3-26-2020

Honeyhive - A Network Intrusion Detection System Framework Utilizing Distributed Internet of Things Honeypot Sensors

Zachary D. Madison

Recommended Citation

Madison, Zachary D., "Honeyhive - A Network Intrusion Detection System Framework Utilizing Distributed Internet of Things Honeypot Sensors" (2020). *Theses and Dissertations*. 3179.
https://scholar.afit.edu/etd/3179

HONEYHIVE – A NETWORK INTRUSION DETECTION SYSTEM FRAMEWORK UTILIZING DISTRIBUTED INTERNET OF THINGS HONEYPOT SENSORS

THESIS

Zachary D. Madison, Capt, USAF

AFIT-ENG-MS-20-M-038

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-20-M-038

HONEYHIVE – A NETWORK INTRUSION DETECTION SYSTEM
FRAMEWORK UTILIZING DISTRIBUTED INTERNET OF THINGS
HONEYPOT SENSORS

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 Computer Science

Zachary D. Madison, B.S.C.S.

Capt, USAF

March 2020

AFIT-ENG-MS-20-M-038

HONEYHIVE – A NETWORK INTRUSION DETECTION SYSTEM
FRAMEWORK UTILIZING DISTRIBUTED INTERNET OF THINGS
HONEYPOT SENSORS

THESIS

Zachary D. Madison, B.S.C.S.
Capt, USAF

Committee Membership:

Barry E. Mullins, Ph.D., P.E.
(Chairman)

Scott R. Graham, Ph.D.
(Member)

Stephen J. Dunlap, M.S., CISSP
(Member)

AFIT-ENG-MS-20-M-038

Abstract

With the ever increasing number of Internet-connected devices, the importance of cyber security also increases. Exploding over the past decade, the number of Internet of Things (IoT) devices connected to the Internet jumped from 3.8 billion in 2015 to 17.8 billion in 2018. A major concern with many IoT devices is that they contain vulnerabilities that are often left unpatched. To make matters worse, many of these IoT devices lack modern security measures found on traditional computing devices, due to their inherent hardware limitations and vendors focusing on functionality and time to market over security.

Honeypots are devices not part of routine network usage that are meant to alert of an attacker's presence, capture an attacker's tools, and record their Tactics, Techniques, and Procedures (TTPs). Honeyd is a framework capable of rapidly creating low-interaction honeypots by simulating the network stack.

HoneyHive is a framework that uses distributed IoT honeypots as Network Intrusion Detection System (NIDS) sensors that beacon back to a centralized Command and Control (C2) server. This research uses the three Honeyd IoT honeypots developed by Stafira, but HoneyHive is flexible enough to support any device capable of running the Python HoneyB Agent script. HoneyHive offers a method for network intrusion detection using the lure of vulnerable IoT devices as distributed honeypot intrusion detection sensors.

The HoneyHive framework consists of the C2 server, transfer server, Snort log parser, Database (DB), and the HoneyB Agent script. The C2 server interacts with all other components and displays honeypot interactions and Snort alerts to network operators in real time. The transfer server receives PCAPs and other Binary Large

iv

Object (BLOB) from the HoneyB Agent. Upon receiving a PCAP, it is passed to Snort and analyzed for matching signatures. Immediately after this, the Snort log parser reads the log file and then sends any alert information back to the C2 server. The DB stores alerts, alert metadata, and PCAPs received from the HoneyB Agent. The HoneyB Agent script captures and reports any traffic interactions with Stafira's honeypots. Captured traffic is sent to the transfer server and honeypot interactions are sent to the C2 server.

The tests in this experiment involve four types of scans and four levels of active honeypots against the HoneyHive framework and a traditional NIDS on the simulated test network. The scan types include No Scan (Control Group), TCP Connect scan, Aggressive scan, and NIDS Avoidance scan. The levels for honeypots are 0, 3, 6, and 9 honeypots. Each of these was run in different combinations with one another for a full factorial experiment resulting in 16 different combinations.

This research successfully created a framework of distributed network intrusion detection IoT honeypot sensors that capture traffic, create alerts, and beacon back to a central C2 server. The HoneyHive framework operated correctly by not alerting on routine network traffic and alerting on non-routine network traffic. Additionally, the HoneyHive framework successfully detected intrusions that traditional NIDSs cannot through the use of distributed IoT honeypot sensors and packet capture aggregation.

To my Wife,

Thank you for all your love, support, and hand-crafted late night lattes. I am so proud of how diligent you work and all your accomplishments. I never want my career or goals to take priority over yours, "Love is taking turns riding shotgun."

To my soon to be born baby girl,

You motivated your Daddy to work extra hard and finish early so that we could have your nursery all setup. I pray that you love the Lord, dream big, and never settle for what's easiest, but fervently pursue justice.

To my Mom and Dad,

Thank you for all your love and encouragement through every step of my life. Thank you for instilling in me a work-ethic and a perseverance to overcome any adversity. I love you both to the moon and back!

PLUS ULTRA!

Acknowledgements

"We all want progress. But progress means getting nearer to the place you want to be and if you have taken a wrong turning, then to go forward does not get you any nearer. If you are on the wrong road, progress means doing an about-turn and walking back to the right road; and in that case, the man who turns back soonest is the most progressive man." - C.S. Lewis

To my advisor, Dr. Barry Mullins, thank you for your shared excitement, expertise, and keeping me from going down too many rabbit holes during my research.

Stephen, even though I grumbled at every design change to the HoneyHive framework and my experiment methodology, thank you. My thesis would not be near the product it is now without all your insight and wonderful suggestions.

<div align="center">Zachary D. Madison</div>

Table of Contents

List of Figures

List of Tables

List of Acronyms

AFNet   Air Force Network

BLOB   Binary Large Object

BPF   Berkeley Packet Filter

C2   Command and Control

CA   Certificate Authority

CEO   Chief Executive Officer

CIKR   Critical Infrastructure and Key Resources

CLI   Command Line Interface

CPU   Central Processing Unit

CSS   Cascading Style Sheets

CSV   Comma-Separated Values

CUT   Component Under Test

CWMP   CPE WAN Management Protocol

DB   Database

DDOS   Distributed Denial of Service

DHS   Department of Homeland Security

DMZ   Demilitarized Zone

DoD   Department of Defense

DODIN   DoD Information Network

DOM   Document Object Model

DOS   Denial of Service

GPS   Global Positioning System

GUI   Graphical User Interface

HTML   HyperText Markup Language

HTTP	HyperText Transfer Protocol
ICS	Industrial Control System
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
JSON	JavaScript Object Notation
LCS	Longest Common Substring
MAC	Media Access Control
MQTT	Message Queue Telemetry Transport
MTU	Maximum Transmission Unit
NIDS	Network Intrusion Detection System
NPM	Node.js Package Manager
OS	Operating System
RFID	Radio-Frequency Identification
SCADA	Supervisory Control and Data Acquisition
SMTP	Simple Mail Transfer Protocol
SPAN	Switch Port Analyzer
SSH	Secure Shell
SUT	System Under Test
TCP	Transmission Control Protocol
TTL	Time to Live
TTPs	Tactics, Techniques, and Procedures
UML	Unified Modeling Language
UWB	Ultra-Wideband
VM	Virtual Machine

VNC   Virtual Network Computing

VPN   Virtual Private Network

Wi-Fi   Wireless Fidelity

WLAN   Wireless LAN

WPAN   Wireless Personal Network

XMPP   Extensible Messaging and Presence Protocol

HONEYHIVE – A NETWORK INTRUSION DETECTION SYSTEM
FRAMEWORK UTILIZING DISTRIBUTED INTERNET OF THINGS
HONEYPOT SENSORS

I. Introduction

1.1   Background

With the ever increasing number of Internet-connected devices, the importance
of cyber security similarly increases. Exploding over the past decade, the number of
Internet of Things (IoT) devices connected to the Internet jumped from 3.8 billion
in 2015 to 17.8 billion in 2018 [1]. A major concern with many IoT devices is that
they contain vulnerabilities that are often left unpatched [2][3]. To make matters
worse, many of these IoT devices lack modern security measures found on traditional
computing devices, due their inherent hardware limitations and vendors focusing on
functionality and time to market over security [2]. While an insecure IoT device
connected to a consumer's probably-already insecure home network does not cause
much worry, insecure IoT devices connected to previously-secure networks do. An
attacker now has a vector into a previously locked down network and can use the
device as a pivot to gain access into the internal network [4]. If these devices were
to become connected to Critical Infrastructure and Key Resources (CIKR) networks,
the results could be catastrophic.

Honeypots are devices not part of routine network usage that are meant to alert of
an attacker's presence, capture tools, and record Tactics, Techniques, and Procedures
(TTPs) [5]. Honeypots come in varying levels of sophistication and are available in

multitudes of frameworks. Honeyd is one such framework that is capable of rapidly creating low-interaction honeypots by simulating the network stack. Lukas Stafira used Honeyd to develop three convincing web-based IoT honeypots which are used in this research. Stafira created IoT honeypots for the TITAThink camera, Proliphix thermostat, and an ezOutlet2 power outlet [6].

Network Intrusion Detection Systems (NIDSs) are devices that analyze network traffic and create alerts if they see traffic of malicious nature and or anomalous traffic. Intrusion detection is split into two categories, signature matching and anomaly detection. Signature matching uses known patterns of malicious traffic and creates alerts upon seeing the pattern. Anomaly detection on the other hand uses baselining and heuristics to create alerts when network traffic deviates from the network baseline. Signature matching is faster and easier to setup than anomaly detection but only alerts on traffic matching installed signatures. This means that signatures must be kept up to date and any previously unknown exploit (zero-day) will not generate an alert. Anomaly detection can detect zero-day exploits but requires much more setup and can create many false positives, or false negatives, if the heuristics are not fine-tuned. Modern NIDS often are a hybrid of the two detection techniques.

1.2   Motivation

Even networks with security measures in place are not immune to compromise; an example is the cyber attack against Ukarine in 2016 where attackers successfully gained internal network access through a phishing campaign. After initial access, attackers then conducted internal network scans and credential harvesting over a period of several months. Using gathered credentials, attackers gained access to Supervisory Control and Data Acquisition (SCADA) networks and took approximately 30 power stations offline, sending the country and more than 230,000 residents into darkness for

several hours [7]. If the network had contained convincing SCADA honeypots then it is possible that network administrators would have detected the attackers' presence and been able to respond in time before the real SCADA systems were taken offline. While honeypots do not guarantee network security nor are they the solution to securing every network, their use is another viable tool for network defense.

Due to IoT devices' lack of sophisticated hardware and vendor support for security updates, other methods must be implemented to secure the network. Because so many IoT devices remain unpatched, unmonitored, and left on, they have become a tantalizing target for attackers to gain network access or add another device to their botnet [8]. Due to IoT device popularity with attackers HoneyHive was developed. HoneyHive is a framework that uses distributed IoT honeypots as NIDSs sensors that beacon back to a centralized Command and Control (C2) server. This research uses the IoT honeypots developed by Stafira, but HoneyHive is flexible enough to support any device capable of running the Python 2.7 HoneyB Agent script. Providing security for all IoT devices with their heterogeneous nature is a monumental task. HoneyHive instead offers another method for network intrusion detection using the lure of vulnerable IoT devices as distributed honeypot intrusion detection sensors.

Because traditional NIDSs typically only monitor Switch Port Analyzer (SPAN) traffic from the switch they are located on, they can miss attacks located on other parts of the network. Typical placement of a NIDS is just inside a network's external firewall in the Demilitarized Zone (DMZ) [9]. If an attacker manages to to infiltrate an internal network without tripping the NIDS, then internal attacks and or scans can be performed without raising an alert, as was the case with the Ukraine power network. The HoneyHive framework addresses this shortcoming of the traditional NIDS construct by using distributed IoT Honeypot NIDS sensors.

1.3 Research Goals

The goal of this research is to first develop the HoneyHive framework and then test its effectiveness in network intrusion detection compared to that of a traditional NIDS.

The hypotheses for this research are:

1. The HoneyHive framework operates correctly by not alerting on routine network traffic and alerting on non-routine network traffic.

2. The HoneyHive framework detects intrusions that traditional NIDSs cannot through the use of distributed IoT honeypot sensors and packet capture aggregation.

1.4 Approach

In order to determine HoneyHive's effectiveness at network intrusion detection, several steps must be taken. These include the development of the HoneyHive framework, setting up the simulated network for experimentation, and then designing and running the experiment.

1.4.1 HoneyHive Framework

To develop the HoneyHive framework, Honeyd and Stafira's IoT honeypots are first setup. Then the individual components of the framework are developed, including the C2 server, transfer server, Snort log parser, Database (DB), and HoneyB Agent script. Snort is also integrated into the framework for increased signature matching. The HoneyHive framework is explained in more depth in Chapter 3.

1.4.2 Simulated Network

A simulated network also needs to be setup in order to run the experiment. The network is composed of IoT devices, Stafira's honeypots (duplicated several times), Windows 10 devices running Ubuntu Virtual Machines (VMs), Ubuntu VMs running the Honeyd honeypots and HoneyB Agent script, the HoneyHive C2 server, Suricata, an Ubuntu attacker machine, and networking devices. The network layout is described in Chapter 3, and the actual devices used on it and in the experiment are described in Chapter 4.

1.4.3 Experiment

After development, HoneyHive's effectiveness at network intrusion detection is tested in a simulation where an attacker has gained access to the internal network, has narrowed down their list of targets through previous reconnaissance, and now is performing internal nmap network scans against the specific Internet Protocol (IP) addresses before launching exploits against them. The attacker launching exploits on scanned devices is not tested in this experiment. The exploitation and propagation stage would hopefully be prevented by network administrators through the use of alerts from the HoneyHive framework. The tests in this experiment involve four types of scans and four levels of active honeypots. The scan types include No Scan (Control Group), TCP Connect scan, Aggressive scan, and NIDS Avoidance scan. The levels for honeypots are 0, 3, 6, and 9 honeypots. Each of these are run in different combinations with one another for a full factorial experiment resulting in 16 different combinations. Each test is performed 30 times for a total of 480 runs. Because of the timing and coordination required to run the experiment, gather results, and reset devices to their initial state after each run, the runExperiment.py script automates this process. This script is discussed further in Chapter 4 and is found in Appendix

D.

1.5 Assumptions and Limitations

1.5.1 Assumptions

The following assumptions are made in this research:

1. Routine network traffic on the simulated network does not contain any traffic a NIDS would treat as malicious.

2. Given the same set rules, NIDS create the same number of distinct alerts and the same number of total alerts when analyzing an identical sample of network traffic.

1.5.2 Limitations

1.5.2.1 HoneyHive

Several limitations currently exist in the HoneyHive framework. The HoneyB Agent script is written in Python 2.7 which is near the end of its life. Additionally, HoneyHive relies on a NIDS (Snort) to perform signature matching instead of being self-contained and possessing its own sophisticated intrusion detection system.

1.5.2.2 Honeypots

While the honeypots in this experiment are useful for testing hypotheses, implementing modern and more sophisticated honeypots would improve the HoneyHive framework. Honeyd is outdated and no longer regularly maintained [10][11]. Also, Stafira's honeypots are low-interaction and are only convincing with web traffic.

1.6 Research Contributions

The HoneyHive framework offers increased network intrusion detection to all networks its deployed to. It can be used for integration in CIKR-based networks since IoT devices share some similarities with Industrial Control System (ICS). In addition, government organizations or commercial companies that work in cyber security could integrate HoneyHive into their existing network security architecture. The impact of this framework is a cross-platform, standalone, NIDS / Network Monitoring solution capable of improving the rate at which network intrusions are detected. While HoneyHive may not be the solution for every network, it is a viable tool for increasing network security through intrusion detection.

1.7 Thesis Overview

Chapter 2 provides background information and related research on the state of IoT devices, IoT and Computer Network Security, NIDS and Network Monitoring, and honeypots and honeytokens. It also provides details about software and programming languages used in this thesis. Chapter 3 describes the HoneyHive framework design and components in depth and explains the rationale behind design decisions. Chapter 4 describes the methodology for running the experiment and the research questions posed for this thesis. The methodology includes all parameters, factors, metrics, and a step-by-step procedure to replicate the experiment. Chapter 5 presents the experiment results and provides analysis. Finally, Chapter 6 provides a summary and conclusion for this thesis as well as future work to improve the HoneyHive framework, and hopefully, IoT and Computer Network Security.

II. Background and Related Research

2.1   Overview

This chapter provides background information on IoT, IoT and Computer Network Security, NIDS, and Network Monitoring, honeypots and honeytokens in Section 2.2. It also covers Honeyd 1.5c, Cyber Deception, and programming languages and tools used in this research. Section 2.3 explores related research and emerging technologies in the field of IoT and honeypots.

2.2   Background

2.2.1   Internet of Things (IoT)

The term IoT covers a myriad of devices and appliances with capabilities to sense the world around them, process information, and share this information with other devices on an internal network or the Internet at large [12]. Simple sensors and appliances now have the computing power for making intelligent decisions, as well as communication abilities for sharing perceived data and being remotely interacted with [13]. Suo et al. break IoT device functionality into four layers: the application layer, support layer, network layer, and perceptual layer. The application layer is the actual service displayed to the user such as a web page, application, or screen. The support layer acts as the intermediary between the application and network layers and involves cloud computing to bring increased performance. The network layer deals with transmitting data between devices through numerous different communication protocols. Finally, the perceptual layer is responsible for collecting data in the physical world and converting it to digital data through the use of sensors, cameras, Radio-Frequency Identification (RFID), Global Positioning System (GPS), transducers, thermostats, etc. [12].

The majority of IoT devices communicate on a Wireless Personal Network (WPAN) with a 10 meter range using one of several different Institute of Electrical and Electronics Engineers (IEEE) protocols. These protocols mainly include Bluetooth (IEEE 802.15.1), Ultra-Wideband (UWB) (IEEE 802.15.3), and Zigbee (IEEE 802.15.4) [13][14][15][16][17]. Some devices utilize Wireless Fidelity (Wi-Fi) (IEEE 802.11) instead for communication over a Wireless LAN (WLAN) with a range up to 100 meters. However, the devices used in this research, and that would be found as honeypots, communicate over Ethernet (IEEE 802.3), which is the focus of this research.

The added functionality of smart devices makes them very appealing and has caused the IoT market to explode over the past decade. As shown in Figure 1, the total number of devices connected to the Internet in 2018 was 17.8 billion, 7 billion of which were IoT devices. By 2025, the total number of devices connected to the Internet is expected to grow to 34.2 billion with IoT devices comprising 21.5 billion of the total devices. The IoT market is anticipated to grow to reach $1.6 trillion by 2025, making it a lucrative and competitive market [1].

With such a competitive market, vendors are scrambling to be the first to release the latest and greatest product, often cutting corners in areas like security to reduce cost and production time. IoT devices are often slapped together with inexpensive, outdated, and insecure third party components that are no longer supported or patchable [2]. Insecure IoT devices are an alarming problem in network security for consumers, corporations, the Department of Homeland Security (DHS), and the Department of Defense (DoD).

2.2.2 IoT and Computer Network Security

IoT creates new possibilities for technologies never before imagined but also opens up new vulnerabilities and attack vectors for malicious hackers [4]. These new attack
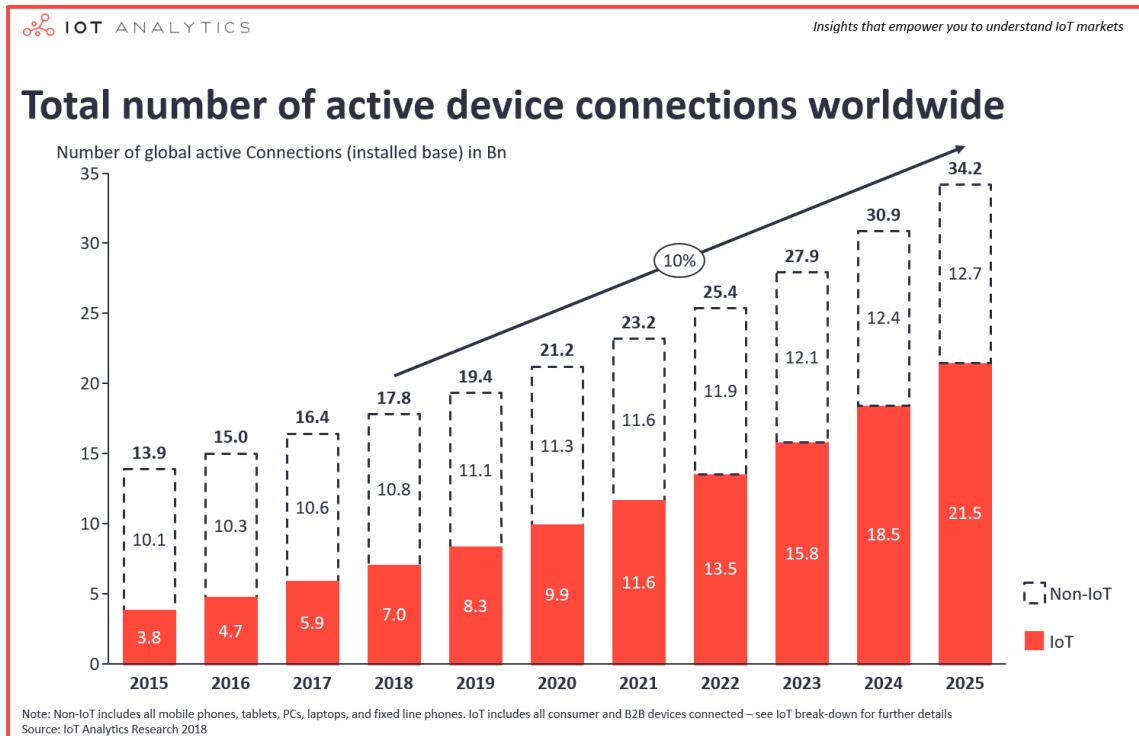
Figure 1. Growth of IoT Devices from 2015-2025 [1]

vectors arise from a lack of security in devices. Unfortunately, some vendors in this market are not primarily concerned with the security of their devices. Their main focus is to rapidly develop innovative and easy to use technology before competitors and turn a profit. This mindset often leaves many IoT devices ripe for exploitation.

Many IoT devices are riddled with vulnerabilities due to vendors focusing on cheap solutions and rapid development in a competitive market. HP performed a study and found that 70 percent of IoT devices contain vulnerabilities. When researching 10 of the most popular IoT devices, they found an average of 25 security vulnerabilities per device, with over 250 vulnerabilities in total [3]. Outdated software is loaded on devices that are then often never updated or very cumbersome to do so for the average user. This results in millions of IoT devices with known unpatched vulnerabilities connected to the Internet, just waiting for attackers to exploit them. Attackers can quickly discover devices with known vulnerabilities using websites such as Shodan

[4][18][19][20].

While being able to lock a house, switch on or off lights, or adjust a thermostat remotely can be desirable, the inclusion of these IoT devices opens up significant vulnerabilities in networks. A once-secure network can now be accessed by exploiting a vulnerable IoT device and using it as a pivot into the otherwise unreachable network. Because these devices are not intended to be accessed by just anyone, unlike web servers, they are not placed inside DMZs, but are instead placed deeper within the network. Attackers are still able to reach these vulnerable IoT devices if they first compromise a DMZ or a different internal device and utilize that device as a pivot to the IoT device. Traffic is already allowed to DMZ devices, but a misconfigured router or router with port forwarding can allow internal devices to be compromised.

The threat of IoT devices being hacked is not just theoretical; in 2014, smart meters were hacked allowing attackers to spoof messages between nodes. With spoofed messages attackers could avoid paying their monthly utility bill or shut down energy from the utility company altogether, without the use of any kinetic effects. The attacker's ability to shut down energy demonstrates the threat of cyber to CIKR networks; they are susceptible and can be disabled as well [21].

Many IoT devices have been found that allow logins with empty, default, or weak passwords. IoT devices are becoming increasingly common targets for use in botnets. While IoT devices may have limited computational power, their sheer number and ease of exploitation have made them an enticing target for attackers. In addition, these devices are not often updated and have limited user interaction allowing attackers to go unnoticed on a network for a prolonged period of time [22]. Take the Mirai (Japanese for "the future") worm for example, hundreds of thousands of IoT devices have been compromised and assimilated as part of botnets since its release in 2016. Mutations of the Mirai worm are even prevalent today because of the lack of

security implemented in IoT devices [8]. These botnets, consisting of up to 400,000 devices, are available for purchase and have been used to execute a Distributed Denial of Service (DDOS) attack on a number of web servers successfully [23].

Because IoT devices lack sophisticated hardware and are so diverse, traditional methods for securing them like installing antivirus software or automatic updates are not possible typically. Kolias et al. argue that the vendor is responsible for implementing automatic updates and better security in device [8]. IoT devices rarely receive updates to fix vulnerabilities, and on the off chance they do, there is even a smaller percentage of users that take the time to manually install the updates [2]. The average user plugs the IoT device into their network without changing default passwords, and never manually checks or installs updates [20].

2.2.3 Network Intrusion Detection System (NIDS)

One common device to increase computer network security is a NIDS. A NIDS can be deployed on networks to detect malicious traffic and intrusions. They can be placed either inline, which can affect network latency as all traffic now passes through the NIDS and is then forwarded to its destination, or mirrored where copies of all traffic are sent to the NIDS as well as the original destination. NIDS can use multiple techniques for intrusion detection which include signature / pattern matching, and or baselining / anomaly detection. Signature-based detection searches network traffic for patterns defined in rule-sets and creates an alert if there is a match. Baselining involves taking a snapshot of normal traffic on a network, and then using heuristics; any behavior that is abnormal (an anomaly), generates an alert. Modern NIDS employ a combination of the techniques as they both have advantages and disadvantages. Signature-based detection is great at alerting on known exploits, but is unable to alert on zero-day exploits, which results in false negatives. Baselining on the

other hand requires creating a network traffic standard that if deviated from causes an alert. If the network changes or routine traffic changes, then a new baseline has to be performed. Anomaly detection can create numerous false alerts (false positives) and require significant setup time for learning the network baseline. However, anomaly detection is capable of detecting previously unknown vulnerabilities.

This research proposes the use of honeypots as NIDS. Using honeypots as a NIDS is not a novel idea. Spitzner argued that they make more effective NIDS than traditional ones since they reduce false positives because any traffic sent to them is suspicious [24].

2.2.4   Networking Monitoring

Network monitoring software works closely with intrusion detection sensors but is focused on overall network traffic patterns and determining whether or not devices are reachable. While NIDS inspect the content of traffic, network monitors record the volume and types of traffic on the network. Network monitoring quickly helps network operators identify overloaded network links and devices. Network monitoring is useful for bringing devices back online and can detect spikes in traffic, indicative of a Denial of Service (DOS) attack. Furthermore, if a device does go down, it can signal that an attacker launched an exploit that resulted in a crashed service. NIDS and network monitoring used together provide a better picture of network health while still closely investigating traffic for malintent. The HoneyHive framework developed in this research provides both network monitoring and network intrusion detection for networks it is deployed to.

2.2.5 Honeypots

One way to mitigate or detect exploited devices is the use of honeypots. Honeypots are used to increase the security of computer networks by emulating real devices attackers might be interested in compromising. A honeypot being interacted with can be one of the first signs of compromise in a network or of an impending attack, and can therefore act as a NIDS. By using honeypots, previously unknown vulnerabilities (zero-day vulnerabilities) may be discovered when an adversary targets and gains access to the device. In addition, honeypots leave known vulnerabilities unpatched so that TTPs of an adversary can be learned and or later used to fingerprint an actor that employs the specific TTPs.

Honeypots come in all kinds of shapes, sizes, and implementations. They range from simple scripts, virtual devices, to physical devices and support low to high-interaction.

Low-Interaction Honeypots Low-Interaction honeypots can simulate common network services and the network stack. However, upon receiving a known exploit, the attacker does not receive full control of the device because the command terminal spawned is simulated. This also means that zero-day exploits are not captured since they are outside what the honeypot knows how to react to. Because the attacker cannot gain full control of the honeypot, this does make them safer for deployment in a network, but at the expense of being easier to detect as a honeypot by attackers.

High-Interaction Honeypots High-Interaction honeypots in contrast do not emulate network services or the network stack but do allow the attacker to gain full control of the device. This not only allows for a more believable honeypot, but also supports gathering more information about the attack such as zero-day exploits,

tools and TTPs used by the hacker. Although there are many benefits to high-interaction honeypots, they also have disadvantages. While the honeypot looks more convincing to the attacker because of the full control allowed, it now presents an increased security risk to the network. In addition, high-interaction honeypots are costly to develop in both time and resources; they require more maintenance and oversight than low-interaction honeypots [24][25][26][27][28].

2.2.6 Honeytokens

Cymmetria breaks honeytokens into several sub categories: breadcrumbs, "beacons", and tokens [29]. Breadcrumbs are data left intentionally for a hacker to find and use to allow them to move throughout the network. However, by using breadcrumbs an attacker only moves through a controlled path of devices monitored by network defenders. All the while, an attacker's TTPs, which include commands, tools, and exploits are being recorded. Cymmetria's "beacons" create alerts whenever they are interacted with. They are not part of routine usage (like honeypots) by the organization so any interaction is considered malicious and can even help to identify insider threats. Examples of "beacons" include decoy shares, documents with embedded macros, and websites that all beacon back to a C2 server when touched. The "beacons" defined by Cymmetria are essentially various types of intrusion detection sensors. Cymmetria's last category, tokens, are Honeydocs (fake documents) that act as a beacon to alert that a file was exfiltrated out of the network. The main difference between beacons and tokens, as defined by Cymmetria, is that "beacons" reside on the organization's internal network and tokens are meant to detect data leaving the network [30].

2.2.7   Honeyd 1.5c

One common framework to create virtual honeypots is Honeyd. The version of Honeyd used and described is 1.5c and was last updated in May of 2007. Honeyd 1.6d is available on Github with a last commit of December 2013, but as noted by Stafira, contains program stability issues [6].

Honeyd simulates the network stack to allow one physical device to act as numerous honeypots. All traffic for the honeypots is sent to Honeyd which makes it look like the devices are running independently on separate IPs. Honeypots can also be customized by using Nmap DB files to deceive scanning and fingerprinting software. The Nmap DB file defines how different Operating Systems (OSs) and their respective versions respond to messages as well as ports and services that are running by default [25][31]. While not identical, the Nmap DB file can be used to closely match network fingerprints of IoT devices. One deficiency Stafira noted about Honeyd was the outdated Nmap DB files [6].

Within the Honeyd configuration file, low-interaction honeypots can be quickly created. Each honeypot can be assigned a personality defined by the Nmap DB file, customize ports to open, filter, and close, and run custom shell scripts on open ports. Running customized scripts is one of the selling features of Honeyd; with sophisticated enough scripts, entire services can be mimicked. In theory, creating scripts to match every service would yield a convincing and very interactive honeypot, but the operating system itself, as well as Inter-Process Communication (IPC) would be painstakingly time consuming and better alternatives such as VMs exist for high-interaction honeypots. Honeyd is designed and better suited for quickly creating numerous honeypots and simulating a handful of services to provide a low-interaction honeypot framework.

The IP and Media Access Control (MAC) addresses of the honeypot are also

configurable, but the IP must be on the same network as Honeyd. MAC addresses can be used to identify the type of device and manufacturer, so allowing customization leads to more convincing honeypots. However, with these customization options, it is imperative that both the IP and MAC addresses be unique on a network in order to prevent collisions [31].

The Honeyd documentation states that with the flag "l" it logs packets and connections to a specified file. However, this logging option contains only time stamps, IP addresses, ports, protocols, and transmission byte counts; the actual packet contents captured by Honeyd are not included. While advanced methods did exist to receive the contents of the packet capture from Honeyd, none are viable now due to the out-of-date libraries and compilation errors. Furthermore, Provos stated that he expected a NIDS or other scripts to be run in tandem with Honeyd [31]. Implementing a full packet capture whose contents are accessible is one of the deficiencies this research plans to address in the Honeyd framework.

Honeyd can be compiled with internal Python services that allow interacting with Honeyd through either honeydctl (Honeyd Control) or Python scripts while it is still running. This allows for the creation of dynamic honeypots. Honeydctl connects to Honeyd and presents the user with a console for issuing commands. These commands allow listing running honeypots, modifying, or deleting them. One noteworthy command is "!" which allows sending Python commands directly to Honeyd. By simply importing the honeyd module in honeydctl (after issuing !) or a Python script, the user now has access to all the data received and transmitted by Honeyd. Having access to this data would allow creating packet captures and signatures for received data. Unfortunately, Honeyd has compilation issues because of out-of-date library dependencies and the version of Honeyd 1.5c that can be installed from the Ubuntu packages list is not built with internal Python services. Therefore, this research per-

forms its own packet capture for signature creation and forensic analysis [31].

Honeydstats and Honeyview Honeydstats and Honeyview are plugins that allow analyzing the log that Honeyd generates from received traffic. Honeydstats is a text-based representation of packet level data received (very similar to Honeyd's 'l' option), while Honeyview is a web-based GUI representation. Both Honeydstats and Honeyview focus on the OS versions, destination ports, country codes, and IP addresses from attackers [31]. Although statistics can be aggregated from across the network, they both still rely on log files. The information they provide is not in depth enough for forensic analysis nor fast enough for today's cyber attacks. This research hopes to provide a framework with real time alerts while capturing detailed evidence for forensic analysis.

2.2.8   Cyber Deception

In their 2017 Cyberthreat Defense Report, the CyberEdge Group recommended that cyber deception technology should include coverage for IoT devices [32]. While not IoT specific, Cymmetria is leading the way on Deception Campaigns. They define cyber deception as "baiting, studying, investigating, fingerprinting, and/or smoking out" attackers. Through the use of cyber deception, organizations can prevent attackers from moving freely throughout their network and impose an increased cost to attack the defended network. Cymmetria explains that cyber deception is more than just implementing everything honey (honeypots, honeynets, and honeytokens), as traditional honey technology is difficult to integrate into a network, expensive to develop and maintain, and easy for attackers to detect. It is the creation, management, and monitoring of these false devices in order to manipulate attackers through a predefined and monitored path of the network (or as Cymmetria defines, "orchestration and virtualization"). Even if the attacker realizes there are fake devices and

documents, the speed of their attack and propagation is hindered because now they have to spend extra time verifying if a target is real or a honeypot. All the while, the attacker's tools and exploits are at risk of being captured. If they are captured, then a signature or patch can be created and propagated throughout the networks, or even worse for the attacker, reported to vendors and antiviruses and distributed throughout the world. This renders the attacker's tools worthless and requires them to spend more time and money to create new ones. While traditional honeypots could still pose this threat to attackers, alerts and all collected information on the attacker are not available in real time [30].

2.2.9 Programming and Languages

JavaScript   JavaScript is one of the staples for web development today. It is an interpreted language and allows interacting and modifying the HyperText Markup Language (HTML) Document Object Model (DOM), which enables creating dynamic web pages without requiring a user to reload it. Through the use of Node.js and Electron, applications such as HoneyHive can be created with dynamic Graphical User Interfaces (GUIs) [33].

