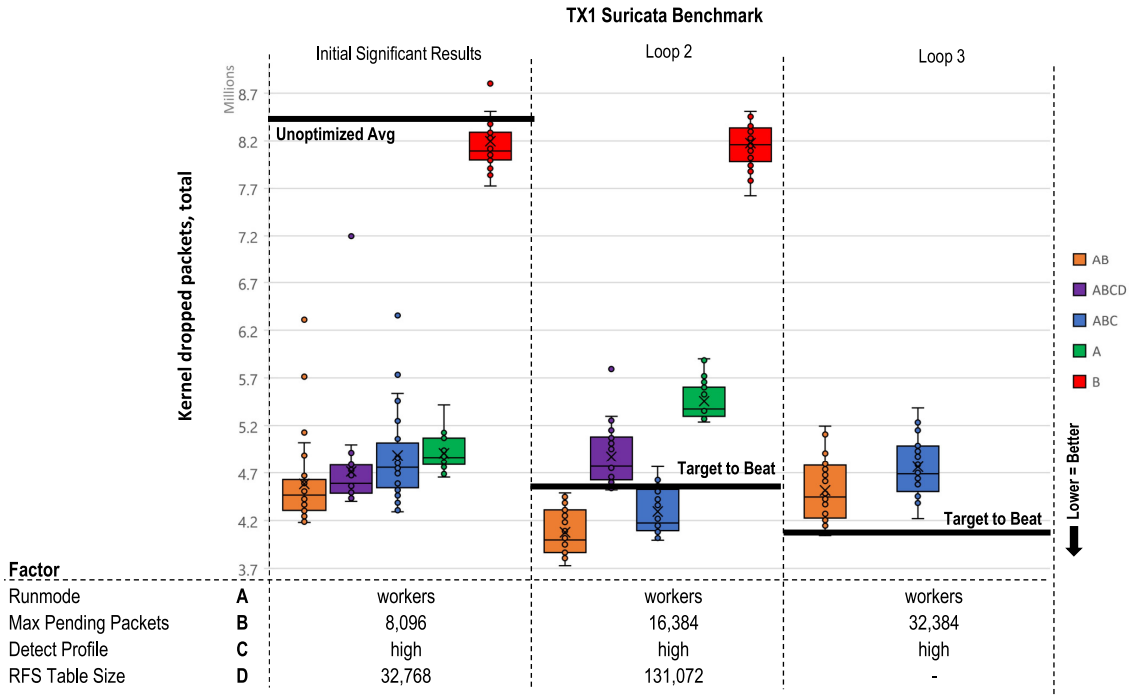
The XAVIER (Figure 45) performed the best of all devices once again but found its optimal configuration after only one iteration. Initially dropping 4.2 million packets this lowered to 2 million (53% decrease) after applying factor AB. Interestingly increasing Max Pending Packets (B) beyond the first iteration yielded worse results unlike the TX1 and TX2. As hinted in the documentation, this may be a cache contention issue. Since the XAVIER has almost twice as many cores, they likely have to compete more for the L3 and L2 caches as the number of simultaneous threads increase. A small diminishing return on B is also visible at the last iteration of the TX2 test.

The RPi3B+ (Figure 46) was unable to complete the first iteration without significant restarts and manual data compilation. The 1GB of RAM on these devices was not enough to keep the device from going unresponsive during bursts of traffic. A few iterations of tests did complete and they align with the common theme that setting the Runmode (A) to workers and increasing Max Pending Packets (B) greatly

reduced packet loss by 28% from 8.5 million to 6.2 million.

Lastly the RPi4 (Figure 47) greatly improved upon switching to Workers mode (A), showing a 44% drop (9.2 million to 5.2 million) in lost packets. Further increasing the Max Pending Packets (B) did not produce any further gains unlike previous results seen in the NVIDIA boards.

Factors C (Detect Profile) and D (RFS Table Size) appeared to have no strong independent or combination effect, only appearing on a few initial results due to the strong outlier pull of factor A.



Factor		Initial Significant Results	Loop 2	Loop 3
Runmode	A	workers	workers	workers
Max Pending Packets	B	8,096	16,384	32,384
Detect Profile	C	high	high	high
RFS Table Size	D	32,768	131,072	-

First Run Result

		Sample Mean	Effect Est.	F0	p-value
n = 32					
k = 4	AB	4,582,472	-138,228	15.223	0.00011
α = 0.5	ABCD	4,711,760	99,073	7.820	0.00537
RV= Kernel packet drops, sum	ABC	4,887,265	74,840	4.463	0.03514
Input=102k packets per sec.	A	4,971,915	-3,323,384	8799.805	0.00000
	B	8,195,639	-418,099	139.274	0.00000
	(N)	8,476,894	-	-	-

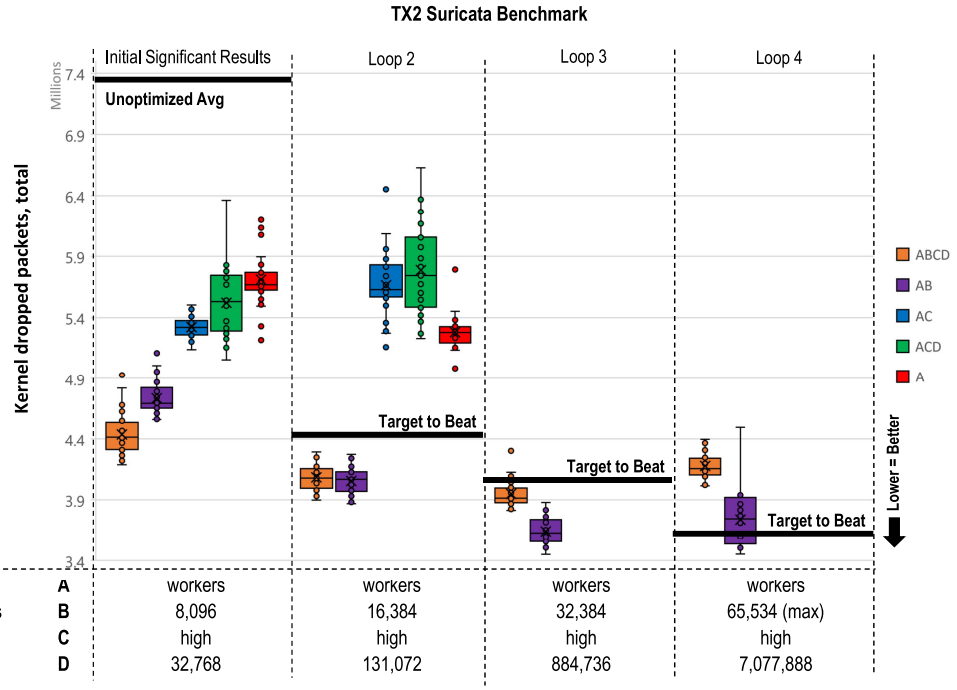
Optimization Loop 2

AB	4,076,585	
ABC	4,293,173	
(Target)	4,582,472	2 Factors Continue
ABCD	4,873,421	
A	5,456,187	
B	8,175,217	

Optimization Loop 3

(Target)	4,076,585	Final Score 72,000	Approx. Maximum
AB	4,513,611		Suricata Packets per second
ABC	4,761,656		

Figure 43: The unoptimized TX1 dropped 8.4 million out of the 14.1 million packets sent (60%) where after switching to workers mode (A) and increasing Max Pending Packets (B) showed a 47% reduction in dropped packets to 4 million

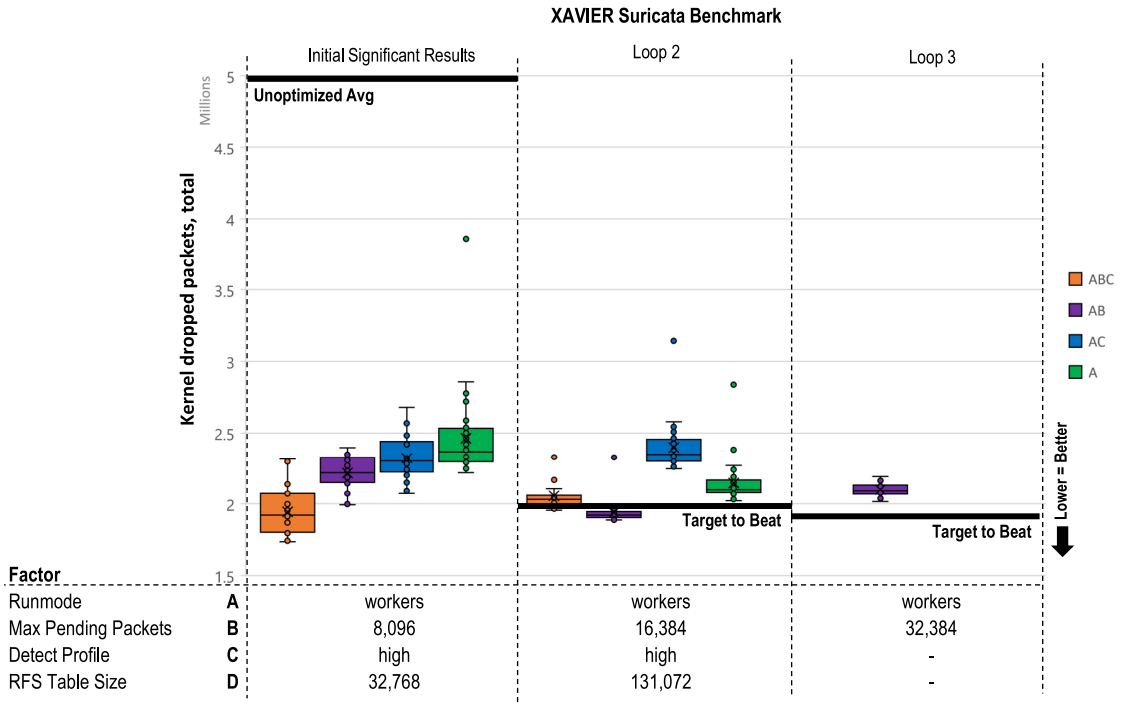


Factor					
Runmode	A	workers	workers	workers	workers
Max Pending Packets	B	8,096	16,384	32,384	65,534 (max)
Detect Profile	C	high	high	high	high
RFS Table Size	D	32,768	131,072	884,736	7,077,888

First Run Result

n = 32		Sample Mean	Effect Est.	F0	p-value
k = 4	ABCD	4,435,810	-212,742	96.422	0.00000
α = 0.5	AB	4,734,817	-624,068	829.729	0.00000
RV= Kernel packet drops,sum	AC	5,322,685	-186,318	73.958	0.00000
Input=148k packets per sec.	ACD	5,520,446	64,955	8.989	0.00285
	A	5,708,719	-2,810,762	16831.425	0.00000
	(N)	7,367,253	-	-	-
Optimization Loop 2					
	AB	4,055,161			
	ABCD	4,084,527			
	(Target)	4,435,810	2 Factors Continue		
	A	5,280,604			
	AC	5,663,832			
	ACD	5,786,426			
Optimization Loop 3					
	AB	3,640,461			
	ABCD	3,950,043			
	(Target)	4,055,161	2 Factors Continue		
Optimization Loop 4					
	(Target)	3,640,461	Final Score	112,000	Approx. Maximum
B Maxed	AB	3,737,923			Suricata Packets per second
B Maxed	ABCD	4,178,399			

Figure 44: The TX2 initially dropped 7.3 million packets out of 14.1 (51%) which after optimizations similar to the TX1 lowered to 3.8 million (48% reduction)



Factor				
Runmode	A	workers	workers	workers
Max Pending Packets	B	8,096	16,384	32,384
Detect Profile	C	high	high	-
RFS Table Size	D	32,768	131,072	-

First Run Result (Top 4)

n = 32		Sample Mean	Effect Est.	F0	p-value
k = 4	ABC	1,944,135	-84,144	10.383	0.00136
α = 0.5	AB	2,216,175	426,079	266.222	0.00000
RV= Kernel packet drops, total	AC	2,322,523	-52,084	3.978	0.04665
Input=159k packets per sec.	A	2,462,694	-2,168,029	6892.766	0.00000
	(N)	5,121,277	-	-	-

Optimization Loop 2

AB	1,937,844		
(Target)	1,944,135	1 Factor Continues	
ABC	2,055,292		
AC	2,398,690		
A	2,146,735		

Optimization Loop 3

(Target)	1,937,844	Final Score	137,000	Approx. Maximum
AB	2,099,777			Suricata Packets per second

Figure 45: The XAVIER performed the best of all devices once again. Initially dropping 4.2 million packets this lowered to 2 million (53% decrease) after applying factor AB. Interestingly increasing Max Pending Packets (B) beyond the second iteration yielded worse results unlike the TX1 and TX2. As hinted in the documentation, this may be a cache contention issue

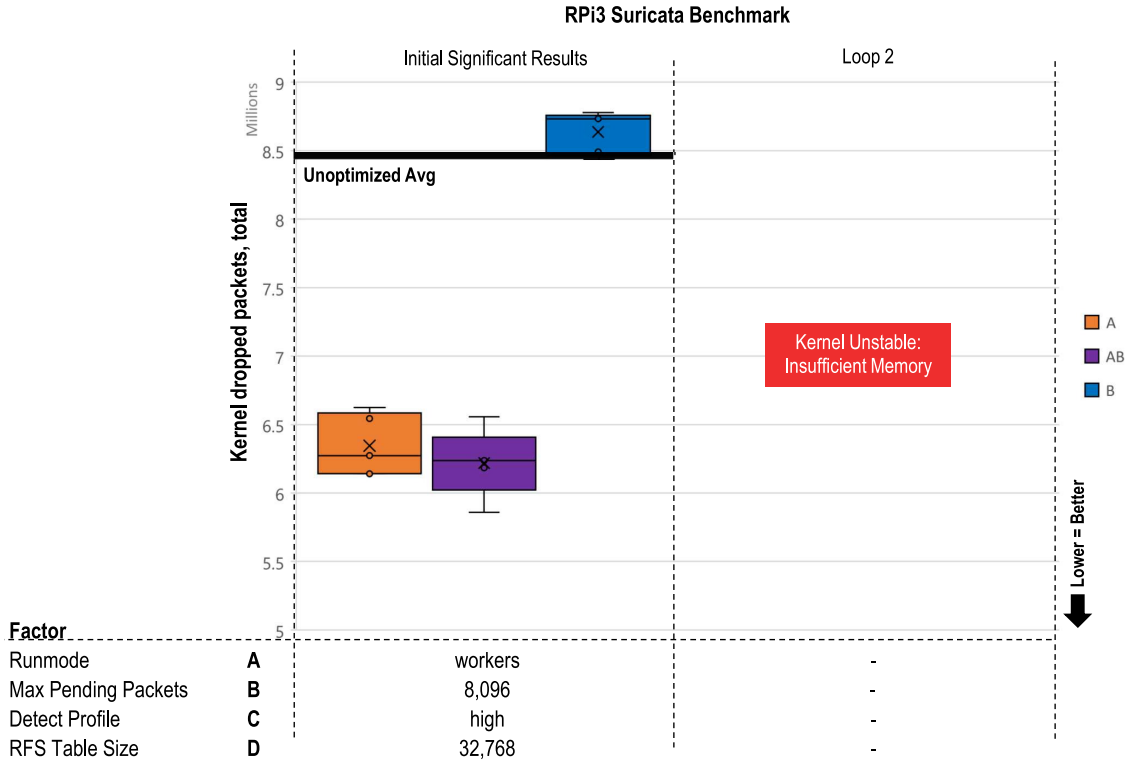
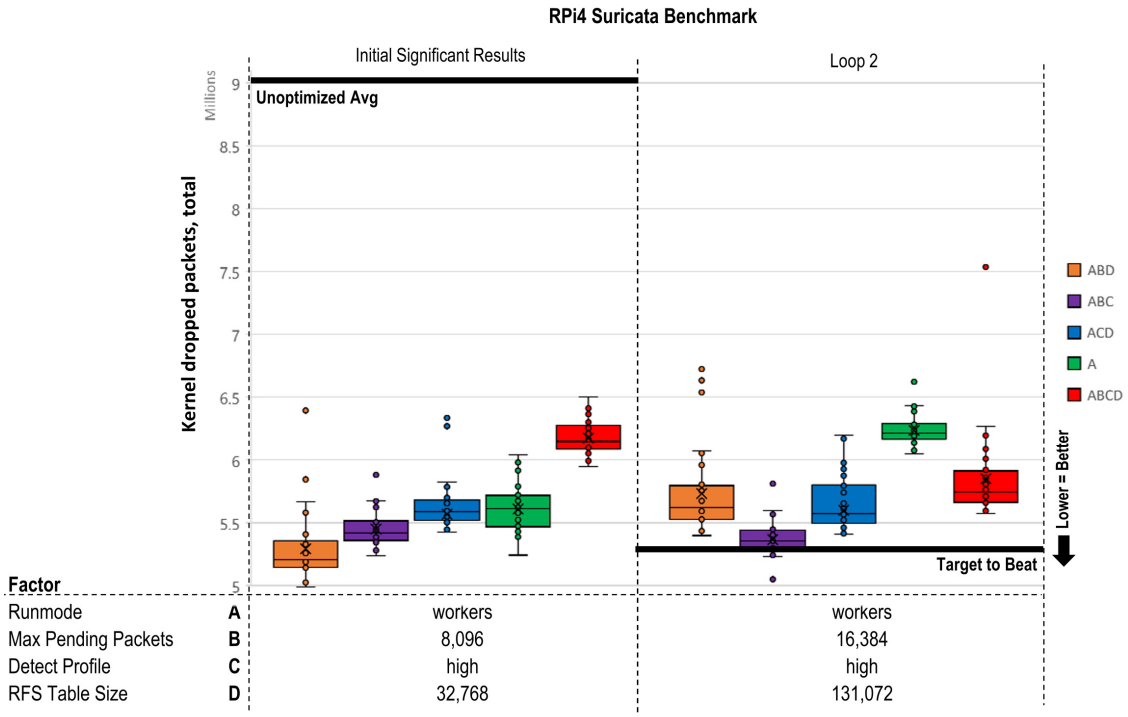


Figure 46: The RPi3B+ was unable to complete the first iteration without significant restarts and manual data compilation. The 1GB of RAM on these devices was not enough to keep the device from going unresponsive during bursts of traffic. A few iterations of tests did complete and they align with the common theme that setting the Runmode (A) to workers and increasing Max Pending Packets (B) greatly reduced packet loss by 28% from 8.5 million to 6.2 million



First Run Result (Top 5)

n = 32
k = 4
α = 0.5
RV= Kernel packet drops, total
Input=159k packets per sec.

	Sample Mean	Effect Est.	F0	p-value
ABD	5,293,205	136,133	31.707	0.00000
ABC	5,450,183	69,712	8.315	0.00410
ACD	5,568,775	-81,128	11.261	0.00085
A	5,609,595	-3,760,560	24195.518	0.00000
ABCD	6,173,994	218,088	81.376	0.00000
(N)	9,291,152	-	-	-

Optimization Loop 2

(Target)	5,293,205	Final Score	103,000	Approx. Maximum
ABC	5,366,304			Suricata Packets per second
ACD	5,416,721			
ABD	5,733,042			
AD	5,710,774			
ABCD	5,851,040			

Figure 47: The RPi4 greatly improved upon switching to Workers mode (A), showing a 44% drop (9.2 million to 5.2 million) in lost packets. Further increasing the Max Pending Packets (B) did not produce any further gains unlike previous results seen in the NVIDIA boards

4.3 Summary

We use the automated test framework described in Section 3.2 to execute three end-to-end experiments. Each test consisted of a unique network related workload with a combination of unique and shared input variables across five different devices. These selected applications not only have shown the validity of the experiment design, automation, and analysis but also have shown the feasibility of using higher end edge devices like the XAVIER, TX2 and RPi4 as a edge network sensors. The next chapter provides some conclusions with respect to the overall effort and presents multiple propositions for future work.

V. Conclusions

5.1 Overall Summary

The objective of this research was to develop a macro level methodology to optimize and benchmark specific workloads on emerging edge devices. Since these devices come in a wide range of capability and the potential workloads to evaluate on them are numerous, the developed process needed to be easy to use, modify and scale as needed. Likewise the workloads needed to be properly tuned for their specific device due to tighter performance envelopes.

The main deliverable was the implementation of an automated workflow using Ansible, an open-source automation, configuration, and application deployment tool [8]. By not requiring any dedicated agent or other prerequisite, deploying a test to a new device is trivial and the human readable task playbooks made it easy to swap workloads and associated variables. As sub-components of the workflow, a multi stage optimization process and robust analysis process ensures the test workload is maximizing the use of available resources.

Optimization is met through a multiple iteration 2^k Full Factorial experiment. This allows the operator to select up to five potential optimization factors and ensures any positive or negative interactions between any factor are appropriately captured. By attempting these factors multiple times at varying levels the "ideal" level for all chosen factors emerges (e.g. sizing a buffer in memory without under or oversizing it). The analysis process utilizes a standard Analysis of Variance (ANOVA) test to provide rigor to the results of the optimization. In cases where ANOVA fails and normality is violated, a simpler heuristic analysis examines the averages of the results.

In order to fully exercise and validate the developed process, a family of network security monitoring applications were run in an end to end scenario. Utilizing edge

devices to fill in network monitoring gaps was a key initial motivator of this research as it is a task that has been traditionally done in only large centralized deployments. The analysis of the results show the developed process met it's original design goals and intentions, with the added fact that edge devices like the XAVIER, TX2 and RPi4 can easily perform as a edge network sensor.

5.2 Future Work

Overall progress on the benchmark is complete, and can be immediately implemented using new workloads. There is still some room to polish and grow as detailed next and is suggested next. More network monitoring related tests and integration's are also proposed in Section 5.2.2 and Appendix B (Employment)

5.2.1 Benchmark Use & Expansion

5.2.1.1 ANOVA Randomness

The randomness over the overall testing is not ideal. When testing a particular factor combination, they are done sequentially (i.e. A-A-A-A-A, AB-AB-AB-AB-AB). This was done to save initial development and testing time. A simple Python script that generates an array of random factor combos could be added before the middle loops begin. This would allow for true replicate testing (i.e. A,ABC,E,B,A...) at the expense of time. Each new factor letter combo would reset the machines and reload that particular configuration.

5.2.1.2 Error Handling

The ANOVA Python script is also very sensitive to errors in the output csv.(see example in Figure 48 If for whatever reason a particular run did not finish it will break the calculations of the contrasts, which causes the entire process to stop till

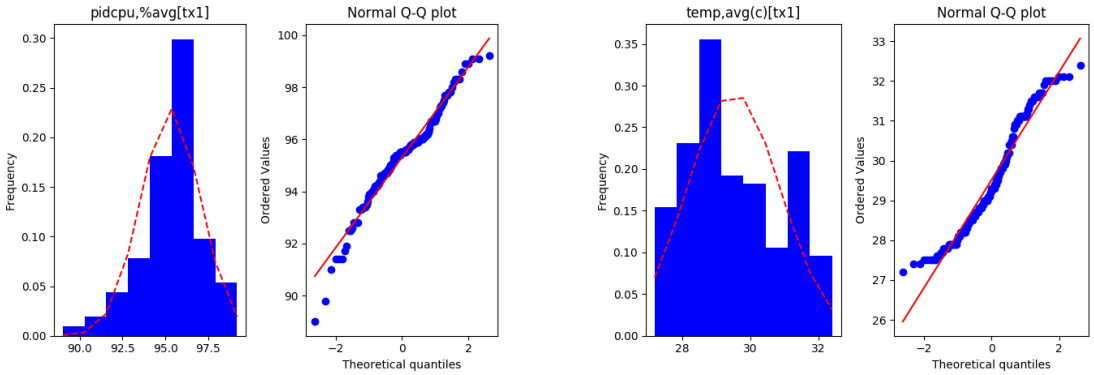
fixed. Manually entering the missing number to the csv, and then resuming with `--skip-tags initial` will resume the outer loop at the point it stopped.