Node.js   Node.js is a standalone runtime environment for JavaScript built and maintained by Google. It utilizes Chrome's V8 engine and possess the functionality to interact with the OS that JavaScript normally does not have since it is sandboxed in the browser. This extra functionality ranges from interacting with local files to networking modules for creating a full fledged server. The OS can be queried for information and concurrency can be implemented through the use of child processes. These are just a handful of built in modules, but using the Node.js Package Manager (NPM), modules can quickly be downloaded and installed for use in an application. Node.js is so versatile because it runs exactly the same across all platforms

[34].

Electron   Electron is a module for Node.js that allows creating cross-platform GUIs. It utilizes HTML, Cascading Style Sheets (CSS), and JavaScript to render the GUI and is essentially the same as coding a standalone web page for the application. This makes creating a GUI that works on any platform relatively fast and easy because of all the CSS frameworks available. Many applications have already been written in Electron, such as Atom, Visual Studio Code, Discord, and Slack, to name a few. Electron applications can even be bundled into executable files for ease of distribution using the Electron Packager module [35].

Python   Python is an interpreted language and currently supports two versions 2.7.X and 3.7.X [36]. The Python code written in this research uses 2.7.X because of the greater community support for Python modules available and because Lukas Stafira, whose work is built upon in this study, created his IoT honeypots with version 2.7.X. Modules can be quickly installed in Python by using its package manager, pip [37].

2.2.10   Tools

This section covers the essential tools used in this research for testing and results. These tools include Nmap, packet capturing software (Wireshark and TCPDump) and VMware Workstation. Docker is also discussed because several related researchers utilize it, and future work on this research references it.

Nmap   Nmap is an open source port scanning tool used by attackers and security professionals alike for reconnaissance and vulnerability analysis on networks and their devices [38]. It provides information on a device's ports, the services running

on the ports, and the suspected OS. All gathered information from ports, running services, and the Time to Live (TTL) in packet responses are compared against the Nmap DB to make a best guess about the target's OS. Nmap scans are customizable in the type of host discovery performed, to the way it scans ports for services, all the way to firewall and NIDS evasion. Using firewall and evasion flags, an attacker is able to spoof their MAC, port, checksum, TTL, and even modify their Maximum Transmission Unit (MTU), which results in smaller fragmented packets. The timing and performance flags allow adjusting timeouts and data transmission for faster or slower scans. If used in combination, these flags can result in scanning a network over a long period of time, without ever alerting a firweall or NIDS [38]. Varying the scan types and parameters of the scan and testing how effectively and quickly a scan is detected is one measure of performance for the HoneyHive framework.

Wireshark and TCPDump Packet capturing software is used to sniff traffic on a network and inspect it in further detail later. Both Wireshark [39] and TCP-Dump [40] utilize the libpcap library to capture network traffic, but Wireshark is GUI-based, while TCPDump is Command Line Interface (CLI)-based. Filters can be used to eliminate unwanted traffic either before or after capture to narrow in on specific hosts or protocols. Although both Wireshark and TCPDump allow quick filtering of data by specifying source / destination hosts and ports, TCPDump allows filtering on specific bytes in frames, packets, datagrams, and applications using the Berkeley Packet Filter (BPF) syntax [41]. Although Wireshark has a command line equivalent (tshark), TCPDump is used to collect traffic in finer granularity and then analyzed in Wireshark for this research [39][40][42].

VMware Workstation VMware Workstation is a hypervisor software solution that allows creating and running VMs [43]. Virtual Machines contain a separate

emulated CPU, OS, memory, and disk space. Hardware from the host system is shared between it and all running VMs and the amount of resources allocated to a VM is highly configurable, see Figure 2. Users can select the amount of memory and disk space a VM has access to, as well as the number of processor cores and peripherals it can use. Not only does VMware allow running multiple OSs on a single computer without having to reboot, as is the case with multi-booting, but it also provides a more secure, sandboxed environment to run applications. If an application becomes compromised in a VM, only that VM's OS is affected; other VMs as well as the host OS are not affected, see Figure 3. An attacker would have to break out of the VM OS, and VMware hypervisor to get at the host OS, a lengthier and more complicated process than escaping docker containers [43] [44].

Figure 2. VMware Workstation Hardware Settings

Docker   Docker is a program that allows running container images. A container image is packaged software with all the dependencies included, i.e., libraries, code, and other tools needed for execution. Because the containerized software includes all dependencies, execution is the same across different infrastructures. This greatly increases the portability and stability of software across different devices. At runtime, container images become containers, which results in isolating the running software from other processes and giving it only user-level privileges. This improves the security of the applications because now if the application is compromised by an attacker, the attacker is limited to access of that container only, essentially a sandbox. Whereas VMs run a guest operating system on top of the hypervisor to sandbox each application, the docker engine runs right on the host operating system. With the elimination of the guest operating system layer from Figure 3, containers utilize less resources (memory, disk, CPU), which allows running more containerized applications than virtual machines, as shown in Figure 4 [44].

Docker claims to increase application and device security, however, Sever and Kišasondi demonstrate that if container images are misconfigured then attackers can compromise other containers, possibly escape the container and compromise the host operating system [45]. While there are configurations and security measures that can be put into place to prevent this, the prepared Dockerfiles and many GitHub images do not use them [45]. Any system that is improperly configured becomes susceptible to exploitation and therefore users of Docker should not assume their systems are secure just because they are running containers. In fact, there are known exploits to escape docker containers as listed in the Exploit Database website [46]. Users should properly configure their containers with security in mind, and then lock down the security of the host operating system as well.

24
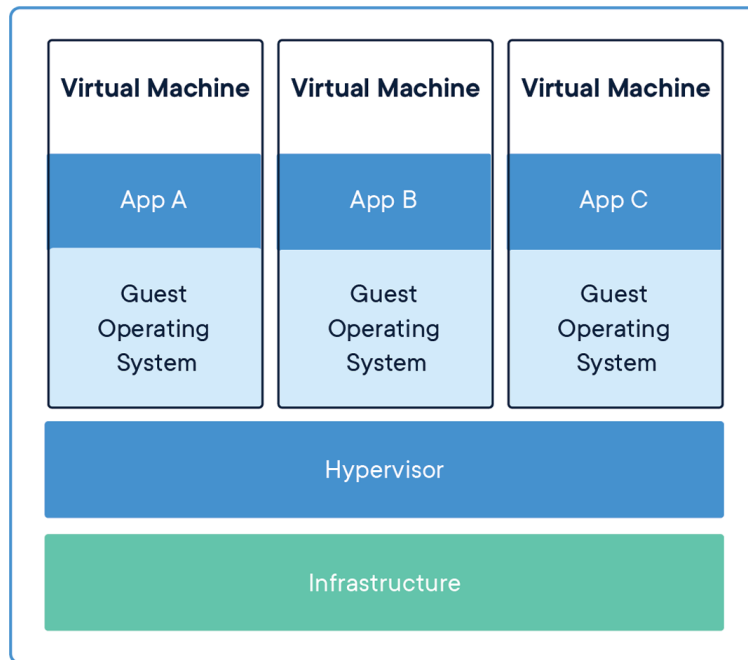
Figure 3. Virtual Machine Structure [44]
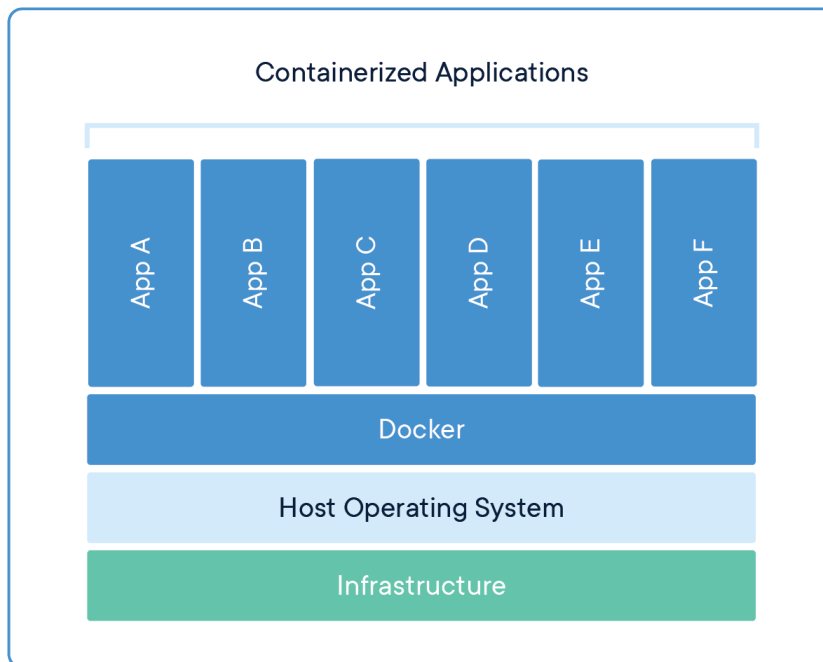
Figure 4. Container Structure [44]

2.3 Related Research

Many different frameworks for building honeypots have been developed and are explored in this section. They range from generic honeypots, ICS / SCADA and IoT honeypots. There are numerous honeypots that serve one purpose such as a specific exploit or service, but the focus of this section is honeypot frameworks that allow the creation of many convincing IoT honeypots.

2.3.1 Conpot

Conpot is developed and maintained by the Honeynet Project and is used to create ICS honeypots. Because IoT devices are used to control things like thermostats, electrical components, and appliances, they bare an ever-increasing resemblance to ICS. Conpot provides a suite of protocols found on ICS networks and throttles their responses to mimic real system response time [47].

2.3.2 IoT Web-Based Honeypots by Lukas Stafira

Using the Honeyd framework and Python, Stafira emulated the web services for three IoT devices to create realistic and interactive web-based honeypots [6]. These devices included the TITAThink Camera, Proliphix Thermostat, and ezOutlet2 Power Outlet. In order to make the devices appear dynamic, Stafira accessed local data, such as time and weather, and used them to generate web pages when responding to HyperText Transfer Protocol (HTTP) requests. Stafira tested whether Honeyd could be used to create near duplicate honeypots that simulate the web traffic of several IoT devices. The honeypots Stafira created successfully mimicked the HTML data of web transmissions for the real devices. He also tested the Transmission Control Protocol (TCP)/IP and HTML header similarity, response time, and Nmap completion time for SYN, UDP, and FIN scans. Stafira compared his results to the physical IoT devices

using Wireshark, Nmap, and custom Python scripts. His test network configuration is shown in Figure 5. Honeyd is shown being able to run all three IoT honeypots on a single VM. Overall, Stafira's results showed that it is possible to create convincing IoT honeypots, and the honeypots he created are used in this research as convincing IoT sensors for network intrusion detection [6].
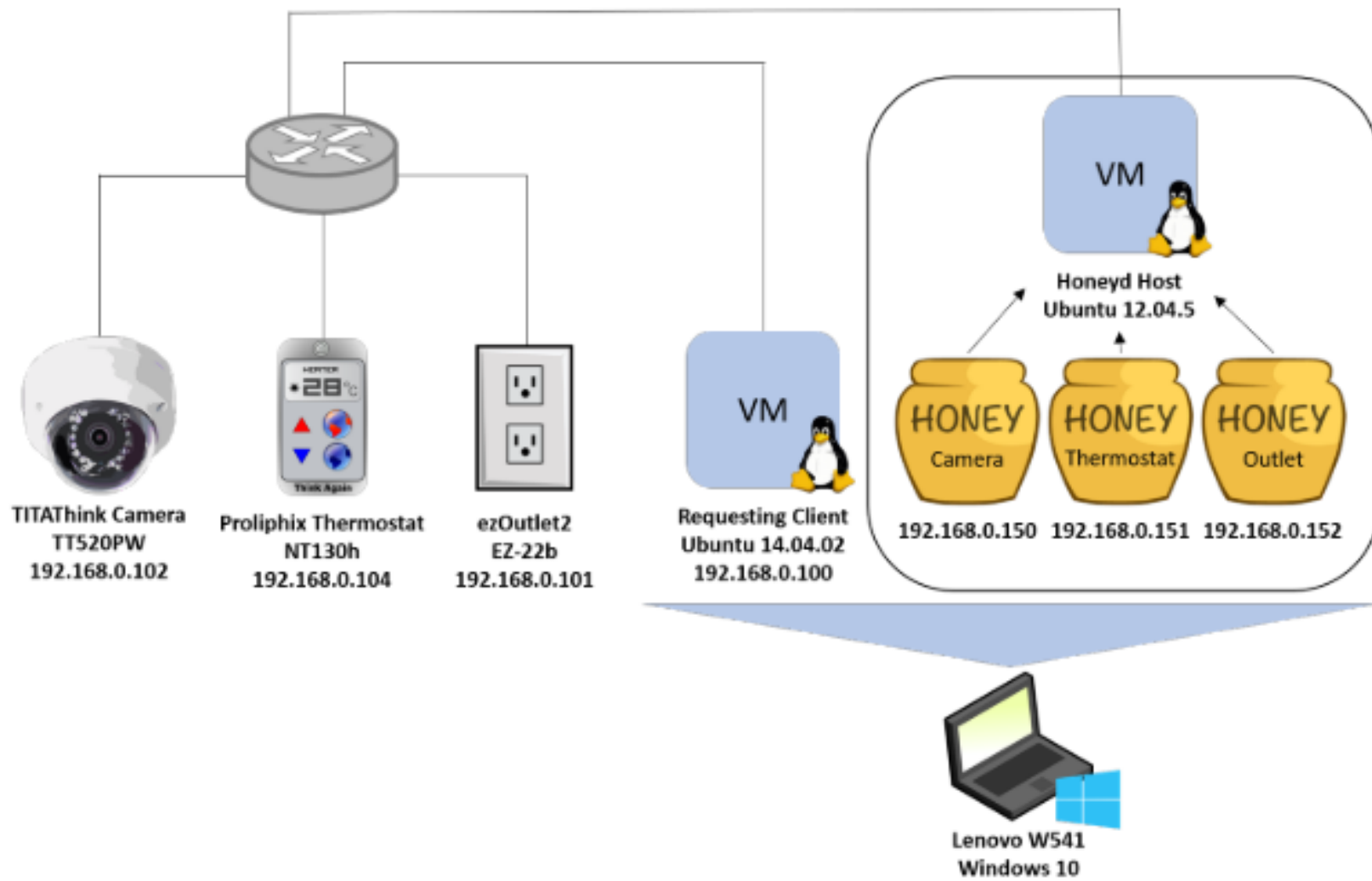
Figure 5. Stafira's Network Configuration [6]

2.3.3 Honeycomb by Christian Kreibich

Honeycomb is a tool for automatic signature generation from malicious network traffic captured with honeypots, specifically those part of the Honeyd framework [48]. Kreibich treats all traffic captured by honeypots as malicious because interaction with them is suspicious and not routine. The signatures generated are formatted for both the Zeek (formerly Bro) and the Snort NIDS [49] [50]. Honeycomb hooks into Honeyd and keeps track of network connections (IP and port combinations), while filtering out traffic received from being scanned, and generates signatures using the Longest Common Substring (LCS) algorithm. Using Honeycomb, Kreibich successfully generated signatures for both the Slammer Worm and the CodeRed II Worm [48].

2.3.4 Honeyd Syslog Solutions

Kiwi Syslog Server Kloet demonstrated how using Kiwi, it is possible to filter Syslog messages generated by Honeyd. The Syslog messages were sent from the host machine running Honeyd to the machine running the Kiwi NIDS [51]. The Kiwi program then filtered Syslog messages and generated Simple Mail Transfer Protocol (SMTP) email alerts based on predefined rules, such as a connection being established to a honeypot. Kloet also mentioned remedies for false positives which include fine tuning the Kiwi alert threshold, creating a static route to null, and excluding the address that the Honeyd daemon listens on [51]. Kloet's solution of forwarding the Honeyd generated Syslog to a program more capable of parsing and displaying the alerts in a readable format is useful for small networks. However, in larger networks, network administrators could easily be flooded by emails, whether the emails are actual alerts or false positives, and it could be difficult to piece together and visualize what is happening. While Kiwi can filter out the noise of false positives, it has no graphical overview of the network for easy real time interpretation by network

operators and administrators [52].

Honeycomb by Lavenya and Kaur   Honeycomb by Lavenya and Kaur is a Honeypot log management tool. It gathers all log files generated by the Honeyd framework, emails them in one file for download, and then allows importing and inspecting them in the web-based GUI [53]. Much like the Kiwi syslog server solution, it is a step towards making the Honeyd log files more manageable and collection of them automated. However, the alerts are still not conveyed to network operators fast enough and the data in the log files is not in-depth enough to capture exploits, tools, or an attacker's TTPs [53].

2.3.5   IoTCandyJar

Ramirez et al. discuss the need for their framework since building custom IoT honeypots or buying the actual physical device to create honeypots are too costly [54]. The vast heterogeneity of IoT devices makes creating custom IoT honeypots time consuming and they often are not high-functioning enough. To combat this, their framework uses machine learning to replicate the behavior of IoT devices, dynamically creating realistic honeypots, and presenting them as convincing devices to attackers [54]. Figure 6 displays the IoTCandyJar framework. This framework consists of three dynamic honeypots that attackers interact with (left), a DB that records responses from scanned IoT devices on the Internet (middle), the IoTScanner which conducts the scanning of IoT devices on the Internet (top right), and the IoTLearner which uses heuristics and training to predict correct responses to attackers (bottom right). Requests from attackers are first sent to IoTCandyJar's dynamic honeypots. These honeypots then query the DB for responses that could be correct. The result of the query is in-turn passed to the IoTLearner which uses heuristics to select what it believes to be the correct response. This response is finally forwarded to the attacker.

The IoTScanner constantly adds to the DB by using attacker requests to scan IoT devices on the Internet [54].

While the framework can quickly imitate any IoT device connected to the Internet, the methodology cannot precisely match responses for exploits without sending actual IoT devices the exploits, which is illegal. IoTCandyJar does use some extent of exploit filtering, but this only works for known exploits. For the known exploits, they either have to manually create the response or drop the connection altogether which means they are back to creating custom low-interaction honeypots. While for unknown exploits, their own system may very well become an attacker itself. In addition, IoT devices have specific ports open and services running on them, which would preclude a single device from responding to a Nmap scan with the exact IoT profile the attacker is targeting [54].

2.3.6   HoneyLab

HoneyLab is a distributed framework for deploying and sharing honeypots between cyber-security researchers that seeks to address the shortcomings Chin et al. described as infrastructure fragmentation, flexibility for deploying devices, and the limited IP address space [55]. HoneyLab runs honeypots in a virtualized environment for high level interaction and attack containment. The framework, shown in Figure 7, is composed of a web interface to register / login and control honeypots, the C2 node called HoneyLab Central, and sensor nodes distributed worldwide that run on Xen servers. The Xen servers deploy honeypot VMs alongside VMs with sensing software. Users can upload custom VM images which allows for maximum flexibility and custom honeypot support. Commands can be issued to honeypots through HoneyLab's web interface but users also have the option of interacting with their honeypots through Virtual Network Computing (VNC) or a remote shell after establishing a Virtual

31
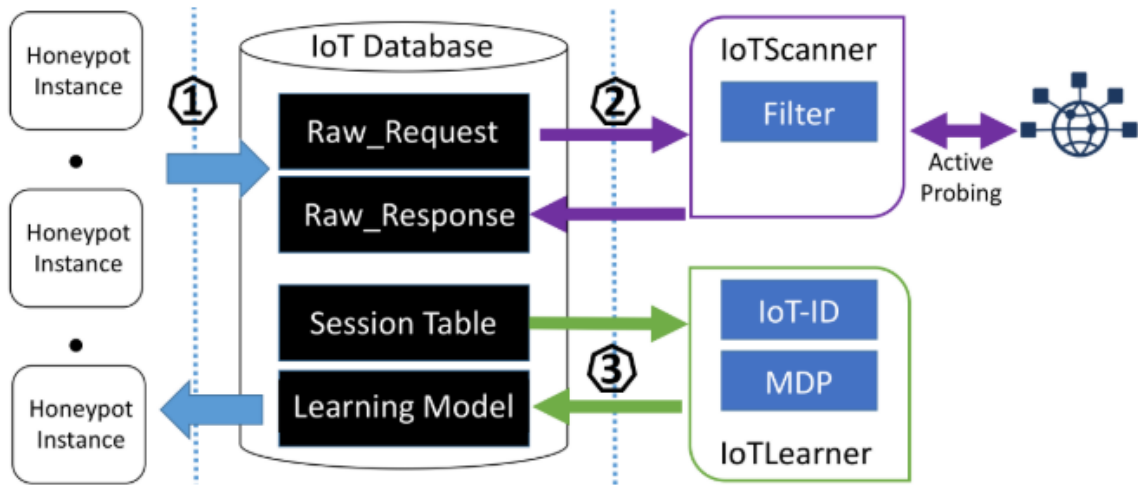
Figure 6. IoTCandyJar Design [54]

Private Network (VPN) connection to the network. All sensor nodes run the Honeylab daemon software to communicate with the C2 HoneyLab Central device to report alerts and receive commands [55].

Limitations with the framework include IP-only level traffic (no Ethernet), all traffic must go through the HoneyLab Central device which could become overburdened, and all outgoing connections (reverse connections) are blocked. These limitations affect the convincingness of the actual honeypots and may not fool attackers. Also, once an attacker is in the target's internal network, they can see that the honeypot is not part of the network and all traffic is forwarded to it. Additionally, it is not apparent how propagation throughout the HoneyLab honeynet from a compromised honeypot is prevented. Finally, the research appears to be discontinued because the website was not found to be up and operational [55]. Like HoneyLab, the IoT honeypot sensors in this research beacon back to a central command and control server for real time alerts.
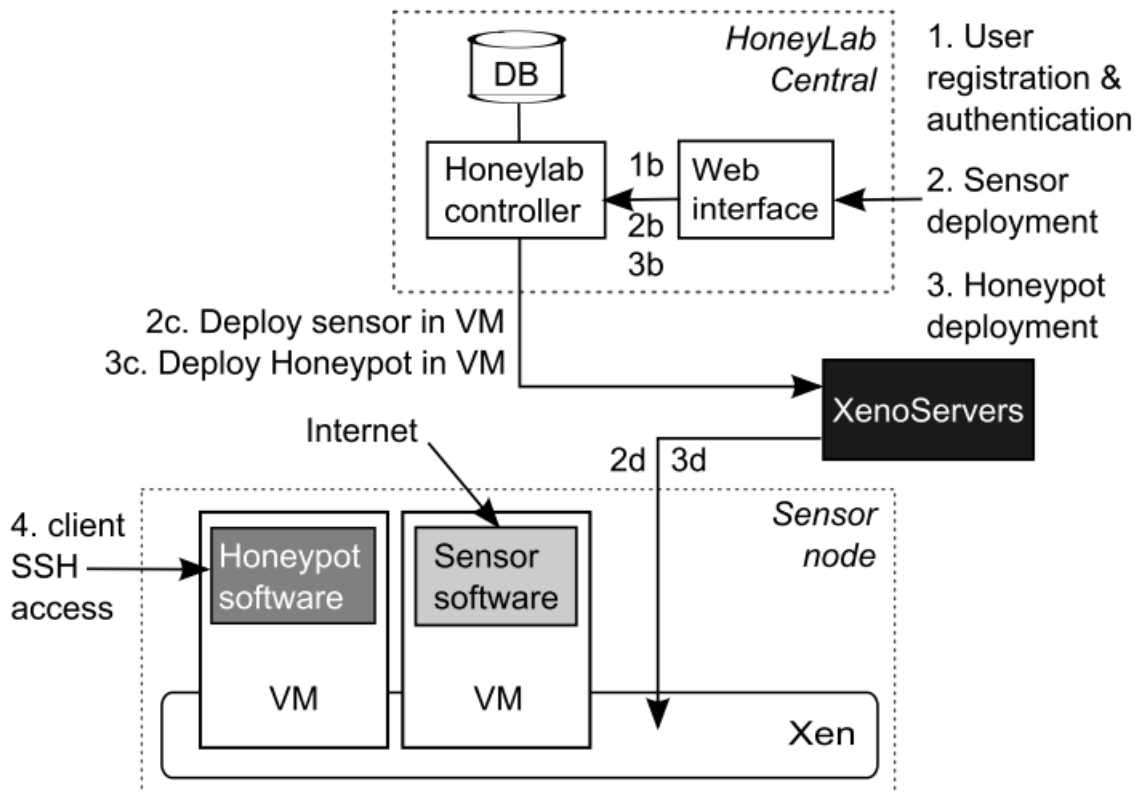
Figure 7. HoneyLab Design [55]

2.3.7 SIPHON

Like HoneyLab, SIPHON is a globally distributed honeynet intended to be a "Scalable, high-Interaction Physical HONeypot" framework [28]. As shown in Figure 8, the framework uses IP addresses distributed around the world from servers rented from cloud providers (Amazon, Digital Ocean, and Linode) that act as "wormholes" – interconnecting the honeynet through SSH tunnels. By using this design, certain geographically located devices that are more desirable to attackers can be simulated. As Figure 9 illustrates, the wormholes send the attacker's traffic to SIPHON's "forwarder" devices that change IP address and perform man-in-the-middle attacks before finally sending the traffic to actual physical IoT devices. This setup is very similar to IoTCandyJar's method of sending traffic to physical devices, but instead of merely recording the devices' responses for replay, SIPHON's network owns the IoT devices and, can, therefore, allow high-interaction and record advanced attacker methodology [28].

2.3.8 HoneyIo4

HoneyIo4 by Alejandro Guerra Manzanares is a low-interaction honeypot with four Python scripts to match the expected Nmap DB scan responses for the following IoT devices: GoPro Hero3 camera, Casio QT6600 cash register, Nintendo Wii video game console, and Oki B4545 printer [23]. HoneyIo4 also includes a web-based GUI that allows starting or stopping each honeypot by simply executing the associated Python script. While HoneyIo4 successfully matched target Nmap DB OS profiles, it appears to be trying to re-invent the wheel as the Python scripts attempt to do what Honeyd does already. Honeyd allows for quickly customizing ports, responses, and specifying an OS profile from a Nmap DB for response traffic to match. Manzanares claims that Honeyd cannot match IoT OS fingerprints, but as long as the OS is in
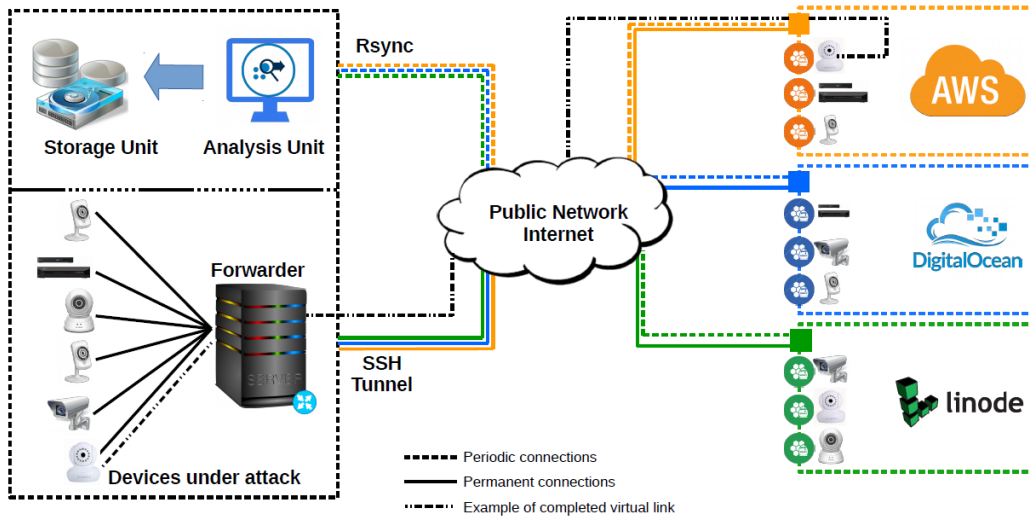
Figure 8. SIPHON Overview [28]
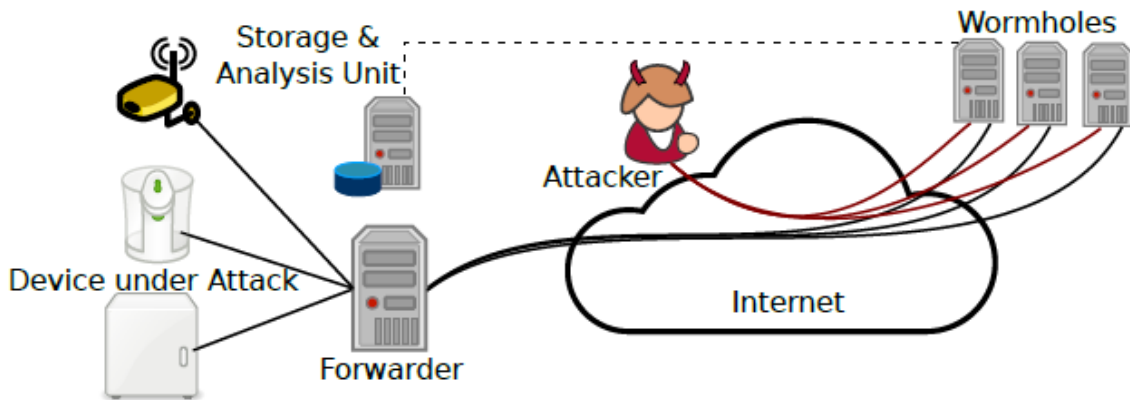
Figure 9. Attacker's Interaction with SIPHON [28]

the supplied Nmap DB file to Honeyd, the traffic can be matched.

Honeyd also has more advanced capabilities than HoneyIo4 such as running multiple honeypots at once on the same physical device, routing of network traffic, and keeping state for each honeypot [23].

2.3.9   IoTPOT and IoTBOX

IoTPOT and IoTBOX is a two-part honeypot system consisting of a Telnet service "frontend" (IoTPOT) and sandboxed "backend" (IoTBOX) [22]. IoTPOT changes its responses to match different IoT devices that an attacker is targeting based on their initial Telnet requests as illustrated in Figure 10. By using this method, IoTPOT can appear to be a vast number of different IoT devices. IoTPOT also logs all traffic which includes login attempts and credentials. Login settings can also be customized to allow authentication on the first attempt, a specific username and password combination, or authenticate only after a set number of attempts. After an attacker successfully authenticates, IoTPOT checks if the command issued has a known, stored response. If it is a known command, IoTPOT responds to the attacker directly. If the command is not known, IoTPOT forwards the command to IoTBOX, stores IoTBOX's response so it can quickly respond to the same command in the future, and then forwards it to the attacker [22].

The design of IoTBOX is shown in Figure 11. Because some commands can be to download malware, IoTBOX is ran in a controlled environment with frequent image resets. IotBOX uses QEMU to emulate eight different Central Processing Unit (CPU) architectures which are then run on the OS OpenWRT. The benefit of this is that malware executables are compiled to run on a specific CPU architecture, and through CPU emulation, the captured malware can be run and analyzed in depth [22].

It is not apparent how IoTPOT would know the correct banner response an at-

tacker is looking for from a specific IoT device. Also, because IoTPOT uses one IP address, an attacker or scanning tool that documents device analysis would notice this single IP responds like multiple different devices and may become suspicious of it being a honeypot.

2.3.10 Multi-Purpose IoT Honeypot

Inspired by IoTPOT, Krishnaprasad created the "Multi-Purpose IoT Honeypot" to handle four protocols commonly used by IoT devices: Secure Shell (SSH), Telnet, HTTP, and CPE WAN Management Protocol (CWMP) [56]. Multi-Purpose IoT Honeypot utilizes a "frontend" proxy that is running a Python script for each of the supported protocols. The frontend logs data about the attack and then forwards it to the corresponding service "backend" which are only two docker machines running the services. While Multi-Purpose IoT Honeypot is running common services for IoT devices, it does not tailor its responses to deceive Nmap scans performed by attackers that it is in fact an IoT device and not a honeypot. Furthermore, if an attacker does connect to a service, they realize it is not an IoT device and not connected to any real network. Krishnaprasad's use of docker to containerize honeypot machines is a concept also used by Cymmetria's Honeycomb framework. This technique allows for high-interaction honeypots that are easier to develop, deploy, and maintain than physical devices or traditional virtual machines [56].

2.3.11 ThingPot

Like IoTPOT and Multi-Purpose IoT Honeypot, ThingPot also has a frontend and backend design [57]. ThingPot classifies itself as a "Medium Interaction Honeypot" simulating Extensible Messaging and Presence Protocol (XMPP) and Message Queue Telemetry Transport (MQTT) and low interaction for HTTP REST traffic. Each of
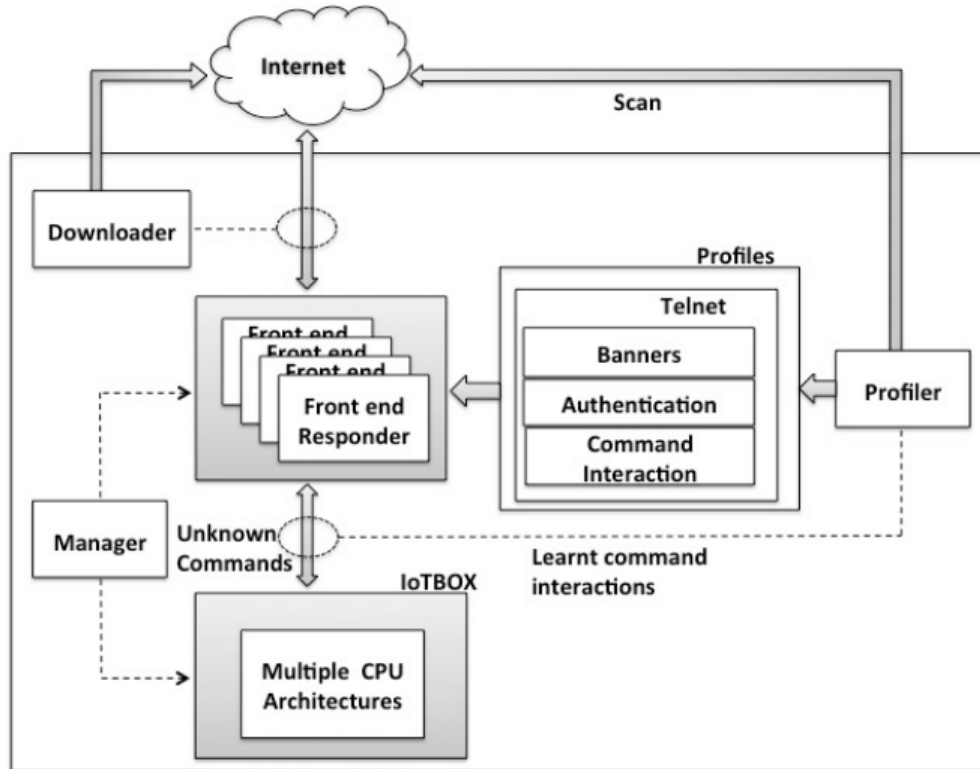
Figure 10. IoTPOT Overview [22]
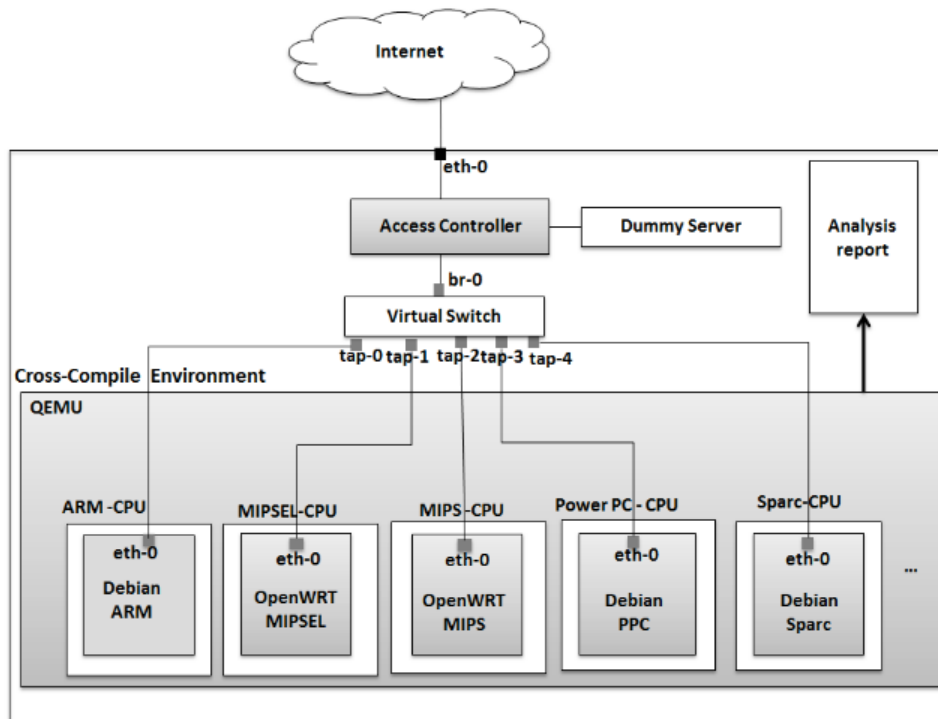
Figure 11. IoTBOX Overview [22]

these services is also run in a virtual environment using docker. An overview of ThingPot's design is shown in Figure 12. The XMPP and REST nodes implement that respective protocol while the controller node logs and stores data. Using this design, ThingPot imitated a Phillips Hue smart light and had an actual attacker try and take control of it [57].