T	U	V	W	X
nicdrop	nicdrop	kerndrop	kerndrop	factors
avg	%	sum	avg	code
4847	2	9047960	120639	CE
3680	1	9378719	123404	CE
1388	0	0	0	ACE
2547	1	5823112	70157	ACE
2885	1	5625612	68605	ACE
2633	1	5478996	66817	ACE
2803	1	5513588	67238	ACE
5925	3	8846967	122874	BCE
5426	2	8388790	121576	BCE
5231	2	8579631	120839	BCE

Figure 48: Example of Runtime Error (Missing Data)

5.2.1.3 Other Applications

While all the results discussed in Chapter IV are related to network applications, the metric gathering subprocess laid out in Section 3.2.3.5 supports 22 possible Response Variable (RV). More can be added to the bash script, or the current ones could be used in future workloads under research. A full example of some of the output data is proved in Appendix C.



(a) Example CPU Utilization Variable

(b) Example CPU Temperature Variable

Figure 49: Example of other response variables supported to use in other application testing

5.2.1.4 Interchangeable Modules

Both the analysis Python and metric gathering scripts are simply called by the Ansible workflow. This means with little effort they can be either modified or swapped for other scripts without impacting the core of the automation.

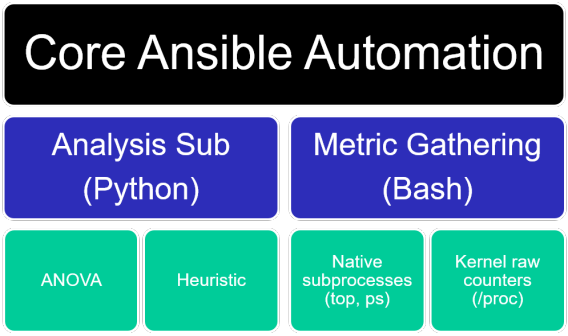


Figure 50: Block diagram showing the sub-components that can be modified or swapped without affecting the automation

5.2.2 Network Monitoring Expansion

5.2.2.1 Other Sensor Roles

The roles tested in Chapter IV were only a small slice of the possible network sensor roles. Selecting a new application to test (like netflow monitoring with Zeek/Bro) should be trivial to implement. More advanced future work, like how an edge sensor could tie into an Elastic database for visualization, or how it may be the solution to other long looming problems like monitoring encrypted traffic and detecting credential abuse are all discussed Appendix B.

5.2.2.2 Suricata Rules Evaluation

Perhaps the most significant performance variable for a signature based Intrusion Detection System (IDS) like Suricata are the signatures enabled [104]. In Suricata the Detect phase takes up to 80% of the engines overall CPU processing effort and

therefore should be limited to only packets worth inspecting [1]. Unfortunately, default rule sets are at best, good at catching low hanging very unsophisticated attacks. Thanks to their open nature it is trivial for an attacker to download the rules, inspect how they are detecting a certain payload or characteristic and modify theirs to evade it. These default rules also also plagued with poorly optimized rules which makes using them in a performance benchmark a tough decision. [105]

As an example, it is quite easy to write poor rules like:

```
alert tcp any any -> any any (pcre: "the_payload")
```

Where every single Transmission Control Protocol (TCP) packet seen by the engine will be deeply inspected using a pearl regular expression. Too many of these kind of inspections can cripple the performance of the system [109] [105]. By properly scoping the rule down to a particular destination (like a web server) the engine will move on much earlier in it's decode phase which only carries about 10-20% of the overall workload. This rule can be further enhanced by taking advantage of fast-pattern matching which furthers this filtering effort only looking at small byte chunks instead of the entire rule first [75].

It should be possible to use our benchmark to implement the methodology of [104] to test how the amount of rules and their composition affects the edge device's performance. Using this test a "max rule count" count be established for a given device.

Appendix A. Annotated Example Results Figure

Figure 51 below also provides a breakdown of how to interpret the main results figures seen in Sections 4.2.1.2, 4.2.2.2, and 4.2.3.2.

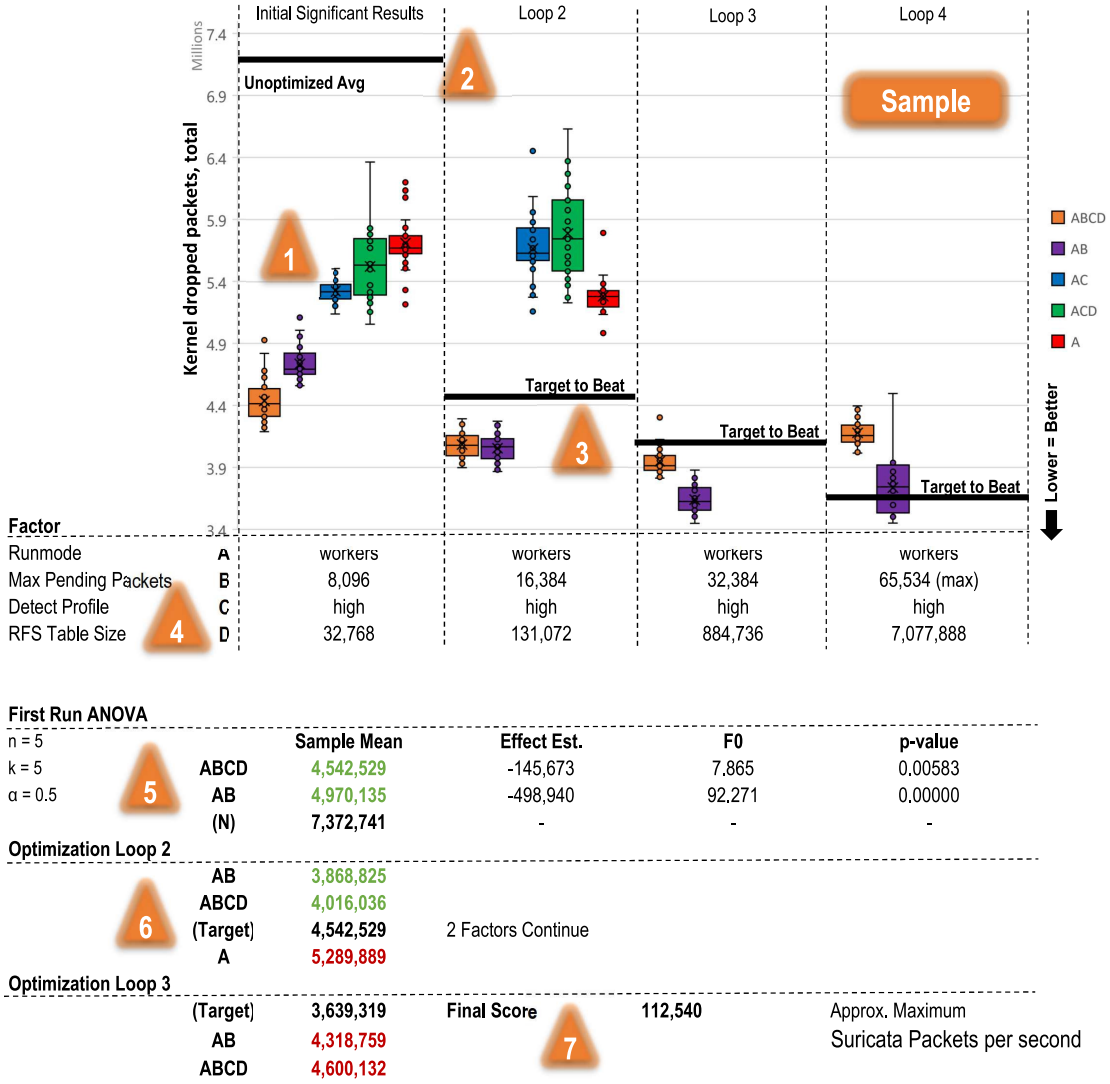


Figure 51: Highlighted example of a results figure. The explanation of each number is listed above

1. Significant results from the first test are extracted from the whole result (Example in Appendix C)
2. The unoptimized baseline average is established, and is updated based on best average of the previous loop

3. The significant factor levels are increased and are compared against the new baseline, determining if additional testing is required
4. The actual variable levels set are displayed for each "loop" column
5. The results of the first analysis test
6. The results of subsequent analysis tests
7. After the last loop shows no improvement, final score is displayed

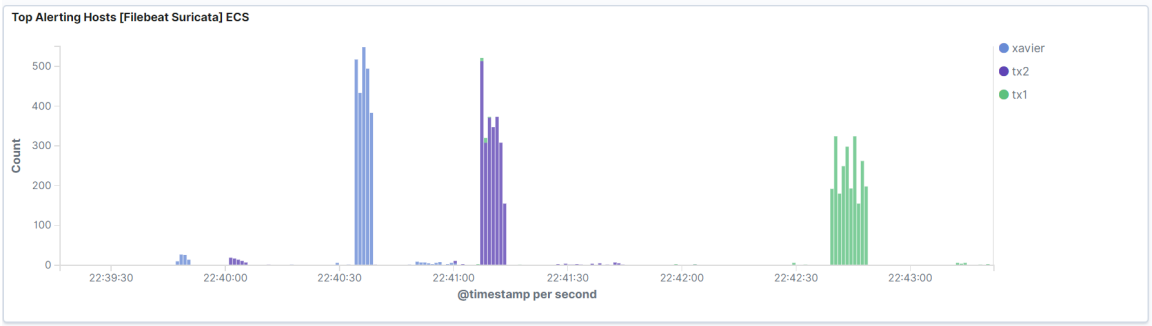
Appendix B. Employment Analysis

Contributing to Indexing & Aggregation

One last capstone test was designed to observe how well the single board computers would play together when sending their alerts back to a central Elasticsearch aggregator. For this standalone test, the 3 NVIDIA boards and the RPi4 were fed the Thursday-WorkingHours-Fixed.pcap file at their appropriate "Suricata Packets Per Second (PPS) Limit." An ARM compatible filebeat agent (log shipper) was built for each device and deployed to ingest the logs that Suricata was generating. At this small level of traffic the filebeats would just report directly back to the Elasticsearch database, instead of typical large scale deployments which go through another aggregator like Logstash or Kafka. Using the results from pretesting in Table 11 we would expect to see around 2,472 alerts on each device (logged locally) and the same amount on the aggregated back end. Table 12 below summarizes the result of this capstone test.

	Alerts Run 1			Alerts Run 2			Alerts Run 3		
	Local	**Indexed**	%	**Local**	**Indexed**	%	**Local**	**Indexed**	%
TX1	2457	2448	99.6	2445	2437	99.7	1727	1417	82.0
TX2	2518	2512	99.8	2515	2433	96.7	2489	2463	98.9
XAVIER	2541	2446	96.2	2547	2386	93.7	2542	2510	98.7
RPi4	1736	1717	98.9	1697	1697	100	1868	1868	100

Table 12: Local Logged Alerts Vs. Elastic Indexed Alerts shows only a slight disparity in a few cases



Top Alert Signatures [Filebeat Suricata] ECS

Alert Signature	Alert Category	Count
ET MALWARE Windows Microsoft Windows DOS prompt command Error not recognized	A Network Trojan was detected	7,124
ET WEB_SERVER Script tag in URI Possible Cross Site Scripting Attempt	Web Application Attack	172
SURICATA ICMPv4 unknown code	Generic Protocol Command Decode	63
SURICATA Kerberos 5 weak encryption parameters	Generic Protocol Command Decode	29
SURICATA SMB malformed request dialects	Generic Protocol Command Decode	28
ET SCAN Possible Nmap User-Agent Observed	Web Application Attack	17
ET SCAN Behavioral Unusual Port 139 traffic Potential Scan or Infection	Misc activity	9
SURICATA Applayer Mismatch protocol both directions	Generic Protocol Command Decode	7
ET POLICY Dropbox.com Offsite File Backup in Use	Potential Corporate Privacy Violation	6

Figure 52: Kibana Dashboard with all alerts reporting. The timestamps are different due to slight wall clock drift between devices

This test verified the Elastic "beat" agent is built to ensure a "at-least-once" delivery of all events it sees. This is accomplished via a registry that tracks the state of each line in each log it is tracking [110]. By essentially acknowledging each individual event it both makes sure it is not reported multiple times but also makes sure every event is processed. Due to this, despite the four devices competing for the 802.11 ad-hoc link all the events do indeed eventually make it to the index. The only exception would be if log rotation were enabled and the file was deleted before it was fully processed. This bodes well for future integration with greater platforms like the Air Force's Cyberspace Vulnerability Assessment / Hunter (CVA/H).

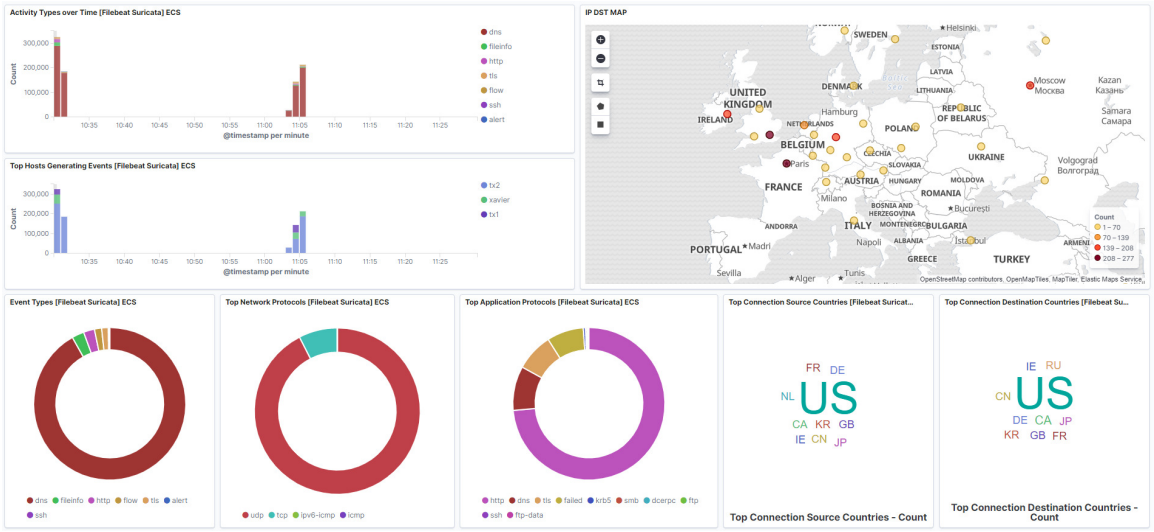


Figure 53: Integration with existing Air Force dashboards should be feasible

Encrypted Traffic Enterprises

While much of the early successes of hunt teams likely came from unencrypted network traffic based investigations (based on the tooling available at the time) the proliferation of encryption across web and domain traffic renders most traditional tools like Intrusion Detection System (IDS) ineffective [111]. At this point the only reasonable way to do threat hunting will be with very strict log and netflow coverage to observe anomalies. This means having a proper data pipeline with all gaps covered, perhaps with edge sensors on the ”last mile” of the enterprise.

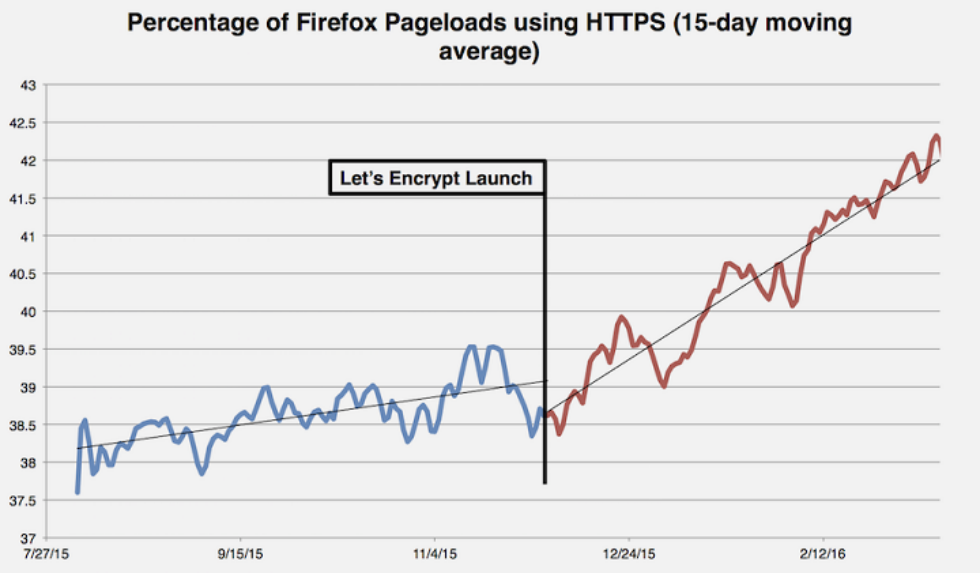


Figure 54: Rapid Increase of Encrypted Web [2]

Forged Kerberos Detection

One of the key strengths of a kerberos based authentication scheme is compartmentalizing authentication to a trusted third party. In the case of Windows Active Directory the Domain Controller (DC) acts as the master or Key Distribution Center (KDC). This protects credentials from being fully exposed should a part of the domain become compromised. It does however create a potential single point of failure should the KDC ever become compromised. Figure 55 below shows a high level overview of a client (grey) requesting a service from the server (black). The Ticket to Grant Tickets (TGT) is the client's initial authentication into the domain, from which they can request further access to other resources like a file share or remote terminal from the Ticket Granting Service (TGS). These tickets are meant to be time limited and point to point constrained tokens that protect against credential theft and Main-in-the-Middle (MITM) attacks [3] [112]. A few design choices (stateless tickets and omitting the last "trust but verify" handshake) in the Windows Active Directory implementation of the protocol however enables three attacks that can be

devastating for an entire enterprise.

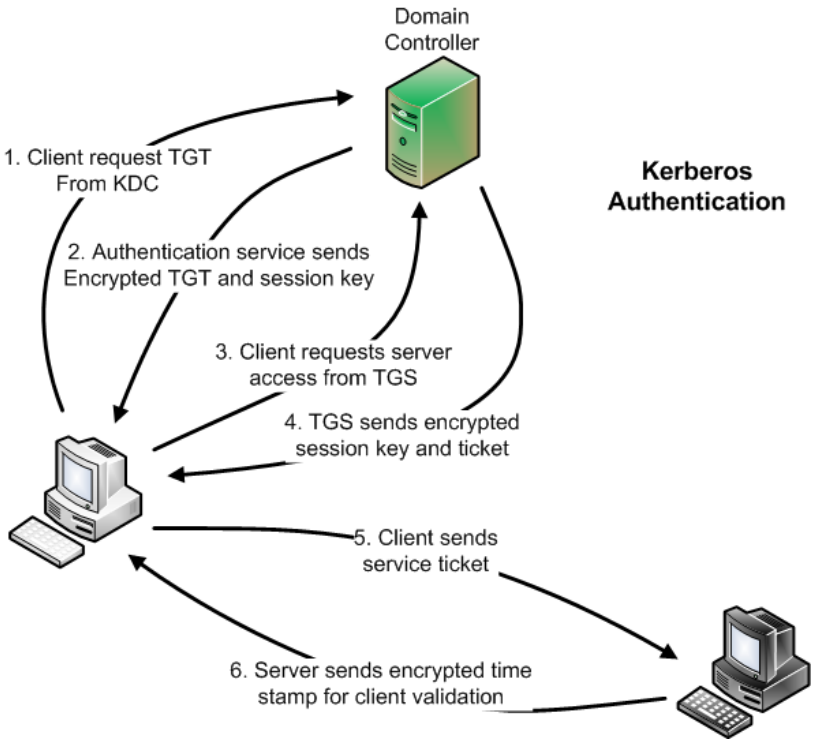


Figure 55: Kerberos Authentication in a Windows Domain[3]

Golden Ticket Attack

The TGT is encrypted by the password hash of a service account called krbtgt on the KDC. Should this password hash be exposed an attacker could forge TGTs as any user in any group on the domain, as the KDC uses the krbtgt hash as a litmus test of is this request legitimate. Since the KDC does not track states it does not mind that steps 1 and 2 above were skipped, and will give the attacker the service ticket they desire. This attack would be evident by looking for the absence of the first two steps in logs or traffic going to the DC. [3] [113] [114]

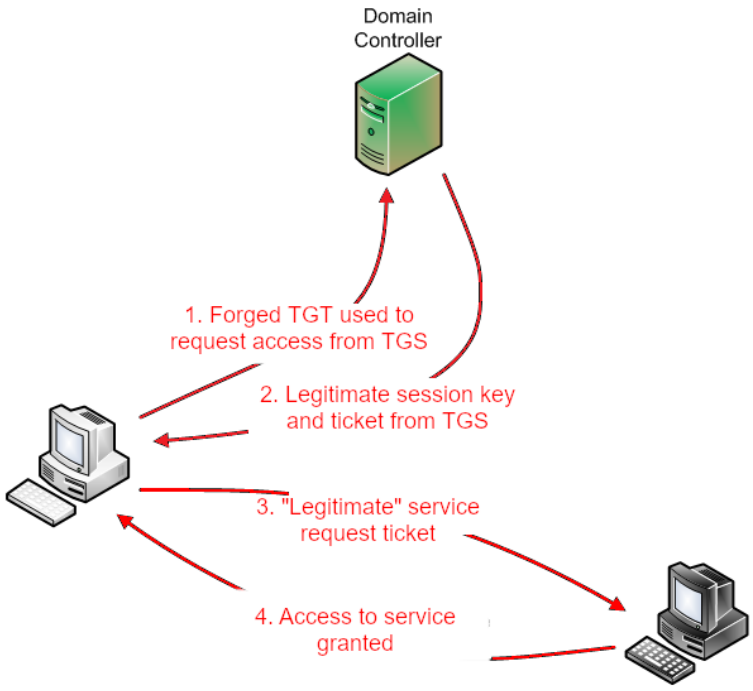


Figure 56: Golden Ticket Attack

Silver Ticket Attack

Individual services are protected by their own service account hash which is used in the generation of the TGS. Should this particular hash be exposed (which can be assumed likely if the main krbtgt hashes were as well), an attacker could forge service tickets as any user in any group on the domain. The service uses its own hash as a litmus test of is this request legitimate and since the service is not aware of any state on the DC it has no idea that steps 1-4 were skipped and assumes the request is okay. This attack while more limited compared to the golden ticket attack is considerably more difficult to detect as no logs appear on the DC. It would only be evident by looking for the absence of steps 3 and 4 in logs or traffic going to the DC compared to what was observed in logs and traffic going to the service in steps 5 and 6. [114] [113]

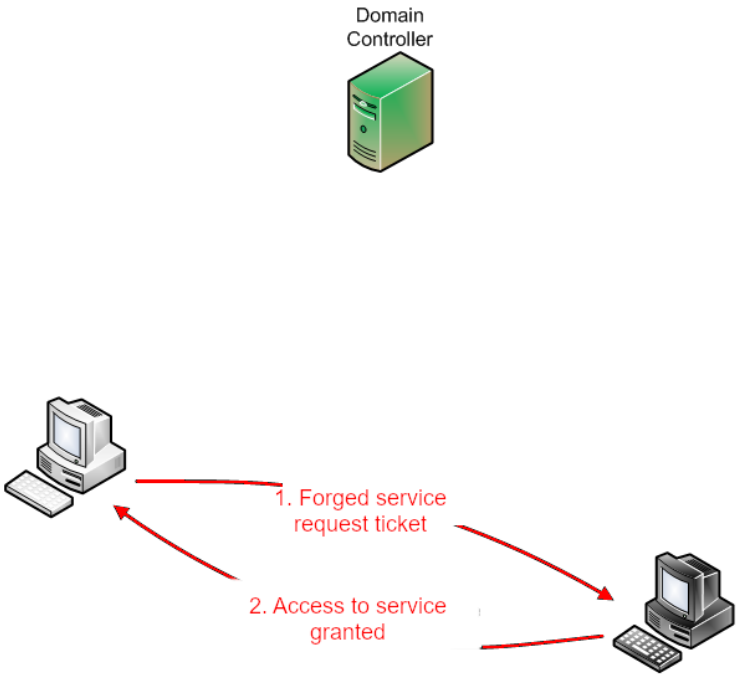


Figure 57: Silver Ticket Attack

Pass-The-Ticket Attack

Pass the ticket is similar to the Golden and Silver attack minus the forging step. Instead, the adversary just steals a valid ticket out of memory and re-uses it. These tickets have a limited lifetime but depending who they stole it from may allow them to elevate even further. The attack signatures will be very similar to the other two attacks as well. [114] [115]

Getting access to the critical logs in a hunt scenario can be administratively challenging and may even tip off the presence of a hunt team to a cautious adversary. The logs themselves may even be manipulated by a deeply persistent threat, as has been demonstrated by multiple Advanced Persistent Threat (APT) [53] [116] [117]. The only remaining method to detect this attack would then require monitoring the network traffic between every host, something large centralized boundary sensors are

likely not doing. By employing a network of edge sensors at these lower levels, these lateral movement anomalies may finally become visible in the network traffic alone.

Appendix C. Raw Data Examples

Full data is available under the "Results" folder on the GitHub Repository [9]. Figure 58 below shows a full box plot example from the interface rate test. Table 13 afterwards is an example of the full data available from the metric gathering subprocess. Some columns were removed to make it fit on a page.

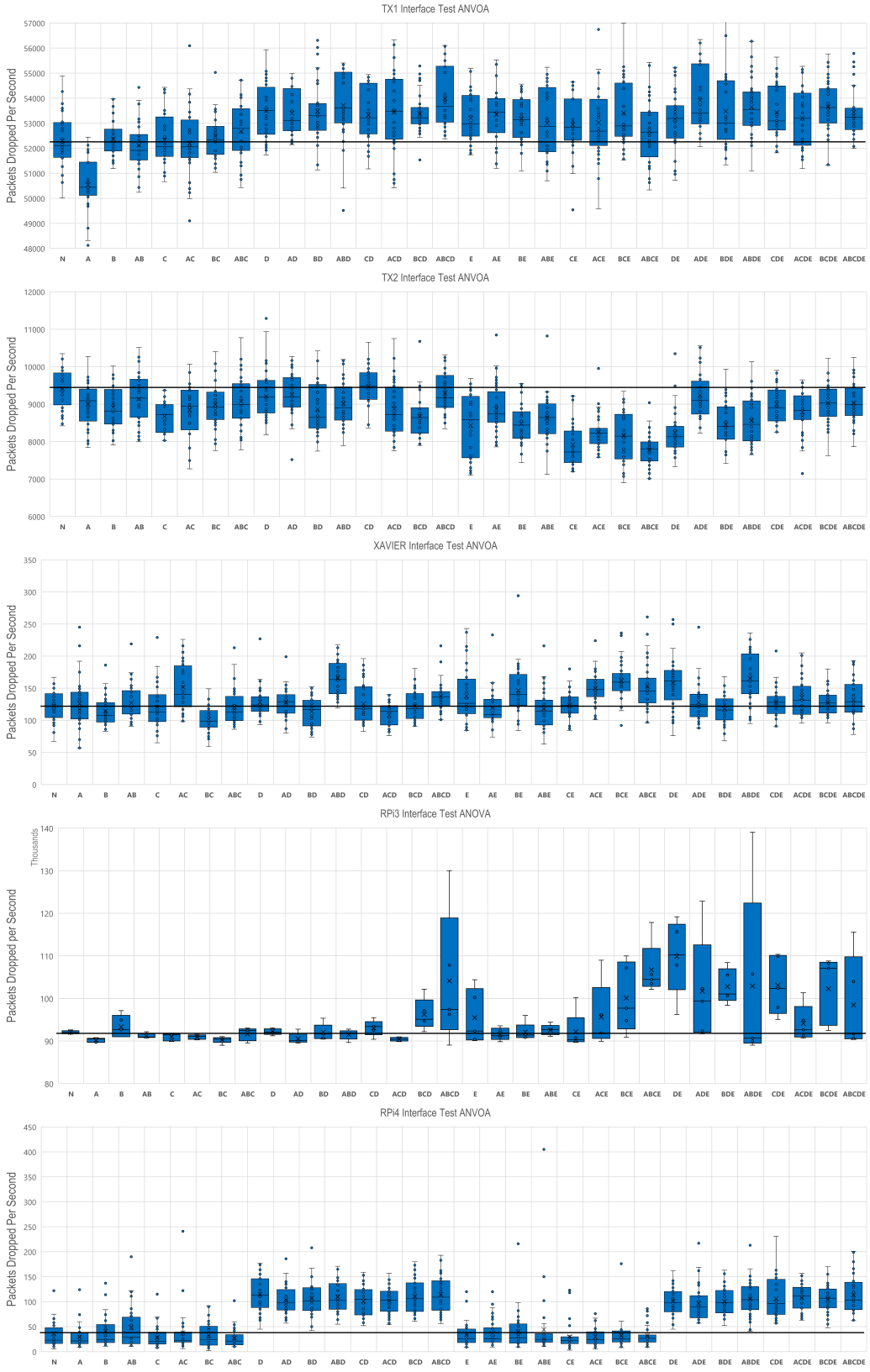


Figure 58: Full Interface Test ANVOA Boxplots

Table 13: Sample of the output of the metric gathering subprocess. Some columns removed to make it fit on page

Input pps	**pid cpu avg**	**pid cpu max**	**sys cpu avg**	**sys cpu max**	**sys MB min**	**temp avg (c)**	**power avg (w)**	**power max (w)**	**rxpps avg**	**rxmbps avg**	**nic drop avg**	**nic drop %**	**test factor code**
160k	90.9	97	27.1	74.2	6865	36.2	5.471	8.857	138572	841	10030	6	N
160k	87.6	96	29.8	82.6	6706	36.4	5.733	8.515	146086	852	9213	5	N
160k	88.8	96	23.9	30.9	6694	36.1	5.243	5.859	145082	849	10082	6	N
160k	87.6	96	20.9	28.7	6711	35.8	4.97	6.087	144742	845	10391	6	N
160k	88.8	96	23.6	32	6707	35.8	5.207	6.087	145118	849	9865	6	N
160k	89.8	97	25.1	49.3	6936	36.6	5.326	7.34	137180	827	10384	6	A
160k	86.3	95	34.7	94.8	6746	36.8	6.007	9.538	142017	856	8670	5	A
160k	89.3	96	23.6	29.8	6715	36.2	5.222	5.783	138093	834	10187	6	A
160k	88.3	95	20.9	27.4	6737	35.7	4.973	5.516	142161	833	9760	6	A
160k	89.1	98	22.8	26	6738	35.7	5.125	5.326	140380	823	10693	6	A
160k	89.3	98	25.3	53.5	6956	36.7	5.344	7.378	136429	825	9941	6	B
160k	86.5	99	31.2	83.7	6737	36.8	5.779	9.348	137915	802	9373	6	B
160k	87.8	98	26.5	93	6711	36.3	5.358	9.123	138919	836	11034	6	B
160k	90.8	95	26.3	87.5	6722	36.1	5.346	9.234	144427	844	9963	6	B
160k	88.4	96	25.5	82.6	6713	36	5.306	8.553	140164	846	10570	6	B
160k	87.4	100	24.9	45.6	6955	36.5	5.326	6.731	139552	809	11747	7	AB
160k	87.5	99	37.9	68.1	6730	37	6.323	7.831	142331	861	8220	5	AB
160k	90.8	97	24.4	36.7	6687	36.2	5.226	6.465	142718	838	10843	7	AB
160k	88.8	98	21.1	25.9	6690	35.9	5.003	5.516	141765	828	10683	6	AB
160k	89.9	99	23.2	33.9	6689	35.8	5.164	6.049	141876	830	10963	6	AB
160k	89.7	99	27.3	98.9	6875	37.1	5.422	9.085	145071	851	10005	6	C
160k	91.3	98	28.1	45.1	6714	36.9	5.614	7.15	140515	852	8649	5	C
160k	90	98	23.6	32.3	6680	36.4	5.207	5.859	145704	856	9295	5	C
160k	90.1	100	20.3	26.4	6688	36	4.932	5.859	138080	831	10722	6	C
160k	92.3	100	23.1	26.1	6684	35.9	5.145	5.516	143721	843	10329	6	C

Table 13 continued from previous page

Input pps	pid cpu avg	pid cpu max	sys cpu avg	sys cpu max	sys MB min	temp avg (c)	power avg (w)	power max (w)	rxpps avg	rxmbps avg	nic drop avg	nic drop %	test factor code
160k	89.4	99	24.6	48.4	6944	36.8	5.319	7.264	141630	828	10318	6	AC
160k	88.7	97	37.7	71.9	6760	37.2	6.284	7.907	146220	860	7866	5	AC
160k	87.8	100	24.3	42.9	6744	36.4	5.252	6.731	144167	845	10980	6	AC
160k	88.2	98	20.7	26.6	6757	35.9	4.958	5.478	139657	845	10793	6	AC
160k	88.7	96	23.1	28.8	6753	35.9	5.136	5.44	138402	836	10457	6	AC
160k	90.1	99	25.3	51	6934	36.7	5.324	6.617	137566	829	10586	6	BC
160k	86.1	100	31.3	93.6	6701	37	5.79	9.5	140975	816	9791	6	BC
160k	89	98	23.7	30.4	6687	36.3	5.208	5.783	143175	837	9393	6	BC
160k	88.8	96	21	26.6	6697	36	4.966	5.859	137878	806	9899	6	BC
160k	87.7	100	23.3	29.1	6689	35.9	5.167	6.087	140275	815	11097	7	BC
160k	87.8	99	25.8	58.1	6954	36.5	5.382	7.682	144838	844	10184	6	ABC
160k	86.5	98	31.3	58	6754	36.6	5.873	7.907	142464	860	8225	5	ABC
160k	86.6	99	23.7	29.5	6733	35.9	5.212	5.478	137365	800	9313	6	ABC
160k	88.2	94	21.1	28	6738	35.6	4.993	5.669	139880	844	10465	6	ABC
160k	87.1	99	24	46.1	6741	35.5	5.193	6.579	140879	826	9448	6	ABC
160k	86.2	100	41.2	98.7	7021	36.4	6.254	9.158	142367	829	12425	7	D
160k	86.7	99	36.5	67.5	6801	35.8	6.242	8.211	136311	795	10526	6	D
160k	87.8	98	24.6	42.9	6735	35	5.29	6.617	143661	841	11615	7	D
160k	85.6	100	22.7	45.5	6723	34.5	5.088	6.427	143049	822	15129	9	D
160k	89	98	23.6	31.5	6736	34.2	5.21	5.631	140375	851	11316	6	D
160k	89.8	97	25	55.4	6899	34.8	5.277	6.845	144250	846	10555	6	AD
160k	88	96	34.4	61	6705	35.1	6.045	8.097	144413	850	7989	5	AD
160k	88.7	97	24.5	42.5	6699	34.3	5.237	6.579	137309	805	9877	6	AD
160k	87.3	99	21	26.9	6698	33.8	4.943	5.326	140207	815	11129	7	AD
160k	0	0	23	26	6695	33.8	5.139	5.44	140557	847	10200	6	AD
160k	92.4	100	26	51.1	6908	34.5	5.326	7.34	142959	834	11107	6	BD

Table 13 continued from previous page

Input pps	pid cpu avg	pid cpu max	sys cpu avg	sys cpu max	sys MB min	temp avg (c)	power avg (w)	power max (w)	rxpps avg	rxmbps avg	nic drop avg	nic drop %	test factor code
160k	88	99	35.7	89.5	6701	34.8	6.062	8.743	141363	828	9357	6	BD
160k	90.2	98	24.9	43.2	6676	34	5.312	6.693	144150	848	10690	6	BD
160k	88.5	100	21.1	27.2	6692	33.5	4.98	5.669	142055	826	11709	7	BD
160k	88.5	100	23.1	26.1	6680	33.4	5.113	5.326	141667	826	11247	7	BD
160k	92.6	97	25.6	42.4	6946	33.9	5.321	6.769	138534	839	11215	7	ABD
160k	87.9	98	37.7	66.5	6719	34.3	6.281	7.983	138429	809	8630	5	ABD
160k	89.4	95	24.6	43.5	6695	33.5	5.221	6.465	141302	826	9525	6	ABD
160k	89.9	100	22	26.7	6715	33	5.019	5.669	139768	816	11427	7	ABD
160k	89.1	100	23.5	33	6708	33	5.123	6.049	135499	821	11226	7	ABD
160k	92	96	25.8	47.9	6931	33.7	5.352	6.807	142657	835	10323	6	CD
160k	87.1	97	30.6	51.3	6708	33.8	5.769	7.831	137329	798	9844	6	CD
160k	89.3	100	24.4	37.4	6682	33.1	5.188	6.465	136153	819	10526	6	CD
160k	90.2	96	20.9	26	6693	32.7	4.943	5.555	144260	847	10891	6	CD
160k	90	98	23.2	26	6691	32.6	5.1	5.745	138868	840	11045	6	CD
160k	90.9	98	26.6	96.1	6900	33.7	5.346	9.082	141098	827	10965	6	ACD
160k	87.4	95	29	44.9	6711	33.7	5.663	6.959	143214	836	7879	5	ACD
160k	89.9	98	23.8	32.1	6672	33.1	5.18	6.315	140960	823	10357	6	ACD
160k	87.9	100	20.5	27.4	6679	32.9	4.904	5.593	141556	829	10116	6	ACD
160k	88.5	97	23.1	25.9	6674	32.9	5.127	5.516	140673	851	10368	6	ACD
160k	90.8	96	25	37	6883	34	5.378	6.277	143080	838	9185	5	BCD
160k	89.6	99	29	44.8	6708	33.9	5.679	7.073	144717	842	9252	5	BCD
160k	91.1	98	23.5	29.8	6702	33.3	5.235	5.821	141538	833	9831	6	BCD
160k	90.8	97	20.4	27.7	6697	33	4.942	5.783	137048	803	10088	6	BCD
160k	91.1	100	23.4	31.5	6699	32.8	5.223	5.897	138100	836	9719	6	BCD
160k	89.4	98	40.4	99.9	6989	34	6.063	9.234	135421	816	11620	7	ABCD
160k	88.4	100	36.4	68.2	6768	33.5	6.125	7.907	143158	835	9979	6	ABCD

Table 13 continued from previous page

Input pps	pid cpu avg	pid cpu max	sys cpu avg	sys cpu max	sys MB min	temp avg (c)	power avg (w)	power max (w)	rxpps avg	rxmbps avg	nic drop avg	nic drop %	test factor code
160k	89.5	97	24.3	35	6708	33	5.225	6.617	141900	832	10536	6	ABCD
160k	87.1	96	24.2	74.9	6703	32.9	5.116	7.831	141578	828	10943	6	ABCD
160k	89.2	97	23.5	31.8	6717	32.6	5.141	6.087	141354	828	10261	6	ABCD
160k	87.7	94.1	43.7	99.9	7008	33.8	6.401	9.234	139736	849	10743	6	E
160k	88.3	97	39	79.8	6770	33.8	6.37	8.287	143220	840	8460	5	E
160k	89.4	98	24.3	38.7	6721	33.1	5.273	6.43	137566	807	9940	6	E
160k	88.6	98	21.5	25.7	6735	32.7	4.999	5.402	140407	819	10755	6	E
160k	89.3	98	23.1	26.3	6722	32.5	5.17	5.44	145062	847	10930	6	E
160k	91.6	97	24.5	31.2	6879	33.7	5.318	6.315	144978	850	9004	5	AE
160k	89	97	27.1	45.4	6716	33.8	5.517	7.111	141959	861	8560	5	AE
160k	88.5	95	23.6	30.1	6687	33.2	5.225	5.821	142276	831	8901	5	AE
160k	89.3	100	20.3	26.3	6696	32.8	4.89	5.478	142771	830	11454	7	AE
160k	90.7	96	23	26	6695	32.9	5.169	5.516	135422	818	8755	5	AE
160k	90.8	95	27.6	93.8	6833	33.9	5.42	8.705	142993	840	8711	5	BE
160k	91.2	97	27.6	44.3	6688	33.7	5.535	7.073	145007	853	8700	5	BE
160k	91	96	23.3	28.7	6682	33.2	5.146	5.859	143401	840	9971	6	BE
160k	89.9	97	19.8	25.3	6688	32.8	4.857	5.555	137536	831	9657	6	BE
160k	88.8	97	23.4	26.4	6678	32.6	5.127	5.669	137824	804	10030	6	BE
160k	88.5	96	30.7	92.1	6872	33.9	5.66	9.199	145922	852	9618	5	ABE
160k	92	98	28.5	45.2	6685	33.7	5.616	7.226	145494	856	9151	5	ABE
160k	88.2	99	23.6	31	6668	33.2	5.155	5.593	142295	828	10643	6	ABE
160k	89.4	100	20.5	27	6685	32.8	4.899	5.326	142596	835	10768	6	ABE
160k	87.1	100	23	26.1	6675	32.9	5.122	5.326	134978	806	12052	7	ABE
160k	89.3	96	42.3	99	6990	34.3	6.252	9.31	136484	825	10025	6	CE
160k	86.7	94	36.8	70.9	6766	33.9	6.238	7.834	138467	808	8693	5	CE
160k	90.1	97	24.8	38.3	6729	33.4	5.291	5.973	142915	839	9442	6	CE

Table 13 continued from previous page

Input pps	pid cpu avg	pid cpu max	sys cpu avg	sys cpu max	sys MB min	temp avg (c)	power avg (w)	power max (w)	rxpps avg	rxmbps avg	nic drop avg	nic drop %	test factor code
160k	88.5	98	21.5	26.9	6735	33	5.008	5.821	135846	818	10650	6	CE
160k	89	96	23.3	30.2	6739	33	5.185	5.478	141093	824	10108	6	CE
160k	90.9	99	24.9	49.6	6939	34	5.318	6.579	143640	842	9758	5	ACE
160k	90.2	95	32.1	86.4	6710	34.3	5.842	8.667	141161	856	7927	4	ACE
160k	88.9	100	24.4	42.8	6693	33.6	5.258	6.731	143146	837	10051	6	ACE
160k	89	96	21.4	30.8	6702	33.2	4.975	6.163	137321	805	10142	6	ACE
160k	88.7	97	23.3	26	6706	33.4	5.145	5.859	140053	847	10175	6	ACE
160k	90.4	95	29.9	89.3	6799	34.4	5.6	9.237	137226	806	8856	5	BCE
160k	0	0	27	95.2	6626	34.3	5.397	9.196	141055	822	10082	6	BCE
160k	87.9	100	23.1	27.4	6625	33.9	5.144	5.631	143917	840	9420	6	BCE
160k	88.2	100	25	89.2	6640	33.6	5.28	9.047	143789	839	10346	6	BCE
160k	87.6	99	24	68.7	6636	33.4	5.175	7.264	144709	846	9441	6	BCE
160k	90.3	97	23.7	35.3	6854	34	5.257	6.354	135911	823	10058	6	ABCE
160k	89.6	96	27.6	93.4	6675	34.1	5.433	8.591	142205	834	9772	6	ABCE
160k	89.4	99	23	26.6	6652	33.8	5.134	5.478	140610	826	10482	6	ABCE
160k	92.7	98	19.8	23.8	6674	33.3	4.84	5.288	143708	843	10627	6	ABCE
160k	92.2	96	22.9	25.8	6660	33.6	5.12	5.821	137362	833	10771	6	ABCE
160k	88.8	96	42.4	99.1	6942	34.7	6.263	9.272	141807	824	11275	7	DE
160k	87.6	100	41.2	94.5	6711	34.9	6.557	9.123	137819	808	9322	6	DE
160k	89.5	98	26	61.5	6658	34.2	5.375	7.416	140859	826	10112	6	DE
160k	92	98	21.9	38.1	6666	33.9	5.061	6.541	134026	812	10215	6	DE
160k	89.9	97	26.8	93.1	6663	34	5.405	9.196	145182	853	9740	6	DE
160k	92.8	98	28.2	85.5	6913	34.9	5.478	8.629	139864	850	9648	6	ADE
160k	90.3	96	30.6	67.8	6736	34.8	5.77	7.945	145661	853	8860	5	ADE
160k	88.8	98	23.4	31.2	6733	34.2	5.147	5.707	141104	823	10754	6	ADE
160k	88.8	96	20.9	83.9	6728	33.9	4.964	8.743	139657	818	10000	6	ADE

Table 13 continued from previous page

Input pps	pid cpu avg	pid cpu max	sys cpu avg	sys cpu max	sys MB min	temp avg (c)	power avg (w)	power max (w)	rxpps avg	rxmbps avg	nic drop avg	nic drop %	test factor code
160k	88.3	100	22.9	26.2	6723	33.8	5.125	5.326	140389	819	10521	6	ADE
160k	89.4	95	29.6	86.2	6987	34.7	5.575	9.196	144634	847	9879	5	BDE
160k	87.9	95	38.1	71	6734	35.1	6.372	8.059	142166	861	8115	5	BDE
160k	88.8	99	26	58.3	6701	34.5	5.365	6.921	142727	834	11306	7	BDE
160k	89.5	96	21.6	52.9	6709	34	4.975	6.769	141240	826	10305	6	BDE
160k	92.6	97	23.3	29.4	6708	34	5.126	5.593	138075	837	10447	6	BDE
160k	91.1	97	24.1	36	6913	34.8	5.216	6.011	140409	823	10274	6	ABDE
160k	91.2	97	32.2	56.9	6730	35.1	5.868	7.34	144582	845	9428	5	ABDE
160k	89.5	97	24.4	42.4	6718	34.4	5.252	6.693	135977	795	10662	6	ABDE
160k	86.4	100	20.6	27	6732	34.1	4.93	5.288	133905	777	13098	8	ABDE
160k	90.3	97	23.1	26	6728	34.1	5.125	5.478	138415	837	11939	7	ABDE
160k	90.8	100	28.8	81.4	6893	35	5.498	9.123	141603	831	10644	6	CDE
160k	86.9	99	34.9	64.7	6692	35.3	6.12	8.287	137381	828	8987	5	CDE
160k	87.6	96	21.1	28.2	6686	34.4	5.093	5.44	143573	839	10568	6	CDE
160k	87.6	99	22.4	86.4	6688	34.2	5.034	8.667	138209	832	11430	7	CDE
160k	89.9	96	23.3	29.4	6688	34.1	5.156	5.44	143285	839	11178	6	CDE
160k	91.3	95	28.9	86.3	6872	35.2	5.552	9.009	138326	835	10736	6	ACDE
160k	90	100	32.9	61.1	6676	35.3	5.935	7.831	143667	839	9159	5	ACDE
160k	88.8	98	23.9	38.9	6642	34.9	5.207	6.201	136525	797	10412	6	ACDE
160k	93.3	97	21.3	27.4	6650	34.7	4.988	5.593	138571	842	11085	6	ACDE
160k	92.1	98	22.9	25.7	6653	34.5	5.107	5.555	138092	836	10731	6	ACDE
160k	87.4	99	29.4	96.5	6917	35.3	5.626	9.199	139682	843	11198	6	BCDE
160k	86.6	96	39	99.8	6688	35.7	6.3	9.424	143123	837	8885	5	BCDE
160k	88	100	24.5	37.9	6678	35.1	5.259	6.807	142136	829	11357	7	BCDE
160k	89.6	99	19.9	23.6	6691	34.7	4.876	5.478	141414	832	10450	6	BCDE
160k	89.7	98	27.7	88.9	6689	35	5.421	8.819	144965	851	9727	6	BCDE

Table 13 continued from previous page

Input pps	pid cpu avg	pid cpu max	sys cpu avg	sys cpu max	sys MB min	temp avg (c)	power avg (w)	power max (w)	rxpps avg	rxmbps avg	nic drop avg	nic drop %	test factor code
160k	91.6	96	29.1	95.3	6877	35.6	5.514	9.161	137533	828	10642	6	ABCDE
160k	90.7	98	28.6	44.6	6684	35.7	5.646	7.035	145106	846	9449	5	ABCDE
160k	89.9	98	23.5	29.7	6670	35.3	5.191	5.821	143651	870	10835	6	ABCDE
160k	88.5	100	19.8	24.2	6681	34.9	4.87	5.176	143307	832	11612	7	ABCDE
160k	89.3	100	27.1	95	6647	35.2	5.409	8.857	141084	853	10361	6	ABCDE

Appendix D. Source Code

Setup and Usage

Full code, and latest user guide is on the GitHub repository [9].