2.3.12   IoTSec

One proposed solution for securing IoT devices is the use of interceding devices called μmboxes as an intermediary between IoT devices, that dynamically configure firewall rules to allow for the specific traffic of IoT devices on the network, essentially acting as a personal firewall or blue coat proxy for the IoT devices [20]. μmboxes work together and alert the centralized IoTSec Control Platform (C2) if an intrusion or anomaly is detected. They utilize several different methods for detecting intrusions: signature matching, network baseline generation, and cross-device policies. Signature matching and network baselining are not new concepts, but cross-device policies are interesting because using the functionality of other IoT devices, a safety check can be performed. The example Yu et al. give is an IoT camera checking that a person is home before a smart oven is allowed to be issued the "on" command [20].

2.3.13   Honeycomb and MazeRunner by Cymmetria

Cymmetria is a cyber-security solutions company based out of Tel Aviv, Israel and was founded in 2014 by Gadi Evron [58]. The Chief Executive Officer (CEO), Gadi Evron, has over 15 years of experience in cyber security and was the former vice president of cyber security strategy at Kaspersky Lab. Cymmetria's flagship product is MazeRunner which utilizes their Honeycomb framework [59].

Honeycomb by Cymmetria, not to be confused with Honeycomb by Kreibich (a
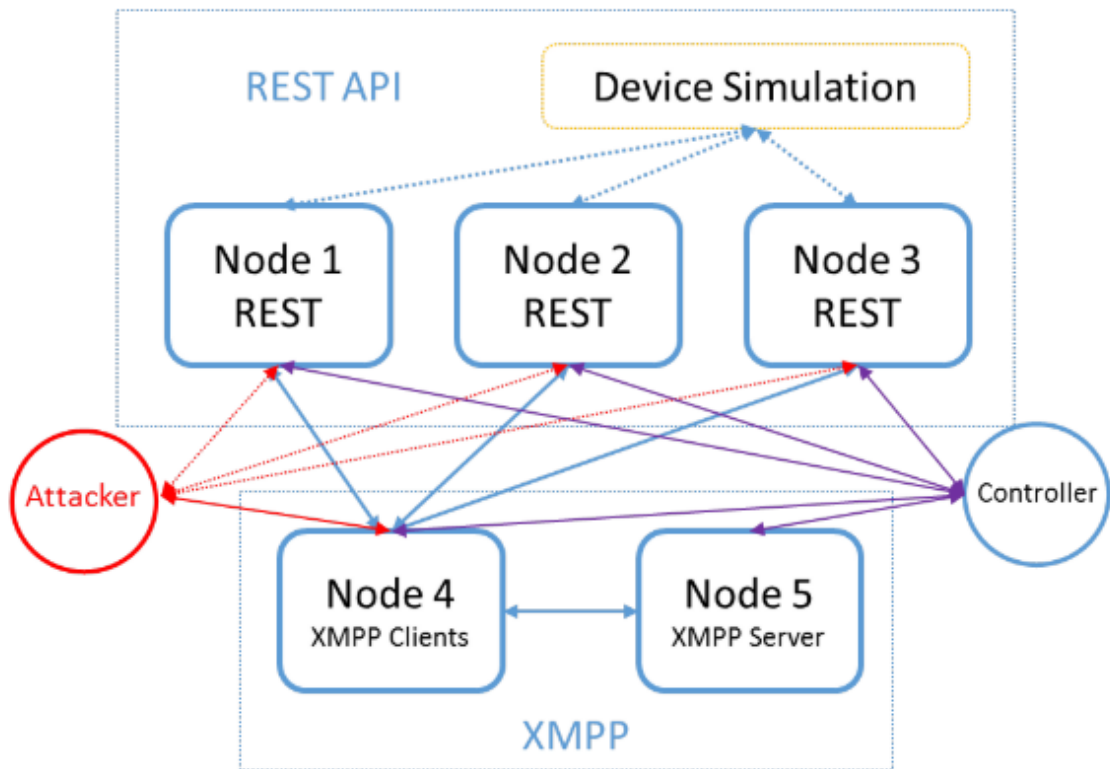
Figure 12. ThingPot Overview [57]

plugin for Honeyd), or Honeycomb by Lavenya and Kaur (a Honeyd log manager), allows for rapid and customized honeypot creation using containers and Python plugins. Honeypots are spawned off through the use of these containers [60].

MazeRunner allows users to create honeypots, add services, and change configurations all rapidly and with a GUI. MazeRunner is the container that manages all the honeypots and acts as the command and control for intrusion detection. It provides an overview of real time alerts of interaction with the honeypots. It implements packet capture, memory dump, shows what commands an attacker ran and allows downloading the tools an attacker used. The tool's hash can then be propagated as a signature to flag on throughout the network. Custom scripts such as Stafira's can be run with the Enterprise edition for even more customized honeypots. The backend uses their Honeycomb framework [61] [62] [63].

MazeRunner contains a component called ActiveSOC which automatically investigates an incident using rules and heuristics to determine if the incident needs further investigation by an analyst. Using ActiveSOC, false positives can be reduced and analysts can focus on investigating actual intrusions [64].

MazeRunner has been successful in catching red teams in NATO exercises [29] as well as APTs such as APT3 (pirpi - a Chinese Threat Actor) in European government networks, defense contractor networks, and several other customer's networks [65]. Additionally, MazeRunner successfully captured the tools and TTPs of the cyberespionage group Patchwork, as Patchwork moved throughout the MazeRunner network. Patchwork, aptly named from the copy-paste code used from online forums, is a targeted attack against government agencies and has infected several thousand machines since 2015 [66].

2.3.14 Comparison of Related Frameworks

This research builds off of the IoT honeypots created by Lukas Stafira. The closest research to the HoneyHive framework is MazeRunner by Cymmetria. However, MazeRunner does not use nor allow creating custom IoT honeypots as network intrusion detection sensors. Table 1 provides a summary and comparison of all the frameworks mentioned. The specific categories compared include the honeypot level of interaction (Honeypot Level), whether or not the framework focuses on IoT honeypots (IoT), whether or not the framework implements full packet capture (PCAP), whether or not the framework is distributed (Distributed), whether or not the framework was developed with the intent for it to be used as a NIDS (NIDS), whether or not the framework reports alerts and receives commands from a C2 server (C2), and the year of the last update on the framework (Last Update). The various levels of honeypot interaction include low, medium, and high. An "X" in a category denotes the framework possess that trait. Neither Conpot nor Honeyd were made to create IoT honeypots specifically, perform full packet capture, have a distributed framework, be implemented as a NIDs, or have a Command and Control structure.

Framework	Honeypot Level	IoT	PCAP	Distributed	NIDS	C2	Last Update
Conpot	Medium						2019
Honeyd	Low						2013
HoneyHive	Low	X	X	X	X	X	2020
HoneyIo4	Low	X					2017
HoneyLab	High			X		X	2009
IoTCandyJar	Low	X					2017
IoTPOT & IoTBOX	High	X					2015
MazeRunner	High		X	X	X	X	2020
Multi-Purpose IoT Honeypot	High	X					2017
SIPHON	High	X	X	X			2017
ThingPot	Medium	X					2017

Table 1. Comparison of Honeypot Frameworks adapted from [19]

2.4 Chapter Summary

This chapter discusses the exponential growth of IoT devices and the need for improved IoT and computer network security. Several methods for improving network security include the use of a NIDS, network monitoring, and honeypots. The HoneyHive framework utilizes a combination of all these aspects for increased network security. Languages and tools for the development of the HoneyHive framework are also covered in detail. Finally, related research in the field of IoT honeypots is explored to understand existing solutions, their shortcomings, and their inspiration in the development of this research.

III. Framework Design

3.1 Overview

This chapter describes the design decisions involved in the creation of the Honey-Hive framework. The HoneyHive framework is comprised of IoT honeypots simulated by Honeyd, a C2 server, and a HoneyB Agent script that beacons back to the C2 server. Using the IoT honeypots as NIDS Sensors, the system can recognize scanning patterns and known IoT exploits to create real time alerts. Full packet capture of all network traffic received by the honeypots is also a feature. Additionally, Snort, Suricata, Nmap, VMware, and Wireshark are tools used for augmentation in the testing of this framework.

3.2 Motivation

While there are numerous NIDS and network monitoring solutions available, none meet all the objectives of this research. For one, most NIDS and network monitoring software are not tailored to honeypots or IoT. Without a convincing, high-interaction honeypot, an attacker will not launch advanced exploits (zero-days), deploy tools, or employ their TTPs [28]. Additionally, many NIDS log to files or send an email to an administrator when alerts are generated. This is neither real time nor manageable for large-scale networks. The closest solution found to this proposed research is MazeRunner by Cymmetria as discussed in Chapter 2. However, the honeypots used in MazeRunner are not IoT devices, and MazeRunner does not currently support running custom scripts for honeypot emulation.

NIDS may also only offer a limited view of the entire network they are defending. On network switches, traffic is sent only to the port of the intended recipient. Some switches contain mirror ports that send a copy of all traffic received out another port

typically for analysis by a NIDS. In addition, for every switch another mirror port and NIDS would be needed, and then only a specific segment of the network would be analyzed individually by each NIDS. This would not provide a global picture of all network traffic across all switches for analysis. The proposed solution in this research scatters IoT honeypot sensors across the network and has them beacon back to a C2 server, where traffic is then combined for analysis. Packets are captured using Scapy on the honeypots, sent to the C2 server, and then analyzed by Snort IDS. In this experiment, Nmap is used to conduct various scans against the HoneyHive framework to test how effective it is at detecting network scans.

One question focused on in the design of the HoneyHive framework was, "What are the qualities of an effective NIDS?" Roesch describes some of the qualities of the Snort NIDS that have made it so effective, even still today. It is lightweight, cross-platform, has a small network footprint, and can be easily configured by an administrator in a short amount of time. Cross-platform means the NIDS should run on a variety of different enterprise systems and OSs. A small network footprint means a device does not generate large amounts of network traffic [67]. Furthermore, an ideal NIDS should have a low rate of false positives and virtually no false negatives (for known exploits), provide meaningful alerts in real time, and not introduce further vulnerabilities into the network.

3.3 Third-Party Software

This section describes the rationale behind software solutions used in the Honey-Hive framework that were developed by other individuals. These solutions includes Honeyd, Stafira's three IoT honeypots, Ubuntu, Snort, Suricata, Nmap, VMware, and Wireshark.

3.3.1   Honeyd 1.5c

This section describes the design decision of using Honeyd 1.5c. Because Lukas Stafira reported stability issues with Honeyd 1.6d and built IoT honeypots using 1.5c, the latter is used in this research [6]. Honeyd 1.5c is installed using apt-get install because compiling the source code generates errors even after following guides and downloading and installing older versions of library dependencies. Unfortunately, the version of Honeyd installed with command "sudo apt-get install honeyd 1.5c" was compiled without Python support. The inability to compile Honeyd prevented compiling it with the Python support as well as the Honeycomb plugin, which requires Honeyd being recompiled with its source files included. This severely impacts the level of control and interaction with the Honeyd program, prevents use of the Honeycomb automatic signature generation plugin, and influenced other HoneyHive framework design decisions.

3.3.2   Stafira's Honeypots

The three honeypots used in the HoneyHive framework are the TITAThink Camera, Proliphix Thermostat, and ezOutlet2 Power Outlet. They simulate the web interface of real IoT devices and were created by Lukas Stafira using Honeyd 1.5c, bash, and Python 2.7. These honeypots were selected because of the high level of web-based authenticity they shared with their real counterparts and the in-depth analysis already performed on them [6].

3.3.3   Ubuntu 12

Ubuntu version 12 was used for two reasons. First, it was used by Lukas Stafira in his research with Honeyd, and second, newer versions of Ubuntu did not have Honeyd available in repositories for installation. Because of the previously mentioned

compilation problems, this ruled out the newer versions of Ubuntu.

3.3.4   Snort

Rather than create a NIDS when there are many solutions already available, Snort was selected because it is lightweight and open source. It is also widely used and has a vast community of support. Community Snort signature rules, that are periodically updated, are readily available for download [50] [67]. Snort is used in the HoneyHive framework to parse PCAPs and create alerts, but was not used as a NIDS in the DMZ. Suricata would have been used as the NIDS to perform signature matching in the HoneyHive framework instead of Snort, but there were issues with getting it to run on Windows 10.

3.3.5   Suricata

Suricata is used in the test network of this experiment but is not part of the HoneyHive framework. In the simulated network, Suricata was selected as the NIDS to monitor SPAN traffic in the DMZ. Suricata was selected as the DMZ NIDS in this experiment because it uses the same rule structure as Snort which allowed the exact same rule-set to be shared between the two. In addition, Suricata employs modern NIDS features such as multi-threading and is regularly updated [68]. In this experiment, it was important to have highly capable NIDS to compare results against, otherwise results would not be as meaningful.

3.3.6   Wireshark

Wireshark itself was used primarily for troubleshooting in pilot studies. However, several tools included in the Wireshark package were used in the actual experiment, Mergecap and Capinfos. Mergecap was used to combine multiple PCAPs into a single

PCAP. Capinfos was used to count the number of packets in the PCAP.

3.4 Programming Languages

This section describes the design decisions behind chosen languages in the development of this framework. Node.js was used for coding the C2 server whereas Python 2.7 was used for the HoneyB Agent that communicates with the C2 server. SQLite was the language selected to create the DB.

3.4.1 Node.js

Node.js was selected over other languages such as C++, Go, and Java for several reasons. Node.js uses Chrome's V8 JavaScript engine to achieve a standalone runtime environment. Any computer with Chrome will be able to run the program with the same execution and output [34]. Another selling feature of Node.js is the Electron package, which allows rapidly creating GUIs using HTML, CSS, and JavaScript [35].

3.4.2 Python 2.7

Python 2.7 was selected instead of 3.X because the IoT honeypots selected were written in 2.7. Originally, the design was to modify the IoT honeypots to beacon back to a server. However, without Python support in Honeyd, there was very little interaction and control provided for the honeypots. Additionally, Ubuntu 12 allowed installation of Python 2.7 but for Python 3.X, only 3.2.X could be installed and the current version is 3.7.X. Python 3.2 also lacked support for modules available in 2.7 and 3.7, such as pyshark and Scapy [69].

3.4.3 SQLite

Because the DB is stored on the same file system as the C2 server, there was no need to authenticate to it over the network like in some languages such as MySQL. This allows for much faster querying and DB modification since the limiting factor is now disk read / write speed and not network latency [70].

3.5 HoneyHive Framework Design

This section describes the individual components that make up the HoneyHive framework. Shown in Figure 13, these components include the C2 server, the HoneyB Agent script, and the DB. After these components are described, the network layout for experimentation and testing is explained. The network layout includes the HoneyHive framework, Honeyd, Stafira's honeypots, Snort, Suricata, the simulated test network, and an attacker machine.
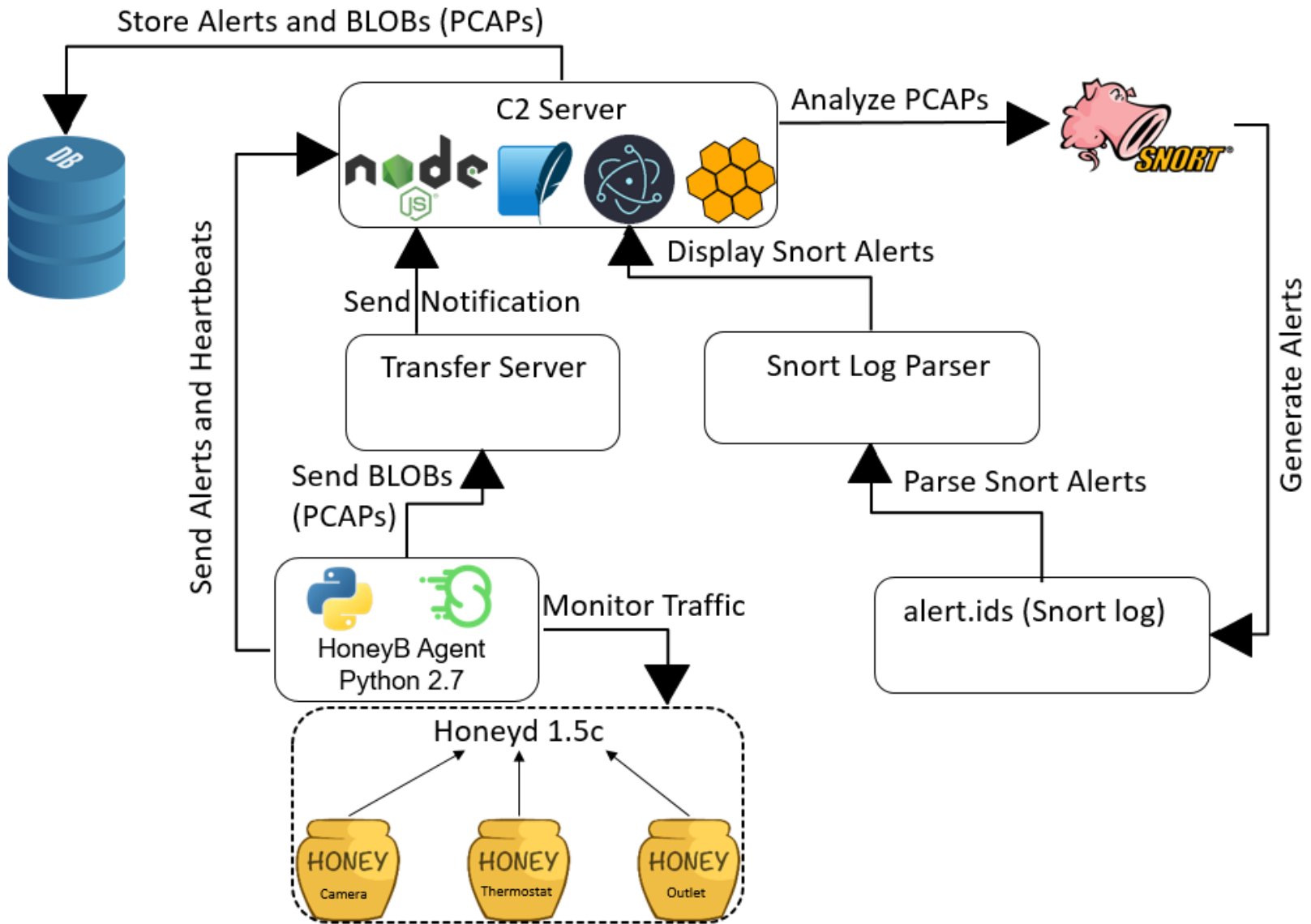
Figure 13. HoneyHive Framework

In Figure 14, the Unified Modeling Language (UML) diagram displays the interaction of all the components that make up the HoneyHive framework. The HoneyB Agent script monitors the Honeyd honeypots for traffic, captures any traffic, alerts the C2 Server, and transfers captured traffic to the C2 server. Credentials, commands, and attempted binary uploads are all captured in network traffic. The HoneyB Agent also periodically reaches out to the C2 server to report it is still functioning, known as a heartbeat.

The C2 server receives alerts and captured traffic from the HoneyB Agent script and then relays all of this to the DB, GUI, and Snort. Although the GUI development is for future work, the overall system was designed with it in mind for ease of use by network operators. The HoneyHive framework functions fully without a GUI and displays relevant information via command line. Additionally, the C2 server provides a kill switch capability to terminate the HoneyB Agent script, Honeyd and the honeypots, and the rest of the framework.

The DB stores alert information and captured traffic. The DB also supports exporting all data, and clearing parts or all data in the DB.
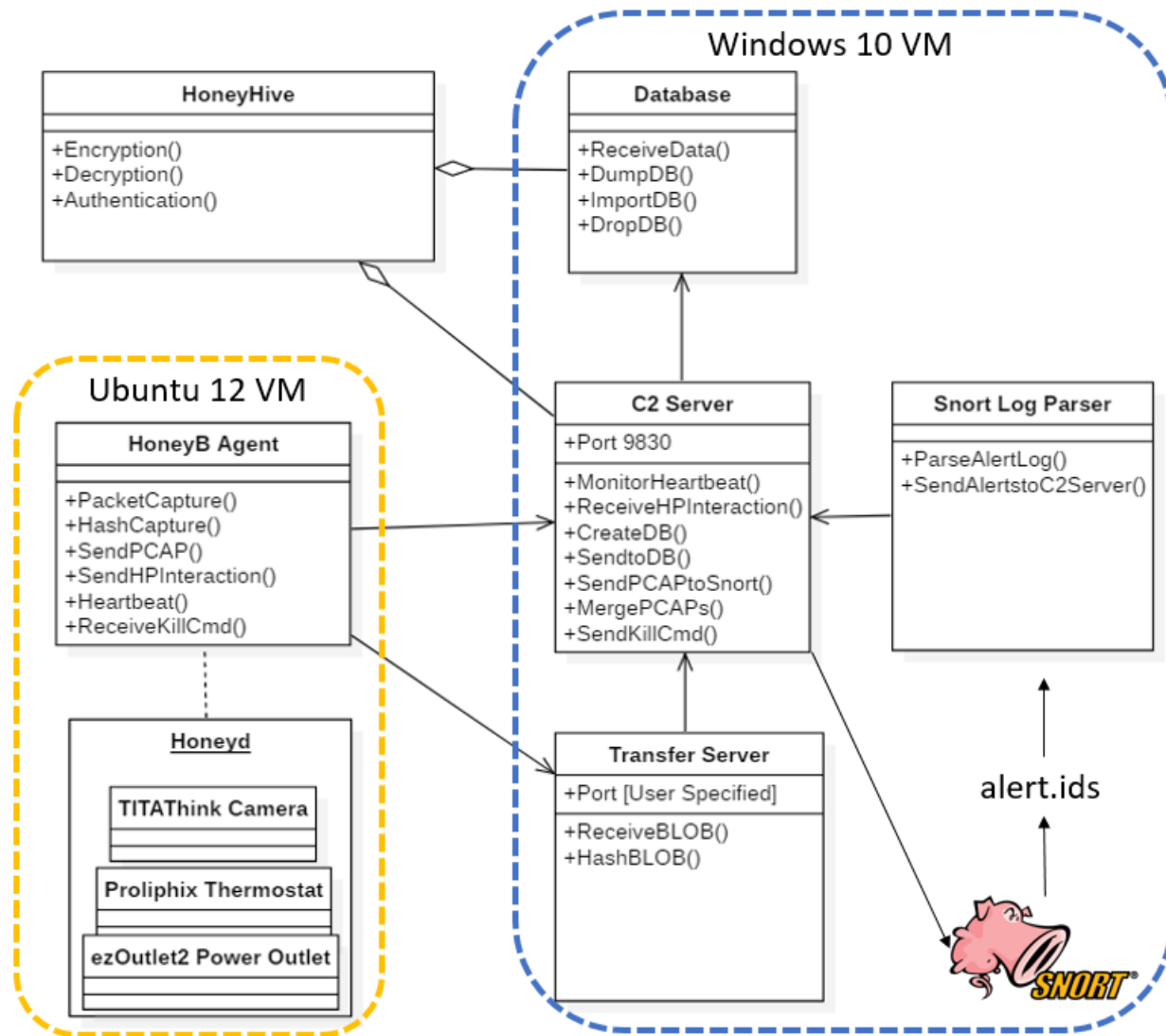
Figure 14. UML Program Design

3.5.1 C2 Server, Transfer Server, and Snort Log Parser

After setting up Honeyd and Stafira's honeypots, the C2 server was the first component developed because every other component interacts with it. The networking features were among the first to be developed. For handling higher loads and lower network latency, the server uses both multithreading and IPC. The C2 server scripts (main.js) are located in Appendix A.

Originally designed with a single server (the main server) that accepted alerts and heartbeats from honeypots, a secondary server (the transfer server) became necessary for handling the transfer of PCAPs. Without a separate transfer server, binary-encoded data would have to be converted to base64 and then back again to send over a channel expecting JavaScript Object Notation (JSON) data. This decreased the server's robustness because additional checks would need to determine if the data was in JSON format or binary data. The conversion to base64 also created a large overhead in transferring files because it increased the number of bytes that needed to be sent in addition to the CPU cycles required for the conversion. These factors ultimately led to the separate transfer server, which listens on a different port for receiving large files.

C2 Server Messages are sent in JSON format from the HoneyB Agent script to the C2 server with one of the following four commands: AUTHENTICATE, ALERT, HEARTBEAT, and PCAP. The AUTHENTICATE command allows Honeyd machines and their respective honeypots to be authenticated and then added to the list of IPs that are tracked by the C2 server. The ALERT command notifies the server of suspicious activity, which is then displayed to the network operator via command line and sent to the DB to store the incident. This allows for real time intrusion alerts and notification of honeypot interaction. HEARTBEAT is the pulse

function used by the HoneyB Agent script to periodically report that Honeyd and the honeypots are all functioning correctly. If the HoneyB Agent misses too many heartbeats, then an alert is generated, displayed, and stored in the DB. This heartbeat alert threshold is a user configurable parameter. Finally, the PCAP signals that the HoneyB Agent script is ready to transfer captured network traffic. To facilitate this, the transfer server is spun up and listens on the port number specified by the HoneyB Agent to download the PCAP. After a PCAP is successfully downloaded, it is stored on disk and in the DB. It is then sent to Snort for analysis, which supports the objective of full packet capture for later forensic investigation.

Additionally, several commands are added to the C2 server for the purpose of running the experiment. These include SNORT, SURICATA, REBOOT, and RESET. While packets were originally processed by Snort immediately after being received, Snort processing on packets was changed on the C2 server to wait until after the SNORT command was received from the runExperiment.py script. This made it so that Mergecap only ran once to merge all PCAPs together, and Snort only ran once on the final merged PCAP, as opposed to subsets of the merged PCAP and creating an incorrect number of alerts in the log file. Capinfos is run immediately after Mergecap in order to gather the number of packets captured. For network deployment, this entire process would be replaced with a user-defined parameter to run Mergecap and Snort on captured traffic after a set amount of time. However, in this experiment, when the SNORT command is received, the C2 server creates a merged PCAP and calls the Snort log parser on each of the individual PCAPs and on the merged PCAP. The alerts from PCAPs individually processed are stored in one log file ("alert.ids"), and the alerts from the merged PCAP are stored in a separate log file. These log files are then parsed and the results are reported back to the C2 server.

SURICATA is a command sent from the suricataConnect.py script to the C2

server and is used to record all transferred metrics from the Suricata machine - Suricata's number of alerts (SuA), Suricata's number of distinct alerts (SuT), and the number of packets Suricata captured (SuP). Upon receiving the REBOOT command from runExperiment.py, the C2 server reboots the Windows VM it is running on to reset everything to the initial state. When the RESET command is received from runExperiment.py, the C2 server transfers all metrics to the runExperiment.py script and then resets all metrics to zero.

Transfer Server   When the C2 server receives the PCAP command from the HoneyB Agent, the transfer server is launched as a child process and listens on the port specified by the HoneyB Agent. The transfer server then downloads the PCAP sent to it by the HoneyB Agent. After the download is complete, the transfer server notifies the C2 server through IPC, and then the transfer server shuts down. The transfer server's code is shown in Appendix A.

Snort Log Parser   The Snort log parser is invoked as a child process after the C2 server receives the SNORT command. The Snort log parser executes Snort on a PCAP specified by the C2 server. After Snort finishes processing the PCAP, the alert information in the log file ("alert.ids") is parsed and then reported back to the C2 server through IPC. The Snort log parser code is shown in Appendix A.

3.5.2   HoneyB Agent

Originally, each honeypot was modified to beacon back to the C2 server itself, but this caused the honeypots and Honeyd to slow down. Additionally, with this design the honeypots beaconed back to the C2 server with every request to the honeypot, instead of once for an entire session. This structure also did not afford a level of control that the HoneyB Agent script does, which is found in Appendix A.

The HoneyB Agent script is multithreaded, performing network packet capture with Scapy, periodically sending a heartbeat, alerts, and packet captures to the C2. The plugin maintains a list of honeypot IP addresses with ongoing network traffic. After a set amount of time without additional traffic to or from an IP address, it ages out; the address is removed from the connections list and its corresponding PCAP file is transferred to the C2 server. The age out time and time between heartbeats are user-defined parameters. In this experiment they are set to a 9000 seconds, a time that exceeded all but two outlier runs in this experiment. This was done so that each honeypot produced a single PCAP to make the test between Snort parsing the PCAP individually versus a merged PCAP fair.

The HoneyB Agent only sends an alert message for honeypots without tracked network activity. Examples of this are honeypots receiving traffic for the first time, or receiving traffic again after it previously aged out. There is a separate thread whose sole function is to monitor the honeypot connection list and detect and remove any IP addresses that have aged out. The traffic for that IP address is then transferred to the C2 server. If a honeypot receives traffic after the age out time, but before it is detected by the watcher thread, then this causes a transfer of the packet capture as well. The capture files are named with the format of "IP_Y-m-d_HM.pcap." IP is the honeypot's IP address, Y is the current year, m is the current month, d is the current day, H is the current hour, and M is the current minute.

After trial and error with other packet capturing software, Scapy was the clear winner. Scapy allowed capturing, analyzing, and running custom functions on packets received. Honeyd.py would have been the ideal solution since Honeyd already performs packet capture and would have also allowed for dynamic control of Honeyd. However, Honeyd on Ubuntu 12 was not compiled with the Python plugin, and re-compiling it generated errors.

The TCPDump and tshark programs supported capturing packets but did not allow processing and running functions on those packets. Although these programs could still have been used, the capture files would need to be manually parsed while still being modified. tshark did have a Python module called pyshark, but only the legacy version could be installed for Python 2.7, which did not work correctly and said to upgrade to the newer version.

The HoneyB Agent also had several commands added for the purpose of testing which include TRANSFER, RESET, KILL, START, STOP, and REBOOT. TRANS-FER signals the HoneyB Agent to transfer and then remove all PCAP files and to remove all tracked IP addresses from the connections list. RESET only removes all the IP addresses from the connection list. KILL terminates Honeyd and then the HoneyB Agent script. START launches Honeyd and registers the honeypots by authenticating to the C2 server. STOP kills only Honeyd. Finally, upon receiving the REBOOT command, the HoneyB Agent script reboots the Ubuntu VM that Honeyd and itself are running on. Originally, this was the chosen method of returning the machine to its initial state between each experimental run. However, Stafira's honeypots did not run correctly when Honeyd was launched on startup from rc.local.

3.5.3 Database Design

The DB schema is shown in Figure 15. The main table that connects all the other tables is the Alert table. Each alert has a unique ID, which serves as the primary key of this table, a timestamp, mandatory layer 3 information (source and destination IP addresses), and associated honeypot (HoneyID which is used as foreign key for the Honeypot table). Optional information includes layer 4 information (source and destination ports), and hashes for the PCAPs and memory dump files. The hashes are used as foreign keys to look up the entries in their respective tables and down-

load their Binary Large Objects (BLOBs). Each alert will only have one associated honeypot, PCAP, and memory dump. However, multiple commands, credentials, and binaries, can be found across multiple alerts and an alert can have multiple of them, creating a many-to-many relationship. To support this design, an intermediary table is used to break the table into a one-to-many relationship for both sides. AlertID is used to lookup all the associated commands, credentials, and binaries for a single alert, whereas the SHA256 is used to lookup all the alerts a specific signature is found in. Additionally, the SHA256 can be used to look up the plain-text command, credentials, and binary BLOB. Note that these tables have MD5 hashes as well to support looking up and distributing signatures generated in this format. The final table is the Honeypot table, which stores information of honeypots that have successfully authenticated. It includes a unique ID (HoneyID), their host name, MAC and IP addresses, OS, and an optional description detailing the type of honeypot and other relevant information.
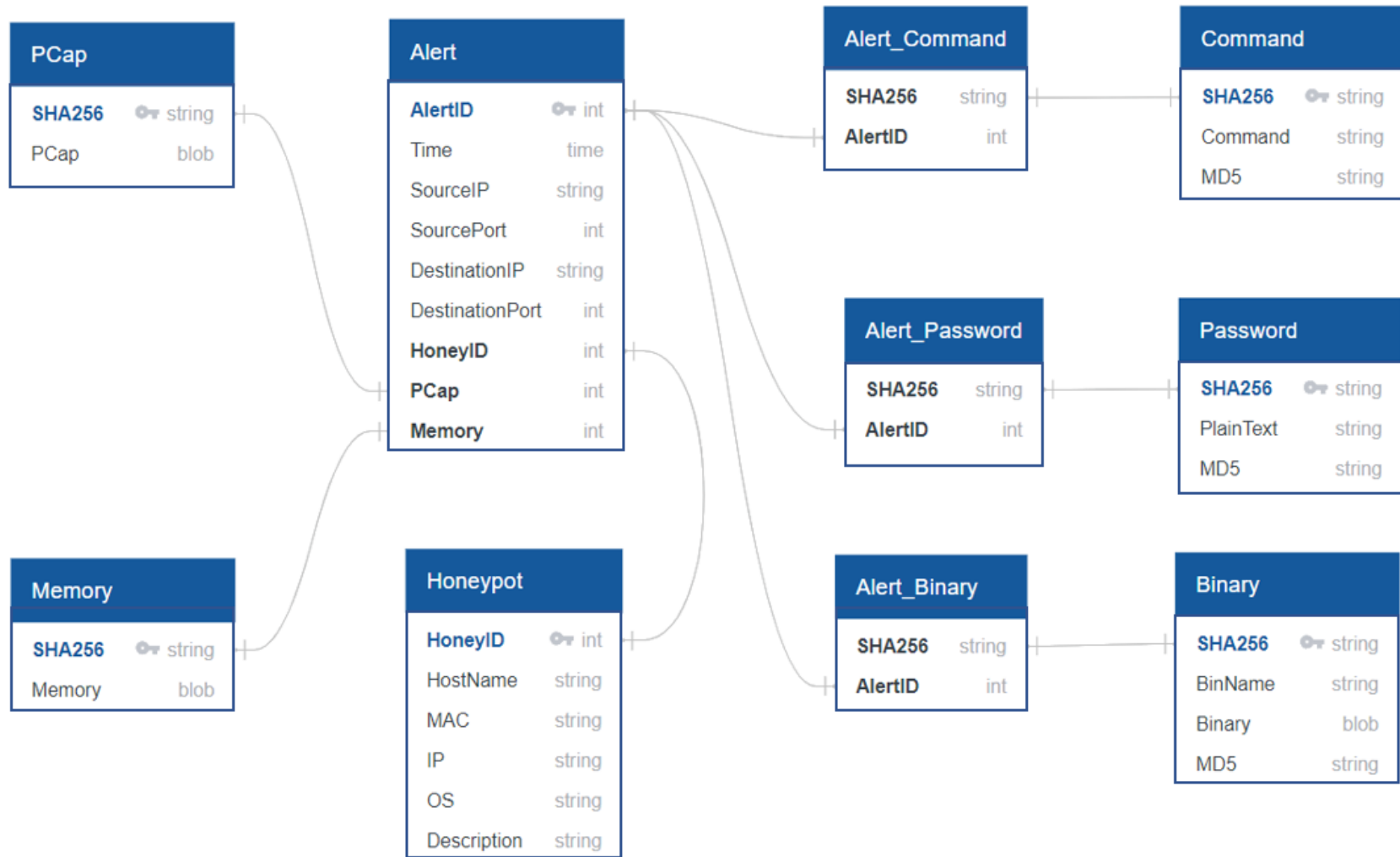
Figure 15. Database Schema

3.5.4 Network Design

Figure 16 shows the overall network design with devices on separate networked hosts to ensure all traffic is not seen by all hosts, which is unrealistic. Each Ubuntu VM housing Honeyd, Stafira's honeypots, and the HoneyB Agent script reside on separate physical machines from that of the attacker. Each Honeyd daemon runs all three of Stafira's honeypots, with only the IP addresses and MAC addresses modified in the Honeyd configuration file and the HoneyB Agent script. Stafira's modified Honeyd configuration file can be found in Appendix B. The IP address scheme for the Honey devices is:

1. TitaThink Camera honeypot 192.168.1.1[5-7]0

2. Prolophix Thermostat honeypot 192.168.1.1[5-7]1

3. ezOutlet2 Power Outlet honeypot 192.168.1.1[5-7]2

4. Ubuntu VM / HoneyB Agent 192.168.1.1[5-7]4

Because of the constant reboots from running the experiment, the C2 server was moved to a Windows 10 VM with an IP address of 192.168.1.233. The Desktop Windows 10 machine, addressed with 192.168.1.235, runs this Windows 10 VM and one of the Ubuntu VMs. The attacker machine (Desktop Ubuntu Machine), was moved from inside a VM to running natively on a physical machine and has an IP address of 192.168.1.230.
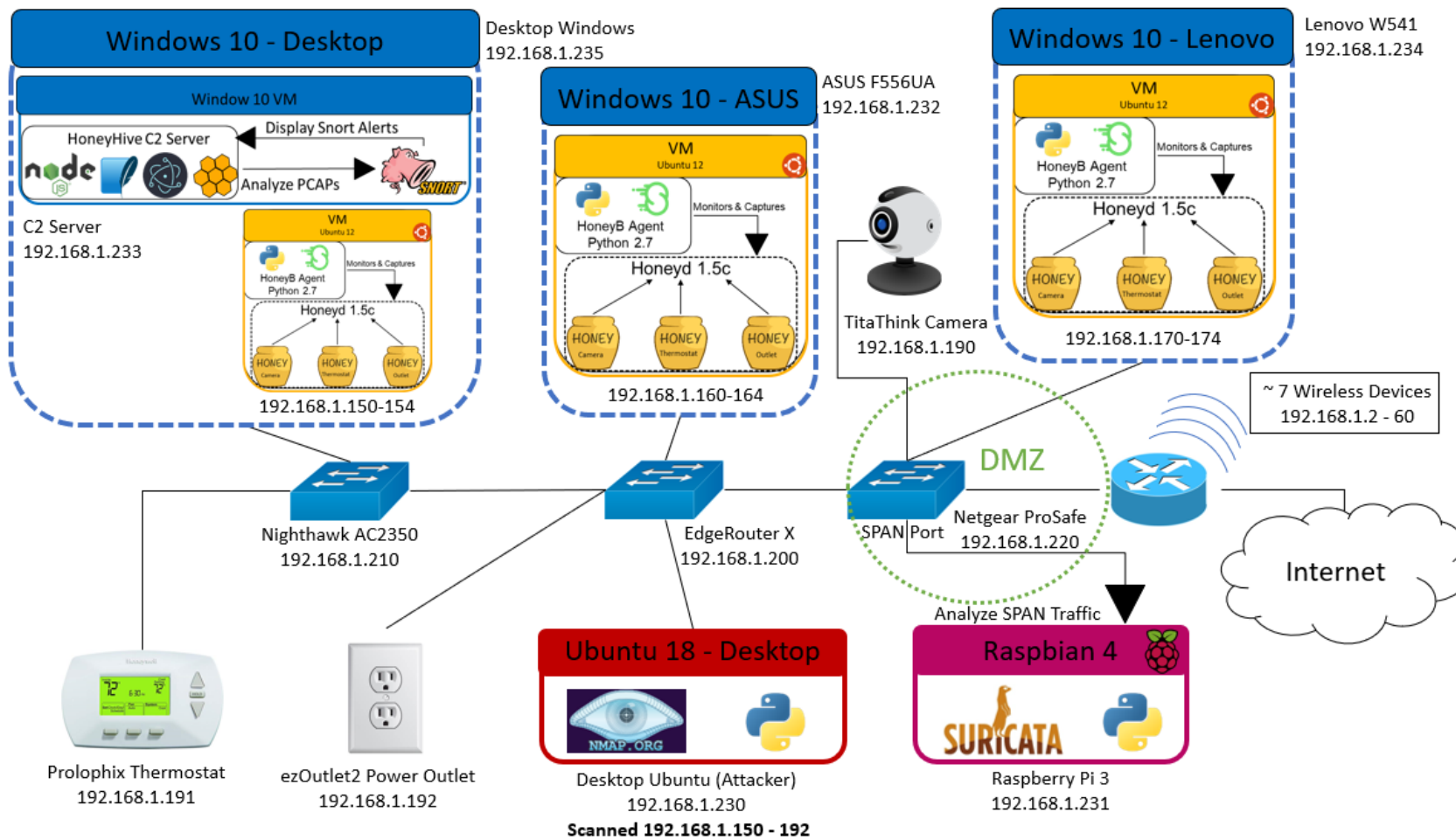
Figure 16. Simulated Test Network - Network Layout

With physical separation of test components complete, the network was designed and developed. Each set of honeypots was placed on separate switches to distribute them throughout the simulated test network. In addition, the IoT devices were spread out over all switches. The IoT devices match their honeypot counterparts IP addressing scheme in the last digit. The TitaThink Camera is 192.168.1.190, the Prolophix Thermostat is 192.168.1.191, and the ezOutlet2 Power Outlet is 192.168.1.192. A NIDS (Suricata), running on the Raspberry Pi with the IP address of 192.168.1.231, was placed in the DMZ on the Netgear ProSafe switch's SPAN port to match their traditional network deployment location [9]. This allowed it to receive a copy of all traffic that traverses the switch. Additionally, the Raspberry Pi was retrofitted with the suricataConnect.py script in order for it receive commands and report results in the experiment. This script is found in Appendix C.

To simulate a real world attack like that of the cyber attack against Ukraine in 2016, the attacker machine was placed in the internal network on the EdgeRouter X switch [7]. The assumption is that the attacker gained internal access through a phishing attack without tripping the NIDS (Suricata) and without alerting network administrators. They are now using the compromised box as a pivot to scan the rest of the internal network. The attacker is using an encrypted channel (SSH) to communicate with the compromised box. Through preliminary scanning and reconnaissance, the attacker has narrowed down their target list to the IP addresses 192.168.1.150-192.168.192 and is now conducting further scanning prior to launching exploits. With this network design, the simulated test network is representative to that of real-world scenarios.

3.6 Summary

In this chapter, the design and justification of decisions for the HoneyHive framework are explained as well as the motivation behind its development. Design decisions explored include the third-party software used, the programming languages selected for the framework's development, the C2 server, the HoneyB Agent script, the DB, and the simulated test network design.

IV. Research Methodology

4.1 Goals

In this research the HoneyHive framework's effectiveness of network intrusion detection is tested and compared against that of traditional NIDS (Snort and Suricata). These tests are conducted using Stafira's IoT honeypots, their physical device counterparts, the HoneyHive framework, Snort, and Suricata. The results from this experiment address the research hypotheses:

1. The HoneyHive framework operates correctly by not alerting on routine network traffic and alerting on non-routine network traffic.

2. The HoneyHive framework detects intrusions that traditional NIDS's cannot through the use of distributed IoT honeypot sensors and packet capture aggregation.

4.2 Approach

In this experiment, the total number of alerts as well as the number of distinct alert types is compared across three categories in order to determine network intrusion detection effectiveness: PCAPs captured by the HoneyB Agent are parsed by Snort individually, PCAPS captured by the HoneyB Agent merged into a single PCAP and parsed by Snort, and Suricata running on a SPAN port. Additionally the number of packets captured to and from the attacking machine by the HoneyHive framework and Suricata are also compared. These metrics are further explained in Sections 4.3 and 4.4.

Different types of scans are conducted against the internal network with varying levels of active honeypots. The type of scans include no scan (control group), TCP

64

Connect scan to simulate an attacker attempting to connect to devices, an Aggressive scan which portrays an attacker attempting to gather as much device information as possible, and a NIDS Avoidance scan to emulate an attacker who wishes to remain undetected on the network while gathering information. Scans are conducted using Nmap, and PCAPs from the scans are created using Scapy. These PCAPs are then ultimately transferred to the C2 server and analyzed by Snort. The number of Snort alerts and types generated by the captured traffic (both individually and merged) and total packet count are then compared to the number of Suricata alerts and types and total packet count.

After initial setup of the physical devices and network, the HoneyB Agent Python script is started on each of the Ubuntu VMs. After the HoneyB Agent scripts are launched, the experiment is conducted by launching the runExperiment.py script on Desktop Ubuntu (Attacker's machine). This script handles running all scan types and honeypot test combinations, device orchestration (starting HoneyB Agents, transferring PCAPs and metrics, analyzing PCAPs via Snort, and resetting devices / HoneyB Agents) between each run, and recording results for later analysis. The runExperiment.py is located in Appendix D.

This experiment simulates an attacker compromising an internal device, unbeknownst to network administrators and the NIDS. The attacker then uses an encrypted channel to the compromised device and uses it as a pivot point to conduct an internal network scan. This is similar to the 2016 cyber attack against Ukraine mentioned in Chapter 1 and Chapter 3 [7].

4.3   System Boundaries

In Figure 17, the System Under Test (SUT), the HoneyHive framework, is shown compromised of the HoneyB Agent, C2 server, and Snort. The specific part under

test, the Component Under Test (CUT), is the C2 server. The C2 server aggregates all captured traffic from the HoneyB Agents, run PCAPs through Snort, and displays alerts.
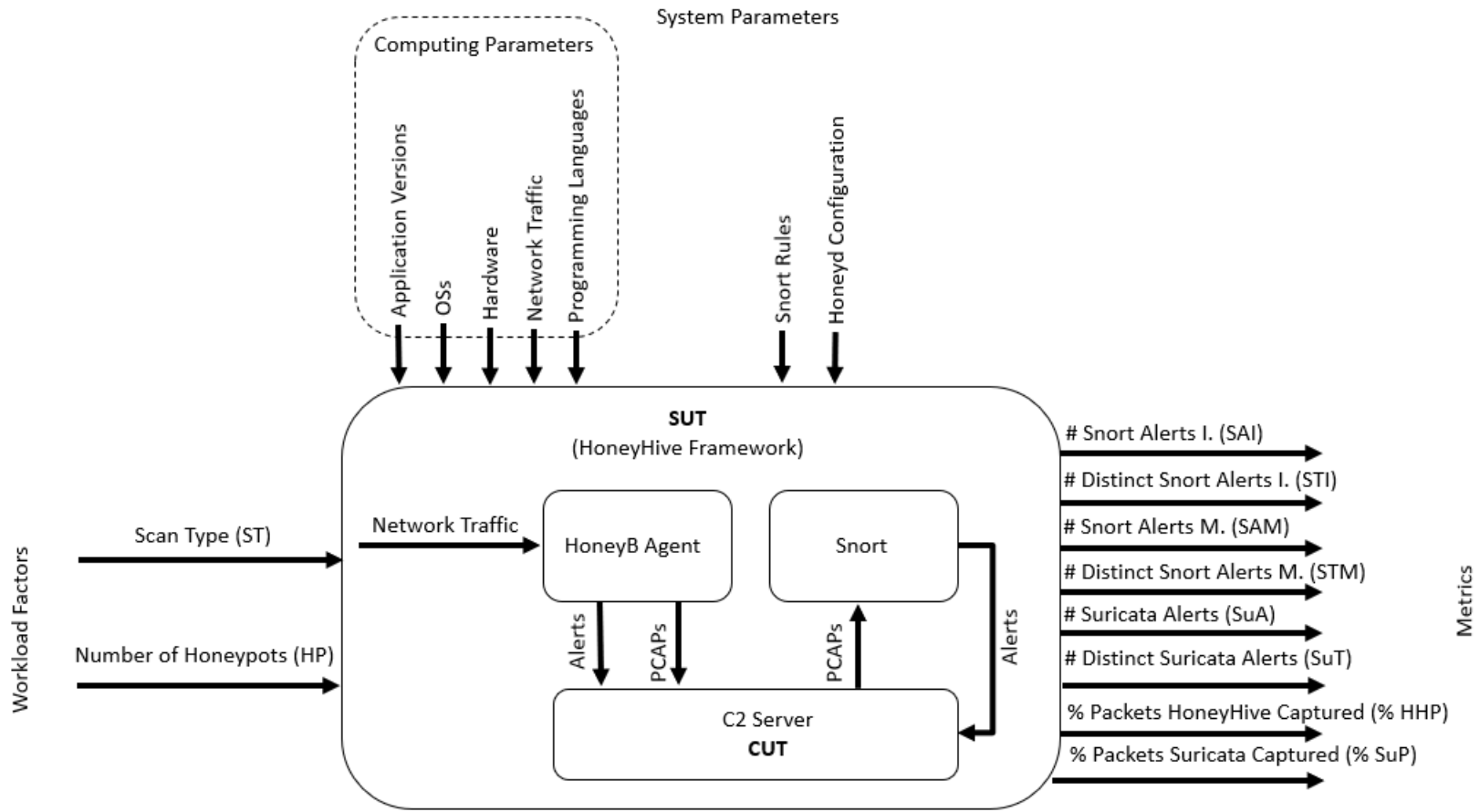
Figure 17. HoneyHive Framework

4.4   Parameters, Factors, and Metrics

4.4.1   Assumptions

During this experiment, several assumptions are made which are as follows:

1. The attacker has already compromised a machine in the internal network and through an encrypted channel, is using it as a pivot to scan the internal network

2. The attacker has performed rudimentary investigation already to narrow down the IP range to those scanned in the experiment

3. The honeypots are convincing enough for the attacker to perform further investigation, i.e., scanning

4.4.2   System Parameters

This section describes all the system parameters used to conduct the experiment. This includes the computing parameters, the programs and languages with their respective versions, and the configurations Honeyd, Snort, and Suricata.

Computing Parameters   The following is a list of physical devices and their hardware specifications used in this experiment:

- **Desktop Windows 10 64 bit**

    - ASUS ATX DDR4 LGA 1151 Motherboard Z170-E

    - Intel Core i7 6700K 4.00 GHz Unlocked Quad Core Skylake Desktop Processor, Socket LGA 1151 [BX80662I76700K]

    - 4 X Crucial Ballistix Sport LT 2400 MHz DDR4 DRAM Desktop Gaming Memory Single 16 GB CL16 BLS16G4D240FSE (Red) - Total 64 GB

- ASUS GeForce GTX 1080 8 GB Turbo Graphic Card TURBO-GTX1080-8G

- Samsung 850 EVO 500 GB 2.5-Inch SATA III Internal SSD (MZ-75E500B/AM)

- Seagate Barracuda ST2000DM001 2 TB 3.5 Internal Hard Drive (ST2000DM008)

- Seagate BarraCuda 4 TB Internal Hard Drive HDD – 3.5 Inch Sata 6 Gb/s 5400 RPM 256 MB Cache for Computer Desktop PC Laptop (ST4000DM004)

- Cooler Master GeminII S524 Version 2 CPU Air Cooler with 5 Direct Contact Heat Pipes (RR-G5V2-20PK-R1)

- NZXT Phantom 410 Mid Tower Computer Case , White/Blue (CA-PH410-W2)

- Corsair CX Series 750 Watt 80 Plus Bronze Certified Modular Power Supply

- **Desktop Ubuntu 18.04.2 64 bit**

  - ASUS ROG Crosshair V Formula-Z AM3+ AMD 990FX + SB950 SATA 6 Gb/s USB 3.0 ATX AMD Gaming Motherboard with 3-Way SLI/Cross-FireX Support and UEFI BIOS

  - AMD FX-8350 Black Edition Vishera 8-Core 4.0 GHz (4.2 GHz Turbo) Socket AM3+ 125W FD8350FRHKBOX Desktop Processor

  - Crucial BX500 240 GB 3D NAND SATA 2.5-Inch Internal SSD - CT240BX500SSD1Z

  - G.SKILL Ripjaws X Series 16 GB (4 x 4 GB) 240-Pin DDR3 SDRAM DDR3 1866 (PC3 14900) Desktop Memory Model F3-14900CL9Q-16GBXL

  - XFX Radeon RS RX 480 DirectX 12 RX-480P836BM 8 GB 256-Bit GDDR5 PCI Express 3.0 CrossFireX Support Video Card

- CORSAIR HX Series HX850 850W ATX12V 2.3 / EPS12V 2.91 SLI Ready CrossFire Ready 80 PLUS GOLD Certified Modular Active PFC Power Supply New 4th Gen CPU Certified Haswell Ready

- Cooler Master Hyper 212 Plus - CPU Cooler with 4 Direct Contact Heat-pipes

- Cooler Master HAF 922 - High Air Flow Mid Tower Computer Case with USB 3.0 and All-Black Interior

- **Lenovo Laptop Windows 10 64 bit**- Lenovo Thinkpad W541 Intel i7-4910MQ processor 2.9 GHz running Windows 10 and 32 GB of RAM. HITACHI HTS725050A7E635 Travelstar Z7K500 Opal 500 GB Hard drive

- **ASUS Laptop Windows 10 64 bit**- ASUS VivoBook F556UA-AB32 Laptop (Windows 10, Intel i3-6100U 2.3 GHz, 15.6" LED-Lit Screen, Storage: 1000 GB, RAM: 4 GB) Black/Silver *** Upgraded to 8 GB memory (added second Crucial 4 GB Single DDR4 2133 MT/s (PC4-17000) SR x8 SODIMM 260-Pin Laptop Memory - CT4G4SFS8213) and replaced HDD with Samsung 850 EVO 500 GB 2.5-Inch SATA III Internal SSD (MZ-75E500B/AM)**

- **Raspberry Pi 3 Model B+ Rev 1.3** - Raspbian 4.19.75-v7. Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz, 1 GB LPDDR2 SDRAM, 16 GB Class 4 SanDisk Edge microSDHC memory card, running Suricata Version 4.1.2

- **R7500-200NAS Nighthawk AC2350 4X4 MU-MIMO Dual Band WiFi Gigabit Router** - Router Firmware Version V1.0.3.16

- **Netgear ProSafe GS108Ev2 Switch** - Firmaware version v1.00.12

- **Ubiquiti EdgeRouter X Model ER-X** - Firmware version v1.9.0

- **TitaThink Camera TT520PW** - Firmware Version 6.10

- **Prolophix Thermostat NT130h** - SW 3.0.3 / HW H.03

- **ezOutlet2 Power Outlet EZ-22b** - Version EZT.8824 (04)

Virtual Machines Two virtual machine images are used in the experiment. The Windows 10 VM is used as the C2 server, and the Ubuntu VM is used to run Stafira's HTTP IoT Honeyd Framework and the HoneyB Agent script. The Ubuntu VM is cloned and then distributed to the Windows Desktop, Lenovo Laptop, and ASUS Laptop in order add physical separation and network distribution between honeypots. Only IP and MAC addresses are modified in settings, scripts, and configs for the cloned VMs. The VMware specifications and versions of installed software are as follows:

- **Ubuntu 12.04 VMs (Honeyd Machines)** - 2 processors, 4.3 GB memory, 20 GB disk space. Honeyd 1.5c installed, bash version 4.2.25, Python version 2.7.3

- **Windows 10 64 bit VM (C2 Server)** - 4 logical processors, 8 GB memory, 40 GB disk space. Git, TortoiseGit, Node.js Version v12.13.0, V8 Engine Version 7.7.299.13-node.12, Snort Version 2.9.15-WIN32 GRE (Build 7), and Wireshark Version 3.0.6 are all installed

Program and Language Versions This section documents the combined programs and programming languages across all devices:

- VMware Workstation 15 Pro Version 15.5.0

- Node.js Version v12.13.0

- V8 Engine Version 7.7.299.13-node.12

- Python 2.7.15+ on Ubuntu Desktop, Python 2.7.3 on Ubuntu VMs, Python 2.7.16 on Raspberry Pi

- Bash 4.4.20 on Ubuntu Desktop, 4.2.25 on Ubuntu VMs, and 5.0.3 on Raspberry Pi

- Honeyd 1.5c

- Nmap version 7.60

- Wireshark Windows Version 3.0.6

- Snort Version 2.9.15-WIN32 GRE (Build 7)

- Suricata Version 4.1.2

Honeyd Configuration The configuration file from Stafira's HTTP IoT Honeyd framework is used as the configuration file for Honeyd 1.5c in this experiment and is shown in Appendix B. Only IP and MAC addresses are modified in this configuration file.

Snort and Suricata Configuration Snort Version 2.9.15-WIN32 GRE (Build 7) is installed on the C2 server and the default snort.conf file is used with some modifications in order to run it on windows with only the emerging-scan.rules rule set. Additionally, Suricata Version 4.1.2 is installed on the Raspberry Pi and the default configuration file, suricata.yaml, is modified to only use the emerging-scan.rules rule set. The snort.conf, suricat.yaml, and emerging-scan.rules are developed and vetted by the Snort community and available for download on their website

[71]. Suricata was built to use the same rule syntax and can successfully implement these community rules as well.

4.4.3   Factors

The following factors were used in this experiment:

- **Scan Type (ST)** - The type of scan performed in the test. These include the control group (no scan), TCP Connect scan, Aggressive scan (OS detection, version detection, script scanning, and traceroute), and NIDS Avoidance scan (scan delay, host randomization, and packet fragmentation). The Nmap parameters of these commands are:

  1. No Scan (Control Group) - Wait 1321 seconds

  2. TCP Connect scan - "sudo nmap -iL ipLst.txt -sT -Pn -oX runNum.xml"

  3. Aggressive scan - "sudo nmap -iL ipLst.txt -A -Pn -oX runNum.xml"

  4. NIDS Avoidance scan - "sudo nmap -iL ipLst.txt –scan-delay 1075ms –randomize-hosts -f 8 -Pn -oX runNum.xml"

- **Number of Honeypots (HP)** - The total number of honeypots spun up in the test. Honeypots are launched in multiples of three with each of the following types: TitaThink Camera, Prolophix Thermostat, and ez-Outlet2 Power Outlet. The level of honeypots are 0, 3, 6, and 9 honeypots.

Factors with their corresponding levels are shown in Table 2.

Factors	Levels			
Scan Type (ST)	No scan (Control Group)	TCP Connect	Aggressive	NIDS Avoidance
Honeypots (HP)	0	3	6	9

Table 2. Factors and Levels

73

4.4.4 Metrics

The metrics from the experiment include:

- **Number of Snort Alerts Individually (SAI)** - Numeric value representing the total number of Snort alerts generated by parsing each PCAP file transferred to the C2 server, individually, and resetting Snort after each run

- **Number of Distinct Types of Snort Alerts Individually (STI)** - Numeric value representing the number of distinct types of Snort rules triggered by parsing each PCAP file transferred to the C2 server, individually, and resetting Snort after each PCAP is parsed

- **Number of Snort Alerts Merged (SAM)** - Numeric value representing the total number of Snort alerts generated by parsing each PCAP file transferred to the C2 server as a merged PCAP file

- **Number of Distinct Types of Snort Alerts Merged (STM)** - Numeric value representing the number of distinct types of Snort rules triggered by parsing each PCAP file transferred to the C2 server as a merged PCAP file

- **Number of Suricata Alerts (SuA)** - Numeric value representing the total number Suricata alerts generated receiving from SPAN traffic

- **Number of Distinct Types of Suricata Alerts (SuT)** - Numeric value representing the number of distinct types of Suricata rules triggered from SPAN traffic

- **Percentage of Packets HoneyHive Captured (% HHP)** - The percentage of packets the HoneyHive framework captured to and from the attacker, measured by the combined total packet count of all transferred PCAPs to the C2

74

server and then divided by the number of received and transferred packets on
the attacker's machine

- **Percentage of Packets Suricata Captured (% SuP)** - The percentage
  of packets the Raspberry Pi captured to and from the attacker, measured by
  counting the number of packets sent and received to and from the attacker,
  divided by the number of received and transferred packets on the attacker's
  machine

The number of HoneyHive Interactions **(HHI)** is a numeric value that represents
the number of times HoneyB Agents collectively reported an interaction with one of
their monitored honeypots. Each honeypot should only create one interaction per
test in this experiment for a total of three interactions per HoneyB Agent. For this
reason, it was not used in statistical analysis against Snort and Suricata but is instead
a verification that the honeypots and HoneyHive framework are operating correctly.
Additionally, the attacker's total number of packets sent and received during each test
(AP) and the Elapsed Time of the test in seconds **(ET)** are useful for understanding
each trial, but not used in analysis. Metrics are bolded in Appendix E.

4.5 Methodology

The procedural steps taken to run the experiment are as follows:

1. Ensure all devices are connected to the network with the correct static IP and
   that the network setup matches the network diagram.

2. Ensure that all program code/scripts are up to date and current with the current
   Git repository. Perform a Git pull for scripts that are not.

3. Ensure the C2 server launches on startup - Windows+R and then type shell:startup.

4. Ensure the suricataConnect.py script launches on startup in rc.local.

5. Ensure no remnants remain from previous experimental runs.

   (a) Remove experiment.txt and all .csv and .xml files from the attacker machine.

   (b) Remove /var/log/suricata/fast.log from the Suricata machine.

6. Ensure emerging-scan.rules is located in "C:\Snort\rules" and that it is the only active rule in the snort.conf file "C:\Snort\etc\snort.conf".

7. Ensure emerging-scan.rules is located in "/etc/suricata/rules" and that it is the only active rule in the suricata.yamnl file "/etc/suricata/suricata.yaml".

8. Reboot all physical machines and virtual machines.

9. On each Honeyd VM, cd to honeyhive/scripts and then run "sudo python connectToServer.py".

10. Ensure the Windows 10 VM is up and the C2 server is running.

11. On the attacker machine, run "sudo python runExperiment.py".

12. Wait for experiment to complete (about 10 to 12 days).

13. Download nmap.csv file and perform analysis on results.

4.5.1 runExperiment.py

This section describes the functions the runExperiment.py executes.

1. Check if experiment.txt exists and is not empty.

   (a) Read in each line as a separate test run (Test type, Number of Honeyd Machines, and Test Type Level) and append it to an array for execution.

76

2. If experiment.txt does not exist.

   (a) Create an array with all combinations of user-specified test types (nmap-Scan), number of Honeyd machines (0-3), and Test Type Level (0-3) for each user-specif iced test. Each Honeyd machine controls 3 honeypots, creating the honeypot levels of 0, 3, 6, and 9. This is all then multiplied by the number of user-specified replicants (30).

   The nmapScan Test Type Levels:

      i. Level 0 - No Scan (Control Group): Wait 1321 seconds for routine network traffic (median amount of time required by the median nmapScan test in pilot studies). The mean was 1341 seconds.

      ii. Level 1 - Nmap TCP Connect scan ("sudo nmap -iL ipLst.txt -sT -Pn -oX runNum.xml").

      iii. Level 2 - Nmap TCP Syn scan with OS detection, version detection, script scanning, and traceroute. (Aggressive scan) ("sudo nmap -iL ipLst.txt -A -Pn -oX runNum.xml").

      iv. Level 3 - Nmap TCP Syn scan with scan delay, host randomization, and packet fragmentation (NIDS Avoidance scan) ("sudo nmap -iL ipLst.txt –scan-delay 1075ms –randomize-hosts -f 8 -Pn -oX runNum.xml"). These parameters were used because Nmap's website listed them as ways to avoid a NIDS [38].

   (b) Randomize the order of the array.

   (c) Create experiment.txt and write contents of the recently created array to the file.

3. Execute all test runs in the array (loop).

(a) Send UDP "START" command to corresponding number of Honeyd machines and to Suricata over command and control port 9830, three times. The following devices are sent "START" based on the number of Honeyd Machines:

   i. 0 - no Honeyd machines are started.

   ii. 1 - 192.168.1.150-154 Honeyd machines are started.

   iii. 2 - 192.168.1.150-154 and 192.168.1.170-174 Honeyd machines are started.

   iv. 3 - 192.168.1.150-154, 192.168.1.160-164, and 192.168.1.170-174 Honeyd machines are started.

Suricata is always started.

(b) Wait 40 seconds for the Honeyd and Suricata programs to finish starting (based on pilot-studies, Honeyd takes 10-15 seconds and Suricata takes 30 seconds).

(c) Record the transmitted (tx_packets) and received packet (rx_packets) count and start time of the attacker machine before executing the test.

(d) Execute the next test run in the execution array and wait for completion.

(e) Record the the transmitted (tx_packets) and received packet (rx_packets) count for the attacker's interface and end time after executing the test. Subtract the end counts from the start counts to get the total packet counts sent and the elapsed time of the test.

(f) Send UDP "RESET" command to all Honeyd Machines and Suricata over command and control port 9830, three times.

(g) Wait 45 seconds for all the PCAPs and Suricata results to be transferred to the C2 server (based on pilot-studies, sending all PCAPs takes 20-30 seconds and sending Suricata results take 3-5.

78

(h) Send the "SNORT" command to the C2 server over TCP port 9830 to signal it to parse all received PCAPs.

(i) Wait 90 seconds for Snort to finish (Snort takes 30-50 seconds to complete, based on pilot studies).

(j) Send the "RESET" command to the C2 server over TCP port 9830 to signal for the C2 Server to send results over the socket and then clear all data. runExperiment.py waits until the results are received.

(k) Write the results received from the C2 server, attacker machine packet count and experiment elapsed time to the corresponding test result file in csv format (nmap.csv).

(l) Remove completed experiment from the experiment.txt file (first line).

(m) Send UDP "REBOOT" command to all Honeyd Machines and Suricata over command and control port 9830, three times (Honeyd Machines set to ignore command).

(n) Send the "REBOOT" command to the C2 server over TCP port 9830 to signal it to reboot the machine.

(o) Wait 45 seconds for all machines to finish rebooting (Windows takes 20-30 seconds, Ubuntu 15-20 seconds, Raspberry Pi 20-25 seconds).

(p) Ensure the C2 Server has booted by continuously trying to connect to it until successful.

4.6   Results

The results from this experiment are saved in Comma-Separated Values (CSV) format in the file nmap.csv. This file / data is imported into Excel and used to create bar graphs and tables to show the differences between the HoneyHive framework

and that of a traditional NIDS (Suricata). Additionally, Anderson-Darling tests for normality are run in Excel [72]. Finally, using the permutation_test.py script in Appendix F, Permutation tests are run for SAI versus SAM, SAI / SAM versus SuA, STI versus STM, STM versus SuT, and % HHP versus % SuP.

The Anderson-Darling test is used to determine how well a set of data fits a specified distribution. In the analysis of results for this experiment this is used to test whether the given data is normal. This is done across all combinations of factors and is important to do because some statistical tests, such as t-tests, assume normal distribution. Suffice it to say that if the p-value from the Anderson-Darling test is less than 0.05 then the data does not follow a normal distribution.

Permutation tests do not require a normal distribution to perform statistical analysis and are used to determine what the probability is of obtaining results that exceed those in two competing data sets, given their initial means and difference. To perform a Permutation test, first the arithmetic mean for each of the two data sets is computed and then the difference between their arithmetic means is recorded as the "test statistic". After that, the data from both sets is combined and shuffled into two new random subsets. The difference between the newly formed sets' means is calculated and compared to the test statistic. This is repeated 900,000 times in this experiment, and then number of permutations whose absolute value exceeds that of the test statistic is divided by the total number of permutations performed to calculate the p-value. If this p-value is less than the chosen significance value, 10% in this experiment, then there is a statistical significance between the two data sets [73] [74] [75]. The code developed to perform Permutation tests on this experiment's results is found in Appendix F.

4.7 Chapter Summary

This chapter outlined the research goals and questions this experiment seeks to answer. The experiment itself was broken down into the SUT, all system parameters, factors, and metrics, the step by step procedure of executing the experiment, and finally how the results of the experiment will be analyzed and presented.

V. Results and Analysis

5.1 Overview

This chapter provides an analysis of experiment results which can be found in Appendix E. Results are presented first with an overall picture and then grouped by each factor to distinguish patterns and the effect of each factor. Results are also grouped by all combinations of factors (scan type with number of honeypots). First tables and graphs present a comparison of each different metric. The Control Group Scan and 0 Honeypot levels are not included in the means of other factors as they create skewed results. This is followed by a table with the results of the Anderson-Darling test for normality with a significance level (α) of 5%. Both metrics that are being compared against one another are required to be normally distributed in order to perform a t-test. While there are a handful of individual metrics that accept the null hypothesis of normal distribution, only the Aggressive scan with 9 honeypots accepted the null hypothesis for SAI and SAM to be compared against one another.

Because only one test set out of eighty is normally distributed, a Permutation test is selected for testing whether results are statistically significant with a significance level of 10% . 900,000 permutations are performed for each test. For number of alerts, SAI and SAM are tested against one another as well as the SAI/SAM (whichever has the higher average mean) against SuA. It appears that Snort consolidates some alerts when PCAPs are analyzed together which is why SAI is often the more fair test against SuA. However, it appears that Snort is able to detect and create more alerts when analyzing packets together for the NIDS Avoidance Scans and when only 3 honeypots are active. When this happens, SAM is compared against SuA. For the number of distinct alerts, STI is compared against STM, and STM is compared against SuT. Snort created more distinct alerts on average when analyzing PCAPs

merged together which is why STM was used over STI. Finally, % HHP is compared against % SuP to compare the percentage of captured packets.