1. Install Ansible package on control workstation (i.e.)

```
apt install ansible
```

2. Install OS and desired network configuration for systems under test

3. If environment has no DNS, add IP addresses to the hosts file of the Ansible control workstation. This is important, do not skip.

```
nano /etc/hosts...
10.0.0.1        tx1
10.0.0.2        tx2
10.0.0.3        rpi3bp
10.0.0.4        rpi4
10.0.0.5        xavier
10.10.10.60     maas-1
10.10.10.61     maas-2
```

4. Build *inventory.yml* file with device specific variables

```
sensors:
        children:
            rpi:
                hosts:
                    rpi4:
                        ...
                vars:
                    ansible_user: pi
                    ansible_become_method: sudo
                    sensor_dir: /sensor
            nvidia:
                hosts:
                    tx2:
                        capture_interface: eth0
                        send_interface: eth3
                        rps_mask: 3E #0011 1110
                        NAPI_budget_best: 300
                        backlog_best: 1000
                        backlog_weight_best: 300

                vars:
                    ansible_user: nvidia
                    ...
```

5. Build *vars.yml*, *static-controls.yml*, and *variable-controls.yml* playbooks with desired experiment variables. See fully implemented tests (interface, pcap, suricata) for examples

6. Fill playbook *benchmark-innerloop.yml* with the workload to test and fill in placeholders.

```
#This playbook is the "inner" loop
  - name: Launch Performance Monitor (Factors {{ current_factor_list }})
    shell: "./gather_stats.bash <<**PID**>> <<**SAMPLE RATE**>> {{
     ↪  current_factor_list }}"
    args:
      chdir: "{{ experiment_dir }}/"
    register: results_async
    poll: 0
    async: 3600
    become: yes
    changed_when: false

   # <<**YOUR WORKLOAD TASK(S) GOES HERE.....**>>
   #       SEE RATELIMIT TEST FOR EXAMPLE
```

7. Replace placeholders (shown as %%%%) in *benchmark-middleloop.yml*, *benchmark-outerloop.yml* and *benchmark-main.yml* with appropriate variable names.

```
- name: Record Initial Variable Levels
    set_fact:
    A_levels: "{{A_levels}} + [ '%%%%%' ]"
    B_levels: "{{B_levels}} + [ '%%%%%' ]"
    C_levels: "{{C_levels}} + [ '%%%%%' ]"
    D_levels: "{{D_levels}} + [ '%%%%%' ]"
    E_levels: "{{E_levels}} + [ '%%%%%' ]"
```

8. Generate some SSH keys if you don't have them already

```
ssh-keygen
```

9. If first time, run *prep-playbook.yml* to setup SSH keys and dependencies

```
ansible-playbook -i inventory.yml --ask-pass --ask-become-pass  prep-playbook.yml
```

10. Run the main playbook once all placeholders have been filled and set:

```
ansible-playbook -i inventory.yml suricata-bench-playbook.yml
```

11. Intermediate and raw .csv results will be generated on each device and copied back to the current working directory

12. At the end of all testing, a final log will be generated that details the best level of each factor and a final performance score.

```
kyle 102K Feb 11 09:49 rpi4-bcmgenet-ksoftirqd0-results-run1.csv
kyle  32K Feb  4 14:59 rpi4-bcmgenet-ksoftirqd0-results-run2.csv
kyle  11K Feb  4 16:19 rpi4-bcmgenet-ksoftirqd0-results-run3.csv
kyle  64K Feb 13 15:00 rpi4-bcmgenet-Suricata-Main-results-run1.csv
kyle  35K Feb 13 09:58 rpi4-bcmgenet-Suricata-Main-results-run2.csv
kyle 4.6K Feb 11 23:05 rpi4-bcmgenet-Suricata-Main-results-run3.csv
kyle  29K Feb  4 10:43 rpi4-bcmgenet-tcpdump-results-run1.csv
kyle  515 Feb  4 16:19 rpi4-interface-final.log
kyle  511 Feb 11 23:05 rpi4-suricata-final.log
```

Tips and Tricks

Overriding variables from command line is done with -e:

```
ansible-playbook -i inventory.yml -e "pps_limit=104000" suricata-benchmark-main.yml
```

Limiting to only certain hosts from inventory is done with -l:

```
ansible-playbook -i inventory.yml -l nvidia,rpi4 pcap-bench-playbook.yml
```

Play only certain factors on certain devices:

```
ansible-playbook -i inventory.yml -l rpi4 -e '{"factor_combos": [E,AE,BE,ABE,CE,ACE,BCE,ABCE,
DE,ADE,BDE,ABDE,CDE,ACDE,BCDE,ABCDE]}' suricata-benchmark-main.yml
```

Jump directly into the third iteration of a optimization loop

```
ansible-playbook -i inventory.yml -l rpi4 -e "test_counter=3" -e '{"significant_factors_array":
[ABCE]}' -e "last_loop_best=3940539" -e "target_to_beat=5084844" suricata-benchmark-main.yml
```

Debug "play" is very useful and can grab stdout from each device:

```
- name: Send Traffic via tcpreplay.
    local_action:
        module: shell
        _raw_params: sudo tcpreplay -i {{send_interface}} -p {{interface_pps}}
         ↪  1.pcap
        warn: false
    ignore_errors: yes
    register: sender

    #Debug
  - name: Generator Debug
    debug:
        var: sender.stdout
```

Common Code Core

gather-stats.bash

```
#!/bin/bash

#Command line args
PPS=$1
PID=$2
IFACE=$3
SAMPLE_RATE=$4
PACKETS_EXPECTED=$5
TUNING_FACTORS=$6


#TOTAL_RUNTIME=$(( $PACKETS_EXPECTED / $PPS + 5 )) #plus for cooldown buffer
#TOTAL_RUNTIME=60

if [ -z "$4" ]; then
 echo "Usage: bash $0 <test pps rate> <monitor pid> <capture interface> <sample rate in sec> optional: <packets expected> <tuning_factors>"
 echo "ex: bash $0 100000 8912 eth0 0.5 2000000 ABCD"
 echo "a negative pid will watch only the interface / softirq handler"
 echo "**sudo access required**"
 exit 1
elif [ -z "$5" ]; then
 TOTAL_RUNTIME=60
 TUNING_FACTORS=N
elif [ -z "$6" ]; then
 TOTAL_RUNTIME=$(( $PACKETS_EXPECTED / $PPS + 5 )) #plus for cooldown buffer
 TUNING_FACTORS=N
elif [ -z "$7" ]; then
 TOTAL_RUNTIME=$(( $PACKETS_EXPECTED / $PPS + 5 )) #plus for cooldown buffer
fi


cd "$(dirname "$0")"
if [ -f gather.pid ]; then
 echo "Unclean shutdown of previous run. Ending it now.."
 sudo kill $(cat gather.pid)
 sleep 2
fi
echo $$ > gather.pid
tmp=$(mktemp -d)


if [ $PID -lt '0' ]; then
 echo "Using interface rate mode only";
 #Watching softirq daemon, that handles the last half of the interrupt from the NIC
 #Thread 0 is most likely on the ARM based boards (first thread)
 #kernel threads like this wont show memory stats
 PID=$(top -b -n1 | grep ksoftirq | head -1 | awk '{ print $1 }');
 PROCESS_NAME=ksoftirqd0;
elif [ ! -d /proc/$PID ]; then
 echo "supplied PID $PID isn't running, exiting";
 exit 1;
else PROCESS_NAME=$(ps -p $PID -o comm=); fi

#Might be a better way to fingerprint the machine
if [ $(sudo lshw -short -c system | awk 'FNR == 3 {print $2}') == 'Raspberry' ]; then DEVICE_FAM=pi;
elif [ $(sudo lshw -short -c system | awk 'FNR == 3 {print $2}') == 'Jetson-TX1' ]; then DEVICE_FAM=nvidia-tx1;
elif [ $(sudo lshw -short -c system | awk 'FNR == 3 {print $2}') == 'quill' ]; then DEVICE_FAM=nvidia-tx2;
elif [ $(sudo lshw -short -c system | awk 'FNR == 3 {print $2}') == 'Jetson-AGX' ]; then DEVICE_FAM=nvidia-xavier;
```

```
59  else DEVICE_FAM=unknown; fi
60
61  #top has to be kept running to gather accurate CPU stats over time.
62  #See man page for how it calcs this. ps doesn't provide useful data, see man page
     ↪  as well
63  #debian buster has newer version of top that defaults to MB, we want KB
64  if [ $(lsb_release -c -s) == 'buster' ]; then
65   top -p $PID -b -d 1 -E k > $tmp/toptmp &
66   sleep 2
67  else
68    top -p $PID -b -d 1 > $tmp/toptmp &
69
70  fi
71
72  #let top warmup..very important. rip my 3 hours troubleshooting this regression
73  sleep 3
74
75  sudo renice -n -15 $(pidof top) &> /dev/null #bump my top process priority
76  sudo renice -n -20 $$ &> /dev/null #bump my priority to max
77
78  #Initialize vars
79  declare -a PID_CPU_PERCENT
80  declare -a PID_MEM_PERCENT
81  declare -a PID_MEM_MB
82  declare -a UTILIZATION_CPU
83  declare -a TEMPERATURE_CPU
84  declare -a POWER_CPU
85  declare -a MEM_AVAIL_PERCENT
86  declare -a MEM_AVAIL_MB
87  declare -a RXPPS
88  declare -a RXBPS
89  declare -a IFACE_DROPS
90  declare -a KERN_DROPS
91
92  NIC_DRIVER=$(ethtool -i $IFACE | head -1 | awk '{ print $2 }')
93  TOTAL_MEM_MB=$(bc <<< 'scale=2; '$(tail -5 $tmp/toptmp | head -n 1 | awk '{ print
     ↪  $4 }')' / 976.562')
94  RX_PKTS_LAST=$(cat /sys/class/net/$IFACE/statistics/rx_packets)
95  RX_PKTS_FIRST=$RX_PKTS_LAST
96  RX_BPS_LAST=$(cat /sys/class/net/$IFACE/statistics/rx_bytes)
97  TIMEFORMAT=%R
98  LOOP_COUNT=0
99  LOOP_TIME_REAL=$SAMPLE_RATE
100 IFACE_DROPS_PERCENT=0
101 KERN_DROPS_PERCENT=0
102 #SUM_PACKETS=0
103
104 KERN_DROP_LAST=0
105 if [ "$NIC_DRIVER" == 'e1000e' ] || [ "$NIC_DRIVER" == 'igb' ] || [ "$NIC_DRIVER"
     ↪  == 'tg3' ] ||  [ "$NIC_DRIVER" == 'bcmgenet' ] ; then
106  IFACE_DROP_LAST=$(cat /sys/class/net/$IFACE/statistics/rx_missed_errors);
107 elif [ "$NIC_DRIVER" == 'lan78xx' ]; then
108  IFACE_DROP_LAST=$(ethtool -S $IFACE | grep "RX Dropped Frames:" | awk '{ print $4
      ↪  }');
109 elif [ "$NIC_DRIVER" == 'eqos' ]; then
110  IFACE_DROP_LAST=$(ethtool -S $IFACE | grep rx_fifo_overflow | awk '{ print $2
      ↪  }');
111 fi
112
113 function captureLap {
114  #Time dependant ("per second") samples below.
115  RX_PKTS_NOW=$(cat /sys/class/net/$IFACE/statistics/rx_packets)
116  RXPPS[$LOOP_COUNT]=$(bc <<< "scale=0; ($RX_PKTS_NOW - $RX_PKTS_LAST) /
      ↪  ($LOOP_TIME_REAL)")
117  RX_PKTS_LAST=$RX_PKTS_NOW
118  RX_BPS_NOW=$(cat /sys/class/net/$IFACE/statistics/rx_bytes)
119  RXBPS[$LOOP_COUNT]=$(bc <<< "scale=0; (($RX_BPS_NOW - $RX_BPS_LAST) / 125000) /
      ↪  ($LOOP_TIME_REAL) ")
120  RX_BPS_LAST=$RX_BPS_NOW
```

```
121
122   #Handle bizzare rare case where a negative number gets calculated on super
       ↪  heavily loaded machine
123   #Happens on Pi based boards, the interface seems to temporarily "reset" its stats
       ↪  counters?
124   if [ ${RXBPS[$LOOP_COUNT]} -lt "0" ]; then RXBPS[$LOOP_COUNT]=0; fi
125   if [ ${RXPPS[$LOOP_COUNT]} -lt "0" ]; then RXPPS[$LOOP_COUNT]=0; fi
126
127   #Specific to suricata...
128   #TX1 and rpi3 version of suricatasc have this in a different spot..
129   if [ "$PROCESS_NAME" == "Suricata-Main" ]; then
130    if [ "$DEVICE_FAM" == 'nvidia-tx1' ]; then
131     KERN_DROP_NOW=$(suricatasc /var/run/suricata-command.socket -c "iface-stat
         ↪  $IFACE" | awk '{ print $5 }'| egrep -o [0-9]+)
132    #elif [ "£NIC_DRIVER" == 'lan78xx' ]; then
133    # KERN_DROP_NOW=£(suricatasc /var/run/suricata-command.socket -c "iface-stat
       ↪  £IFACE" | awk '{ print £11 }'| egrep -o [0-9]+)
134    else
135     KERN_DROP_NOW=$(suricatasc /var/run/suricata-command.socket -c "iface-stat
         ↪  $IFACE" | awk '{ print $7 }'| egrep -o [0-9]+)
136    fi
137    KERN_DROPS[$LOOP_COUNT]=$(bc <<< "scale=0; ($KERN_DROP_NOW - $KERN_DROP_LAST) /
       ↪  $LOOP_TIME_REAL ")
138    KERN_DROP_LAST=$KERN_DROP_NOW
139   else
140    KERN_DROPS[$LOOP_COUNT]=NA
141   fi
142
143   #Driver specific stat locations
144   if [ "$NIC_DRIVER" == 'lan78xx' ]; then
145    IFACE_DROP_NOW=$(ethtool -S $IFACE | grep "RX Dropped Frames:" | awk '{print
       ↪  $4}')
146    IFACE_DROPS[$LOOP_COUNT]=$(bc <<< "scale=0; ($IFACE_DROP_NOW-$IFACE_DROP_LAST) /
       ↪  $LOOP_TIME_REAL ")
147    IFACE_DROP_LAST=$IFACE_DROP_NOW
148   elif [ "$NIC_DRIVER" == 'e1000e' ] || [ "$NIC_DRIVER" == 'igb' ] || [
       ↪  "$NIC_DRIVER" == 'tg3' ] || [ "$NIC_DRIVER" == 'bcmgenet' ]; then
149    IFACE_DROP_NOW=$(cat /sys/class/net/$IFACE/statistics/rx_missed_errors)
150    IFACE_DROPS[$LOOP_COUNT]=$(bc <<< "scale=0; ($IFACE_DROP_NOW-$IFACE_DROP_LAST) /
       ↪  $LOOP_TIME_REAL ")
151    IFACE_DROP_LAST=$IFACE_DROP_NOW
152   elif [ "$NIC_DRIVER" == 'eqos' ]; then
153    IFACE_DROP_NOW=$(ethtool -S $IFACE | grep rx_fifo_overflow | awk '{ print $2
       ↪  }');
154    IFACE_DROPS[$LOOP_COUNT]=$(bc <<< "scale=0; ($IFACE_DROP_NOW-$IFACE_DROP_LAST) /
       ↪  $LOOP_TIME_REAL ")
155    IFACE_DROP_LAST=$IFACE_DROP_NOW
156   fi
157
158   #Device specific sensors, not super efficient
159   if [ "$DEVICE_FAM" == 'pi' ]; then
160    TEMPERATURE_CPU[$LOOP_COUNT]=$(vcgencmd measure_temp | grep -ow
       ↪  "[0-9][0-9].[0-9]")
161    POWER_CPU[$LOOP_COUNT]=$(expr $(vcgencmd measure_clock arm | grep -oP "([0-9]+)"
       ↪  | tail -1) / 1000000)
162   elif [ "$DEVICE_FAM" == 'nvidia-tx1' ]; then
163    TEMPERATURE_CPU[$LOOP_COUNT]=$(bc <<< 'scale=1; '$(cat
       ↪  /sys/devices/virtual/thermal/thermal_zone1/temp)' / 1000')
164    POWER_CPU[$LOOP_COUNT]=$(bc <<< 'scale=3; '$(cat
       ↪  /sys/devices/7000c400.i2c/i2c-1/1-0040/iio_device/in_power0_input)' /
       ↪  1000')
165   elif [ "$DEVICE_FAM" == 'nvidia-tx2' ]; then
166    TEMPERATURE_CPU[$LOOP_COUNT]=$(bc <<< 'scale=1; '$(cat
       ↪  /sys/devices/virtual/thermal/thermal_zone1/temp)' / 1000')
167    POWER_CPU[$LOOP_COUNT]=$(bc <<< 'scale=3; '$(cat
       ↪  /sys/bus/i2c/drivers/ina3221x/0-0041/iio_device/in_power0_input)' / 1000')
```

```
168   elif [ "$DEVICE_FAM" == 'nvidia-xavier' ]; then
169    TEMPERATURE_CPU[$LOOP_COUNT]=$(bc <<< 'scale=1; '$(cat
       ↪  /sys/devices/virtual/thermal/thermal_zone0/temp)' / 1000')
170    POWER_CPU[$LOOP_COUNT]=$(bc <<< 'scale=3; '$(cat
       ↪  /sys/bus/i2c/drivers/ina3221x/1-0040/iio_device/in_power1_input)' / 1000')
171   else
172    TEMPERATURE_CPU[$LOOP_COUNT]=NA
173    POWER_CPU[$LOOP_COUNT]=NA
174   fi
175
176   #Regular sensors / reports
177   PID_MEM_PERCENT[$LOOP_COUNT]=$(ps -p $PID -o pmem --no-headers)
178   PID_MEM_MB[$LOOP_COUNT]=$(bc <<< 'scale=0; '$(ps -p $PID -o rss --no-headers)' /
       ↪  976.562' )
179   PID_CPU_PERCENT[$LOOP_COUNT]=$(tail -1 $tmp/toptmp | awk '{ print $9 }')
180   MEM_AVAIL_MB[$LOOP_COUNT]=$(bc <<< 'scale=0; '$(tail -5 $tmp/toptmp | head -n 1 |
       ↪  awk '{ print $6 + $10 }')' / 976.562' )
181   MEM_AVAIL_PERCENT[$LOOP_COUNT]=$(bc <<< "scale=1; ${MEM_AVAIL_MB[$LOOP_COUNT]} /
       ↪  $TOTAL_MEM_MB * 100" )
182   UTILIZATION_CPU[$LOOP_COUNT]=$(tail -6 $tmp/toptmp | head -n 1 | awk '{ print $2
       ↪  + $4 + $6 + $10 + $12 + $14  }')
183
184
185   # uncomment for live debugging
186   # echo txPPS\: £PPS - \%CPU\: £{PID_CPU_PERCENT[£LOOP_COUNT]} - TOTAL CPU\:
       ↪  £{UTILIZATION_CPU[£LOOP_COUNT]} - \%MEM\: £{PID_MEM_PERCENT[£LOOP_COUNT]} -
       ↪  MEM MB\: £{PID_MEM_MB[£LOOP_COUNT]} - \
187   # MB FREE\: £{MEM_AVAIL_MB[£LOOP_COUNT]} - TEMPERATURE_CPU\(C\)\:
       ↪  £{TEMPERATURE_CPU[£LOOP_COUNT]} - CPU POWER\: £{POWER_CPU[£LOOP_COUNT]} -
       ↪  rxPPS\: £{RXPPS[£LOOP_COUNT]} - \
188   #rxmbps\: £{RXBPS[£LOOP_COUNT]} - iface drps\: £{IFACE_DROPS[£LOOP_COUNT]}, krn
       ↪  drps\: £{KERN_DROPS[£LOOP_COUNT]}, loop\: £LOOP_TIME_REAL
189
190   (( LOOP_COUNT=LOOP_COUNT+1 ))
191  }
192
193  function buildFinalStats {
194
195   #Moved out of critial loop region
196   IFS=$'\n'
197   MAX_PID_CPU_PERCENT=$(echo "${PID_CPU_PERCENT[*]}" | sort -nr | head -n1)
198   MAX_PID_MEM_PERCENT=$(echo "${PID_MEM_PERCENT[*]}" | sort -nr | head -1)
199   MAX_PID_MEM_MB=$(echo "${PID_MEM_MB[*]}" | sort -nr | head -1)
200   MAX_UTILIZATION_CPU=$(echo "${UTILIZATION_CPU[*]}" | sort -nr | head -1)
201   MIN_MEM_AVAIL_PERCENT=$(echo "${MEM_AVAIL_PERCENT[*]}" | sort -nr | tail -1)
202   MIN_MEM_AVAIL_MB=$(echo "${MEM_AVAIL_MB[*]}" | sort -nr | tail -1)
203   MAX_TEMPERATURE_CPU=$(echo "${TEMPERATURE_CPU[*]}" | sort -nr | head -1)
204   MAX_POWER_CPU=$(echo "${POWER_CPU[*]}" | sort -nr | head -1)
205   MAX_RXBPS=$(echo "${RXBPS[*]}" | sort -nr | head -1)
206   MAX_RXPPS=$(echo "${RXPPS[*]}" | sort -nr | head -1)
207
208   #Averages.
209   #Zeros may throw off averages if they're not likely (fully overloaded,etc)
210   #Can count the number of zeros in the array so they dont throw off averages
211   #( All items in array / (Array size - zero count) )
212   IFS='+'
213   (( RX_PKTS_TOTAL=RX_PKTS_LAST-RX_PKTS_FIRST ))
214
215   #currently only suricata gives access to real time kernel drops. dont remove
       ↪  zeros
216   if [ "$PROCESS_NAME" == "Suricata-Main" ]; then
217    SUM_KERN_DROPS=$(echo "${KERN_DROPS[*]}"|bc)
218    echo "kern drop number: ${#KERN_DROPS[@]}"
219    AVG_KERN_DROPS=$(echo "(${KERN_DROPS[*]}) / ${#KERN_DROPS[@]}" | bc 2>
       ↪  /dev/null)
220    KERN_DROPS_PERCENT=$(bc <<< "scale=2; $SUM_KERN_DROPS / $RX_PKTS_TOTAL * 100")
221   elif [ "$PROCESS_NAME" == "tcpdump" ]; then
```

```
222     AVG_KERN_DROPS=NA
223     SUM_KERN_DROPS=$(cat counters | awk ' FNR == 4 {print $1}')
224     KERN_DROPS_PERCENT=$(bc <<< "scale=3; $SUM_KERN_DROPS / $RX_PKTS_TOTAL")
          ↪  #Percent dropped after making it past the first round...
225     rm -rf counters
226     rm -rf tcpdump.pid
227   else
228     AVG_KERN_DROPS=NA
229     SUM_KERN_DROPS=NA
230     KERN_DROPS_PERCENT=NA
231   fi
232
233   SUM_IFACE_DROPS=$(echo "${IFACE_DROPS[*]}"|bc)
234   AVG_IFACE_DROPS=$(echo "(${IFACE_DROPS[*]}) / (${#IFACE_DROPS[*]} - $(echo
          ↪  ${IFACE_DROPS[*]} | grep -ow '0' | wc -l))"|bc 2> /dev/null)
235   IFACE_DROPS_PERCENT=$(bc <<< "scale=2; $SUM_IFACE_DROPS / $PACKETS_EXPECTED *
          ↪  100")
236   AVG_RXPPS=$(echo "(${RXPPS[*]}) / (${#RXPPS[*]} - $(echo ${RXPPS[*]} | grep -ow
          ↪  '0' | wc -l))"|bc 2> /dev/null)
237   AVG_RXBPS=$(echo "(${RXBPS[*]}) / (${#RXBPS[*]} - $(echo ${RXBPS[*]} | grep -ow
          ↪  '0' | wc -l))"|bc 2> /dev/null)
238   AVG_PID_MEM_PERCENT=$(echo "scale=1; (${PID_MEM_PERCENT[*]}) /
          ↪  (${#PID_MEM_PERCENT[*]} - $(echo ${PID_MEM_PERCENT[*]} | grep -ow '0.0' | wc
          ↪  -l))"|bc 2> /dev/null)
239   AVG_PID_CPU_PERCENT=$(echo "scale=1; (${PID_CPU_PERCENT[*]}) /
          ↪  (${#PID_CPU_PERCENT[*]} - $(echo ${PID_CPU_PERCENT[*]} | grep -ow '0.0' | wc
          ↪  -l))"|bc 2> /dev/null)
240   AVG_POWER_CPU=$(echo "scale=3; (${POWER_CPU[*]}) / (${#POWER_CPU[*]} - $(echo
          ↪  ${POWER_CPU[*]} | grep -ow '0' | wc -l))"|bc 2> /dev/null)
241   AVG_TEMPERATURE_CPU=$(echo "scale=1; (${TEMPERATURE_CPU[*]}) /
          ↪  (${#TEMPERATURE_CPU[*]} - $(echo ${TEMPERATURE_CPU[*]} | grep -ow '0.0' | wc
          ↪  -l))"|bc 2> /dev/null)
242   AVG_UTILIZATION_CPU=$(echo "scale=1; (${UTILIZATION_CPU[*]}) /
          ↪  (${#UTILIZATION_CPU[*]} - $(echo ${UTILIZATION_CPU[*]} | grep -ow '0.0' | wc
          ↪  -l))"|bc 2> /dev/null)
243
244   unset IFS
245 }
246
247 function printVerboseStats {
248
249   echo "New run -- tx PPS: $PPS -- Sample rate: $4 -- Driver: $NIC_DRIVER -- tuning
          ↪  factors $TUNING_FACTORS" >>
          ↪  $HOSTNAME-$NIC_DRIVER-$PROCESS_NAME-results-verbose.csv
250   echo
          ↪  "txpps,%pidcpu,%totalcpu,%pidmem,pidmemMB,memavailMB,%memavail,cpu_temp(c),cpu_power(w),rxpps,
          ↪  >> $HOSTNAME-$NIC_DRIVER-$PROCESS_NAME-results-verbose.csv
251
252   for ((i = 0; i < $LOOP_COUNT; i++ )); do
253     echo
          ↪  $PPS,${PID_CPU_PERCENT[$i]},${UTILIZATION_CPU[$i]},${PID_MEM_PERCENT[$i]},${PID_MEM_MB[$i]},$
254     ${POWER_CPU[$i]},${RXPPS[$i]},${RXBPS[$i]},${IFACE_DROPS[$i]},${KERN_DROPS[$i]}>>
          ↪  $HOSTNAME-$NIC_DRIVER-$PROCESS_NAME-results-verbose.csv
255   done
256 }
257
258 function finish {
259   rm -rf "$tmp"
260   rm -rf gather.pid
261
262   killall top 2> /dev/null
263   buildFinalStats
264   printVerboseStats
265
266   if [ ! -f $HOSTNAME-$NIC_DRIVER-$PROCESS_NAME-results.csv ]; then
```

```
267     echo
        ↪  'tx,pidcpu,pidcpu,syscpu,syscpu,pidmem,pidmem,pidmem,sysmemfree,sysmemfree,temp,temp,power,po
        ↪  >> $HOSTNAME-$NIC_DRIVER-$PROCESS_NAME-results.csv;
268     echo
        ↪  'pps,%avg,%max,%avg,%max,%avg,%max,MBmax,MBmin,%min,avg(c),max(c),avg,max,avg,max,avg,max,sum
        ↪  >> $HOSTNAME-$NIC_DRIVER-$PROCESS_NAME-results.csv;
269   fi
270
271   #handle some empty cases before writing to file
272   if [ -z "$AVG_IFACE_DROPS" ]; then AVG_IFACE_DROPS=0; fi
273   if [ -z "$AVG_KERN_DROPS" ]; then AVG_KERN_DROPS=0; fi
274   if [ -z "$AVG_RXBPS" ]; then AVG_RXBPS=0; fi
275   if [ -z "$AVG_RXPPS" ]; then AVG_RXPPS=0; fi
276   if [ -z "$AVG_PID_CPU_PERCENT" ]; then AVG_PID_CPU_PERCENT=0.0; fi
277   if [ -z "$AVG_PID_MEM_PERCENT" ]; then AVG_PID_MEM_PERCENT=0.0; fi
278   if [ -z "$AVG_UTILIZATION_CPU" ]; then AVG_UTILIZATION_CPU=0.0; fi
279   if [ ${TEMPERATURE_CPU[0]} == 'NA' ]; then AVG_TEMPERATURE_CPU=NA;
      ↪  MAX_TEMPERATURE_CPU=NA; fi
280   if [ ${POWER_CPU[0]} == 'NA' ]; then AVG_POWER_CPU=NA; MAX_POWER_CPU=NA; fi
281
282   echo
      ↪  $PPS,$AVG_PID_CPU_PERCENT,$MAX_PID_CPU_PERCENT,$AVG_UTILIZATION_CPU,$MAX_UTILIZATION_CPU,$AVG_
283   $MAX_PID_MEM_MB,$MIN_MEM_AVAIL_MB,$MIN_MEM_AVAIL_PERCENT,$AVG_TEMPERATURE_CPU,$MAX_TEMPERATURE_CPU,
284   $MAX_RXBPS,$SUM_IFACE_DROPS,$AVG_IFACE_DROPS,$IFACE_DROPS_PERCENT,$SUM_KERN_DROPS,$AVG_KERN_DROPS,$
      ↪  >> "$HOSTNAME-$NIC_DRIVER-$PROCESS_NAME-results.csv"
285
286
287   echo "Total packets past kernel phase: $RX_PKTS_TOTAL"
288   #echo "Iface Drops: £SUM_IFACE_DROPS £IFACE_DROPS_PERCENT%"
289   #echo "kern drops: £SUM_KERN_DROPS £KERN_DROPS_PERCENT%"
290
291   (head -n2 && tail -n1) < $HOSTNAME-$NIC_DRIVER-$PROCESS_NAME-results.csv | column
      ↪  -t -s ,
292   #column -t -s , £HOSTNAME-£NIC_DRIVER-£PROCESS_NAME-results.csv
293   exec 3>&1- 4>&2-
294   exit 0
295 }
296
297 ##"main" function below
298 #sleep 2 #brief warmup
299 trap finish EXIT  #Capture ctrl-c or kill signals so I can cleanup
300
301 echo "Watching process: $PROCESS_NAME ($PID)"
302 echo "I'm running on a: $DEVICE_FAM board with a $NIC_DRIVER interface "
303 echo "runtime will be $TOTAL_RUNTIME with factors $TUNING_FACTORS"
304 SECONDS=0
305 exec 3>&1 4>&2 #bash magic to get the output of the time command and save the
    ↪  functions stdout/stderr
306 while [[ -d /proc/$PID && $SECONDS -lt $TOTAL_RUNTIME ]]
307 do
308   #This needs to be as close as possible to SAMPLE_RATE sec for "per second"
      ↪  calculations to be accurate
309   #As system load nears 100% the loop will likely drift, so try to account for it.
310   #Still not perfect, but close enough for now.
311   { time {
312     sleep $SAMPLE_RATE & captureLap 1>&3 2>&4;
313     if [ ${RXPPS[$LOOP_COUNT-1]} -lt '10' ]; then SECONDS=0; fi #Dont start the
        ↪  countdown till packets start arriving. 10 accounts for random broadcasts
314     wait $!; } } 2>"$tmp/lastloop"
315   LOOP_TIME_REAL=$(cat $tmp/lastloop)
316 done
```