5.1.1   Number of Alerts Overview

In Table 3 and Figure 18, the mean number of alerts across all groupings and combination of factors is shown. Suricata generated more alerts in all factor groupings except for when 9 honeypots were active in an Aggressive scan. Additionally, in the TCP Connect scan Snort and Suricata created a similar amount of alerts when 9 honeypots were active. However, Suricata created a significant number more alerts on average in the NIDS Avoidance scan when 9 honeypots were active. Table 4 displays the Anderson-Darling test for normality results conducted on the experiment results. Only the Aggressive Scan with 9 honeypots was normal for both SAI and SAM. Additionally, SAI was normal for the TCP Connect scan with 6 honeypots. In Table 5 the Permutation test results are shown. In the SAI versus SUA column, cells italicized and denoted with a "*" indicate that SAM was used for comparison instead because it had a higher mean for alerts than SAI. SAI and SAM were not statistically significant in test. However, Snort versus SuA was significantly different in all tests except the TCP Connect Scan with 9 honeypots and 9 honeypots overall. This indicates that the null hypothesis (Snort and Suricata create the same number of alerts) is rejected. This suggests Suricata outperforms Snort in all alert tests where its mean is higher and statistically significant. This is true for all but the Aggressive Scan with 9 honeypots, where Snort outperforms Suricata and is statistically significant.

While HoneyHive created alerts in the majority of runs, there were 3/270 runs (runs 270, 291, and 465) that it did not create alerts (HHI) when it should have. Additionally, HoneyHive is currently using Snort for a higher level of signature matching and alert creation. However, Snort did not create alerts for 32/270 runs that it should

have. This means roughly 10% of intrusions did not have successful signature matching performed on packet captures. This is either from Snort crashing, not finishing in a timely manner, simply not creating alerts, or an error in the HoneyHive framework.

	HHI	SAI	SAM	SuA
Control Group	0	0	0	0
CG 0	0	0	0	0
CG 3	0	0	0	0
CG 6	0	0	0	0
CG 9	0	0	0	0
TCP Connect	5.933333333	24.14444444	22.07777778	33.95555556
TCP 0	0	0	0	24.2
TCP 3	2.8	12.86666667	14.03333333	23
TCP 6	6	23.53333333	22.5	39.2
TCP 9	9	36.03333333	29.7	39.66666667
Aggressive	5.255555556	82.68888889	77.14444444	112.4
Agg 0	0	0	0	112.5
Agg 3	2.833333333	40.56666667	42.63333333	103.5333333
Agg 6	5	74.06666667	71.83333333	126.5666667
Agg 9	7.933333333	133.4333333	116.9666667	107.1
NIDS Avoidance	5.966666667	22.92222222	27.45555556	49.36666667
NIDS 0	0	0	0	26.96666667
NIDS 3	3	13.86666667	17.23333333	28
NIDS 6	6	22.6	25.83333333	58.16666667
NIDS 9	8.9	32.3	39.3	61.93333333
0 Honeypots	0	0	0	54.55555556
3 Honeypots	2.877777778	22.43333333	24.63333333	51.51111111
6 Honeypots	5.666666667	40.06666667	40.05555556	74.64444444
9 Honeypots	8.611111111	67.25555556	61.98888889	69.56666667
Overall	5.718518519	43.25185185	42.22592593	65.24074074

Table 3. Overview - Mean Alerts by Level
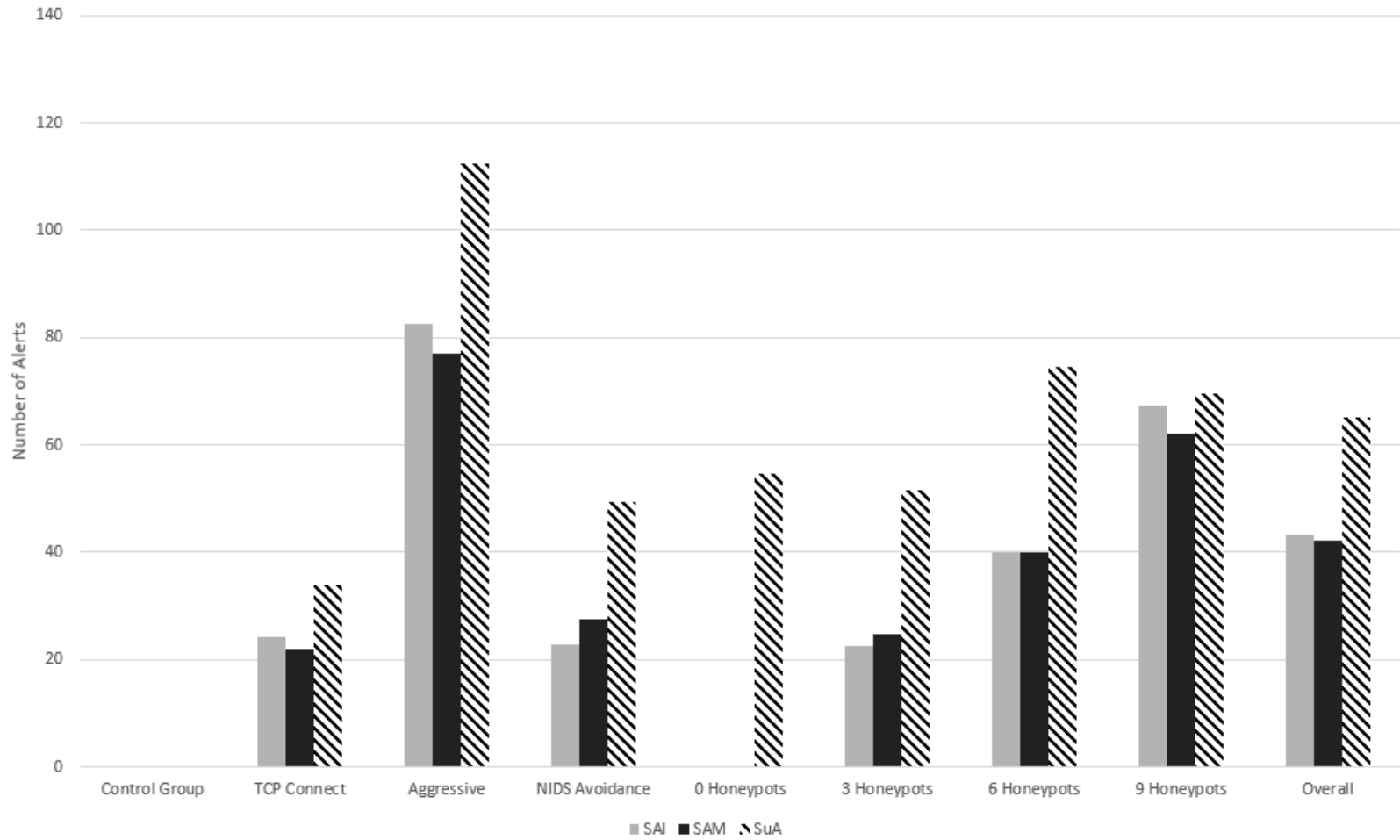
Figure 18. Overview - Mean Number of Alerts

	SAI	SAM	SuA
TCP Connect	8.4179E-10	2.10219E-06	8.45677E-05
TCP 3	1.86436E-09	1.80995E-09	0.001764216
TCP 6	0.07823086	0.018595137	3.75442E-05
TCP 9	0.003989696	0.04930214	0.001424823
Aggressive	0.00231741	0.009967393	4.53772E-10
Agg 3	0.001050125	0.000460888	4.02293E-10
Agg 6	0.01229756	0.045563122	0.000516952
Agg 9	0.118392405	0.136474624	0.002484829
NIDS Avoidance	0	7.24625E-25	4.65401E-19
NIDS 3	1.03809E-09	1.02987E-07	4.93212E-11
NIDS 6	9.41431E-15	4.06702E-13	3.23667E-12
NIDS 9	1.18093E-13	4.58503E-11	5.75098E-11
3 Honeypots	1.90455E-12	1.2306E-10	7.56855E-23
6 Honeypots	1.14242E-13	3.13753E-13	7.32928E-21
9 Honeypots	3.86714E-12	1.15597E-11	1.65326E-15
Overall	0	0	0

Table 4. Anderson-Darling Test - Number of Alerts

	SAI v. SAM	SAI v. SuA
TCP Connect	0.489953333	0.00048
TCP 3	0.74778	*0.0034967 *
TCP 6	0.762092222	0.0001189
TCP 9	0.339487778	0.49631
Aggressive	0.497806667	*0.0006667 *
Agg 3	0.76365	0
Agg 6	0.817308889	0.0016511
Agg 9	0.275418889	0.0702367
NIDS Avoidance	0.310293333	*0 *
NIDS 3	0.462257778	*0.0015556 *
NIDS 6	0.687624444	*3.33E-06 *
NIDS 9	0.453642222	*0.0007711 *
3 Honeypots	0.532037778	*0 *
6 Honeypots	0.996777778	3.33E-06
9 Honeypots	0.555283333	0.7710211
Overall	0.795945556	0

Table 5. Permutation Test - Number of Alerts

5.1.2 Number of Distinct Types of Alerts Overview

Table 6 and Figure 19 show the mean number of distinct alerts across all tests. Suricata created more distinct number of alerts (SuT) on average than Snort. PCAPs merged together and then analyzed by Snort (STM) created more distinct alerts on average as compared with each packet being parsed individually by Snort (STI). Furthermore, not only did the average number of distinct alerts grow for STM and

STI as the number of honeypots increased but the average difference in the number of distinct alerts between STM and STI grew as well. In addition, the difference between STM and SuT shrank. This supports that distributed data / PCAP aggregation exhibits merit for NIDS. Table 7 shows the results of the Anderson-Darling test. No data sets fit the normal distribution. Table 8 displays the results of Permutation tests for STI compared with STM and STM compared with SuT. STI versus STM was statistically significant in all tests except the TCP Connect Scan and NIDS Avoidance Scan each with 3 honeypots. These tests were both very close to 10% but did not meet the required significance level. Because STM created more distinct alerts on average for each test over STI this means that it performed better for all tests except the two aforementioned. This supports the hypothesis that the HoneyHive framework can detect intrusions that traditional NIDS cannot through distributed sensors and packet aggregation. However, Suricata created more distinct alerts on average than SAM in all categories and was statistically significant in all tests except the Aggressive Scan with 9 Honeypots. This indicates there was not an advantage of combining packets over that of just listening on a SPAN port.

	STI	STM	SuT
Control Group	0	0	0
CG 0	0	0	0
CG 3	0	0	0
CG 6	0	0	0
CG 9	0	0	0
TCP Connect	4.477777778	5.544444444	7.411111111
TCP 0	0	0	7.166666667
TCP 3	4.333333333	5.033333333	6.733333333
TCP 6	4.633333333	5.866666667	7.5
TCP 9	4.466666667	5.733333333	8
Aggressive	6.3	7.822222222	10.13333333
Agg 0	0	0	12
Agg 3	5.7	7.1	11.36666667
Agg 6	6.233333333	7.733333333	9.6
Agg 9	6.966666667	8.633333333	9.433333333
NIDS Avoidance	3.633333333	4.911111111	7.233333333
NIDS 0	0	0	6
NIDS 3	3.333333333	4.233333333	5.8
NIDS 6	3.7	4.466666667	7.966666667
NIDS 9	3.866666667	6.033333333	7.933333333
0 Honeypots	0	0	8.388888889
3 Honeypots	4.455555556	5.455555556	7.966666667
6 Honeypots	4.855555556	6.022222222	8.355555556
9 Honeypots	5.1	6.8	8.455555556
Overall	4.803703704	6.092592593	8.259259259

Table 6. Overview - Mean Number of Distinct Alerts by Level
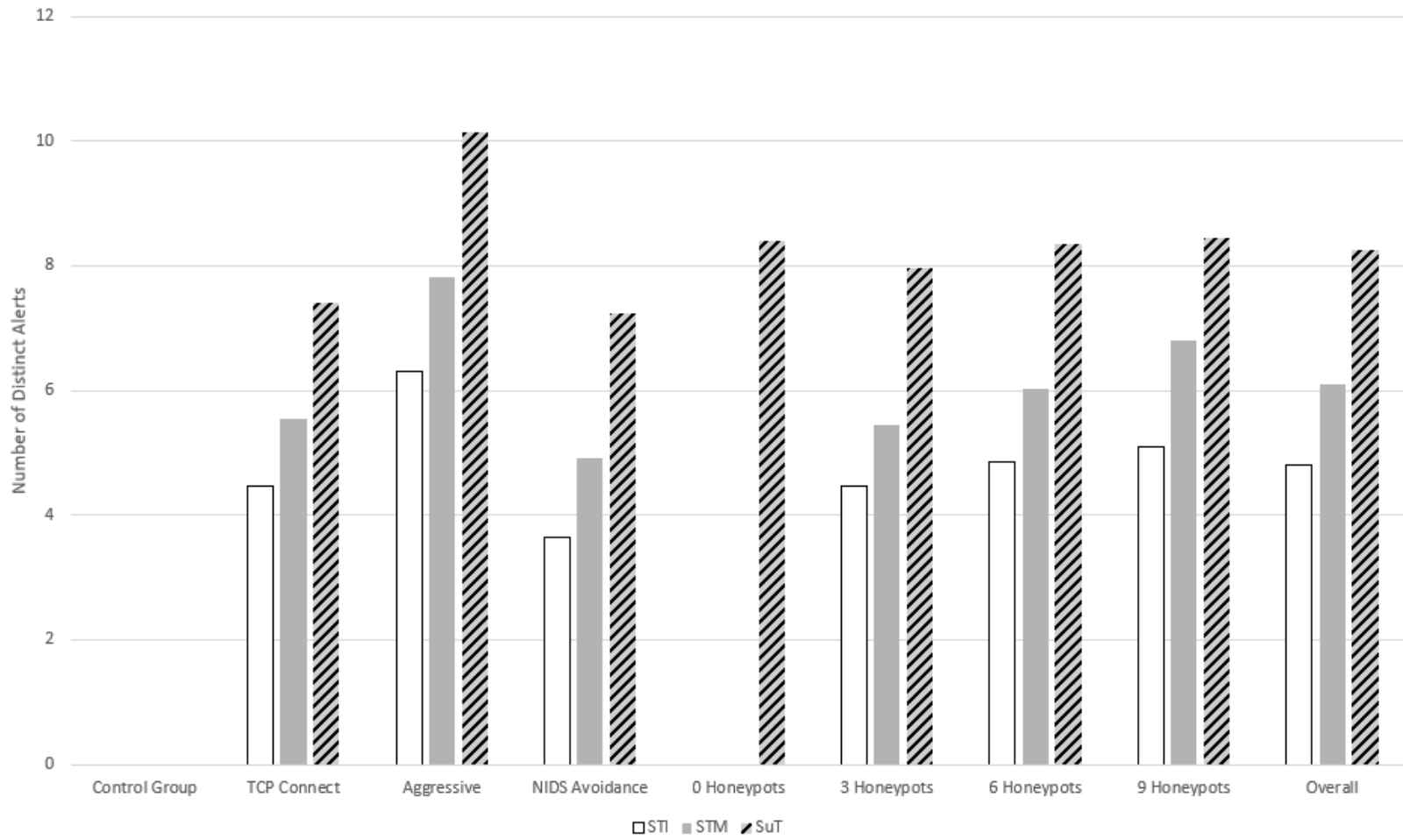
Figure 19. Overview - Mean Number of Distinct Alerts

	STI	STM	SuT
TCP Connect	1.42E-30	2.26E-18	0
TCP 3	2.26E-10	1.04E-06	3.82E-08
TCP 6	4.44E-12	2.19E-13	3.78E-16
TCP 9	1.47E-10	2.38E-08	3.96E-15
Aggressive	0	0	1.1E-12
Agg 3	9.08E-12	3.92E-14	5.21E-20
Agg 6	2.25E-14	2.56E-16	0.005961
Agg 9	4.67E-08	6.4E-13	7.33E-05
NIDS Avoidance	9.44E-29	1.93E-20	0
NIDS 3	6.37E-09	2.95E-10	2.66E-08
NIDS 6	1.61E-08	9.65E-12	8.99E-24
NIDS 9	4.02E-13	1.5E-13	2.28E-27
3 Honeypots	2.01E-10	4.23E-11	2.33E-09
6 Honeypots	2.16E-10	1.93E-10	1.72E-20
9 Honeypots	1.34E-08	7.63E-15	4.72E-22
Overall	3.26E-27	1.58E-28	0

Table 7. Anderson-Darling Test - Number of Distinct Alerts

	STI v. STM	STM v. SuT
TCP Connect	0.00062	0
TCP 3	0.1014511	0.0011956
TCP 6	0.0273578	0.0015211
TCP 9	0.0310778	0
Aggressive	0.0003433	0
Agg 3	0.0726156	0
Agg 6	0.0417956	0.0032133
Agg 9	0.0042444	0.1583944
NIDS Avoidance	0	0
NIDS 3	0.1012789	0.0982622
NIDS 6	0.0897022	0
NIDS 9	1.22E-05	0
3 Honeypots	0.0084811	0
6 Honeypots	0.0026778	0
9 Honeypots	1.00E-05	0
Overall	0	0

Table 8. Permutation Test - Number of Distinct Alerts

5.1.3 Packet Capture Percentage Overview

The percentages of scanned devices that are monitored for each level of honeypots in the test network for the HoneyHive framework and Suricata, are shown in Table 9. As the number of honeypots increases, the total number of active devices on the network that are scanned also increases. HoneyHive's percentage of monitored devices increses as the number of honeypots increases. However, the percentage of monitored

93

devices for Suricata decreases from 0 to 3 honeypots, increase from 3 to 6 honeypots, and then decreases from 6 to 9 honeypots. This is because the honeypots activated for 6 honeypots reside on Suricata's switch (the DMZ switch).

The percentage of packets captured by the HoneyHive framework versus Suricata is shown in Table 10 and Figure 20. Overall, HoneyHive and Suricata captured roughly the same average percentage of packets. However, when 9 honeypots are active the HoneyHive framework significantly outperforms Suricata in all categories except the NIDS Scan. At the level of three honeypots, packets captured are within several percentage points of another which is to be expected as Suricata does not have active honeypots on its switch. Similarly, at 6 honeypots HoneyHive and Suricata stayed within a couple percentage points of one another since they both increase by three honeypots. The one category this does not hold for is the NIDS Avoidance scan in which Suricata captures significantly more traffic than HoneyHive. Part of the reason for this could be how packets are counted in Snort versus Suricata. Capinfos counts all the packets in the PCAP and could be combining the fragmented parts together, reducing the actual number. Suricata reports by using Scapy to increment its packet for every packet sent or received with the attacker's IP address. Table 11 shows the Anderson-Darling normality test for the percentage of packets captured but only % HPP for the TCP Connect Scan with 6 honeypots and % SuP for the Aggressive Scan with 9 honeypots are normal. Table 12 shows the Permutation test results for percentage of packets captured. All but TCP Connect overall, TCP Connect Scan

	HoneyHive	Suricata
0	0 / 6 = 0.00%	2 / 6 = 33.33%
3	3 / 9 = 33.33%	2 / 9 = 22.22%
6	6 / 12 = 50.00%	5 / 12 = 41.67%
9	9 / 15 = 60.00%	5 / 15 = 33.33%

Table 9. Percentage of Scanned Devices Monitored in Test Network

with 3 and 6 Honeypots (although very close), Aggressive Scan with 6 honeypots, and overall results are statistically significant. % HHP performed significantly better than % SuP in the Aggressive Scan overall and all tests with 9 honeypots, minus the NIDS Avoidance Scan. % SuP performed significantly better in all NIDS Avoidance Scans, Aggressive Scan with 3 honeypots, and 3 and 6 Honeypots overall. Even though the HoneyHive framework does not at this moment create more alerts or more distinct alerts than Suricata, the fact that it significantly outperforms Suricata in capturing network traffic when all 9 honeypots are active supports it will be able to detect more than a traditional NIDS could. Also of note, Suricata did not significantly outperform the HoneyHive framework in any of the TCP Connect scans.

Figure 21 shows the trend of an increase in mean capture packet percentage (% HHP) for the HoneyHive framework as the ratio of honeypots to devices scanned on a network also increases. In the TCP Connect scan each addition of 3 honeypots increases % HHP by 9%. For the Aggressive scan there is a 36% increase for % HHP from 3 to 6 honeypots and then a 39% increase from 6 to 9 honeypots. The NIDS Avoidance scan only has a 5% increase from 3 to 6 honeypots and an 8% increase from 6 to 9 honeypots. Finally, % HHP for each level of honeypot overall increase 17% from 3 to 6 and 18% from 6 to 9. Assuming this trendline continues, at 12 and 15 honeypots % HHP would be 62.5% and 80% respectively. This well exceeds % SuP which would continue to decrease with the addition of honeypots not connected to the switch it monitors.

	% HHP	% SuP
Control Group	0%	21%
CG 0	0%	5%
CG 3	0%	6%
CG 6	0%	7%
CG 9	0%	49%
TCP Connect	22%	19%
TCP 0	0%	19%
TCP 3	13%	10%
TCP 6	22%	28%
TCP 9	31%	21%
Aggressive	39%	27%
Agg 0	0%	4%
Agg 3	2%	4%
Agg 6	38%	35%
Agg 9	77%	41%
NIDS Avoidance	21%	32%
NIDS 0	0%	32%
NIDS 3	15%	23%
NIDS 6	20%	40%
NIDS 9	28%	34%
0 Honeypots	0%	18%
3 Honeypots	10%	12%
6 Honeypots	27%	34%
9 Honeypots	45%	32%
Overall	27%	26%

Table 10. Overview - Mean Packet Capture Percentage
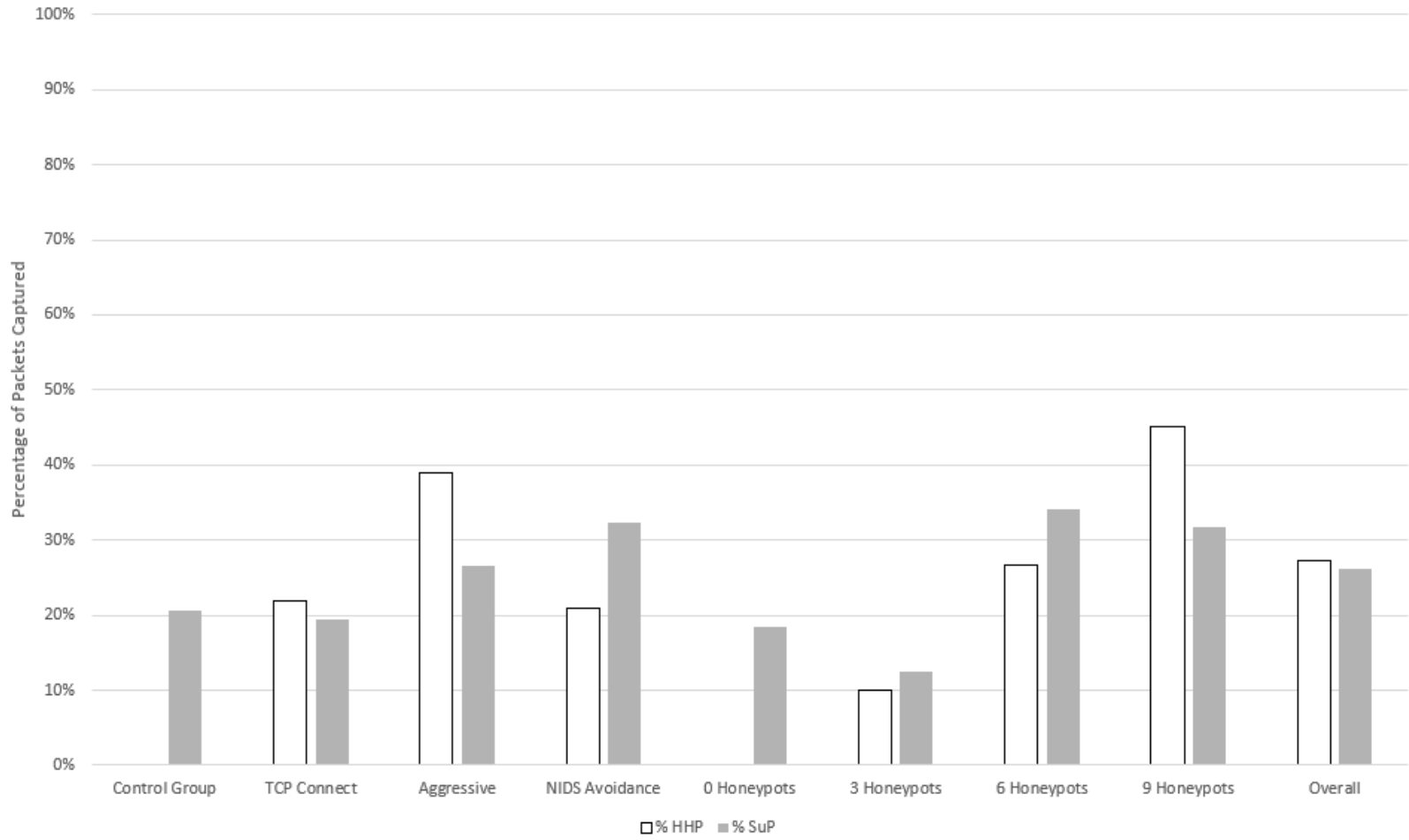
Figure 20. Overview - Mean Packet Capture Percentage
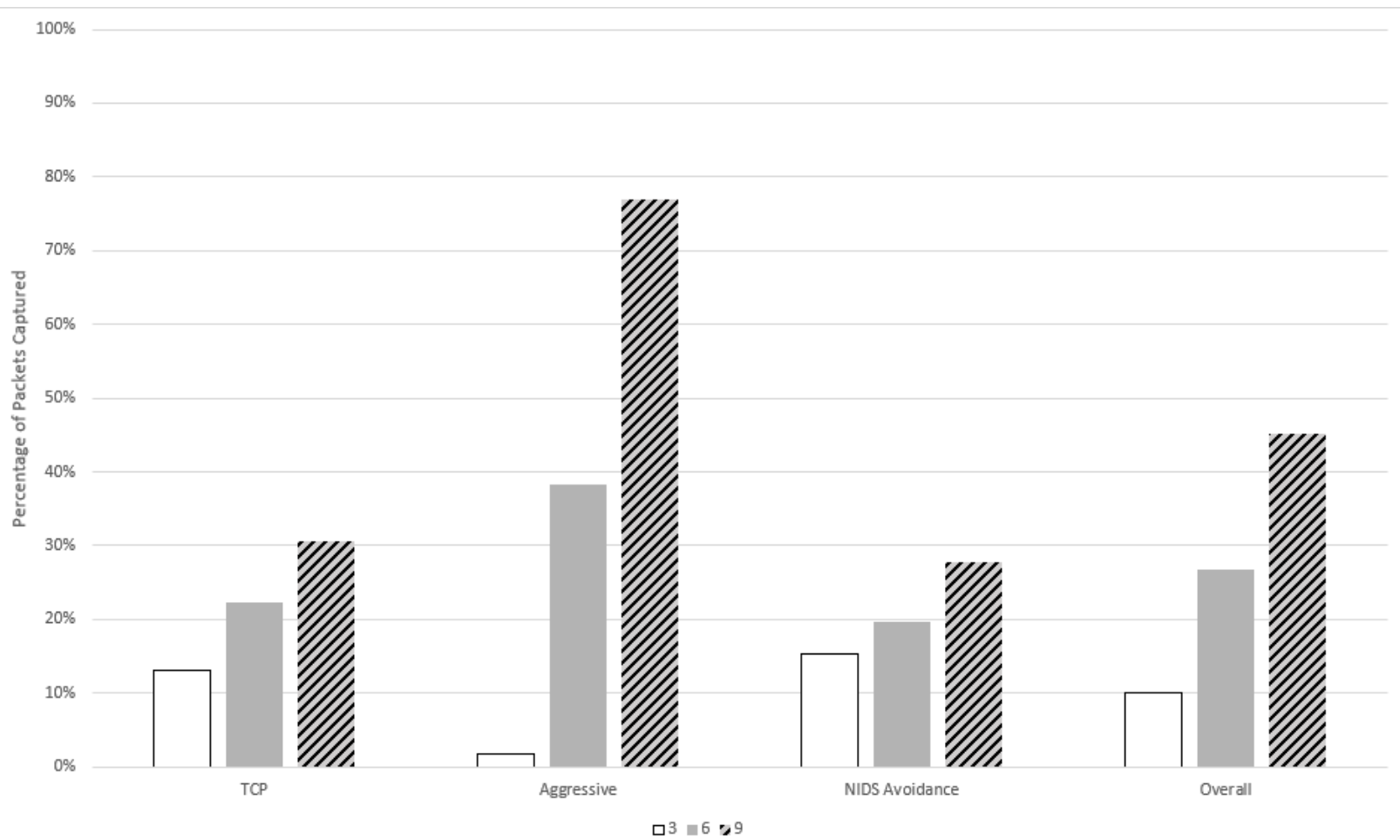
Figure 21. HoneyHive Framework Mean Packet Capture Percentage (% HHP) by Level

	% HHP	% SuP
TCP Connect	2.4739E-05	1.18E-08
TCP 3	1.7612E-08	0.001415
TCP 6	0.08450075	7.71E-10
TCP 9	0.00013778	0.013687
Aggressive	1.2062E-18	2.86E-14
Agg 3	1.2349E-07	3.13E-08
Agg 6	6.8951E-05	2.95E-05
Agg 9	5.1989E-06	0.474596
NIDS Avoidance	0.00022313	1.95E-05
NIDS 3	9.268E-09	1.77E-08
NIDS 6	3.2239E-06	1.11E-10
NIDS 9	2.6557E-08	0.00294
3 Honeypots	2.018E-08	2.23E-13
6 Honeypots	1.2178E-15	3.65E-05
9 Honeypots	8.2817E-10	4.19E-05
Overall	0	9.39E-07

Table 11. Anderson-Darling Test - Packet Capture Percentage

	% HHP v % SuP
TCP Connect	0.20446
TCP 3	0.109532222
TCP 6	0.10659
TCP 9	0.003485556
Aggressive	0.010246667
Agg 3	0
Agg 6	0.667023333
Agg 9	0.003394444
NIDS Avoidance	0
NIDS 3	0
NIDS 6	0
NIDS 9	0
3 Honeypots	0.081776667
6 Honeypots	0.019293333
9 Honeypots	3.78E-05
Overall	0.537306667

Table 12. Permutation Test - Packet Capture Percentage

5.2 Scan Type

This section groups experiment results by each of the different scan types for analysis. These scan types are No Scan (Control Group), TCP Connect, Aggressive, and NIDS Avoidance.

5.2.1 Control Group

For the control group, where no scan was performed, no alerts were generated for any of the metrics (HHI, SAI, SAM, and SuA). Similarly, without any alerts there were also no distinct types of alerts (STI, STM, and SuT). This supports that the HoneyHive framework operates correctly without any malicious traffic and does not create false positives. The packets captured by the HoneyHive framework were zero for all different levels of honeypots as is expected since no packets were sent to the honeypots. However, Suricata still captured traffic as shown in Figure 22. Although no malicious traffic was sent by the attacker, routine network traffic still takes place and this is the traffic that is captured by Suricata. One interesting aspect about the data is the sharp jump in percentage when 9 honeypots are active as compared to the other levels of honeypots. Roughly seven times the traffic captured in other levels is captured in this level. This is indicative that more traffic traverses the switch Suricata is on. Upon further investigation, one trial had a massive outlier of 1247% captured. When removing this outlier and reanalyzing, the mean falls in line with the other honeypot levels. Because Suricata's packet count only contains traffic to or from the attacker and the percentage was much higher than what the attacker sent, some device must have been sending traffic to the attacker's IP. It is speculated that this is attributed to Honeyd expiring all scan connections made by the attacker in the previous trial (234) aggressive scan.
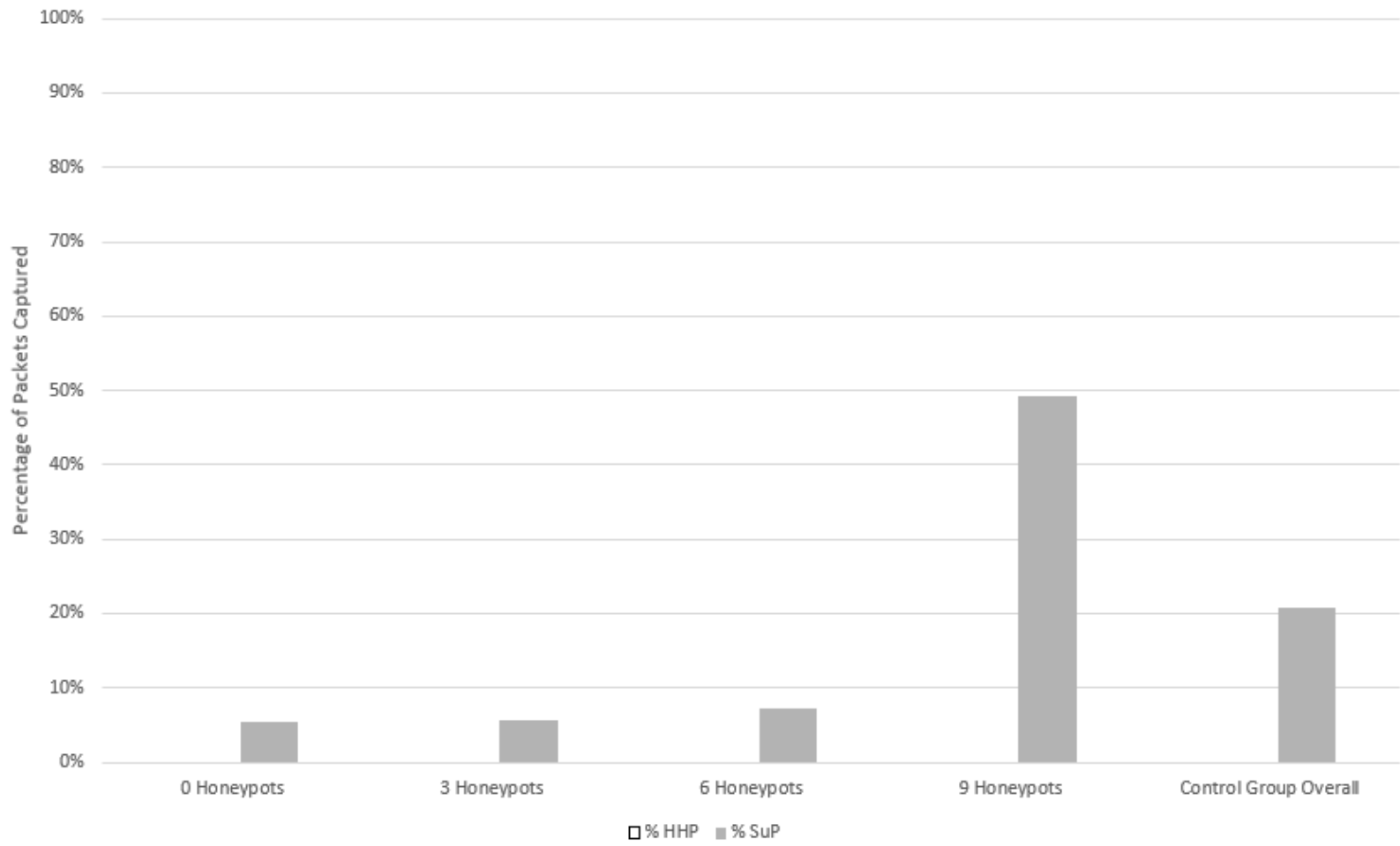
Figure 22. Control Group - Mean Percentage of Packets Captured

5.2.2 TCP Connect

In Figure 23 Suricata generates more alerts on average in all levels of honeypots for the TCP Connect scan. When 9 honeypots are active, SAI is only a few alerts away from SuA. One aspect of note is that after 3 honeypots, SAI becomes and remains higher than SAM. It is speculated that this is partially due to Snort aggregating some alerts it sees relatively close to one another to reduce the number to parse through. Additionally, by parsing combined packets, it appears that Snort is able to create more alerts at levels of honeypots with a lower amount of traffic.