anova.py

```
#The pandas .convert_objects() function is deprecated
#Couldnt get the new function to work properly, didnt want to waste more time on it
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
import matplotlib.pyplot as plt
import pandas as pd
from scipy import stats
from scipy.stats import norm
import numpy as np
import seaborn as sns
import statsmodels.api as sm
from statsmodels.formula.api import ols
import statsmodels.stats.multicomp
import probscale
import sys

#ANOVA
#Thanks to Marvin for Python help

if len(sys.argv) <= 5:
 print("Not enough args usage: anova.py <*.csv> <rv1,rv2> <factors> <replicates> <alpha> optional: device")
 print("ex: anova.py testdata.csv nicdrop,avg 5 10 .05 TX1")
 print("<rv> is response variable, note comma in example")
 print("\"Device\" is used in normplot of effects, omit to skip graph")
 exit()

n = int(sys.argv[4]) #replicates
k = int(sys.argv[3]) #factors
alpha = float(sys.argv[5])
rv = sys.argv[2].split(',')
input_csv_parse = sys.argv[1].split('-')


if k > 5 or k < 1:
 print("Max factors is 5, Min is 1")
 exit()


data2 = pd.read_csv(sys.argv[1], header=[0,1])
response_var = data2[[rv[0],'factors']]
response_var.columns = response_var.columns.get_level_values(1)
#print(response_var.groupby('code').mean().sort_values(by=[rv[1]]).round(0))

if(k >= 1):
 df_index=['A', 'Error', 'Total']
 one = response_var[response_var['code'] == 'N'].loc[:,rv[1]].to_numpy()
 a  = response_var[response_var['code'] == 'A'].loc[:,rv[1]].to_numpy()
 means_all = np.array([np.mean(a)])
 total = np.array([one, a])
 contrast_A = np.sum(-one + a)
 contrasts_all = np.array([contrast_A])
if(k >= 2):
 df_index=['A', 'B', 'AB', 'Error', 'Total']
 one = response_var[response_var['code'] == 'N'].loc[:,rv[1]].to_numpy()
 a  = response_var[response_var['code'] == 'A'].loc[:,rv[1]].to_numpy()
 b  = response_var[response_var['code'] == 'B'].loc[:,rv[1]].to_numpy()
 ab  = response_var[response_var['code'] == 'AB'].loc[:,rv[1]].to_numpy()
 means_all = np.array([np.mean(a), np.mean(b), np.mean(ab)])
 total = np.array([one, a, b, ab])
 contrast_A = np.sum(-one + a - b + ab)
 contrast_B = np.sum(-one - a + b + ab)
 contrast_AB = np.sum(one - a - b + ab)
 contrasts_all = np.array([contrast_A, contrast_B, contrast_AB])
if(k >= 3):
```

```
  df_index=['A', 'B', 'AB', 'C', 'AC', 'BC', 'ABC', 'Error', 'Total']
  c  = response_var[response_var['code'] == 'C'].loc[:,rv[1]].to_numpy()
  ac  = response_var[response_var['code'] == 'AC'].loc[:,rv[1]].to_numpy()
  bc  = response_var[response_var['code'] == 'BC'].loc[:,rv[1]].to_numpy()
  abc = response_var[response_var['code'] == 'ABC'].loc[:,rv[1]].to_numpy()
  means_all = np.array([np.mean(a), np.mean(b), np.mean(ab), np.mean(c),
   ↪  np.mean(ac), np.mean(bc), np.mean(abc)])
  total = np.array([one, a, b, ab, c, ac, bc, abc])
  contrast_A = np.sum(-one + a - b + ab - c + ac - bc + abc)
  contrast_B = np.sum(-one - a + b + ab - c - ac + bc + abc)
  contrast_AB = np.sum(one - a - b + ab + c - ac - bc + abc)
  contrast_C = np.sum(-one - a - b - ab + c + ac + bc + abc)
  contrast_AC = np.sum(one - a + b - ab - c + ac - bc + abc)
  contrast_BC = np.sum(one + a - b - ab - c - ac + bc + abc)
  contrast_ABC= np.sum(-one+ a + b - ab + c - ac - bc + abc)
  contrasts_all = np.array([contrast_A, contrast_B, contrast_AB, contrast_C,
   ↪  contrast_AC, contrast_BC, contrast_ABC])
if(k >= 4):
  df_index=['A', 'B', 'AB', 'C', 'AC', 'BC', 'ABC', 'D', 'AD', 'BD', 'ABD', 'CD',
   ↪  'ACD', 'BCD', 'ABCD', 'Error', 'Total']
  d  = response_var[response_var['code'] == 'D'].loc[:,rv[1]].to_numpy()
  ad  = response_var[response_var['code'] == 'AD'].loc[:,rv[1]].to_numpy()
  bd  = response_var[response_var['code'] == 'BD'].loc[:,rv[1]].to_numpy()
  abd = response_var[response_var['code'] == 'ABD'].loc[:,rv[1]].to_numpy()
  cd  = response_var[response_var['code'] == 'CD'].loc[:,rv[1]].to_numpy()
  acd = response_var[response_var['code'] == 'ACD'].loc[:,rv[1]].to_numpy()
  bcd = response_var[response_var['code'] == 'BCD'].loc[:,rv[1]].to_numpy()
  abcd = response_var[response_var['code'] == 'ABCD'].loc[:,rv[1]].to_numpy()
  means_all = np.array([np.mean(a), np.mean(b), np.mean(ab), np.mean(c),
   ↪  np.mean(ac), np.mean(bc), np.mean(abc), np.mean(d),
       np.mean(ad), np.mean(bd), np.mean(abd), np.mean(cd), np.mean(acd),
        ↪  np.mean(bcd), np.mean(abcd)])
  total = np.array([one, a, b, ab, c, ac, bc, abc, d, ad, bd, abd, cd, acd, bcd,
   ↪  abcd])
  contrast_A = np.sum(-one + a - b + ab - c + ac - bc + abc - d + ad - bd + abd -
   ↪  cd + acd - bcd + abcd)
  contrast_B = np.sum(-one - a + b + ab - c - ac + bc + abc - d - ad + bd + abd -
   ↪  cd - acd + bcd + abcd)
  contrast_AB = np.sum(one - a - b + ab + c - ac - bc + abc + d - ad - bd + abd +
   ↪  cd - acd - bcd + abcd)
  contrast_C = np.sum(-one - a - b - ab + c + ac + bc + abc - d - ad - bd - abd +
   ↪  cd + acd + bcd + abcd)
  contrast_AC = np.sum(one - a + b - ab - c + ac - bc + abc + d - ad + bd - abd -
   ↪  cd + acd - bcd + abcd)
  contrast_BC = np.sum(one + a - b - ab - c - ac + bc + abc + d + ad - bd - abd -
   ↪  cd - acd + bcd + abcd)
  contrast_ABC= np.sum(-one+ a + b - ab + c - ac - bc + abc - d + ad + bd - abd +
   ↪  cd - acd - bcd + abcd)
  contrast_D =  np.sum(-one- a - b - ab - c - ac - bc - abc + d + ad + bd + abd +
   ↪  cd + acd + bcd + abcd)
  contrast_AD=  np.sum(one - a + b - ab + c - ac + bc - abc - d + ad - bd + abd -
   ↪  cd + acd - bcd + abcd)
  contrast_BD = np.sum(one + a - b - ab + c + ac - bc - abc - d - ad + bd + abd -
   ↪  cd - acd + bcd + abcd)
  contrast_ABD= np.sum(-one + a + b - ab - c + ac + bc - abc + d - ad - bd + abd +
   ↪  cd - acd - bcd + abcd)
  contrast_CD = np.sum(one + a + b + ab - c - ac - bc - abc - d - ad - bd - abd +
   ↪  cd + acd + bcd + abcd)
  contrast_ACD = np.sum(-one + a - b + ab + c - ac + bc - abc + d - ad + bd - abd -
   ↪  cd + acd - bcd + abcd)
  contrast_BCD = np.sum(-one - a + b + ab + c + ac - bc - abc + d + ad - bd - abd -
   ↪  cd - acd + bcd + abcd)
  contrast_ABCD = np.sum(one - a - b + ab - c + ac + bc - abc - d + ad + bd - abd +
   ↪  cd - acd - bcd + abcd)
```

```
  contrasts_all = np.array([contrast_A, contrast_B, contrast_AB, contrast_C, contrast_AC, contrast_BC, contrast_ABC,
                            contrast_D, contrast_AD, contrast_BD, contrast_ABD, contrast_CD, contrast_ACD, contrast_BCD, contrast_ABCD])
if(k >= 5):
  df_index=['A', 'B', 'AB', 'C', 'AC', 'BC', 'ABC', 'D', 'AD', 'BD', 'ABD', 'CD', 'ACD', 'BCD', 'ABCD', 'E', 'AE', 'BE', 'ABE',
  'CE', 'ACE', 'BCE', 'ABCE','DE', 'ADE', 'BDE', 'ABDE', 'CDE', 'ACDE', 'BCDE', 'ABCDE', 'Error', 'Total']
  e  = response_var[response_var['code'] == 'E'].loc[:,rv[1]].to_numpy()
  ae  = response_var[response_var['code'] == 'AE'].loc[:,rv[1]].to_numpy()
  be  = response_var[response_var['code'] == 'BE'].loc[:,rv[1]].to_numpy()
  abe = response_var[response_var['code'] == 'ABE'].loc[:,rv[1]].to_numpy()
  ce  = response_var[response_var['code'] == 'CE'].loc[:,rv[1]].to_numpy()
  ace = response_var[response_var['code'] ==  'ACE'].loc[:,rv[1]].to_numpy()
  bce = response_var[response_var['code'] == 'BCE'].loc[:,rv[1]].to_numpy()
  abce = response_var[response_var['code'] == 'ABCE'].loc[:,rv[1]].to_numpy()
  de  = response_var[response_var['code'] == 'DE'].loc[:,rv[1]].to_numpy()
  ade = response_var[response_var['code'] == 'ADE'].loc[:,rv[1]].to_numpy()
  bde = response_var[response_var['code'] == 'BDE'].loc[:,rv[1]].to_numpy()
  abde = response_var[response_var['code'] == 'ABDE'].loc[:,rv[1]].to_numpy()
  cde = response_var[response_var['code'] == 'CDE'].loc[:,rv[1]].to_numpy()
  acde = response_var[response_var['code'] == 'ACDE'].loc[:,rv[1]].to_numpy()
  bcde = response_var[response_var['code'] == 'BCDE'].loc[:,rv[1]].to_numpy()
  abcde = response_var[response_var['code'] == 'ABCDE'].loc[:,rv[1]].to_numpy()
  means_all = np.array([np.mean(a), np.mean(b), np.mean(ab), np.mean(c), np.mean(ac), np.mean(bc), np.mean(abc), np.mean(d),
       np.mean(ad), np.mean(bd), np.mean(abd), np.mean(cd), np.mean(acd), np.mean(bcd), np.mean(abcd),np.mean(e),
       np.mean(ae),np.mean(be),np.mean(abe),np.mean(ce),np.mean(ace),np.mean(bce),np.mean(abce),np.m
       np.mean(bde),np.mean(abde),np.mean(cde),np.mean(acde),np.mean(bcde),np.mean(abcde)])
  total = np.array([one, a, b, ab, c, ac, bc, abc, d, ad, bd, abd, cd, acd, bcd, abcd,
     e,ae,be,abe,ce,ace,bce,abce,de,ade,bde,abde,cde,acde,bcde,abcde])
  contrast_A = np.sum(-one + a  - b  + ab  - c  + ac  - bc  + abc  - d  + ad  - bd  + abd  - cd  + acd  - bcd  + abcd
       - e  + ae - be + abe - ce + ace - bce + abce - de + ade - bde + abde - cde + acde - bcde + abcde)
  contrast_B = np.sum(-one - a  + b  + ab  - c  - ac  + bc  + abc  - d  - ad  + bd  + abd  - cd  - acd  + bcd  + abcd
       - e  - ae + be + abe - ce - ace + bce + abce - de - ade + bde + abde - cde - acde + bcde + abcde)
  contrast_AB = np.sum(one - a  - b  + ab  + c  - ac  - bc  + abc  + d  - ad  - bd  + abd  + cd  - acd  - bcd  + abcd
       + e  - ae - be + abe + ce - ace - bce + abce + de - ade - bde + abde + cde - acde - bcde + abcde)
  contrast_C = np.sum(-one - a  - b  - ab  + c  + ac  + bc  + abc  - d  - ad  - bd  - abd  + cd  + acd  + bcd  + abcd
       - e  - ae - be - abe + ce + ace + bce + abce - de - ade - bde - abde + cde + acde + bcde + abcde)
  contrast_AC = np.sum(one - a  + b  - ab  - c  + ac  - bc  + abc  + d  - ad  + bd  - abd  - cd  + acd  - bcd  + abcd
       + e  - ae + be - abe - ce + ace - bce + abce + de - ade + bde - abde - cde + acde - bcde + abcde)
  contrast_BC = np.sum(one + a  - b  - ab  - c  - ac  + bc  + abc  + d  + ad  - bd  - abd  - cd  - acd  + bcd  + abcd
       + e  + ae - be - abe - ce - ace + bce + abce + de + ade - bde - abde - cde - acde + bcde + abcde)
  contrast_ABC= np.sum(-one + a  + b  - ab  + c  - ac  - bc  + abc  - d  + ad  + bd  - abd  + cd  - acd  - bcd  + abcd
       - e  + ae + be - abe + ce - ace - bce + abce - de + ade + bde - abde + cde - acde - bcde + abcde)
  contrast_D =  np.sum(-one - a  - b  - ab  - c  - ac  - bc  - abc  + d  + ad  + bd  + abd  + cd  + acd  + bcd  + abcd
```

```
150         - e  - ae - be - abe - ce - ace - bce - abce + de + ade + bde + abde + cde
             ↪  + acde + bcde + abcde)
151   contrast_AD=  np.sum(one - a  + b  - ab  + c  - ac  + bc  - abc  - d  + ad  - bd
     ↪  + abd  - cd  + acd  - bcd  + abcd
152         + e  - ae + be - abe + ce - ace + bce - abce - de + ade - bde + abde - cde +
             ↪  acde - bcde + abcde)
153   contrast_BD = np.sum(one + a  - b  - ab  + c  + ac  - bc  - abc  - d  - ad  + bd
     ↪  + abd  - cd  - acd  + bcd  + abcd
154         + e  + ae - be - abe + ce + ace - bce - abce - de - ade + bde + abde - cde -
             ↪  acde + bcde + abcde)
155   contrast_ABD= np.sum(-one + a + b  - ab  - c  + ac  + bc  - abc  + d  - ad  - bd
     ↪  + abd  + cd  - acd  - bcd  + abcd
156         - e + ae + be - abe - ce + ace + bce - abce + de - ade - bde + abde + cde -
             ↪  acde - bcde + abcde)
157   contrast_CD = np.sum(one + a  + b  + ab  - c  - ac  - bc  - abc  - d  - ad  - bd
     ↪  - abd  + cd  + acd  + bcd  + abcd
158         + e  + ae + be + abe - ce - ace - bce - abce - de - ade - bde - abde + cde +
             ↪  acde + bcde + abcde)
159   contrast_ACD = np.sum(-one + a  - b  + ab  + c  - ac  + bc  - abc  + d  - ad  +
     ↪  bd  - abd  - cd  + acd  - bcd  + abcd
160           - e  + ae - be + abe + ce - ace + bce - abce + de - ade + bde - abde - cde
               ↪  + acde - bcde + abcde)
161   contrast_BCD = np.sum(-one - a  + b  + ab  + c  + ac  - bc  - abc  + d  + ad  -
     ↪  bd  - abd  - cd  - acd  + bcd  + abcd
162           - e  - ae + be + abe + ce + ace - bce - abce + de + ade - bde - abde - cde
               ↪  - acde + bcde + abcde)
163   contrast_ABCD = np.sum(one - a  - b  + ab  - c  + ac  + bc  - abc  - d  + ad  +
     ↪  bd  - abd  + cd  - acd  - bcd  + abcd
164           + e  - ae - be + abe - ce + ace + bce - abce - de + ade + bde - abde + cde
               ↪  - acde - bcde + abcde)
165   contrast_E = np.sum(-one - a  - b  - ab  - c  - ac  - bc  - abc  - d  - ad  - bd
     ↪  - abd  - cd  - acd  - bcd  - abcd
166         + e  + ae + be + abe + ce + ace + bce + abce + de + ade + bde + abde + cde +
             ↪  acde + bcde + abcde)
167   contrast_AE = np.sum(one - a  + b  - ab  + c  - ac  + bc  - abc  + d  - ad  + bd
     ↪  - abd  + cd  - acd  + bcd  - abcd
168           - e  + ae - be + abe - ce + ace - bce + abce - de + ade - bde + abde -
               ↪  cde + acde - bcde + abcde)
169   contrast_BE= np.sum(one + a  - b  - ab  + c  + ac  - bc  - abc  + d  + ad  - bd
     ↪  - abd  + cd  + acd  - bcd  - abcd
170         - e - ae + be + abe - ce - ace + bce + abce - de - ade + bde + abde - cde -
             ↪  acde + bcde + abcde)
171   contrast_ABE= np.sum(-one + a  + b  - ab  - c  + ac  + bc  - abc  - d  + ad  + bd
     ↪  - abd  - cd  + acd  + bcd  - abcd
172          + e  - ae - be + abe + ce - ace - bce + abce + de - ade - bde + abde + cde
               ↪  - acde - bcde + abcde)
173   contrast_CE= np.sum(one + a  + b  + ab  - c  - ac  - bc  - abc  + d  + ad  + bd
     ↪  + abd  - cd  - acd  - bcd  - abcd
174         - e - ae - be - abe + ce + ace + bce + abce - de - ade - bde - abde + cde +
             ↪  acde + bcde + abcde)
175   contrast_ACE= np.sum(-one + a  - b  + ab  + c  - ac  + bc  - abc  - d  + ad  - bd
     ↪  + abd  + cd  - acd  + bcd  - abcd
176          + e  - ae + be - abe - ce + ace - bce + abce + de - ade + bde - abde - cde
               ↪  + acde - bcde + abcde)
177   contrast_BCE= np.sum(-one - a  + b  + ab  + c  + ac  - bc  - abc  - d  - ad  + bd
     ↪  + abd  + cd  + acd  - bcd  - abcd
178           + e  + ae - be - abe - ce - ace + bce + abce + de + ade - bde - abde -
               ↪  cde - acde + bcde + abcde)
179   contrast_ABCE= np.sum(one - a  - b  + ab  - c  + ac  + bc  - abc  + d  - ad  - bd
     ↪  + abd  - cd  + acd  + bcd  - abcd
180           - e + ae + be - abe + ce - ace - bce + abce - de + ade + bde - abde + cde
               ↪  - acde - bcde + abcde)
181   contrast_DE= np.sum(one + a  + b  + ab  + c  + ac  + bc  + abc  - d  - ad  - bd
     ↪  - abd  - cd  - acd  - bcd  - abcd
182         - e - ae - be - abe - ce - ace - bce - abce + de + ade + bde + abde + cde +
             ↪  acde + bcde + abcde)
```

```
contrast_ADE= np.sum(-one + a  - b  + ab  - c  + ac  - bc  + abc  + d  - ad  + bd  - abd  + cd  - acd  + bcd  - abcd
      + e  - ae + be - abe + ce - ace + bce - abce - de + ade - bde + abde - cde  + acde - bcde + abcde)
contrast_BDE= np.sum(-one - a  + b  + ab  - c  - ac  + bc  + abc  + d  + ad  - bd  - abd  + cd  + acd  - bcd  - abcd
      + e  + ae - be - abe + ce + ace - bce - abce - de - ade + bde + abde - cde  - acde + bcde + abcde)
contrast_ABDE= np.sum(one - a  - b  + ab  + c  - ac  - bc  + abc  - d  + ad  + bd  - abd  - cd  + acd  + bcd  - abcd
      - e  + ae + be - abe - ce + ace + bce - abce + de - ade - bde + abde + cde  - acde - bcde + abcde)
contrast_CDE= np.sum(-one - a  - b  - ab  + c  + ac  + bc  + abc  + d  + ad  + bd  + abd  - cd  - acd  - bcd  - abcd
      + e  + ae + be + abe - ce - ace - bce - abce - de - ade - bde - abde + cde  + acde + bcde + abcde)
contrast_ACDE= np.sum(one - a  + b  - ab  - c  + ac  - bc  + abc  - d  + ad  - bd  + abd  + cd  - acd  + bcd  - abcd
      - e  + ae - be + abe + ce - ace + bce - abce + de - ade + bde - abde - cde  + acde - bcde + abcde)
contrast_BCDE= np.sum(one + a  - b  - ab  - c  - ac  + bc  + abc  - d  - ad  + bd  + abd  + cd  + acd  - bcd  - abcd
      - e - ae + be + abe + ce + ace - bce - abce + de + ade - bde - abde - cde  - acde + bcde + abcde)
contrast_ABCDE= np.sum(-one + a  + b  - ab  + c  - ac  - bc  + abc  + d  - ad  - bd  + abd  - cd  + acd  + bcd  - abcd
      + e  - ae - be + abe - ce + ace + bce - abce - de + ade + bde - abde + cde - acde - bcde + abcde)

contrasts_all = np.array([contrast_A, contrast_B, contrast_AB, contrast_C, contrast_AC, contrast_BC, contrast_ABC,
      contrast_D, contrast_AD, contrast_BD, contrast_ABD, contrast_CD, contrast_ACD, contrast_BCD, contrast_ABCD,
      contrast_E, contrast_AE, contrast_BE, contrast_ABE, contrast_CE, contrast_ACE, contrast_BCE, contrast_ABCE,
      contrast_DE, contrast_ADE, contrast_BDE, contrast_ABDE, contrast_CDE, contrast_ACDE, contrast_BCDE, contrast_ABCDE])


# Sum Squares
num_effects = np.power(2,k)-1
num_elements = num_effects+2
sum_squares = np.ones(num_elements) #All effects plus error and total
for i in range(num_effects):
    sum_squares[i] = np.square(contrasts_all[i])/(n*np.power(2,k))
total_mean = np.mean(total)
SST = np.sum(np.square(total - total_mean))
SSE = SST - np.sum(sum_squares[0:num_effects])
sum_squares[num_effects] = SSE
sum_squares[num_effects+1] = SST

#Degrees of Freedom
DF = np.ones(num_elements)
DF[num_effects] = np.power(2,k)*(n-1) # Error DoF
DF[num_effects+1] = n*np.power(2,k)-1 # Total DoF

#Mean Squares
mean_squares = np.ones(sum_squares.size)
for i in range(num_elements):
    mean_squares[i] = sum_squares[i]/DF[i]
MSE = mean_squares[num_effects]

#F-values
f_vals = np.ones(num_elements)
f_vals[num_effects:] = -1
f_crits = np.ones(num_elements)
f_crits[num_effects:] = -1
```

```
232
233 #P-values
234 p_vals = np.ones(num_elements)
235 p_vals[num_effects:] = -1
236
237 #Effect Estimates
238 effects = np.ones(num_elements)
239 effects[num_effects:] = -1
240
241 #Response variable averages
242 means = np.ones(num_elements)
243 means[num_effects:] = -1
244
245 #Build datafile
246 for i in range(num_effects):
247     F0 = mean_squares[i]/MSE
248     f_vals[i] = F0
249     f_crits[i] = stats.f.ppf(1-alpha,DF[i],DF[num_effects])
250     p_vals[i] = 1 - stats.f.cdf(F0, DF[i],DF[num_effects])
251     effects[i] = contrasts_all[i]/(n*np.power(2,k-1))
252     means[i] = means_all[i]
253
254 anova_df_numpy = np.array([means, effects, sum_squares, DF, mean_squares, f_vals,
     ↪  f_crits, p_vals])
255 anova_df_pandas = pd.DataFrame(data=anova_df_numpy.T, index=df_index,
     ↪  columns=['Sample Mean','Effect Est.','Sum of Squares', 'df', 'Mean Square',
     ↪  'F0', 'F Threshold', 'p-value'])
256 anova_df_pandas = anova_df_pandas.replace(to_replace=-1,value='')
257
258 #The deprecated function. Changes p-value to float64 so it can be rounded properly
259 anova_df_pandas['p-value']=
     ↪  anova_df_pandas['p-value'].convert_objects(convert_numeric=True)
260 print("Unoptimized Mean: " + str(np.mean(one)))
261 print(anova_df_pandas.round(5))
262
263 ##Significant Factors
264 significant_factors = anova_df_pandas[(anova_df_pandas['p-value'] <
     ↪  alpha)].round(5)
265
266 #Ideal candidates are less then the unoptimized mean...
267 candidiate_factors = significant_factors[(significant_factors['Sample Mean'] <
     ↪  np.mean(one))].sort_values(by=['Sample Mean'])
268
269 longest = ""
270
271 #Print all the significant factors at chosen p-value
272 if significant_factors.empty == False:
273  anova_df_pandas[(anova_df_pandas['p-value'] < alpha)].to_csv("results/anova/" +
      ↪  sys.argv[6]+"-"+input_csv_parse[2]+"-"+rv[0]+"-"+rv[1]+"-anova-significant.csv")
274
275 #Print them all to another file
276 anova_df_pandas.to_csv("results/anova/" + sys.argv[6]+"-"+
     ↪  input_csv_parse[2]+"-"+rv[0]+"-"+rv[1]+"-anova-all.csv")
277
278 #If I have significant candidate factors with sample mean < unoptimized mean:
279 if candidiate_factors.empty == False:
280  print("\nSignificant Factors (alpha = " + str(alpha) + ")")
281  print(significant_factors.sort_values(by=['Sample Mean']))
282  #candidiate_factors_index = candidiate_factors[(candidiate_factors['Sample Mean']
      ↪  < np.mean(one))].sort_values(by=['Sample Mean']).index.array
283  candidiate_factors_index = candidiate_factors.sort_values(by=['Sample
      ↪  Mean']).index.array
284
285  for x in candidiate_factors_index:
286   if len(x) > len(longest):
287    longest = x
288
289  print("\n!!--Statistically Significant Effects--!! ")
```

```
290   all = ""
291   for y in candidiate_factors_index:
292    all = all + y + ","
293   print("Lowest observed mean (Target to Beat)")
294   print(int(significant_factors['Sample Mean'].min()))
295   print("Effects")
296
297  #If all my significant candidate factors actually have a worse sample mean:
298  else:
299   print("\n ***No statistically significant effects with sample mean < unoptimized
       ↪  mean***\n")
300   candidiate_factors_index = anova_df_pandas[(anova_df_pandas['Sample Mean'] <
       ↪  np.mean(one))].sort_values(by=['Sample Mean']).index.array
301
302   all = ""
303   for y in candidiate_factors_index:
304    all = all + y + ","
305
306   if len(all) == 0:
307
308    print("Unoptimized Mean")
309    print(str(np.mean(one)))
310    print("***ALL effects attempted have sample mean > unoptimized mean***")
311    print("NONE")
312    exit()
313
314   #Take the factor combo with the best sample mean and keep trying
315   print("Lowest observed sample mean (Target to Beat)")
316   print(int(anova_df_pandas[(anova_df_pandas['Sample Mean'] <
       ↪  np.mean(one))]['Sample Mean'].min()))
317   print("Next best guesses (produced a sample mean lower than unoptimized)")
318
319  if len(all) == 0:
320   print("NONE - ERROR") #I shouldn't get here...
321  else:
322   print(all.rstrip(','))
323
324  #Normplot of effects
325  if len(sys.argv) == 7:
326   fig = plt.figure(figsize=(6,4))
327   probscale.probplot(effects,plottype='prob',probax='y',problabel='Standard Normal
       ↪  Probabilities',bestfit=True)
328   plt.xlabel("Normal Probability Plot of Effect Estimates")
329   plt.title(rv[0] +" - " + rv[1] + " [" + sys.argv[6] +"]")
330   plt.tight_layout()
331   plt.savefig("results/anova/"+sys.argv[6]+"-"+input_csv_parse[2]+"-"+rv[0]+"-"+rv[1]+"-anova-normplo
```