In Figure 24, STM significantly outperforms STI in honeypot levels 6, 9, and overall. However, Suricata significantly outperforms STM in all TCP connect tests. One interesting aspect to note is that the average at 6 honeypots is higher than that of 9 for STI and STM. Upon looking at the individual data, there were four runs where Snort reports 0 for the number of alerts and the distinct number of alerts in the 9 honeypot level and three runs for the 6 honeypot level. Upon removing these, the new averages are STI:5.15, STM: 6.62 for 9 honeypots and STI:5.15, STM: 6.52, for 6 honeypots, supporting the trend of more honeypots creating more distinct alerts with diminishing returns. The 0 runs are most likely from Snort crashing or not finishing before results were collected.

Figure 25 shows the average percentage of packets captured by the HoneyHive framework compared to that of Suricata. The HoneyHive framework begins with a higher capture percentage at the 3 honeypots level but then Suricata captures more at 6 honeypots. This is interesting because the number of honeypots increases equally for the HoneyHive framework and Suricata. However, at 9 honeypots and for TCP Connect scan overall the HoneyHive framework captures more traffic on average than Suricata. The capture percentage of HoneyHive at 9 honeypots is statistically higher than that of Suricata. No other levels were statistically significant. Another trend

expected to repeat is a drop in Suricata packet capture percentage from 6 to 9 due to an increase in attacker traffic that the SPAN port does not receive.
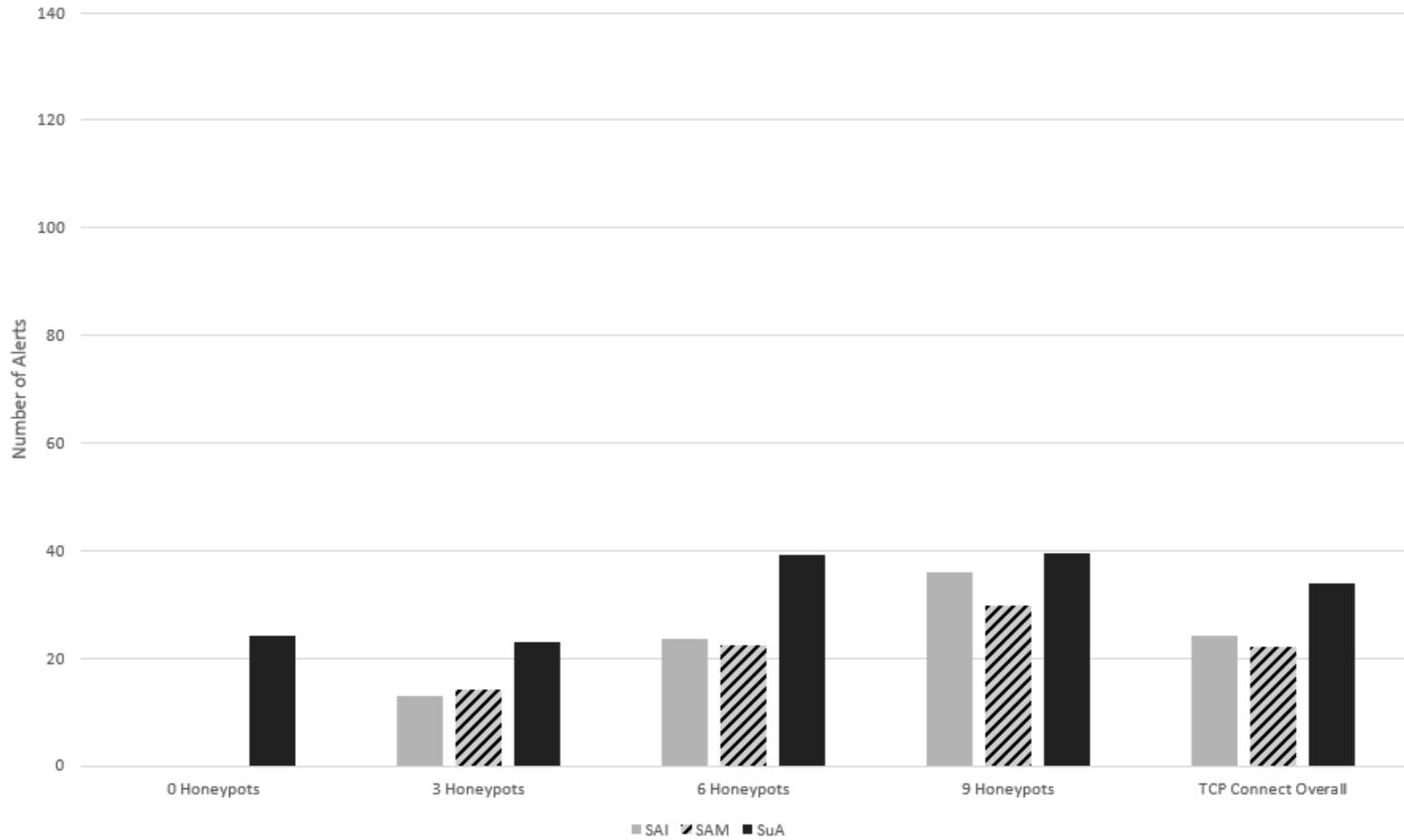
Figure 23. TCP Connect - Mean Number of Alerts
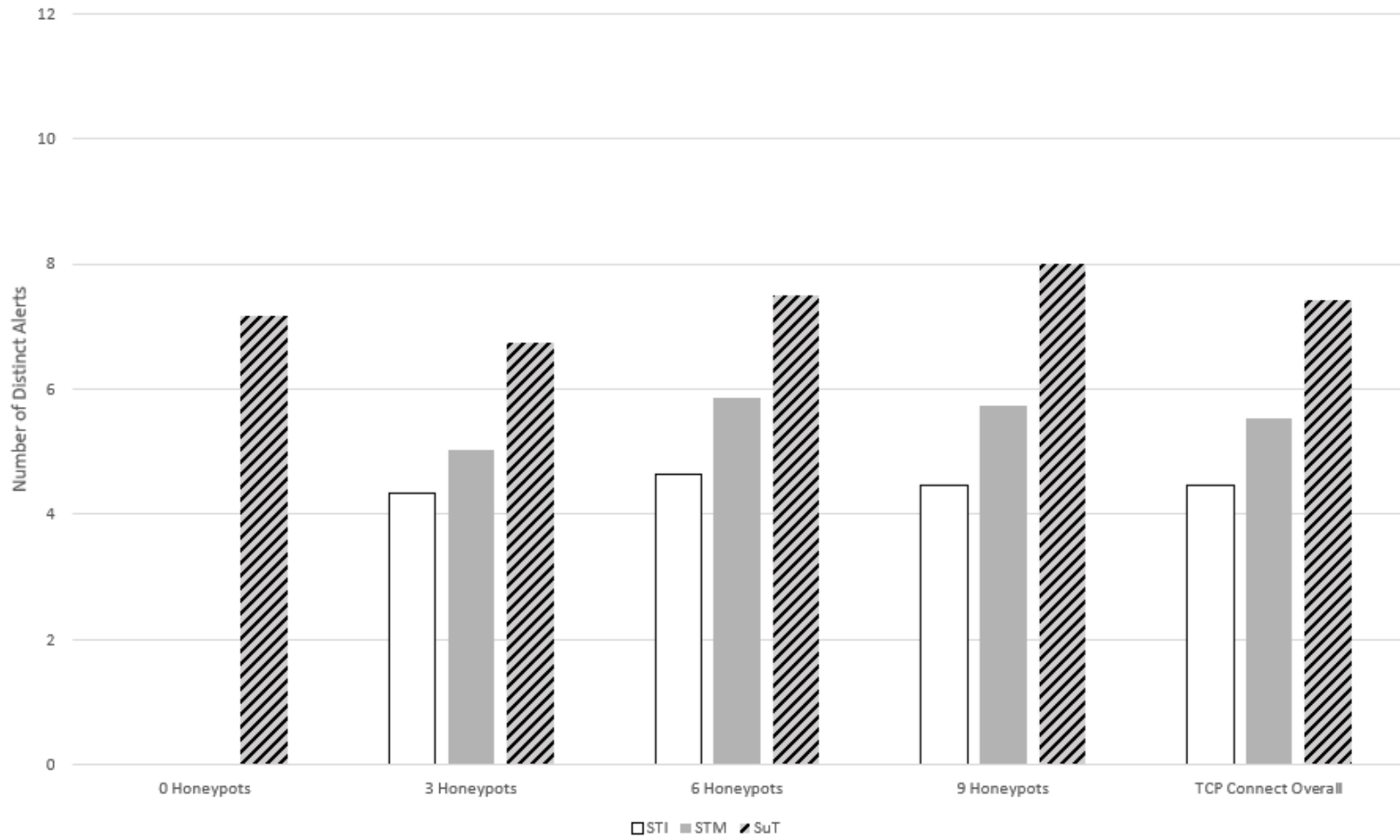
Figure 24. TCP Connect - Mean Number of Distinct Alerts
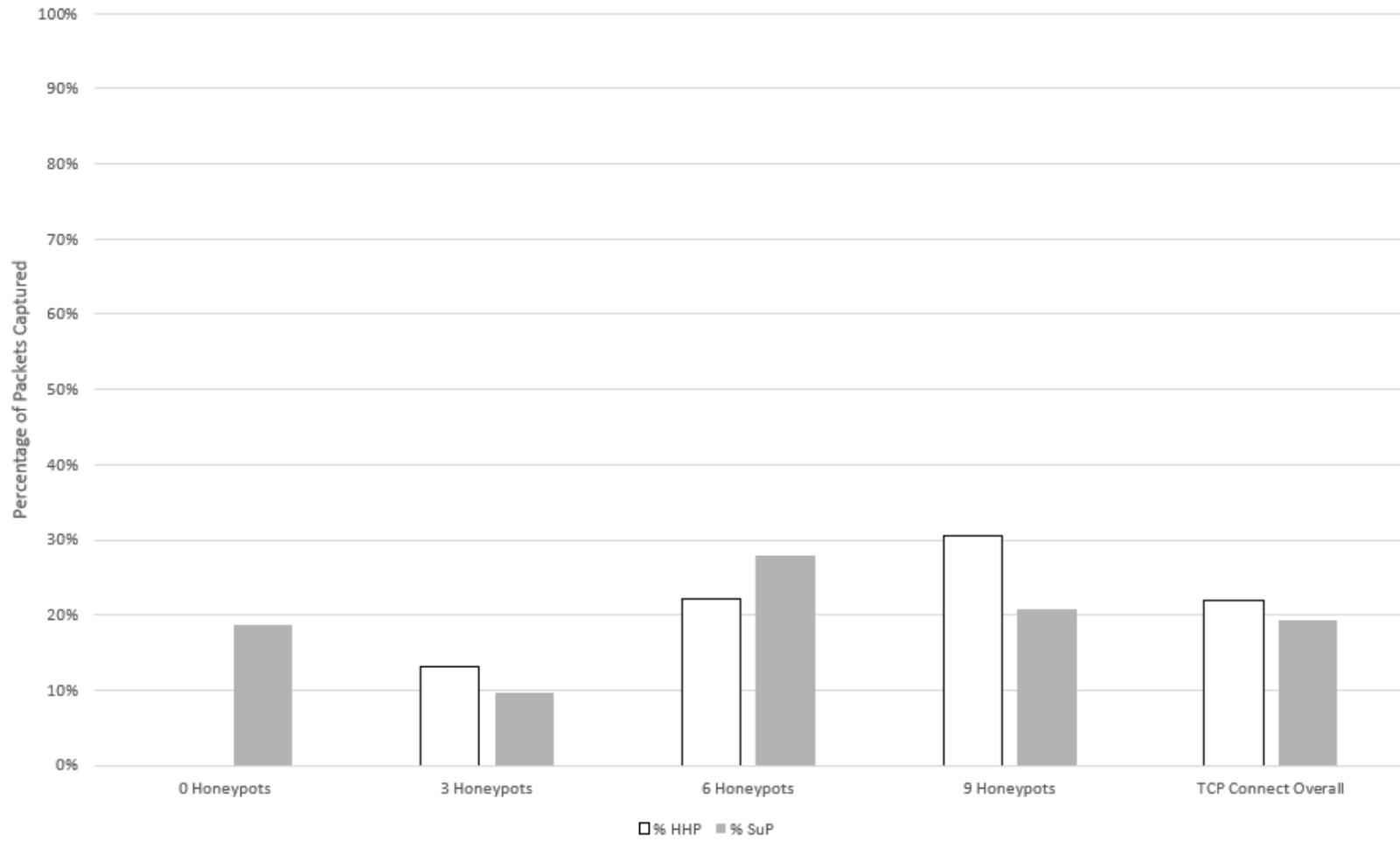
Figure 25. TCP Connect - Mean Packet Capture Percentage

5.2.3 Aggressive

In Figure 26 the trend holds for SAM creating more alerts than SAI at 3 honeypots and then SAI creating more alerts than SAM at 6 and 9 honeypots. Suricata performs significantly better overall and when 3 or 6 honeypots are active. SAI performs significantly better than SuA when 9 honeypots are active, supporting the hypothesis that distributed sensors can detect intrusions that traditional NIDS cannot. This is the only test in which the HoneyHive framework creates more alerts than Suricata although at 9 honeypots in the TCP connect scan, it is also very close to that of Suricata.

Figure 27 also follows the trend of the corrected data in the TCP connect scan, an increase in distinct number of alerts in STI and STM with STM being significantly greater than STI. SuT is significantly greater overall and for 3 and 6 honeypots. Even though it is larger, it is not significantly greater than STM at 9 honeypots. One interesting trend is that the number of distinct alerts for SuT actually decreases as the number of honeypots increases. This could be caused by the increased time spent scanning each host as more honeypots are activated and not tripping rules that are triggered by consecutive scanning in a certain amount of time.

The average packet percentage for each honeypot level in the Aggressive Scan is shown in Figure 28. The HoneyHive framework captures a higher percentage on average overall and for 6 honeypots and significantly more for 9 honeypots. Even though Suricata does not drop in packet percentage captured as it did in the TCP Connect scan, it does not increase nearly as much as it did from 3 to 6 honeypots.
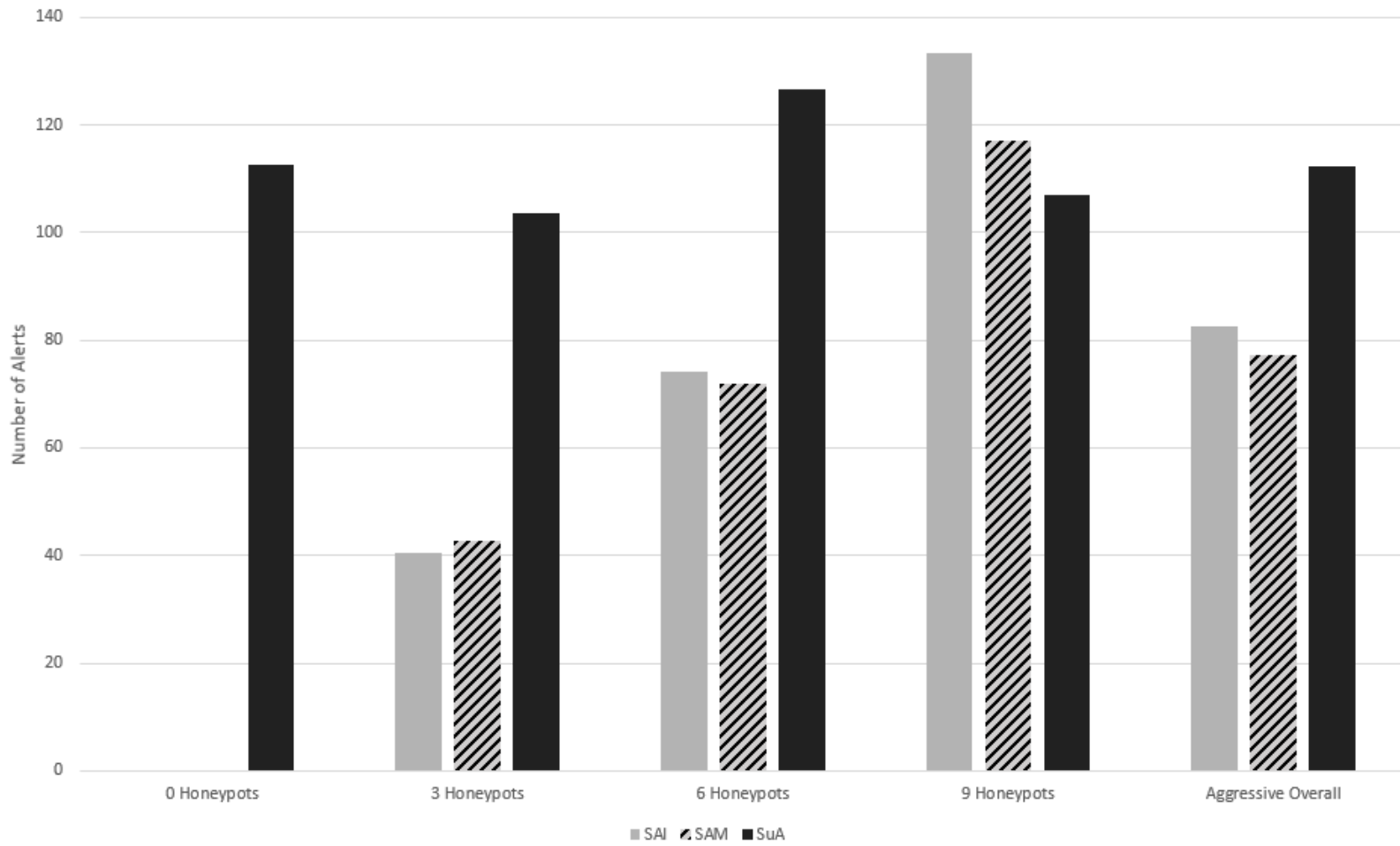
Figure 26. Aggressive - Mean Number of Alerts
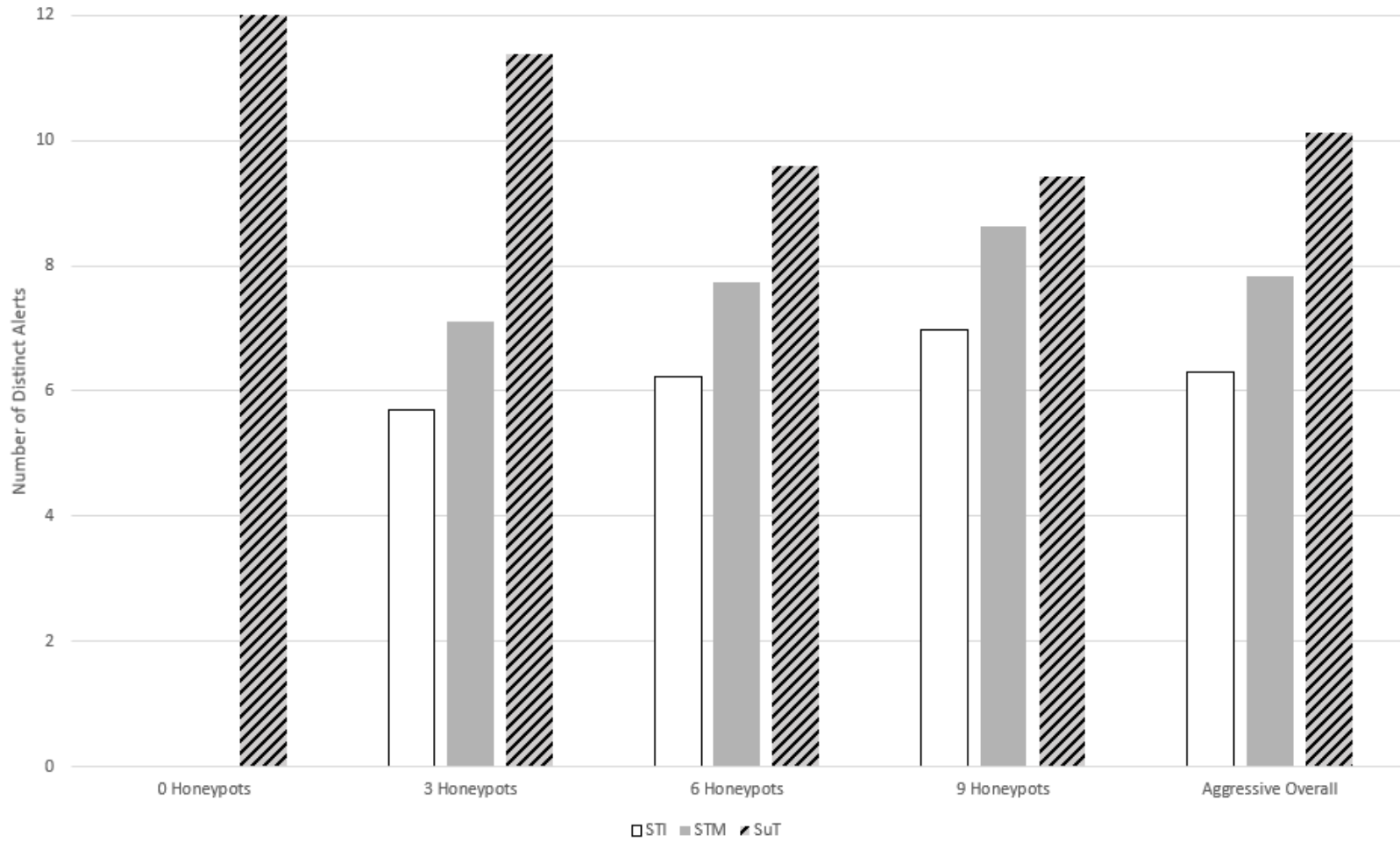
Figure 27. Aggressive - Mean Number of Distinct Alerts
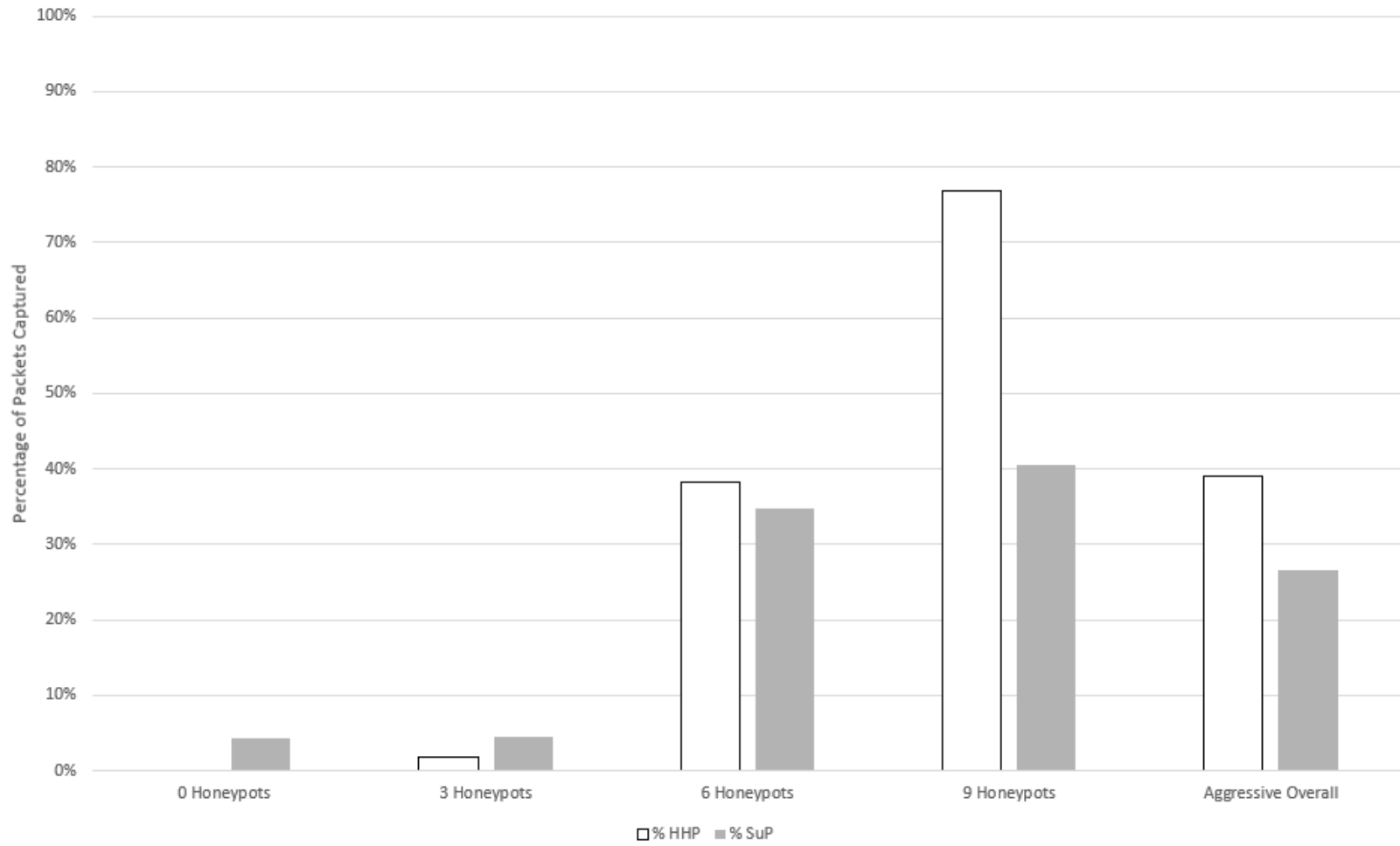
Figure 28. Aggressive - Mean Packet Capture Percentage

5.2.4   NIDS Avoidance

The average number of alerts for the NIDS Avoidance Scan with each level of honeypot is shown in Figure 29. While not statistically significant, SAM performs better than STI in all levels of honeypots. This is most likely due to the packet fragmentation and scan delay not tripping as many Snort alerts, although more alerts are created as more honeypots are active and more packets are aggregated. Because SAM creates more alerts on average over SAI, it is compared against SuA. Unlike the TCP Connect scan with 9 honeypots and Aggressive scan with 9 honeypots, Suricata significantly outperforms Snort in the 9 honeypots level and all other tests.

Figure 30 shows the average distinct number of alerts for the NIDS Avoidance scan, broken down by each honeypot level and overall. All metrics increase as the number of honeypots they monitor increase as well. STM outperforms STI in all tests and is statistically significant in all tests except with 3 honeypots (just over 10%). SuT is statistically superior in performance in all tests for the NIDS Avoidance scan.

The trend of the percentage of packets captured by Suricata declining from 6 to 9 honeypots holds and is shown in Figure 31. However, in the NIDS Avoidance Scan % SuP significantly outperforms % HHP in all tests, even when 9 honeypots are active. % HHP still increases with each honeypot level though. As mentioned in the overview, the packet count may be different if Capinfos is combining fragmented packets together, resulting in a reduced packet count.
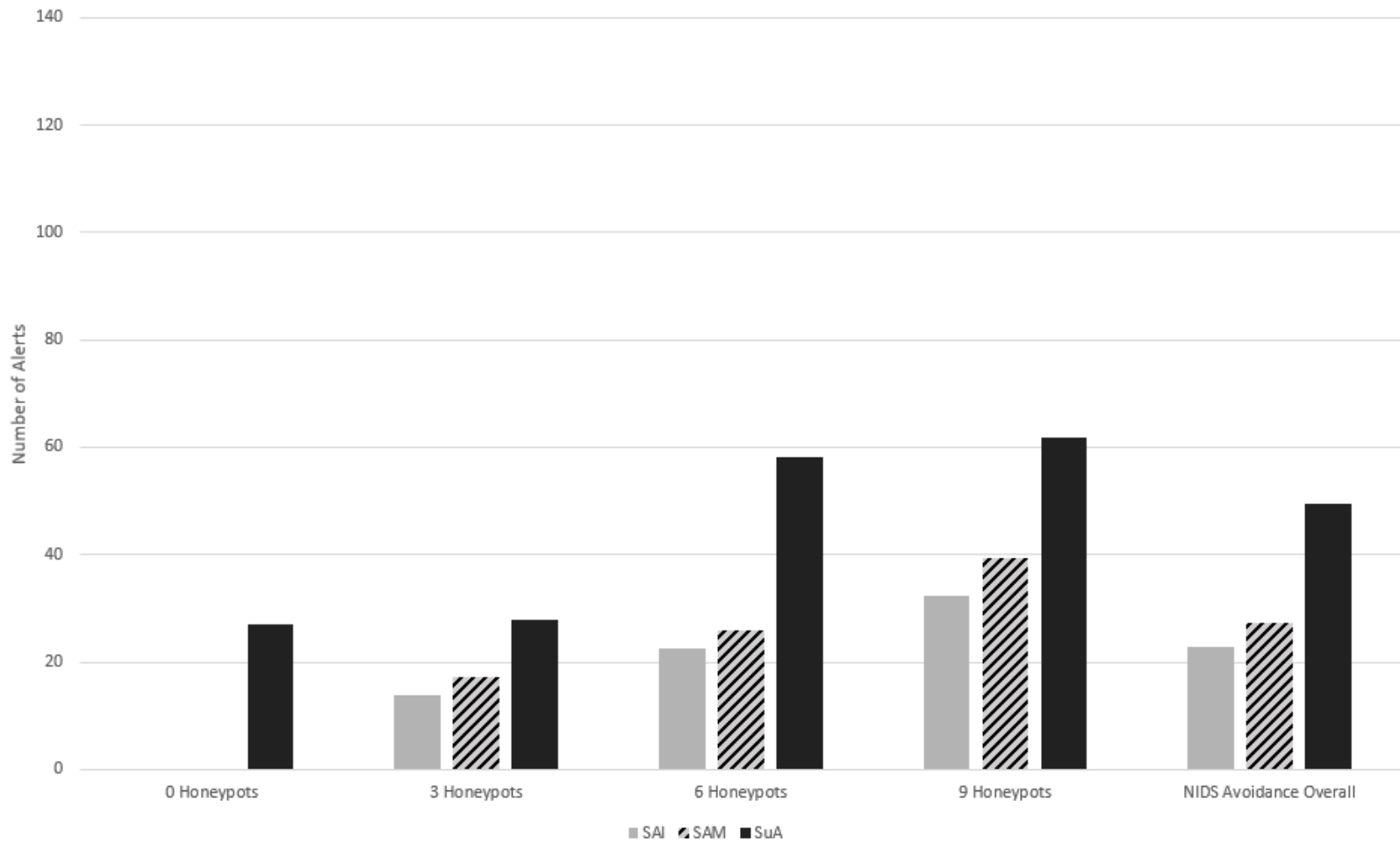
Figure 29. NIDS Avoidance - Mean Number of Alerts
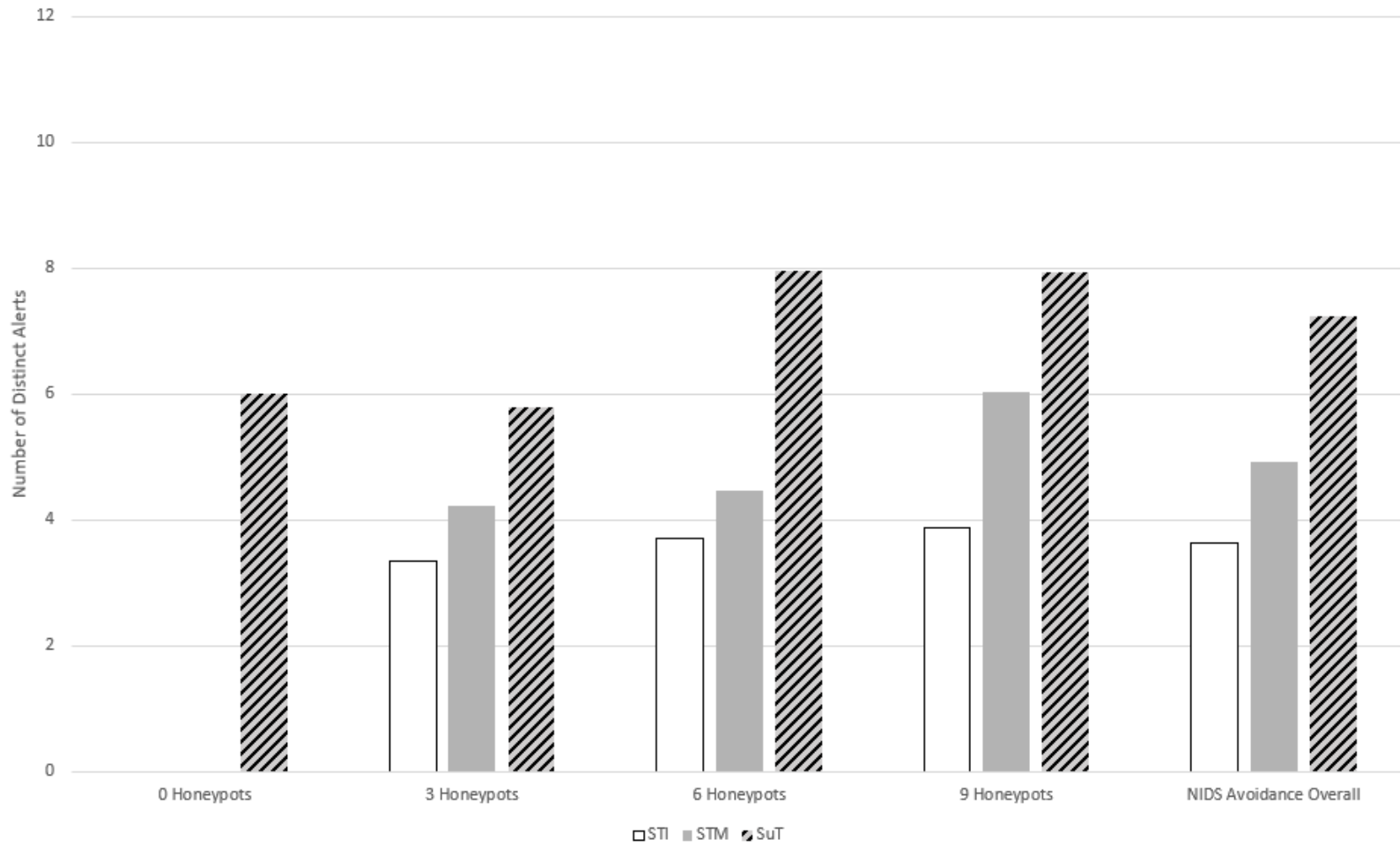
Figure 30. NIDS Avoidance - Mean Number of Distinct Alerts
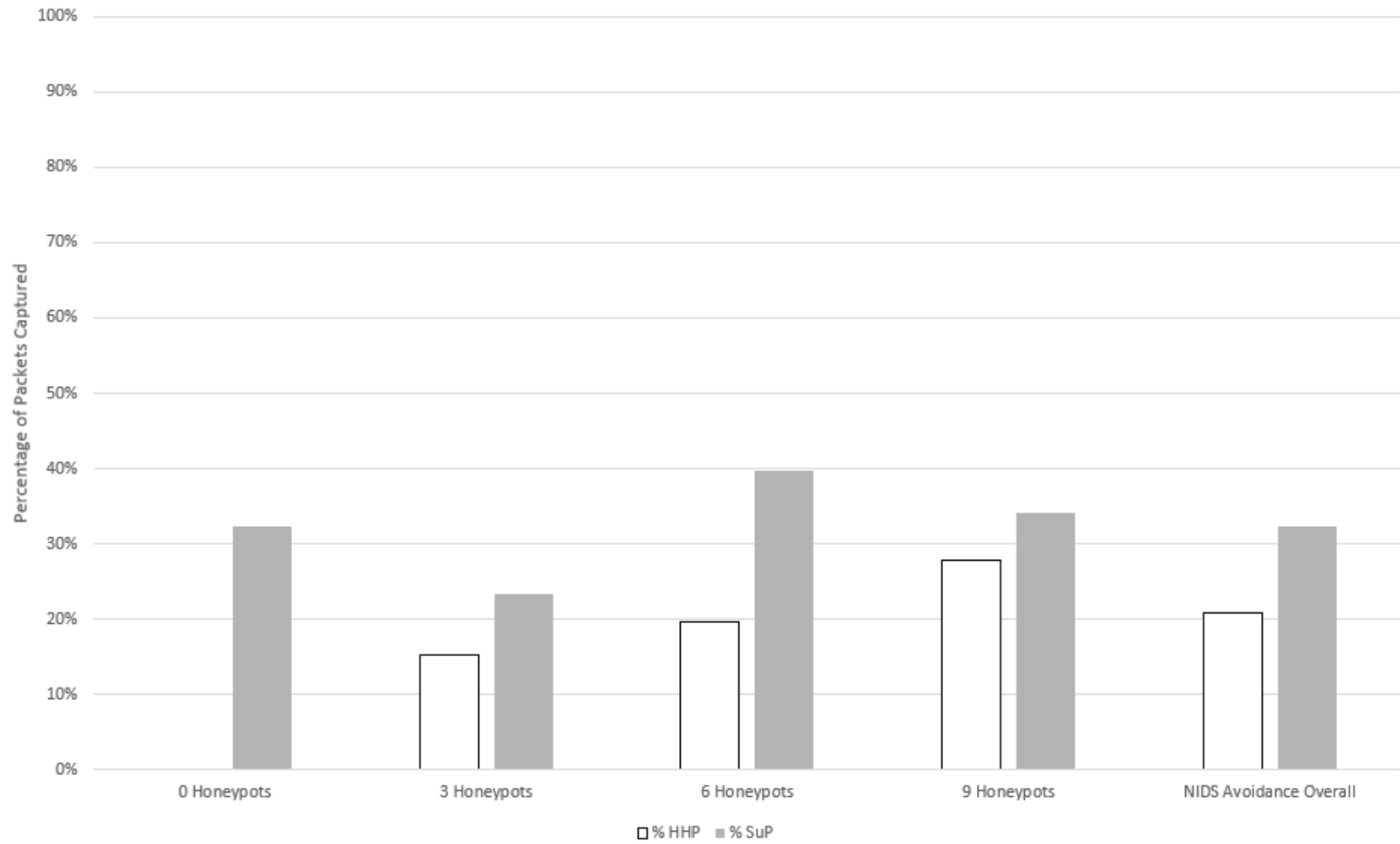
Figure 31. NIDS Avoidance - Mean Packet Capture Percentage

5.3 Number of Honeypots

This section groups experiment results by each of the different levels of honeypots for analysis. These levels are 0, 3, 6, and 9.