best-mean.py

```
import pandas as pd
import numpy as np
import sys

#Best Mean Test
if len(sys.argv) <= 3:
 print("Not enough args usage: anova.py <*.csv> <rv1,rv2> <target to beat>")
 print("ex: best-mean.py testdata.csv nicdrop 95000")
 print("<rv> is response variable")
 exit()

target_to_beat = int(sys.argv[3]) #factors
rv = sys.argv[2].split(',')

data = pd.read_csv(sys.argv[1], header=[0,1])
response_var = data[[rv[0],'factors']]
response_var.columns = response_var.columns.get_level_values(1)

print("Re-run factor means")
print(response_var.groupby('code')[rv[1]].mean())

print("Lowest observed sample mean (target to beat)")
print(response_var.groupby('code')[rv[1]].mean().min())

#print factors still remaining as viable
candidiate_factors_index =
 ↪  response_var.groupby('code')[rv[1]].mean().index.array.to_numpy() #all
 ↪  factors from csv
improved_factors_bools = (response_var.groupby('code')[rv[1]].mean() <
 ↪  target_to_beat).to_numpy() #boolean series
all = ""
i=0
for y in candidiate_factors_index:
 if improved_factors_bools[i]:
  all = all + y + ","
 i=i+1
print("Effects")
if len(all) == 0:
 print("NONE")
 exit()
print(all.rstrip(','))
```

prep-playbook.yml

```
############Setup pre-reqs##############
#https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variables-discovered-fr
- name: Prepare Sensors
  connection: ssh
  gather_facts: true
  hosts: all
  vars_files: vars.yml

###Install environment
  tasks:

    #Only supporting debian sensors at this point though yum should not be a
     ↪  problem as future work
  - name: Check OS Version
    fail: msg="Currently only supports Debian based sensors."
    when: ansible_facts['os_family'] != "Debian"

  #This assumes all your SSH passwords are the same.
  #Or, run this playbook once per host with "-t auth -l "hostname" -k -K"
  - name: Set SSH Keys
    authorized_key:
        user: "{{ansible_user}}"
        state: present
        key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
    tags: auth

  - name: Setup Passwordless sudo
    lineinfile:
        path: /etc/sudoers
        state: present
        regexp: '^%sudo'
        line: '%sudo ALL=(ALL) NOPASSWD: ALL'
        validate: 'visudo -cf %s'
    tags: auth
    become: yes

  - name: Install Prerequisites (may take awhile)
    apt:
      name: "{{ packages }}"
      #update_cache: yes
      force_apt_get: yes
    vars:
        packages:
        - lshw
        - build-essential
        - libyaml-0-2
        - libyaml-dev
        - pkg-config
        - zlib1g
        - zlib1g-dev
        - libnet1-dev
        - libpcre3
        - libpcre3-dbg
        - libpcre3-dev
        - libyaml-dev
        - libpcap-dev
        - python-yaml
        - libcap-ng-dev
        - libcap-ng0
        - libmagic-dev
        - liblz4-dev
        - libhtp-dev
        - libjansson-dev
        - libnspr4-dev
        - libnss3-dev
        - rustc
        - libgeoip-dev
        - liblua5.1-dev
        - libhiredis-dev
        - libevent-dev
```

```
        - cargo
        - bc
        - git
        - tcpdump
        - python-apt
        - make
        - nano
        - locate
    ignore_errors: yes
    become: yes

  - name: Create Sensor Directory
    file:
      path: "{{ sensor_dir }}"
      state: directory
      owner: "{{ ansible_user }}"
      group: "{{ ansible_user }}"
      mode: '0777'
    become: yes

  - name: Create Log Directory
    file:
      path: "{{ sensor_dir }}/log"
      state: directory
      mode: '0777'

  - name: Copy Default Suricata Rules
    copy:
      src: suricata.rules
      dest: "{{ sensor_dir }}/suricata.rules"
      mode: '0766'
    tags: suricata

  - name: Download and Unpack Suricata Source
    unarchive:
      src: https://www.openinfosecfoundation.org/download/suricata-5.0.0.tar.gz
      dest: "{{ sensor_dir }}"
      remote_src: yes
      creates: "{{ sensor_dir }}/suricata-5.0.0/"
    register: suricata_source
    tags: suricata

  - name: Check Previuos Installs
    stat:
      path: "/usr/bin/suricata"
    register: previous_failure
    tags: suricata

  - name: Run configure
    shell: "./configure --prefix=/usr/ --sysconfdir=/etc/ --localstatedir=/var/"
    args:
      chdir: "{{ sensor_dir }}/suricata-5.0.0"
      creates: "{{ sensor_dir }}/suricata-5.0.0/Makefile"
    register: suricata_version
    when: suricata_source is changed or not previous_failure.stat.exists
    tags: suricata

  - name: Build Suricata Latest
    shell: make
    args:
      chdir: "{{ sensor_dir }}/suricata-5.0.0/"
    register: suricata_build
    when: suricata_version is changed
    tags: suricata

  - name: Install Suricata Latest
    shell: |
        make install-conf
        make install
    args:
      chdir: "{{ sensor_dir }}/suricata-5.0.0/"
```

```
141       become: yes
142       tags: suricata
143       when: suricata_build is changed
144
145     - name: Standardize Hostname
146       shell: |
147           hostnamectl set-hostname {{ ansible_hostname }}
148           echo "127.0.0.1    {{ ansible_hostname }}" > /etc/hosts
149       become: yes
150       when: ansible_facts['hostname'] != ansible_hostname
151
152     - name: Clean Up old Results
153       shell: |
154           rm -rf {{ sensor_dir }}/*.csv
155           rm -rf {{ sensor_dir }}/*.pid
156           rm -rf counters
157       args:
158           warn: false
159       become: yes
160       tags: wipe
161
162     - name: Reboot to apply updates
163       reboot:
164       become: yes
165       tags: reboot
166
167   ##Generator Setup##
168   #This assumes your ansible host is also the traffic generator. It doesnt have to
        ↪  be..
169   - name: Prepare Traffic Generator
170     connection: local
171     gather_facts: true
172     hosts: localhost
173
174     tasks:
175     - name: Install Prerequisites (may take awhile)
176       when: ansible_facts['os_family'] == "Debian"
177       apt:
178         name: "{{ packages }}"
179         force_apt_get: yes
180       vars:
181           packages:
182           #this may be missing some things to build fragroute support
183           #fragrout and tcpreplay-edit are needed to fix dataset pcaps that have
                ↪  jumbo (up to 64K) frames
184           - build-essential
185           - libpcap-dev
186           - dnet-common
187           - libdumbnet-dev
188           - libdnet
189           - libevent1-dev
190           - libdnet-dev
191           - libdumbnet1
192           - nano
193           - locate
194           - docker.io
195       ignore_errors: yes
196       become: yes
197
198     - name: Set Docker Permissions
199       shell: usermod -aG docker $USER
200       become: yes
201       ignore_errors: yes
202
203     - name: Create Docker Network
204       docker_network:
205         name: experiment
206
207     - name: Grab required Python packages
208       pip:
209          name:
```

```
        - matplotlib
        - pandas
        - scipy
        - numpy
        - researchpy
        - seaborn
        - probscale
        - statsmodels
        - pytest
      become: yes

    - name: Create Generator Directory
      file:
        path: |
          "{{ generator_dir }}"
          "{{ generator_dir }}/results"
          "{{ generator_dir }}/results/verbose"
          "{{ generator_dir }}/results/anova"
        state: directory
        mode: '0777'
      become: yes


      ## netmap ##
    - name: Clone Latest netmap Source
      git:
          repo: https://github.com/luigirizzo/netmap.git
          dest: "{{ generator_dir }}/netmap"
      register: netmap_source
      tags: netmap

      #Using this specific NIC so select it's driver manually
      #Netmaps auto detect is flaky esp on kernel 5+
    - name: Build netmap Makefile
      shell: ./configure --select-version=igb:5.3.5.39 --driver-suffix=-netmap
      args:
        chdir: "{{ generator_dir }}/netmap"
        #creates: "{{ generator_dir }}/netmap/config.status"
      register: netmap_version
      when: netmap_source is changed
      tags: netmap

    - name: Build netmap
      shell: make
      args:
          chdir: "{{ generator_dir }}/netmap"
          #creates: "{{ generator_dir }}/netmap/netmap.ko"
      register: netmap_build
      when: netmap_version is changed
      tags: netmap

    - name: Install netmap
      shell: make install
      args:
          chdir: "{{ generator_dir }}/netmap"
      become: yes
      register: netmap_install
      when: netmap_build is changed
      tags: netmap

      #anytime the kernel is updated these modules will need rebuilt
      #if this fails after a reboot the kernel may have changed,
      #delete the netmap/ folder and rebuild it
    - name: Enable Netmap Drivers
      shell: |
       rmmod igb 2> /dev/null
       rmmod igb_netmap 2> /dev/null
       rmmod netmap 2> /dev/null
       insmod {{ generator_dir }}/netmap/netmap.ko
       insmod {{ generator_dir }}/netmap/igb/igb-netmap.ko
      become: yes
```

```
    tags: netmap

  ## tcpreplay ##
- name: Clone tcpreplay Source
  unarchive:
      src:
       ↪  https://github.com/appneta/tcpreplay/releases/download/v4.3.1/tcpreplay-4.3.1.tar.xz
      dest: "{{ generator_dir }}"
      remote_src: yes
      creates: "{{ generator_dir }}/tcpreplay-4.3.1/"
  register: tcpreplay_source

- name: Build tcpreplay Makefile
  shell: "./configure --with-netmap={{ generator_dir }}/netmap"
  args:
    chdir: "{{ generator_dir }}/tcpreplay-4.3.1"
    #creates: "{{ generator_dir }}/tcpreplay-4.3.1/Makefile"
  when: tcpreplay_source is changed or netmap_build is changed
  register: tcpreplay_version

- name: Build tcpreplay
  shell: make
  args:
    chdir: "{{ generator_dir }}/tcpreplay-4.3.1/"
    #creates: "{{ generator_dir }}/tcpreplay-4.3.1/src/tcpreplay"
  register: tcpreplay_build
  when: tcpreplay_version is changed

- name: Install tcpreplay
  shell: make install
  args:
    chdir: "{{ generator_dir }}/tcpreplay-4.3.1/"
  become: yes
  when: tcpreplay_version is changed

- name: Pull Elasticsearch Docker Image
  docker_container:
      name: elastic
      image: elasticsearch:7.5.0

- name: Pull Kibana Docker Image
  docker_container:
      name: kibana
      image: kibana:7.5.0



#   docker run -d --name kibana --net experiment -p 5601:5601 kibana:7.5.0
#    docker run -d --name elasticsearch --net experiment -p 9200:9200 -p 9300:9300
 ↪   -e "discovery.type=single-node" elasticsearch:7.5.0
```