5.3.1   0 Honeypots

Without any active honeypots, no alerts should be created and no packets captured. This section is useful in understanding the baseline performance of Suricata across different scan types. The average number of alerts is shown in Figure 32. Suricata creates alerts in all scan types with the most in the Aggressive scan, followed by the NIDS Avoidance scan and then the TCP Connect scan. The number of alerts increases as the number of honeypots increases for TCP Connect and NIDS but actually decreases for all honeypots not monitored by it (all but 6). This is most likely due to a decrease in alerts being triggered as hosts being scanned in sequence takes longer since each scan takes longer. Suricata performed correctly by not creating any alerts when no scan was performed.

Figure 30 displays the average number of distinct alerts created by Suricata for each different type of scan. The Aggressive scan creates the most number of distinct alerts followed by TCP Connect scan and then the NIDS Avoidance scan. The distinct number of alerts actually decreases for the Avoidance scan as each honeypot level increases. The TCP Connect scan decreases for the 3 honeypots but then increases for 6 and 9 honeypots. The NIDS Avoidance scan similarly decreases for 3 honeypots, increases for 6, but then decreases slightly again for 9 honeypots.

The average percentage of packets captured by Suricata for each test is shown in Figure 34. The NIDS Avoidance scan captured the largest percentage of packets on average, followed by the TCP Connect scan, the Control Group, and then the Aggressive Scan. The NIDS Avoidance scan capture percentage being so high with

116

0 honeypots supports that it is being skewed in other tests. The TCP Connect scan results are within the expected amount for only one monitored device out of three. With the attacker sending no scan packets in the control group, one might wonder why packets are being captured. These packets are command and control packets for running tests. It is surprising that the Aggressive scan packet capture percentage was so low. This could be because the actual IoT devices cannot keep up with the scan so ones not on this switch generate lots of re-transmissions that are not captured or Nmap quickly dismisses the IoT device on the switch and moves on.
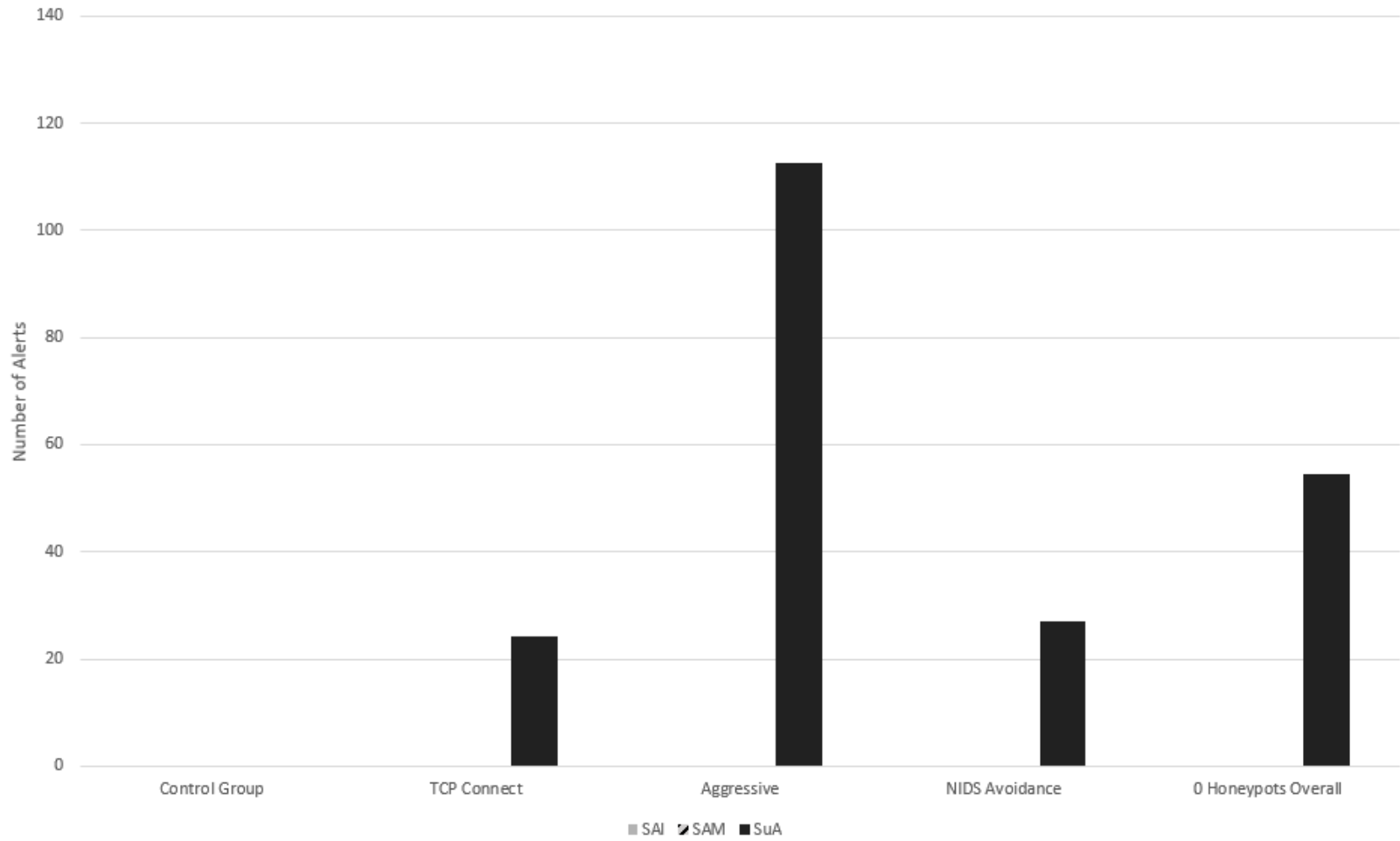
Figure 32. 0 Honeypots - Mean Number of Alerts
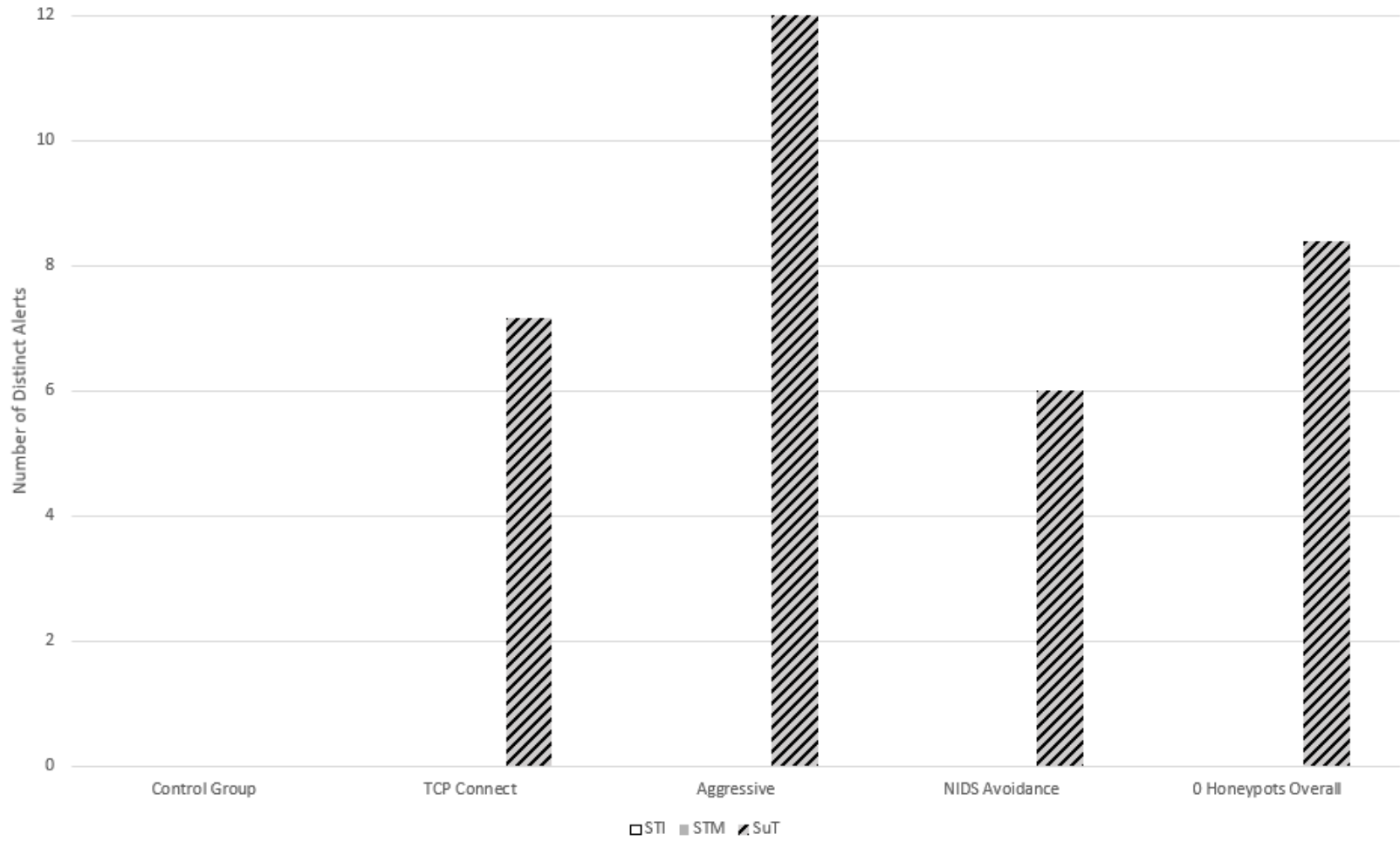
Figure 33. 0 Honeypots - Mean Number of Distinct Alerts
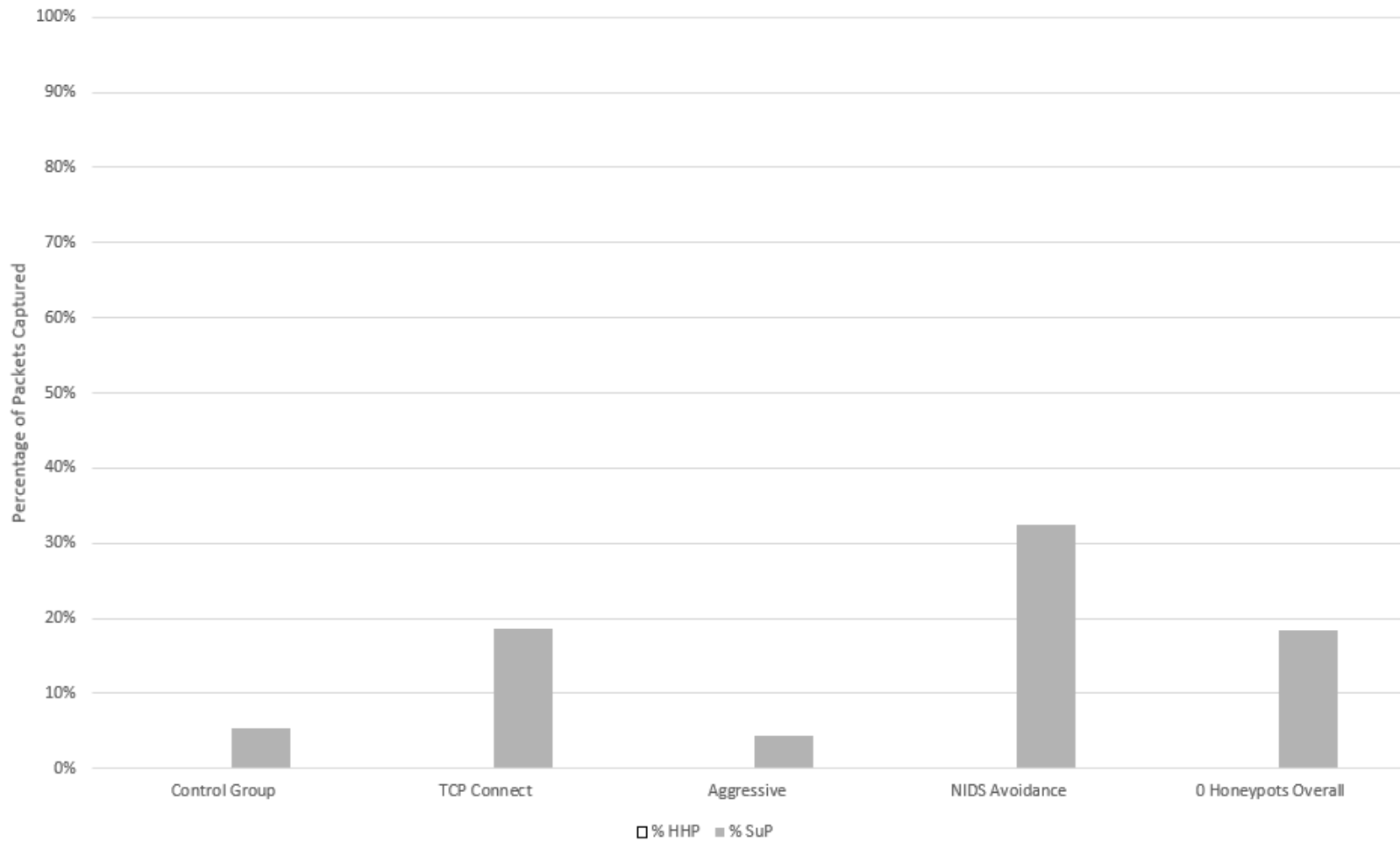
Figure 34. 0 Honeypots - Mean Packet Capture Percentage

5.3.2   3 Honeypots

Figure 35 shows the average number of alerts with 3 honeypots active grouped by different scan types. All three metrics create more alerts in the Aggressive scan than the other scan types. This is followed by the NIDS Avoidance Scan, and the TCP Connect scan. Suricata creates the most alerts in all tests followed by SAM and then SAI. Even though the NIDS Avoidance scan was designed to trigger the least number of alerts, it appears that the NIDS have the most difficulty in creating alerts for the TCP Connect scan. This is probably because the TCP Connect scan is difficult to distinguish from normal connections while the NIDS Avoidance scan sends fragmented packets, which might be treated as suspicious. The NIDS clearly have no trouble detecting Aggressive Scans which makes sense because the scan generates much more traffic by requesting OS and service information, giving it a very identifiable signature.

Figure 36 displays the number of distinct alerts for each scan with 3 honeypots active. Based on the sheer number of alerts generated by the Aggressive scan, it makes sense that the distinct number would also be higher than the other scan types. However, even though the NIDS Avoidance scan generated more alerts than the TCP Connect scan, the TCP Connect scan had a larger number of distinct alerts than that of the NIDS Avoidance scan for STI, STM, and SuT. This suggest that the NIDS Avoidance scan is repeatedly generating the same alerts while TCP connect has a broader spectrum.

In Figure 37, % SuP captures more traffic than % HHP in all scans except the TCP Connect scan. The NIDS Avoidance scan has the highest percentage of captured for both % SuP and % HHP followed by the TCP Connect and then the Aggressive Scan (although the Control Group is higher for % SuP). Because packets are fragmented and sent at a much slower rate, it appears that Suricata and HoneyHive are able to better capture them in the NIDS Avoidance scan with three honeypots active. With

121

0 honeypots active on its switch, it would appear that a single IoT device scanned creates more traffic than three scanned honeypots. In the TCP Connect scan, % HHP exceeds % SuP suggesting that IoT devices handle normal connections better than ones with long delays and fragmented packets. The aggressive scan being the lowest makes sense with the large amount of traffic that is generated and with only a small fraction of the devices being scanned being active honeypots.
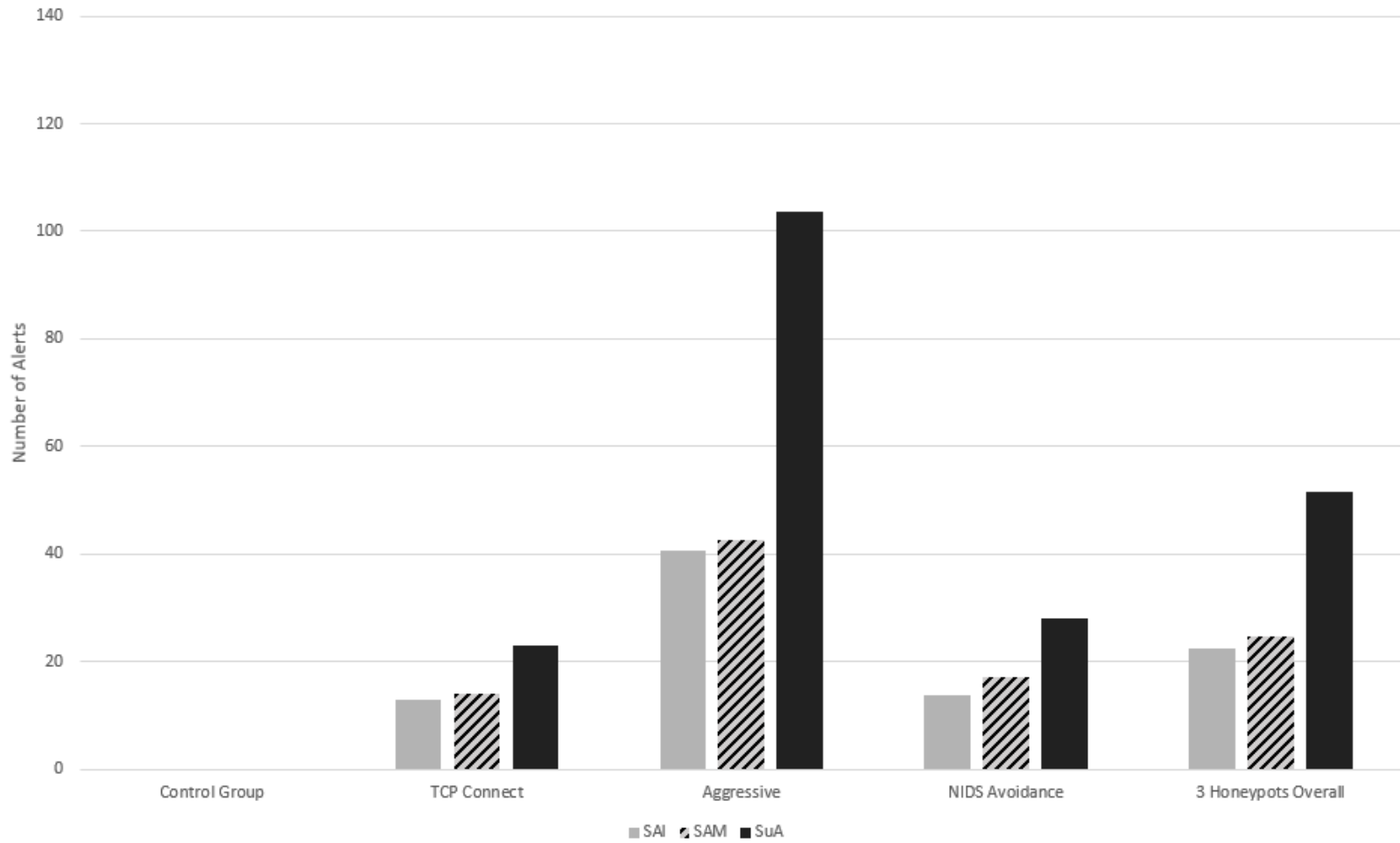
Figure 35. 3 Honeypots - Mean Number of Alerts
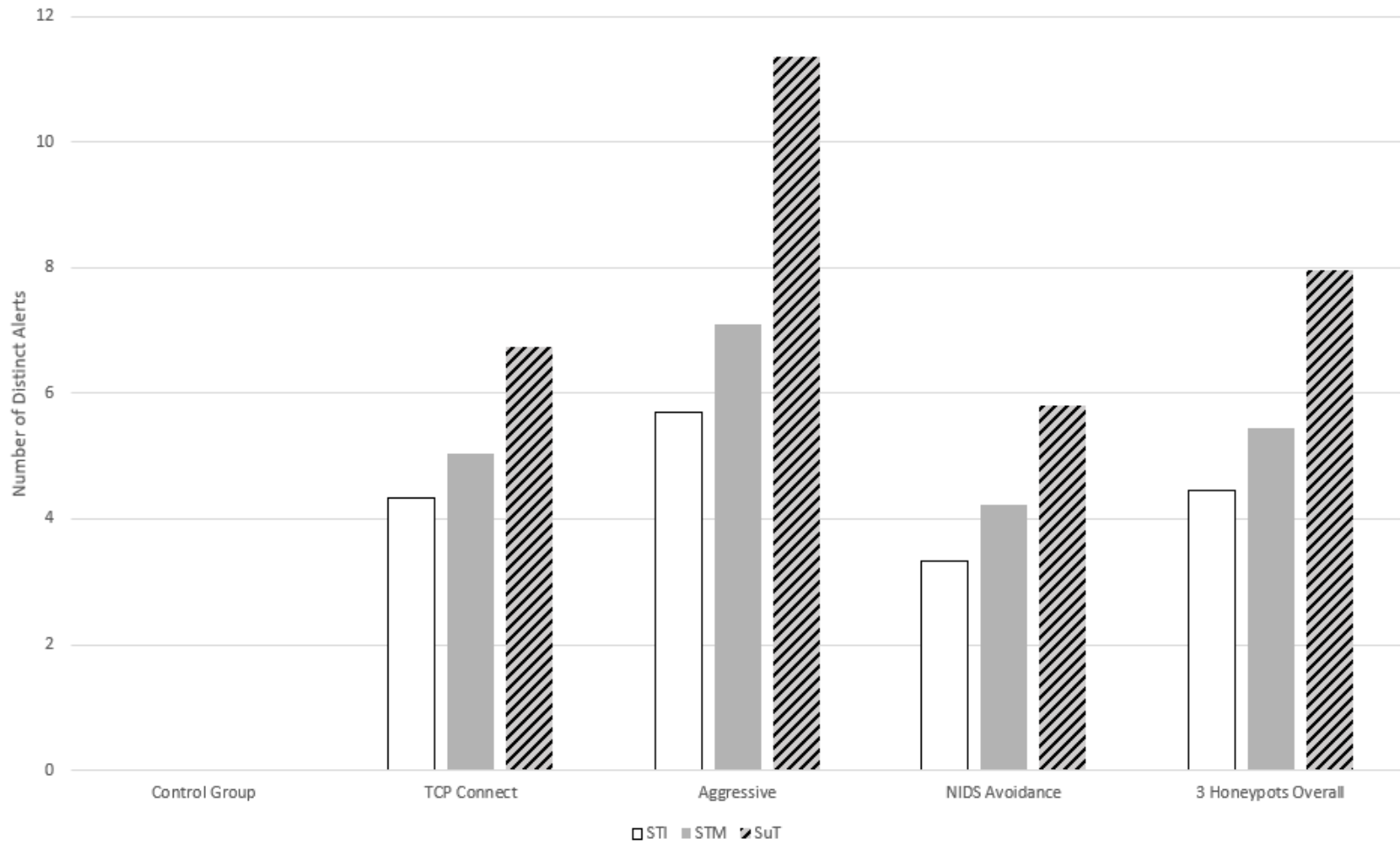
Figure 36. 3 Honeypots - Mean Number of Distinct Alerts
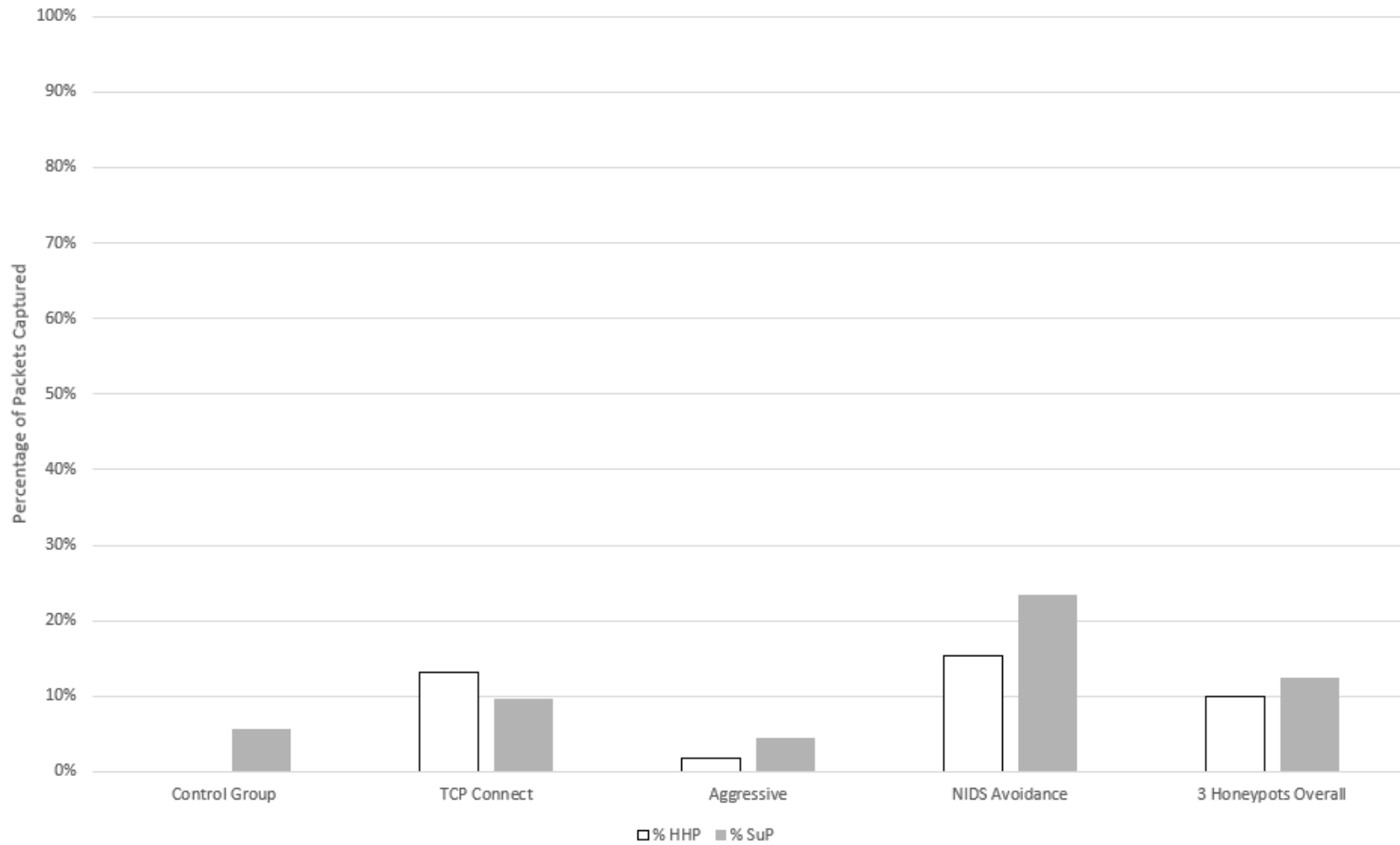
Figure 37. 3 Honeypots - Mean Packet Capture Percentage

5.3.3   6 Honeypots

The same pattern for average number of alerts holds for Figure 38 for 6 honeypots as it did with 3. The main differences are more alerts are generated in each scan and now SAI create more alerts than SAM in the TCP Connect and Aggressive scans. SAM still creates more alerts than SAI in the NIDS Avoidance scan.

With 6 honeypots the trend for the average number of distinct alerts is mostly the same as 3 honeypots and is shown in Figure 39. SuT is still the highest in all scan types but the average distinct number decreases in the Aggressive Scan. Also of note, SuT creates more alerts in the NIDS Avoidance scan instead of the TCP Connect scan, this was the opposite with 3 honeypots. STM remains higher than STI in all scan tests, and averages for both increase. Additionally, the difference between STM and SuT for the average number of distinct alerts decreases in all scans except the NIDS Avoidance scan.