inventory.yml

```yaml
#Interface numbers like to jump around on reboot...
all:
  children:
    sensors:
        children:
            rpi:
                hosts:
                    rpi3bp:
                        send_interface: eth7
                        capture_interface: eth0
                        line_pps_limit: 160000

                        #Facts inherited from previous tests
                        interface_pps_limit: 70000
                        rps_mask: 0 #0000 All made it worse
                        NAPI_budget_best: 300
                        backlog_best: 32768
                        backlog_weight_best: 1200

                    rpi4:
                        send_interface: eth8
                        capture_interface: eth0
                        line_pps_limit: 160000

                        #Facts inherited from previous tests
                        interface_pps_limit: 158000
                        rps_mask: 0 #0000 All made it worse
                        NAPI_budget_best: 300
                        backlog_best: 1000
                        backlog_weight_best: 1200
                vars:
                    ansible_user: pi
                    ansible_become_method: sudo
                    experiment_dir: /exp
                    ansible_python_interpreter: /usr/bin/python

            nvidia:
                hosts:
                    tx1:
                        capture_interface: eth0
                        send_interface: eth2
                        line_pps_limit: 160000

                        #Facts inherited from previous tests
                        interface_pps_limit: 102000
                        rps_mask: E #1110 (4 cpu total, CPU0 handling IRQ)
                        NAPI_budget_best: 1200
                        backlog_best: 32768
                        backlog_weight_best: 1200
                    tx2:
                        capture_interface: eth1
                        send_interface: eth6
                        line_pps_limit: 160000

                        #Facts inherited from previous tests
                        interface_pps_limit: 148000
                        rps_mask: 3E #0011 1110 (6 cpu total, CPU0 handling IRQ)
                        NAPI_budget_best: 300
                        backlog_best: 1000
                        backlog_weight_best: 300
                    xavier:
                        capture_interface: eth0
                        send_interface: eth7 #shared with rpi3bp due to quad nic
                         ↪  limit
                        line_pps_limit: 160000

                        #Facts inherited from previous tests
                        interface_pps_limit: 159000
                        rps_mask: FE #1111 1110 (8 cpu total) CPU0 handling IRQ
                        NAPI_budget_best: 1200
```

```
                        backlog_best: 1000
                        backlog_weight_best: 300
                vars:
                    ansible_user: nvidia
                    ansible_become_method: sudo
                    experiment_dir: /exp

vars:
  ansible_host_dir: /ansible_working_dir
```

vars.yml

```
###Master variable file for all playbooks

### 5 Factors
total_factors: 5
factor_combos: [N,A,B,AB,C,AC,BC,ABC,
                D,AD,BD,ABD,CD,ACD,BCD,ABCD,
                E,AE,BE,ABE,CE,ACE,BCE,ABCE,
                DE,ADE,BDE,ABDE,CDE,ACDE,BCDE,ABCDE]

### 4 Factors
#total_factors: 4
#factor_combos: [N,A,B,AB,C,AC,BC,ABC,
#                D,AD,BD,ABD,CD,ACD,BCD,ABCD]

total_combinations: "{{2 ** total_factors }}" #2^k

#replicates: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
replicates: [1,2,3,4,5]
#replicates: [1,2]
#must be an array like this for it to work


###Variable controls###

#Used to store values actually attempted
A_levels: []
B_levels: []
C_levels: []
D_levels: []
E_levels: []
significant_factors_history: []
last_loop_best_history: []

#New API (NAPI) IRQ budget
#default = 300. Will eventually hit hard time limit of "2 jiffies"
NAPI_budget: 1200

#Max Kernel Backlog
#default = 1000
backlog:  32768

#Socket receive queue memory, in bytes
#default = 212992
rmem_max: 83886080

#Receive Flow Steering (RFS) hash table size and per queue flow count
#default = off
rfs_table: 32768
rfs_flow_cnt: 32768

#NAPI backlog poll loop weight
#default = 64
backlog_weight: 1200

#libpcap buffer size, in KiB
#default = 2 MB  = ~2000 KiB
libpcap_buffer: 250000 #64 MB
libpcap_buffer_size: 2000 #initial

#suricata runmode
#default = autofp (flow pin)
suricata_runmode: workers

#suricata max-pending-packets
#default = 1024
suricata_max_pending: 8096

#suricata detect memory grouping aggressiveness
#default = medium
```

```
suricata_detect_profile: high

#af-packet memory map ring feature
#default = no
af_packet_mmap: 'yes'

###Static controls###

#RX_checksumming on/off
#More research may be worthwhile here. It seemed to boost suricata when off?
#lan78xx driver on rpi3 fails to read random traffic from pkt-gen when on.
#probably due to UDP header / payload mismatch
#default = on
rx_checksum_status: 'off'

#RX_timestamping on/off
#Moves timestamping of rx packets to after they enter load balanced RPS queue vs
 ↪  before
#default = on
rx_timestamp_status: '0'

#Large Receive / Generic Receive offload on/off.
#Off for suricata
#default = on
lro_status: 'off'
gro_status: 'off'


###other vars###
packet_size_max: "1500"
packet_size_min: "64"
num_packets_30sec: "{{ line_pps_limit|int * 30 }}"
num_packets_cic_monday: 17997887 #avg 590 bytes
num_packets_cic_thursday: 14106798 #avg size 576 bytes
test_counter: 1
loop_multiplier: 1
```

Ansible Templates

These are the template playbooks used to build the three specific tests
See begining of Appendix D for preparatory setup and general tips & tricks

general-benchmark-main.yml

```yaml
#This playbook acts as the "main" function
#It has to be its own file since the "tasks:" keyword
#can only appear once within all the loops
- name: Generic Experiment Template
  connection: ssh
  hosts: all
  vars_files: vars.yml
  gather_facts: true

  tasks:
    - name: Clear gathered facts from all currently targeted hosts
      meta: clear_facts #start fresh

    - name: Clean Up old Results
      shell: |
          rm -rf "{{ experiment_dir }}"/*.csv
      args:
          warn: false
      become: yes

    - name: Record Initial Variable Levels
      set_fact:
        A_levels: "{{A_levels}} + [ '%%%%%' ]"
        B_levels: "{{B_levels}} + [ '%%%%%' ]"
        C_levels: "{{C_levels}} + [ '%%%%%' ]"
        D_levels: "{{D_levels}} + [ '%%%%%' ]"
        E_levels: "{{E_levels}} + [ '%%%%%' ]"

    - name: Begin New Test
      include_tasks: general-benchmark-outerloop.yml


   - name: Done Looping, Store Final Result Facts
      set_fact:
        your_result: "{{ %%%%% }}"
        A_best: "{{ A_levels[-1] }}"
        B_best: "{{ B_levels[-1] }}"
        C_best: "{{ C_levels[-1] }}"
        D_best: "{{ D_levels[-1] }}"
        E_best: "{{ E_levels[-1] }}"
        cacheable: yes

    - name: Store Results File
      local_action:
          module: shell
          _raw_params: |
              echo "{{inventory_hostname}}  Test Final"\
              > results/"{{inventory_hostname}}"-testxyz-final.log

              echo "- A levels: {{ A_levels | to_yaml }}\
              - B levels: {{ B_levels | to_yaml }}\
              - C levels: {{ C_levels | to_yaml }}\
              - D levels: {{ D_levels | to_yaml }}\
              - E levels: {{ E_levels | to_yaml }}"\
              >> results/"{{inventory_hostname}}"-testxyz-final.log

              echo "- Best (A): {{ A_best | to_yaml }}\
              - Best (B): {{ B_best | to_yaml }}\
              - Best (C): {{ C_best | to_yaml }}\
              - Best (D): {{ D_best | to_yaml }}\
              - Best (E): {{ E_best | to_yaml }}"\
              >> results/"{{inventory_hostname}}"-testxyz-final.log

              echo "- Number of Loops: {{ test_counter | to_yaml }}\
              - Factor History: {{ significant_factors_history | to_yaml }}\
              - Loop Bests: {{ last_loop_best_history | to_yaml }}\
              - your_result: {{ %%%%% }}"\
```

```
68                >> results/"{{inventory_hostname}}"-testxyz-final.log
69        ignore_errors: yes
70
71
72    - name: Display Variable States
73      debug:
74        msg: "
75        - A levels: {{ A_levels | to_yaml }}
76        - B levels: {{ B_levels | to_yaml }}
77        - C levels: {{ C_levels | to_yaml }}
78        - D levels: {{ D_levels | to_yaml }}
79        - E levels: {{ E_levels | to_yaml }}"
80
81    - name: Display Factor History
82      debug:
83        msg: "
84        - Number of Loops: {{ test_counter | to_yaml }}
85        - Factor History: {{ significant_factors_history | to_yaml }}"
86
87    - name: Display Final Optimization Results
88      debug:
89        msg: "
90        - Best (A): {{ A_best | to_yaml }}
91        - Best (B): {{ B_best | to_yaml }}
92        - Best (C): {{ C_best | to_yaml }}
93        - Best (D): {{ D_best | to_yaml }}
94        - Best (E): {{ E_best | to_yaml }}"
95
96    - name: Final Result
97      debug:
98        msg: "Your final score: {{ %%%%% }}"
```

general-benchmark-outerloop.yml

```
- name: Begin First Middle Loop
  include_tasks: general-benchmark-middleloop.yml
  loop: "{{ 5_factor_combos }}" #This will run 2^(#factors) times
  loop_control:
   loop_var: current_factor_list
   index_var: factor_idx
   extended: yes
  when: "test_counter == 1"
  tags:
    - workload
    - initial

- name: Run Initial ANOVA Test
  local_action:
    module: shell
    _raw_params: |
        python anova.py 'results/{{ inventory_hostname }}-results-run1.csv'
         ↪  <<**RESPONSE VARIABLE**>> "{{total_factors}}" "{{replicates|length}}"
         ↪  0.05 {{ inventory_hostname }}
  register: anova
  tags: anova
  ignore_errors: yes
  changed_when: false
  when: "test_counter == 1"

- name: Set Initial Significant Factors
  set_fact:
     significant_factors_array: "{{anova.stdout_lines[anova.stdout_lines|length
      ↪  -1].split(\",\")}}"
     significant_factors_string: "{{anova.stdout_lines[anova.stdout_lines|length
      ↪  -1]}}"
     significant_factors_history: "{{significant_factors_history}} + [
      ↪  '{{anova.stdout_lines[anova.stdout_lines|length -1]}}' ]"
     target_to_beat: "{{anova.stdout_lines[anova.stdout_lines|length -3]|int}}"
     last_loop_best: "{{anova.stdout_lines[anova.stdout_lines|length -3]|int}}"
     last_loop_best_history: "{{last_loop_best_history}} + [
      ↪  '{{anova.stdout_lines[anova.stdout_lines|length -3]|int}}' ]"
  when: "test_counter == 1"

- name: Record Significant A Level
  set_fact:
   A_levels: "{{A_levels}} + [ '{{%%%A%%%}}' ]"
  when: "'A' in significant_factors_string"

- name: Record Significant B Level
  set_fact:
   B_levels: "{{B_levels}} + [ '{{%%%B%%%}}' ]"
  when: "'B' in significant_factors_string"

- name: Record Significant C Level
  set_fact:
   C_levels: "{{C_levels}} + [ '{{%%%C%%%}}' ]"
  when: "'C' in significant_factors_string"

- name: Record Significant D Level
  set_fact:
   D_levels: "{{D_levels}} + [ '{{%%%D%%%}}' ]"
  when: "'D' in significant_factors_string"

- name: Record Significant E Level
  set_fact:
   E_levels: "{{E_levels}} + [ '{{%%%E%%%}}' ]"
  when: "'E' in significant_factors_string"

- name: Increment Test Counter
  set_fact:
     test_counter: "{{ test_counter | int + 1 }}"

```

```
- name: Reset Results
  shell: |
      rm -rf "{{ experiment_dir }}"/*.csv
  args:
      warn: false
  become: yes

- name: Target To Beat
  debug:
    msg: "Target to Beat is {{target_to_beat}}. Using Factors
     ↪  {{significant_factors_array | to_yaml}}"

- name: Increment Loop Multiplier
  set_fact:
     loop_multiplier: "{{ test_counter|int**test_counter|int }}"

- name: Continue Middle Loop
  include_tasks: general-benchmark-middleloop.yml
  loop: "{{significant_factors_array}}"
  loop_control:
    extended: yes
    loop_var: current_factor_list
    index_var: inner_index
  when: "'NONE' not in significant_factors_array"

- name: Last Loop Results
  debug:
    msg: "Last Loop best was {{last_loop_best}}. Needs to beat {{target_to_beat}}.
     ↪  Was Iteration {{ test_counter }}"

- name: Check Last Loop Results
  block:
    - name: Check Recursive Base Case
      fail:
        #Maybe run a set number of times....
        msg: "Continuing {{test_counter}} < 4"
      when: "test_counter | int < 4"

       #Or have a target
        #msg: "Still room to improve {{last_loop_best}} < {{target_to_beat}}"
      #when: "last_loop_best < target_to_beat"

  rescue:
    - name: Update Target to Beat
      set_fact:
         target_to_beat: "{{ last_loop_best }}"
         loop_multiplier: "{{ test_counter|int**test_counter|int }}"

    #Recursively call self to keep going
    - name: Begin New Round
      include_tasks: general-benchmark-recursive.yml
```

general-benchmark-middleloop.yml

```
#This playbook is the "middle" loop
- name: Reboot to Defaults. Beginning Factor {{current_factor_list}}
  ↪  ({{ansible_loop.index}} of {{ansible_loop.length}})
  reboot:
  become: yes
  tags: skippable

  #If any configuration changes are not undone with a reboot,
  #add a playbook here to manually "revert" them

- name: Set Static Controls
  include_tasks: general-static-controls.yml

- name: Set Variable Factor Controls
  include_tasks: general-variable-controls.yml

- name: Copy Performance Monitor
  copy:
    src: gather_stats.bash
    dest: "{{experiment_dir}}/gather_stats.bash"
    mode: '0755'

  #Run repeats
- name: Begin Inner Loop
  include_tasks: general-benchmark-innerloop.yml
  loop: "{{ replicates }}"
  loop_control:
    extended: yes
    loop_var: inner_counter
    index_var: inner_idx
  tags: workload

- name: End of Run Best Mean Test {{test_counter}}
  local_action:
    module: shell
    _raw_params: |
        python best-mean.py 'results/{{ inventory_hostname
        ↪  }}-results-run{{test_counter}}.csv' <<**RESPONSE VARIABLE**>>
        ↪  "{{target_to_beat}}"
  register: anova
  tags: anova
  #if the last item in loop and not inital run
  when: "ansible_loop.revindex == 1 and test_counter > 1"
  changed_when: false
  ignore_errors: yes

- name: Update Last Middle Loop Best
  set_fact:
   last_loop_best: "{{anova.stdout_lines[anova.stdout_lines|length -3]|int}}"
   significant_factors_array: "{{anova.stdout_lines[anova.stdout_lines|length
   ↪  -1].split(\",\")}}"
   significant_factors_string: "{{anova.stdout_lines[anova.stdout_lines|length
   ↪  -1]}}"
   significant_factors_history: "{{significant_factors_history}} + [
   ↪  '{{anova.stdout_lines[anova.stdout_lines|length -1]}}' ]"
   last_loop_best_history: "{{last_loop_best_history}} + [
   ↪  '{{anova.stdout_lines[anova.stdout_lines|length -3]|int}}' ]"
  when: "ansible_loop.revindex == 1 and test_counter > 1"
```

general-static-controls.yml

```
  ###Blanket Optimizations (always apply to all)

- name: Bump RPi Throttling Temp (3B+ only)
  lineinfile:
    path: /boot/config.txt
    regex: "temp_soft_limit="
    line: temp_soft_limit=70.0
  when: "'nvidia' not in group_names"
  become: yes

- name: Set MAXN Power Profile on NVIDIA Boards
  shell: |
    nvpmodel -m 0
    jetson_clocks
  become: yes
  when: "'nvidia' in group_names"
```

general-variable-controls.yml

```
 ##FACTORS UNDER EXPERIMENT

  ###FACTOR A###
- name: (Factor A) <%%%>
  shell: <%%%>
  become: yes
  when: "'A' in current_factor_list"

  ###FACTOR B###
- name: (Factor B) <%%%>
  shell: <%%%>
  become: yes
  when: "'B' in current_factor_list"

  ###FACTOR C###
- name: (Factor C) <%%%>
  shell: <%%%>
  become: yes
  ignore_errors: yes
  when: "'C' in current_factor_list"

  ###FACTOR D###
- name: (Factor D) <%%%>
  shell: <%%%>
  become: yes
  ignore_errors: yes
  when: "'D' in current_factor_list"

  ###FACTOR E###
- name: (Factor E) <%%%>
  shell: <%%%>
  become: yes
  ignore_errors: yes
  when: "'E' in current_factor_list"

```

general-benchmark-innerloop.yml

```
#This playbook is the "inner" loop
  - name: Launch Performance Monitor (Factors {{ current_factor_list }})
    shell: "./gather_stats.bash <<**PID**>> <<**SAMPLE RATE**>> {{ current_factor_list }}"
    args:
      chdir: "{{ experiment_dir }}/"
    register: results_async
    poll: 0
    async: 3600
    become: yes
    changed_when: false

   # <<**YOUR WORKLOAD TASK(S) GOES HERE.....**>>
   #       SEE RATELIMIT TEST FOR EXAMPLE

  - name: Stop Everything
    shell: kill "$(cat gather.pid)"
    args:
      chdir: "{{ experiment_dir }}"
    become: yes
    ignore_errors: yes
    failed_when: false

  - name: Wait for Results
    async_status: jid="{{ results_async.ansible_job_id }}"
    become: yes
    register: results
    until: results.finished
    retries: 30
    failed_when: false
    ignore_errors: yes

  - name: Copy Verbose Results
    fetch:
      src: "{{experiment_dir}}/{{ inventory_hostname }}-results-verbose.csv"
      dest: "results/verbose/{{ inventory_hostname }}-results-run{{test_counter}}-verbose.csv"
      flat: yes
    changed_when: false

  - name: Copy Totals
    fetch:
      src: "{{experiment_dir}}/{{ inventory_hostname }}-results.csv"
      dest: "results/{{ inventory_hostname }}-results-run{{test_counter}}.csv"
      flat: yes
    changed_when: false

  - name: Display Running Results
    debug:
        var: results.stdout
```

Bibliography

1. OISF, "Runmodes - Suricata Documentation," 2019. [Online]. Available: https://suricata.readthedocs.io/en/latest/performance/runmodes.html

2. S. Helme, "The encrypted web is coming!" 2016. [Online]. Available: https://scotthelme.co.uk/the-encryted-web-is-coming/

3. M. Pilkington, "Protecting Privileged Domain Accounts: Network Authentication In-Depth," 2012. [Online]. Available: https://digital-forensics.sans.org/blog/2012/09/18/protecting-privileged-domain-accounts-network-authentication-in-depth

4. D. Franklin, "NVIDIA® Jetson TX1 Supercomputer-on-Module Drives Next Wave of Autonomous Machines," 2015. [Online]. Available: https://devblogs.nvidia.com/nvidia-jetson-tx1-supercomputer-on-module-drives-next-wave-of-autonomous-machines/

5. Imagimob, "Edge computing needs Edge AI," 2018. [Online]. Available: https://www.imagimob.com/blog/edge-computing-needs-edge-ai

6. S. M. Z. Iqbal, Y. Liang, and H. Grahn, "ParMiBench - An open-source benchmark for embedded multiprocessor systems," *IEEE Computer Architecture Letters*, vol. 9, no. 2, pp. 45–48, 7 2010.

7. MITRE, "Lateral Movement," 2019. [Online]. Available: https://attack.mitre.org/tactics/TA0008/

8. RedHat, "How Ansible Works," 2020. [Online]. Available: https://www.ansible.com/overview/how-ansible-works

9. K. Smathers, "EdgeBench Repository," 2020. [Online]. Available: https://github.com/Jump-Industries/EdgeBench

10. MarketWatch, "Cyber Security Market Size and Share 2019 Global Industry Demand, Growth Analysis, Revenue and Forecast 2023," 2019. [Online]. Available: https://www.marketwatch.com/press-release/cyber-security-market-size-and-share-2019-global-industry-demand-growth-analysis-revenue-and-forecast-2023-2019-05-30

11. C. Timberg, "The real story of how the Internet became so vulnerable," 2015. [Online]. Available: https://www.washingtonpost.com/sf/business/2015/05/30/net-of-insecurity-part-1/?utm_term=.37f5b487e842

12. D. D. Clark, "The design philosophy of the DARPA internet protocols," in *Symposium Proceedings on Communications Architectures and Protocols, SIGCOMM 1988*, vol. 18. ACM, 1988, pp. 106–114. [Online]. Available: http://portal.acm.org/citation.cfm?doid=52325.52336

13. K. Ingham and S. Forrest, "A history and survey of network firewalls," *University of New Mexico, Tech. Rep*, 2002.

14. C. Stoll, "Stalking the wily hacker," *Commun. ACM*, vol. 31, no. 5, pp. 484–497, 1988.

15. B. Cheswick, "An Evening with Berferd in which a cracker is Lured, Endured, and Studied," in *Proc. Winter USENIX Conference, San Francisco*, 1992, pp. 20–24.

16. C. Barker, "Hackers and defenders continue cybersecurity game of cat and mouse," 2016. [Online]. Available: https://www.zdnet.com/article/hackers-and-defenders-continue-cyber-security-game-of-cat-and-mouse/

17. K. Rekouche, "Early Phishing," *CoRR*, vol. abs/1106.4, 2011. [Online]. Available: http://arxiv.org/abs/1106.4692

18. E. Chien, "W97M.Melissa.A — Symantec," 2000. [Online]. Available: https://www.symantec.com/security-center/writeup/2000-122113-1425-99

19. ——, "VBS.LoveLetter.Var," 2000. [Online]. Available: https://www.symantec.com/security-center/writeup/2000-121815-2258-99

20. I. Grigg, "Financial Cryptography: GP4.3 - Growth and Fraud - Case #3 - Phishing," 2005. [Online]. Available: https://www.financialcryptography.com/mt/archives/000609.html

21. G. Keizer, "Suspected Chinese spear-phishing attacks continue to hit Gmail users," 2011. [Online]. Available: https://www.computerworld.com/article/2510237/suspected-chinese-spear-phishing-attacks-continue-to-hit-gmail-users.html

22. C. Drew and J. Markoff, "SecurID Breach Suggested in Hacking Attempt at Lockheed - The New York Times," 2011. [Online]. Available: https://www.nytimes.com/2011/05/28/business/28hack.html

23. Anti Phishing Working Group, "APWG — Phishing Activity Trends Reports," 2018. [Online]. Available: http://www.antiphishing.org/trendsreports/

24. C. Doman, "The First Cyber Espionage Attacks: How Operation Moonlight Maze made history," 2016. [Online]. Available: https://medium.com/@chris_doman/the-first-sophistiated-cyber-attacks-how-operation-moonlight-maze-made-history-2adb12cc43f7

25. B. Brewin and D. Verton, "Cyberattacks spur talk of 3rd DOD network – FCW," 1999. [Online]. Available: https://fcw.com/Articles/1999/06/20/Cyberattacks-spur-talk-of-3rd-DOD-network.aspx

26. NSA, "Defense In Depth," 2015. [Online]. Available: https://apps.nsa.gov/iaarchive/customcf/openAttachment.cfm?FilePath=/iad/library/ia-guidance/archive/assets/public/upload/Defense-in-Depth.pdf&WpKes=aF6woL7fQp3dJizPwAnfAMjTrkVrn52VccuPXz

27. S. Woodside, "Defence in Depth: The medieval castle approach to internet security," 2016. [Online]. Available: https://medium.com/@sbwoodside/defence-in-depth-the-medieval-castle-approach-to-internet-security-6c8225dec294

28. P. Small, "Defense in Depth: An Impractical Strategy for a Cyber World," 2011. [Online]. Available: https://www.sans.org/reading-room/whitepapers/warfare/defense-depth-impractical-strategy-cyber-world-33896

29. L. J. Berman, "DoD (Finally) Begins Transition to RMF," 2014. [Online]. Available: https://www.itdojo.com/dod-finally-begins-transition-to-rmf/

30. B. Graham, "Hackers Attack Via Chinese Web Sites," 2005. [Online]. Available: http://www.washingtonpost.com/wp-dyn/content/article/2005/08/24/AR2005082402318.html

31. K. Zetter, "Google Hack Attack Was Ultra Sophisticated, New Details Show — WIRED," 2010. [Online]. Available: https://www.wired.com/2010/01/operation-aurora/

32. ——, *Countdown to Zero Day.* Broadway Books, 2014. [Online]. Available: https://gizmodo.com/the-incredible-tale-of-stuxnet-a-weapon-for-the-digita-1656811897

33. ——, "Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid — WIRED," 2016. [Online]. Available: https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/

34. M. Quinn and Soltra, "Threat context TLP WHITE Cyber security panel," Soltra Edge, Tech. Rep., 2015. [Online]. Available: https://slideplayer.com/slide/8390853/

35. S. Losey, "Go cyber: Airmen can earn cash and promotions, get set up for civilian life," 2017. [Online]. Available: https://www.airforcetimes.com/news/your-air-force/2017/10/10/go-cyber-airmen-can-earn-cash-and-promotions-get-set-up-for-civilian-life/

36. Mandiant FireEye, "M-Trends 2019," Mandiant, Tech. Rep., 2019. [Online]. Available: https://content.fireeye.com/m-trends

37. MITRE, "Defense Evasion - Enterprise," 2019. [Online]. Available: https://attack.mitre.org/tactics/TA0005/

38. R. A. Grimes and R. Lee, "Who wants to go threat hunting?" 2018. [Online]. Available: https://www.csoonline.com/article/3269779/who-wants-to-go-threat-hunting.html

39. Air Forces Cyber, "92ND CYBERSPACE OPERATIONS SQ," 2018. [Online]. Available: https://www.afcyber.af.mil/About-Us/Fact-Sheets/Display/Article/962008/92nd-cyberspace-operations-squadron/

40. D. Miessler, "The Difference Between Red, Blue, and Purple Teams," 2019. [Online]. Available: https://danielmiessler.com/study/red-blue-purple-teams/

41. USCYBERCOM, "U.S. Cyber Command History," 2019. [Online]. Available: https://www.cybercom.mil/About/History/

42. O. Pawlyk, "Calling up the Reserves: Cyber mission is recruiting," *Air Force Times*, 2015. [Online]. Available: https://www.airforcetimes.com/education-transition/jobs/2015/01/03/calling-up-the-reserves-cyber-mission-is-recruiting/

43. DHS, "Information Sharing Specifications for Cybersecurity — CISA," 2020. [Online]. Available: https://www.us-cert.gov/Information-Sharing-Specifications-Cybersecurity

44. MITRE, "Known Groups," 2019. [Online]. Available: https://attack.mitre.org/groups/

45. J. Leffall, "Are Patches Leading to Exploits? – Redmondmag.com," 2007. [Online]. Available: https://redmondmag.com/articles/2007/10/12/are-patches-leading-to-exploits.aspx

46. AOL, "Moloch Estimators," 2019. [Online]. Available: https://molo.ch/estimators

47. OISF, "Suricata High Performance Configuration," 2019. [Online]. Available: https://suricata.readthedocs.io/en/latest/performance/high-performance-config.html

48. G. Grudo, "USAFs Network Gateways Changing Hands, Eliminating Blind Spots - Air Force Magazine," 2017. [Online]. Available: https://www.airforcemag.com/usafs-network-gateways-changing-hands-eliminating-blind-spots/

49. ——, "Hackers Infiltrate DOD, Earn Priciest Government Bug Bounty Reward Ever - Air Force Magazine," 2017. [Online]. Available: https://www.airforcemag.com/hackers-infiltrate-dod-earn-priciest-government-bug-bounty-reward-ever/

50. T. Spring, "Critical Bug Opens Millions of HP OfficeJet Printers to Attack — Threatpost," 2018. [Online]. Available: https://threatpost.com/def-con-2018-critical-bug-opens-millions-of-hp-officejet-printers-to-attack/134972/

51. Microsoft, "Remove-EventLog," 2019. [Online]. Available: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/remove-eventlog?view=powershell-5.1

52. MITRE, "Lazarus Group, HIDDEN COBRA, Guardians of Peace, ZINC, NICKEL ACADEMY," 2019. [Online]. Available: https://attack.mitre.org/groups/G0032/

53. ——, "APT29: YTTRIUM, The Dukes, Cozy Bear, CozyDuke," 2019. [Online]. Available: https://attack.mitre.org/groups/G0016/

54. J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," Massachusetts Institute of Technology, Tech. Rep., 1988. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.606&rep=rep1&type=pdf

55. Novetta, "Operation Blockbuster: Tools Report," Operation Blockbuster Report, Tech. Rep., 2016. [Online]. Available: www.novetta.com

56. MITRE, "APT1: Comment Crew, Comment Group, Comment Panda," 2019. [Online]. Available: https://attack.mitre.org/groups/G0006/

57. ——, "APT3: Gothic Panda, Pirpi, UPS Team, Buckeye, Threat Group-0110, TG-0110," 2019. [Online]. Available: https://attack.mitre.org/groups/G0022/

58. ——, "BRONZE BUTLER, REDBALDKNIGHT," 2019. [Online]. Available: https://attack.mitre.org/groups/G0060/

59. ——, "APT32: SeaLotus, OceanLotus, APT-C-00," 2019. [Online]. Available: https://attack.mitre.org/groups/G0050/

60. Acalvio Threat Research Labs, "WannaCry Ransomware Analysis: Lateral Movement Propagation," 2017. [Online]. Available: https://www.acalvio.com/wannacry-ransomware-analysis-lateral-movement-propagation/

61. A. Singh, "Spreading techniques used by malware," 2016. [Online]. Available: https://www.virusbulletin.com/virusbulletin/2016/12/spreading-techniques-used-malware/

62. D. Berman, "The Complete Guide to the ELK Stack — Logz.io," 2019. [Online]. Available: https://logz.io/learn/complete-guide-elk-stack/#common-pitfalls

63. RockNSM Foundation, "Response Operation Collection Kit," 2019. [Online]. Available: https://rocknsm.io/

64. Air Forces Cyber, "Cyberspace Vulnerability Assessment/Hunter," 2018. [Online]. Available: https://www.afcyber.af.mil/About-Us/Fact-Sheets/Display/Article/1186672/cyberspace-vulnerability-assessment-hunter-weapon-system/

65. American Airlines, "Special items American Airlines," 2020. [Online]. Available: https://www.aa.com/i18n/travel-info/baggage/specialty-and-sports.jsp

66. RaspberryPiFoundation, "Raspberry Pi Foundation - About Us." [Online]. Available: https://www.raspberrypi.org/about/

67. Google, "Coral." [Online]. Available: https://coral.ai/

68. NVIDIA, "Jetson AGX Xavier Series Thermal Design Guide," NVIDIA, Tech. Rep., 2019.

69. J. Hughes, "Raspberry Pi Documentation," 2019. [Online]. Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/frequency-management.md

70. Kangalow, "NVPModel - NVIDIA Jetson AGX Xavier Developer Kit - JetsonHacks," 2018. [Online]. Available: https://www.jetsonhacks.com/2018/10/07/nvpmodel-nvidia-jetson-agx-xavier-developer-kit/

71. C. Rush, "How to save power on your Raspberry Pi Pi Supply Maker Zone." [Online]. Available: https://learn.pi-supply.com/make/how-to-save-power-on-your-raspberry-pi/#turn-off-hdmi

72. G. Khalil, "Open Source IDS High Performance Shootout," 2015. [Online]. Available: https://www.sans.org/reading-room/whitepapers/intrusion/open-source-ids-high-performance-shootout-35772

73. X. Bu, "Benchmarking Suricata in Different Isolation Systems Using TCPreplay," 2017. [Online]. Available: https://github.com/xybu/cs590-nfv/tree/master/experiments/suricata

74. D. J. Day, D. J. Day, and B. M. Burns, "A Performance Analysis of Snort and Suricata Network Intrusion Detection and Prevention Engines," in *ICDS 2011: The Fifth International Conference on Digital Society*, 2011. [Online]. Available: https://www.researchgate.net/publication/241701294

75. A. Kirk, "Comprehensive Threat Intelligence: Using Snort fast patterns wisely for fast rules," 2010. [Online]. Available: https://blog.talosintelligence.com/2010/04/using-snort-fast-patterns-wisely-for.html

76. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *2001 IEEE International Workshop on Workload Characterization,*

WWC 2001. Institute of Electrical and Electronics Engineers Inc., 2001, pp. 3–14.

77. C. P. Kruger and G. P. Hancke, "Benchmarking Internet of things devices," in *Proceedings - 2014 12th IEEE International Conference on Industrial Informatics, INDIN 2014*. Institute of Electrical and Electronics Engineers Inc., 11 2014, pp. 611–616.

78. Amido, "A Case Study of DevOps at Netflix ," 2018. [Online]. Available: https://amido.com/blog/a-case-study-of-devops-at-netflix/

79. Guru99, "Best 8 Ansible Alternatives in 2020," 2020. [Online]. Available: https://www.guru99.com/ansible-alternative.html

80. D. C. Montgomery, "The 2ˆk Factorial Design," in *Design and Analysis of Experiments*, 9th ed. Wiley, 2017, ch. 6.

81. AFLCMC/HNCDV, "CYBERSPACE VULNERABILITY ASSESSMENT/HUNTER (CVA/H) WEAPON SYSTEM (WS)," 2019. [Online]. Available: https://confluence.di2e.net/display/THISISCVAH/CVAH+Home

82. Smashicons, "Smashicons Icon Sets," 2019. [Online]. Available: https://www.flaticon.com/authors/smashicons

83. L. Rizzo, "netmap: a novel framework for fast packet I/O," Usenix ATC'12, Tech. Rep., 2012.

84. A. Turner, F. Klassen, and AppNeta, "Tcpreplay - Pcap editing and replaying utilities," 2013. [Online]. Available: https://tcpreplay.appneta.com/

85. I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization," *ICISSP*, 2018.

86. L. Rizzo, "netmap/README.md at master · luigirizzo/netmap," 2019. [Online]. Available: https://github.com/luigirizzo/netmap/blob/master/LINUX/README.md

87. F. Klassen and AppNeta, "tcprewrite," 2019. [Online]. Available: http://tcpreplay.appneta.com/wiki/tcprewrite.html#dealing-with-mtu-problems

88. R. Donato, "The Journey of a Frame through a Linux Based System," 2017. [Online]. Available: https://www.fir3net.com/UNIX/Linux/the-journey-of-a-frame-through-a-linux-based-system.html

89. packagecloud, "Monitoring and Tuning the Linux Networking Stack: Receiving Data - Packagecloud Blog," 2016. [Online]. Available: https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-

linux-networking-stack-receiving-data/#general-advice-on-monitoring-and-tuning-the-linux-networking-stack

90. G. Schudel, "Bandwidth, Packets Per Second, and Other Network Performance Metrics," 2018. [Online]. Available: https://www.cisco.com/c/en/us/about/security-center/network-performance-metrics.html

91. CAIDA, "The CAIDA UCSD Statistical information for the CAIDA Anonymized Internet Traces," 2019. [Online]. Available: https://www.caida.org/data/passive/passive_trace_statistics.xml

92. W. E. Leland, M. S. Taqqu, and D. V. Wilson, "On the Self-Similar Nature of Ethernet Traffic (Extended Version)," *IEEE/ACM Transactions on Networking*, vol. 2, no. 1, pp. 1–15, 1994.

93. M. E. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: Evidence and possible causes," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 835–846, 1997.

94. SUSE Support, "Improving network performance using Receive Packet Steering (RPS) — Support — SUSE," 2014. [Online]. Available: https://www.suse.com/support/kb/doc/?id=7015585

95. Mellanox Technologies, "Linux sysctl Tuning," 2018. [Online]. Available: https://community.mellanox.com/s/article/linux-sysctl-tuning

96. T. Herbet and W. de Bruijn, "Scaling in the Linux Networking Stack," 2010. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt

97. ARM, "ARM ® Generic Interrupt Controller Architecture version 2.0 Architecture Specification," ARM, Tech. Rep., 2008.

98. mingo, "Linux Kernel IO-APIC." [Online]. Available: https://www.kernel.org/doc/Documentation/x86/i386/IO-APIC.txt

99. J. Corbet, "Receive packet steering [LWN.net]," 2009. [Online]. Available: https://lwn.net/Articles/362339/

100. J. Bainbridge and J. Maxwell, "Red Hat Enterprise Linux Network Performance Tuning Guide," Red Hat Enterprise Linux, Tech. Rep., 2015. [Online]. Available: www.redhat.com

101. J. Real, "Measuring the impact of tcpdump on Very Busy Hosts," 2015. [Online]. Available: https://www.percona.com/blog/2015/04/10/measuring-impact-tcpdump-busy-hosts/

102. Samsung, "Introducing eMMC 5.1: The Next Step in Relentless Flash Innovation," 2015. [Online]. Available: https://www.samsung.com/semiconductor/newsroom/tech-trends/introducing-emmc-5-1-the-next-step-in-relentless-flash-innovation/

103. SD Association, "Speed Classes," 2020. [Online]. Available: https://www.sdcard.org/developers/overview/speed_class/index.html

104. L. Schaelicke, T. Slabach, B. Moore, and C. Freeland, "Characterizing the Performance of Network Intrusion Detection Sensors," in *RAID 2003*, 2003. [Online]. Available: http://www.cs.utah.edu/~lambert/pdf/nids_raid03.pdf

105. NetSec Support Notes, "Bad Snort Rules — NetSec Support Notes and Ramblings," 2014. [Online]. Available: https://netsecsupport.wordpress.com/2014/06/22/bad-snort-rules/

106. Emerging Threats, "Suricata 5 ET Rules," 2020. [Online]. Available: https://rules.emergingthreats.net/open/suricata-5.0/

107. X. Bu, "Performance Characterization of Suricata's Thread Models," 2017. [Online]. Available: https://xbu.me/article/performance-characterization-of-suricata-thread-models/

108. OISF, "Advanced Suricata Engine Documentation," 2020. [Online]. Available: https://suricata.readthedocs.io/en/latest/configuration/suricata-yaml.html#detection-engine

109. Martin, "Open-Source Security Tools: Network Intrusion Detection Systems," 2011. [Online]. Available: http://ossectools.blogspot.com/2011/04/network-intrusion-detection-systems.html

110. Elasticsearch, "Filebeat Reference [7.5]," 2020. [Online]. Available: https://www.elastic.co/guide/en/beats/filebeat/current/how-filebeat-works.html

111. L. H. Newman, "A Controversial Plan to Encrypt More of the Internet — WIRED," 2019. [Online]. Available: https://www.wired.com/story/dns-over-https-encrypted-web/

112. M. Pilkington, "Kerberos in the Crosshairs: Golden Tickets, Silver Tickets, MITM, and More," 2014. [Online]. Available: https://digital-forensics.sans.org/blog/2014/11/24/kerberos-in-the-crosshairs-golden-tickets-silver-tickets-mitm-more

113. A. Duckwall and C. Campbell, "Still Passing the Hash 15 Years Later: Mimikatz and Golden Tickets... What's the BFD?" 2014. [Online]. Available: http://passing-the-hash.blogspot.com/2014/08/mimikatz-and-golden-tickets-whats-bfd.html

114. E. Nabigaev, "PROTECTING WINDOWS NETWORKS KERBEROS ATTACKS," 2015. [Online]. Available: http://web.archive.org/web/20160216185049/http://dfir-blog.com/2015/12/13/protecting-windows-networks-kerberos-attacks/

115. MITRE, "Kerberoasting," 2019. [Online]. Available: https://attack.mitre.org/techniques/T1208/

116. ——, "APT28: SNAKEMACKEREL, Swallowtail, Group 74, Sednit, Sofacy, Pawn Storm, Fancy Bear, STRONTIUM, Tsar Team, Threat Group-4127, TG-4127," 2019. [Online]. Available: https://attack.mitre.org/groups/G0007/

117. ——, "Indicator Removal on Host- MITRE ATT&CK," 2020. [Online]. Available: https://attack.mitre.org/techniques/T1070/

Acronyms

AFIT Air Force Institute of Technology. 9

ANOVA Analysis of Variance. 4, 30, 31, 34, 39, 43, 71, 72, 73, 110

APT Advanced Persistent Threat. 9, 12, 13, 17, 55, 123

CAIDA The Center for Applied Internet Data Analysis. 61

CCRI Command Cyber Readiness Inspection. 10

CIC Canadian Institute for Cybersecurity. 58, 63, 70

CPT Cyber Protection Team. 13

CPU Central Processing Unit. 25, 28, 39, 51, 58, 60, 73

CVA/H Cyberspace Vulnerability Assessment / Hunter. 20, 54, 98, 118

DC Domain Controller. 120, 121, 122

DIACAP DoD Information Assurance Certification and Accreditation Process. 10

DISA Defense Information Systems Agency. 10

DMA Direct Memory Access. 60

DNS Domain Name Service. 38

DOD Department of Defense. 9

DODI DoD Instruction. 10

GPU Graphics Processing Unit. 48

GRO Generic Receive Offload. 58, 74

GSO Generic Segmentation Offload. 58, 64

GUI Graphical User Interface. 25

HBSS Host Based Security System. 16

I/O Input / Output. 22, 56

IDS Intrusion Detection System. 20, 26, 54, 57, 98, 113, 119

IOC Indicator of Compromise. 13, 14

IoT Internet Of Things. iv, 1

IPI Inter-Processor Interrupt. 60, 75

IRQ Interrupt Request. 60

JRSS Joint Regional Security Stack. 15

KDC Key Distribution Center. 120, 121

LRO Large Receive Offload. 58, 74

MITM Main-in-the-Middle. 120

MTU Maximum Transmission Unit. 58, 59

NAPI New API. 59, 60, 78, 80

NIC Network Interface Card. 25, 54, 60, 73, 76, 81

NIDS Network Intrusion Detection System. 20

NIPRNET Non-classified Internet Protocol Router Network. 9

NIST National Institute of Standards and Technology. 10

NSA National Security Agency. 10

OS Operating System. 15, 29

PID Process ID. 49

PPS Packets Per Second. 61, 64, 69, 73, 81, 101, 117

RAM Random Access Memory. 28

RFS Receive Flow Steering. 60, 79, 80, 81, 82, 90, 91, 100

RockNSM Response Operations Collection Kit Network Security Monitor. 19

RPS Receive Packet Steering. 60, 74, 75, 79, 80

RV Response Variable. 112

SBC Single Board Computer. 22, 23

SDK Software Development Kit. 25

SIEM Security Information and Event Management. 20

SIPRNET Secret Internet Protocol Router Network. 9

skb Socket Buffer. 75, 78

SOC System on a Chip. 55

SSH Secure Shell. 35, 41

STIG Security Technical Implementation Guide. 10

STIX Structured Threat Information eXpression. 14

TAXII Trusted Automated eXchange of Indicator Information. 14

TCP Transmission Control Protocol. 25, 114

TGS Ticket Granting Service. 120, 122

TGT Ticket to Grant Tickets. 120, 121

TSO TCP Segmentation Offload. 58, 64

USCYBERCOM United States Cyber Command. 13

YAML Yet Another Markup Language. 36, 43, 66

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
26–03–2020	Master's Thesis	Sept 2018 — Mar 2020

4. TITLE AND SUBTITLE

A General Methodology to Optimize and Benchmark Edge Devices

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Smathers, Kyle J., Capt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-20-M-062

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

A macro level methodology is proposed that iteratively benchmarks and optimizes specific workloads on edge devices. With automation provided by Ansible, a multi stage 2^k full factorial experiment and robust analysis process ensures the test workload is maximizing the use of available resources before establishing a final benchmark score. By framing the validation tests with a family of network security monitoring applications an end to end scenario fully exercises and validates the developed process. This also provides an additional vector for future research in the realm of network security. The analysis of the results show the developed process met it's original design goals and intentions, with the added fact that the latest edge devices like the XAVIER, TX2 and RPi4 can easily perform as a edge network sensor.

15. SUBJECT TERMS

edge device, benchmarking, optimization, network monitoring

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Lt Col Mark DeYoung, AFIT/ENG
U	U	U	UU	193	19b. TELEPHONE NUMBER *(include area code)* (937)-255-6565 x7216 mark.deyoung@afit.edu