Unlike scans with 3 honeypots, % HHP is the largest in the Aggressive scan and the least in the NIDS Avoidance scan, as shown in Figure 40. % SuP is still largest in the NIDS Avoidance scan but the Aggressive scan captures a higher pecentage than the TCP Connect scan. % SuP also remains higher than % HHP in the NIDS Avoidance scan, surpasses %HHP in the TCP Connect scan, and falls below % HHP in the Aggressive scan.
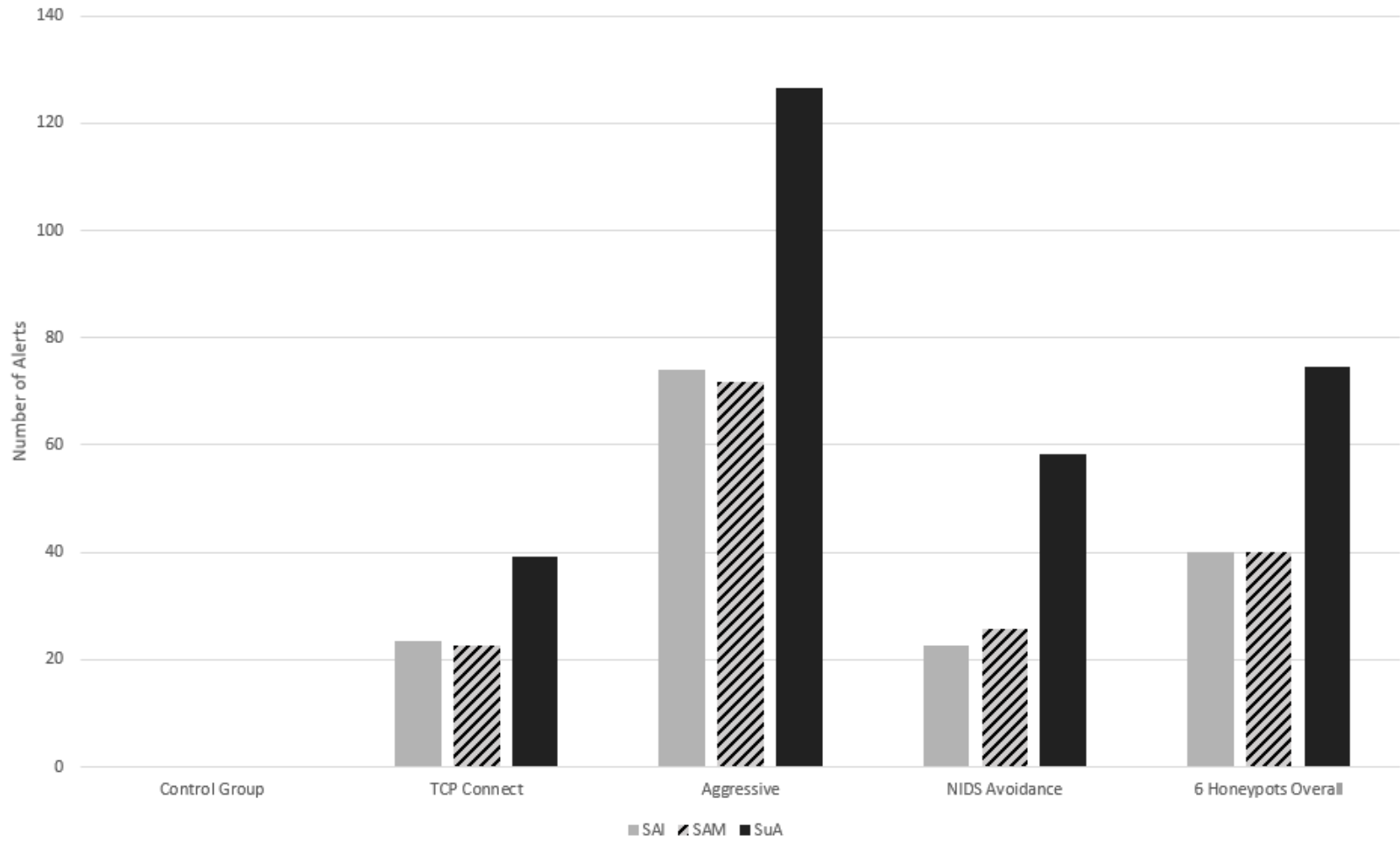
Figure 38. 6 Honeypots - Mean Number of Alerts
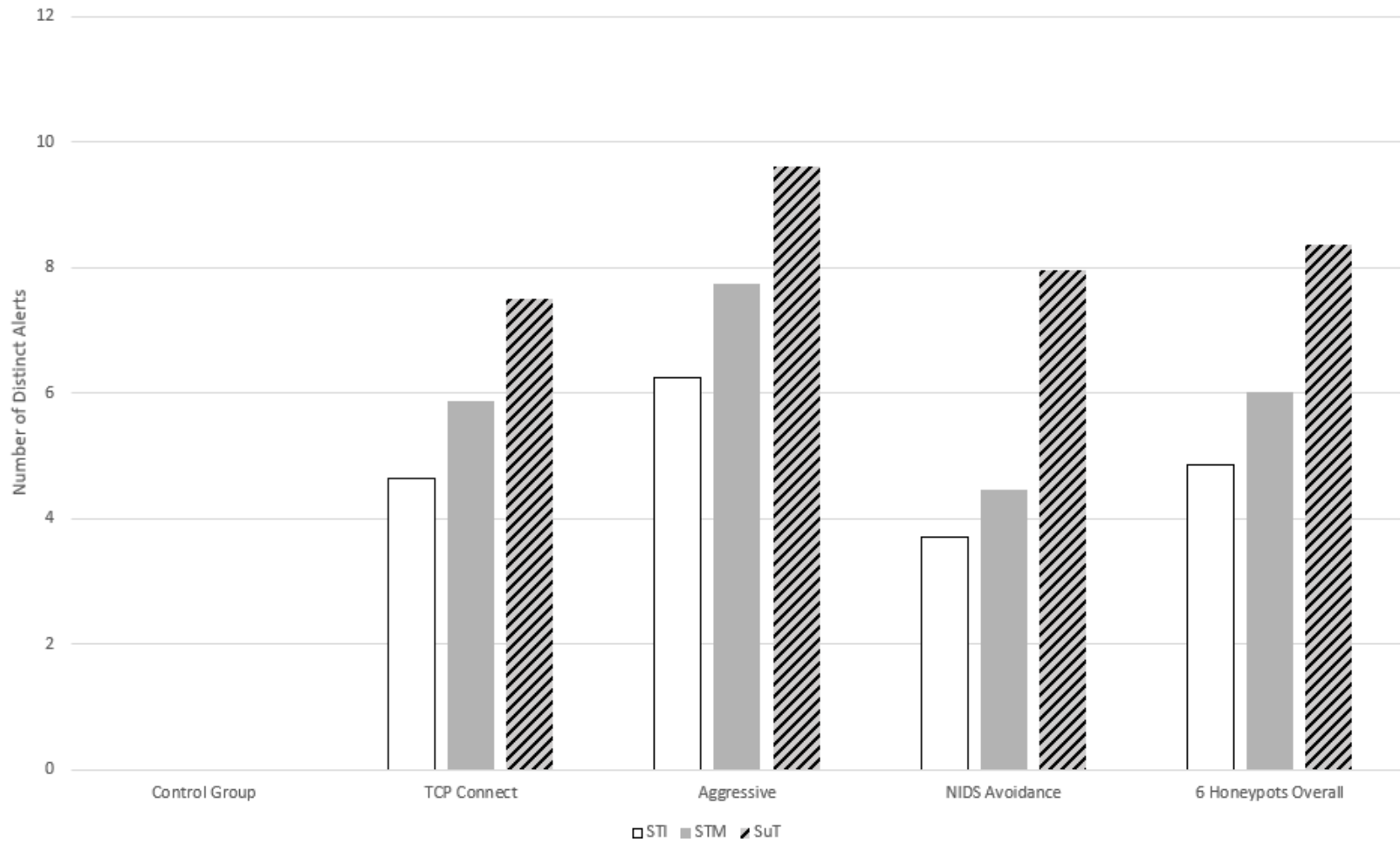
Figure 39. 6 Honeypots - Mean Number of Distinct Alerts
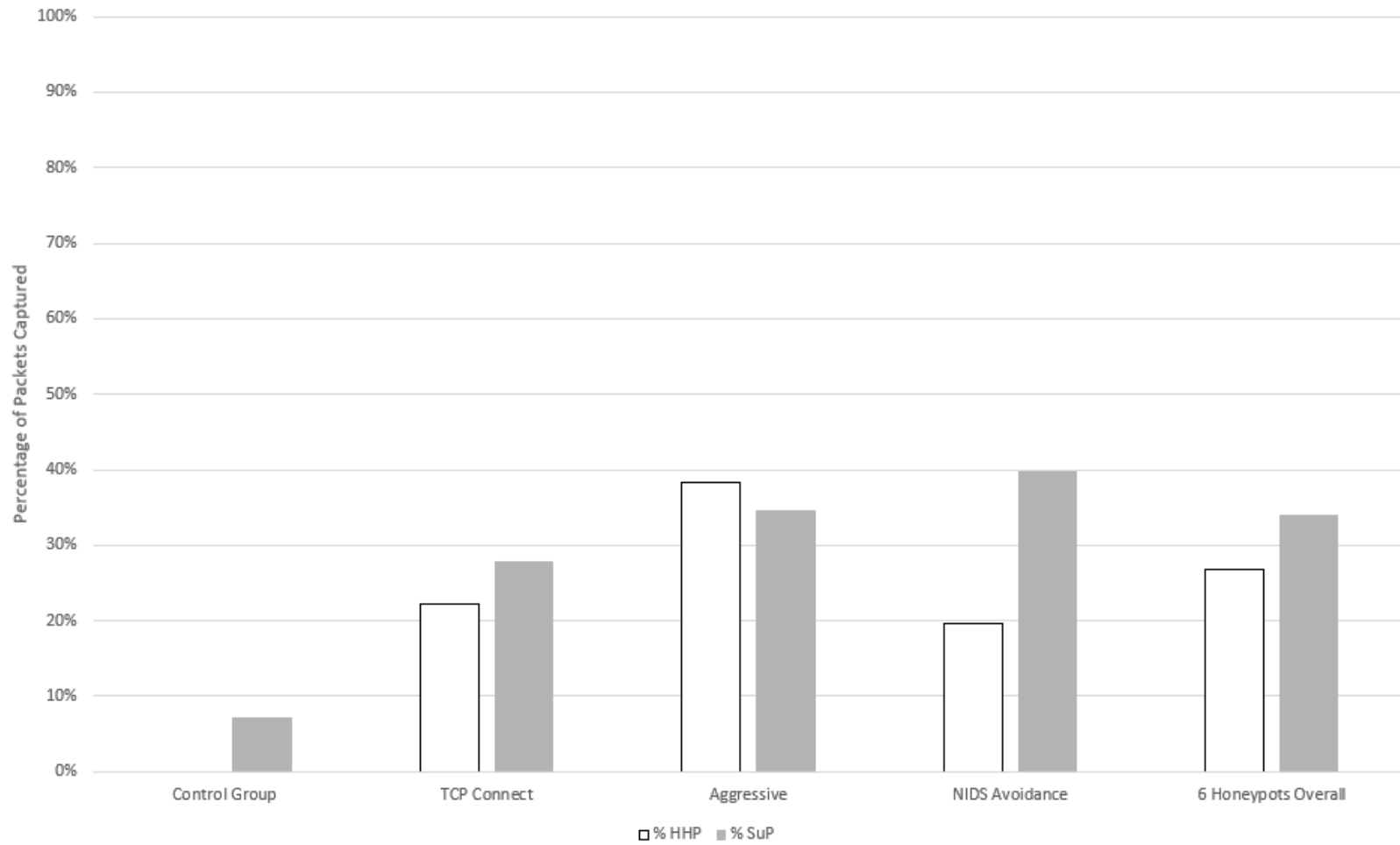
Figure 40. 6 Honeypots - Mean Packet Capture Percentage

5.3.4   9 Honeypots

With 9 honeypots active SAI approached SuA in the TCP Connect scan and significantly surpassed SuA in the Aggressive scan, shown in Figure 41. SuT remained on top and STM remained higher than the STI in the NIDS avoidance scan. SuT actually decreased in the number alerts generated from 6 honeypots to 9 honeypots in the Aggressive scan.

Figure 42 shows the average number of distinct alerts across different scans with 9 honeypots active. Noticeable changes are STM closing in on SuT and there no longer being a significant difference between the two, STM being higher in the NIDS Avoidance scan than the TCP Connect scan, and SuT being approximately the same in the TCP Connect and NIDS Avoidance scans, although TCP Connect is now higher. In fact, the average number of alerts actually decreases for SuT in the Aggressive and NIDS Avoidance scans.

In Figure 43, % HHP now surpasses % SuP in all scans except the NIDS Avoidance scan. Another noticeable difference is % SuP is now higher in the Agressive scan compared with the NIDS Avoidance scan. The percentage of packets captured by % SuP actually decreases for both the TCP Connect and NIDS Avoidance scans.
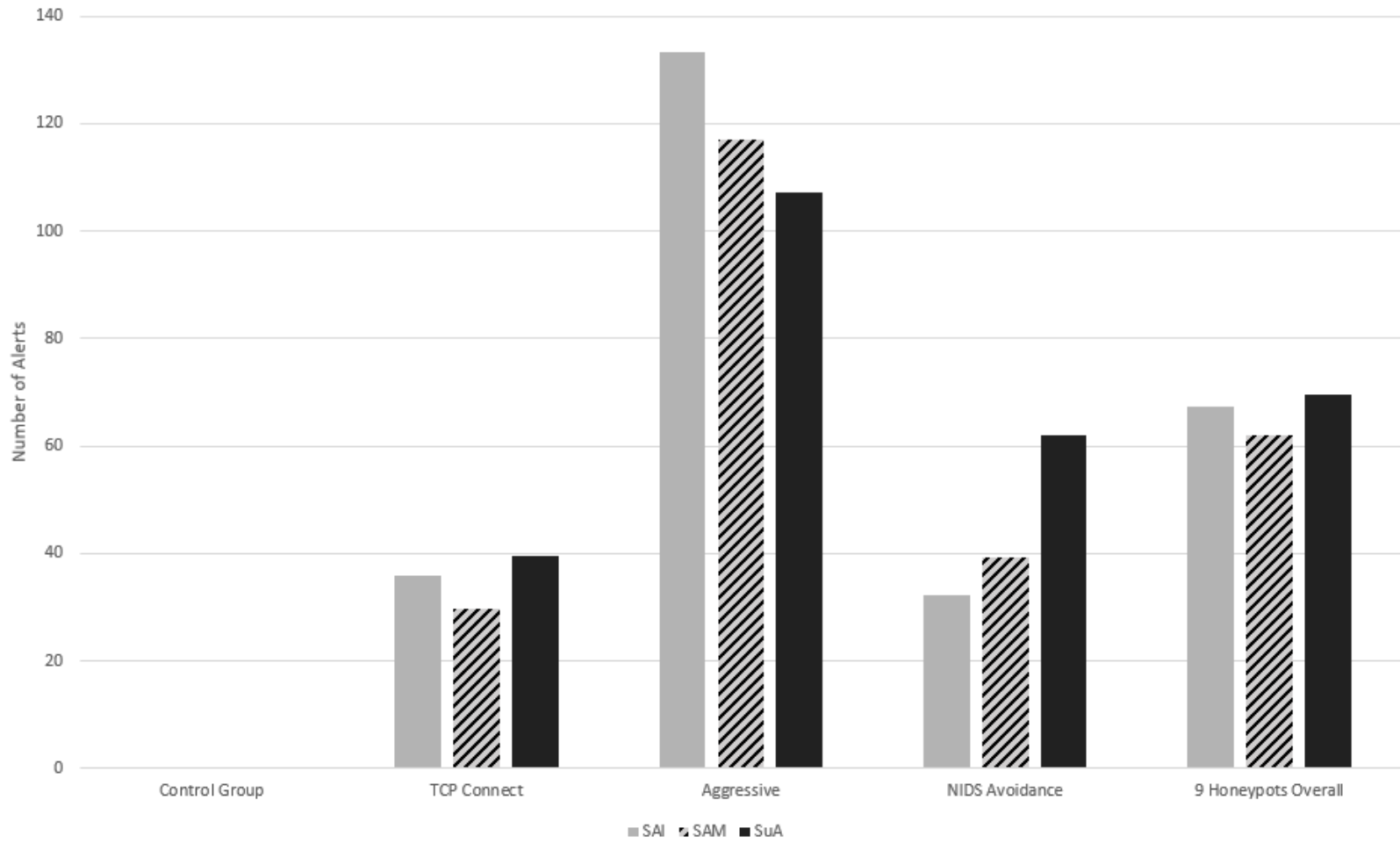
Figure 41. 9 Honeypots - Mean Number of Alerts
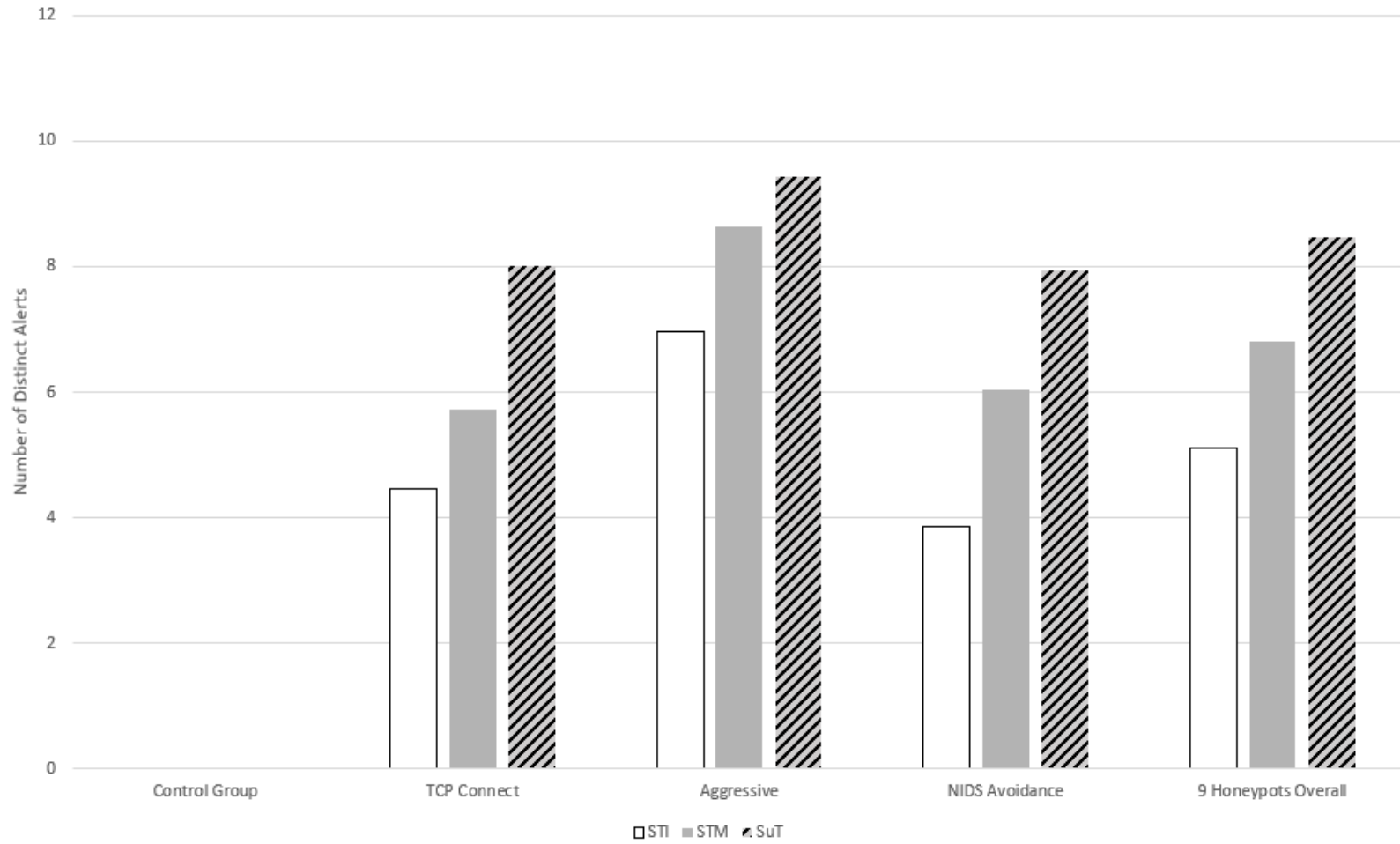
Figure 42. 9 Honeypots - Mean Number of Distinct Alerts
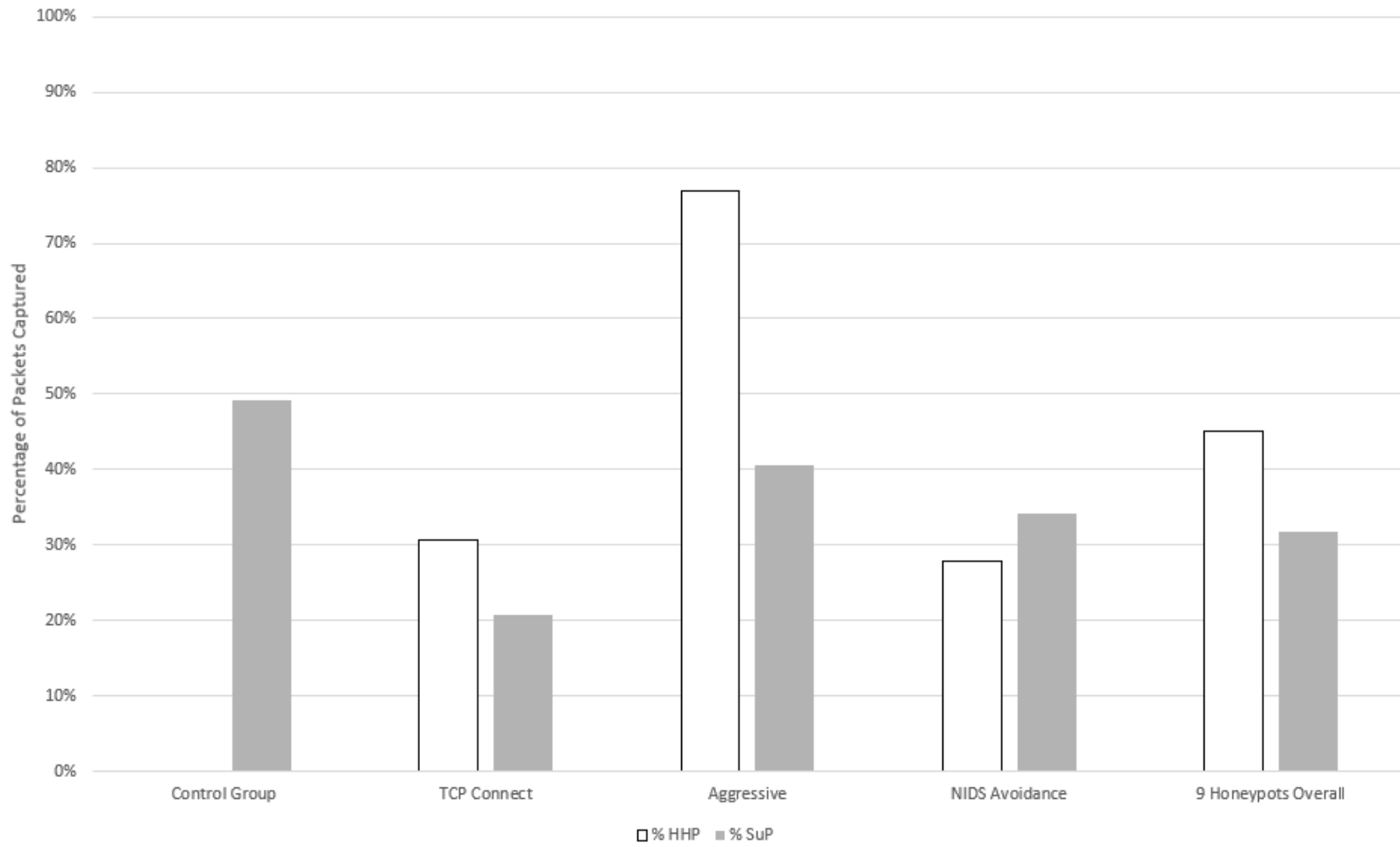
Figure 43. 9 Honeypots - Mean Packet Capture Percentage

5.4   Summary

This chapter presents and analyzes results from the experiment in Chapter 4. Results are explored with different grouping of factors to distinguish patterns and analyze the effects each factor has on the experiment. Explanations for observed patterns were also explored. Anderson-Darling tests for normality are completed on all grouping of factors to determine if a t-test was suitable for statistical testing. Because only one set was normal, a Permutation test is used to compare statistical significance with a significance level of 10%. SuA is statistically superior in all tests except for 9 honeypots overall and the TCP Connect and Aggressive scans with 9 honeypots. SAI is statistically superior to SuA in the Aggressive scan with 9 honeypots. STM is statistically superior to STI in all tests except the TCP Connect and NIDS Avoidance scans with 3 honeypots active. SuT statistically outperforms STM in all tests except the Aggressive scan with 9 honeypots active. Finally, % HHP is statistically significant in comparison to % SuP in the Aggressive Scan overall and all tests with 9 honeypots, except the NIDS Avoidance Scan. % SuP is statically significant in all NIDS Avoidance scans, the Aggressive scan with 3 honeypots, and 3 and 6 honeypots overall. Assuming the trendline continues, % HHP would perform better than % SuP with 12 and 15 honeypots in all scans.

VI. Conclusions

6.1   Introduction

In this chapter the conclusions drawn from the experimental results are discussed in Section 6.2 for each category of metrics, which includes the number of alerts, the number of distinct alerts, and the percentage of packets captured. The significance of this research is discussed in Section 6.3. Section 6.4 describes the limitations of this research. In Section 6.5, this thesis concludes with potential future work in the research field of honeypots, network intrusion detection, and the HoneyHive framework.

6.2   Research Conclusions

This research successfully creates a framework of distributed network intrusion detection IoT honeypot sensors that capture traffic, create alerts, and beacon back to a central C2 server. The first hypothesis from Chapter 1 is mostly supported with experiment results while the second is only partially supported by the trendline of experiment results:

1. The HoneyHive framework operates correctly by not alerting on routine network traffic and alerting on non-routine network traffic.

2. The HoneyHive framework detects intrusions that traditional NIDSs cannot through the use of distributed IoT honeypot sensors and packet capture aggregation.

6.2.1 Number of Alerts

The first hypothesis is a two part question, but both parts are supported by the experiment results. When No Scan is performed (the Control Group), no false positive alerts are created by HoneyHive across all runs and varying levels of honeypots. Therefore, HoneyHive operates correctly with routine network traffic.

The next part of the question requires delving into experimental results. While HoneyHive performed admirably in the majority of runs, there were 3/270 runs (runs 270, 291, and 465) that it did not create alerts (HHI) when it should have. Additionally, HoneyHive is currently using Snort for a higher level of signature matching and alert creation. However, Snort did not create alerts for 32/270 runs that it should have. This means roughly 10% of intrusions did not have successful signature matching performed on packet captures. This is either from Snort crashing, not finishing in a timely manner, simply not creating alerts, or an error in the HoneyHive framework. Although Snort is only an augmentation for the HoneyHive framework, it is used extensively in this research for generating alerts. Because of this, the HoneyHive framework, in its current configuration, alerts on non-routine traffic only around 90% of the time. For network intrusions, it is ideal for this to be as close to 100% as possible.

The second hypothesis is supported only partially by metrics and partially by trendlines. To start, the average number of alerts steadily increased as the number of honeypots increased. Even though the number of alerts in the HoneyHive framework only statistically exceeded Suricata in the Aggressive scan with 9 honeypots it was very close to Suricata in the TCP Connect scan with 9 honeypots. If this trend continued with the additions of honeypots, then it is expected that at 12 or 15 honeypots HoneyHive would surpass Suricata. Furthermore, if the experiment was modified to match an internal network scan that did not scan the DMZ, then Suricata would

not receive any non-routine traffic. This was not done in this experiment because then there would not have been any data to run statistical tests against. All this supports the hypothesis that distributed IoT honeypot sensors can detect intrusions that traditional NIDS cannot through packet capture aggregation.

6.2.2   Number of Distinct Types of Alerts

The second hypothesis is also supported by the number of distinct types of alerts from the experiment. While, HoneyHive did not ever exceed Suricata in the number of distinct alerts, Suricata was no longer statistically significant for the Aggressive scan with 9 honeypots. This once again suggests that with more honeypots HoneyHive could outperform Suricata. What is supportive though is that Snort PCAPs merged together (STM) created a larger number of distinct alerts with statistical significance for almost all tests compared with that of PCAPs parsed individually by Snort (STI). This supports the hypothesis because alerts that were not generated by analyzing each PCAP individually were generated when the PCAPs were merged and analyzed as one with statistical significance.

6.2.3   Percentage of Packets Captured

Finally, the percentage of packets captured to and from the attacker increased in all tests for the HoneyHive framework as the number of honeypots increased. The HoneyHive framework was statistically superior to Suricata with 9 honeypots active in all tests except the NIDS Avoidance scan. Assuming the trendline holds with % HHP, with more honeypots active, a higher percentage of attacker packets can be captured. Additionally, if areas of a network are scanned without a NIDS then a traditional NIDS will not see this traffic and not create alerts. Because HoneyHive is a distributed system, it can capture this traffic that a traditional NIDS cannot by

placing IoT honeypots in different enclaves throughout the network. This supports the second hypothesis that HoneyHive can detect intrusions that traditional NIDS cannot through the use of distributed IoT honeypots.

6.3   Research Significance

While there currently exist many NIDSs, none of the existing research explored in Chapter 2 is tailored to IoT honeypots. In addition, not all NIDSs alert in real time nor offer a complete view of the network with C2 capabilities. Furthermore, few offer automated distributed packet aggregation for intrusion analysis. The HoneyHive framework addresses all these shortcomings. The HoneyB Agent is also deployable to more than just IoT honeypots. It is deployable to any host with Python 2.7 installed. This makes it very versatile and offers immediate monitoring for the host with very little setup.

The HoneyHive framework offers benefits that allow the Air Force in defending the Air Force Network (AFNet) and the DoD to protect the DoD Information Network (DODIN). It can also be used for integration in CIKR-based networks since IoT devices share some similarities with ICS. Moreover, any company that works in the realm of network security could integrate HoneyHive into their existing network security. The impact of this framework is a cross-platform, standalone, NIDS / Network Monitoring solution capable of improving the rate at which network intrusions are detected. While HoneyHive will not be the solution for every network, it is another viable tool for increasing network security through intrusion detection.

6.4   Research Limitations

While the HoneyHive framework is a great success, there are several limitations in its current configuration. The IoT honeypots from Stafira used in this research

are low-interaction, web-based honeypots. Furthermore, they rely on Honeyd 1.5c and Python 2.7, both of which are at the end of their life [76]. Honeyd 1.5c was last updated in 2007 [10] and the current version, Honeyd 1.6d was last updated in 2013, but Stafira reported stability issues with Honeyd 1.6d [6][11]. The IoT devices that Stafira's IoT honeypots emulate are also not the most up to date or popular IoT devices in the United States [6]. Furthermore, not enough honeypots were used in this experiment and only Nmap scans were performed against them.

The HoneyB Agent script is also written in Python 2.7. To ensure future usability, it would need to be rewritten in Python 3.x or node.js. While this would not be too challenging, it could take several days or weeks to ensure correct functionality after the upgrade.

Currently the HoneyHive framework relies on a NIDS (Snort) and its community of rules to perform a higher level of signature detection as opposed to simply alerting based on interactions with honeypots. If the HoneyHive framework possessed its own self-contained, sophisticated combination of heuristics and signature matching then it would perform even more effectively and be easier to deploy to networks. Despite these limitations, the HoneyHive framework has great potential as a tool for network intrusion detection. In this experiment, Snort should have been used for both the HoneyHive framework and the DMZ NIDS listening on the SPAN port as it appears that Snort and Suricata do not create the same alerts given the same rule-set and network traffic.

The test network scans were performed on was of limited size whereas enterprise networks have hundreds if not thousands of devices. Additionally, all Honeypots were on the same network but enterprise networks often have different sub-networks comprising their internal network.

6.5   Future Work

One area of future work that would benefit the research of honeypots, network intrusion detection, and the HoneyHive framework is developing high-interaction honeypots in Docker (with secure configurations). Doing so would create more realistic honeypots and extend the capabilities of the HoneyHive framework. Memory dumps of an attacker's interaction with honeypots could be performed to capture TTPs and help identify them based on their tradecraft. In addition, more modern and popular versions of IoT devices could be developed as honeypots. This would allow HoneyHive to blend into more networks and not be as identifiable. The DB currently stores alert information, captured traffic, and is designed to store captured passwords, binaries, and memory dumps in the event that future work in HoneyHive is able to capture these.

A major component of future work for the HoneyHive framework includes the development of GUI to help network operators quickly identify and respond to intrusions. The proposed GUI design is shown in Figure 44 and can display alerts and honeypot interactions to network operators. The left side of the GUI displays the health status of the monitored honeypots, green representing no alerts, yellow scan alerts, red exploit detected, and black meaning unreachable. Honeypots are nested under their Honeyd controller, and the Honeyd controller takes on the most severe status of the honeypots under it.

The center displays a network map with the same status indicators shown on the left side. Notice that the device 192.168.45.42 is not a monitored honeypot, yet it is still flagged red. This is because an internal scan originating from the device has been detected, so the device itself is inferred to be compromised.

Individual alerts are viewable on the right side, and the actual packet capture is available for download by clicking the green down arrow. The two alerts currently

displayed show the TITAThink and ezOutlet2 honeypots being scanned internally by 192.168.45.42, which matches the color coding of what is shown by the honeypot status (left), and the network map (middle).
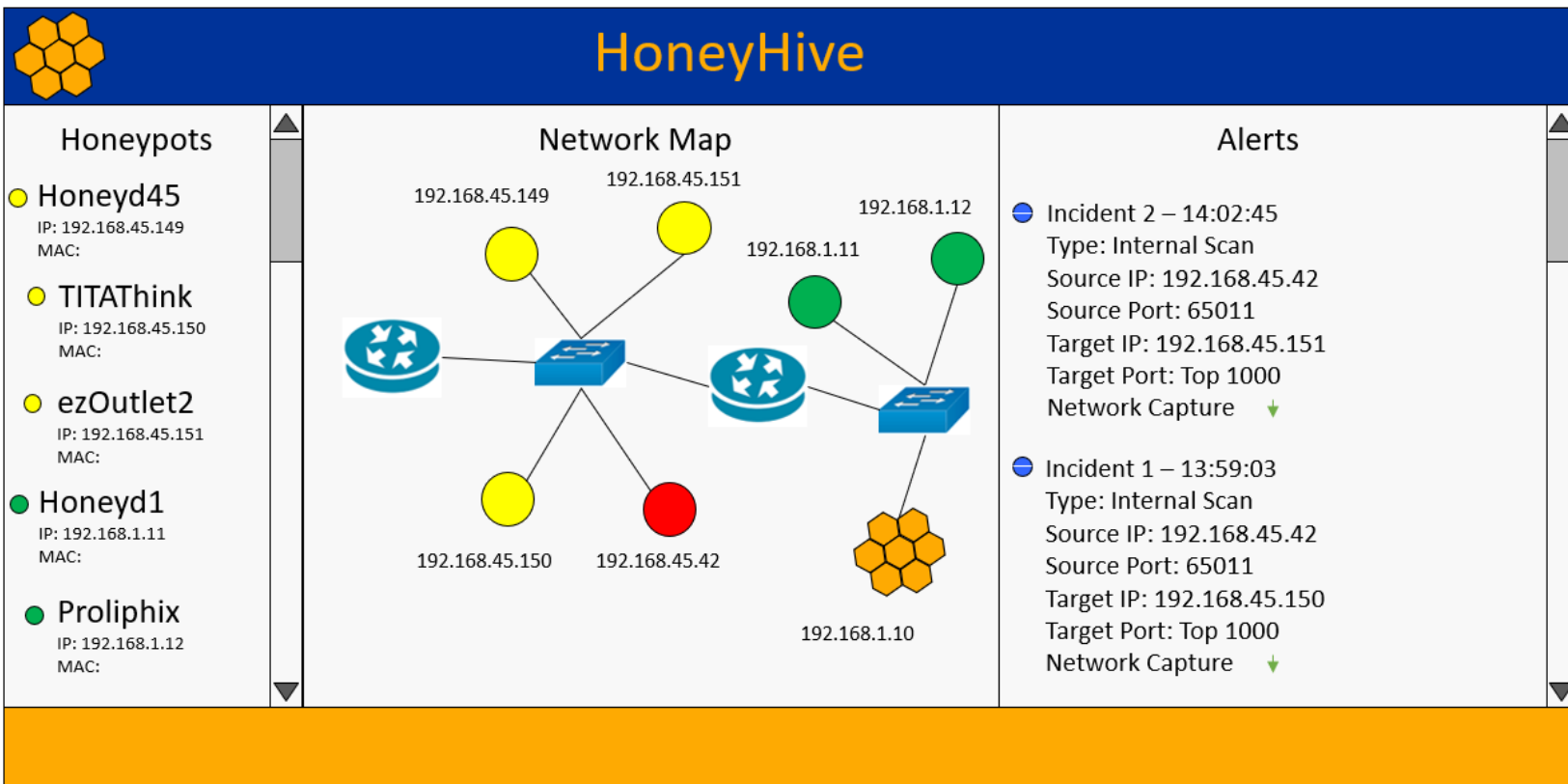
Figure 44. Proposed HoneyHive GUI

Other future work includes hashing and distributing signatures across a multi-server HoneyHive framework for faster detection of binaries, exploits, and TTPs. The HoneyB Agent script could also be improved to be a sophisticated and self-contained alert engine. In addition, the HoneyHive framework currently does not implement encryption of network traffic and device authentication, as it was not required for testing, but must before deployment. All traffic sent must be encrypted before network deployment to thwart sniffing. Symmetric encryption is the proposed implementation method of encryption, as opposed to asymmetric encryption, because distributing a single shared passphrase is easier than setting up a Certificate Authority (CA), generating two keys per device, and then registering all keys with the CA. Device authentication is also important to ensure the authenticity of traffic, security of the framework, and prevent erroneous alerts from overloading network operators.

If all these areas of future work are implemented then HoneyHive could be used for more than just intrusion detection. It could be used for cyber deception much like Cymmetria's MazeRunner framework [29].

Appendix A. HoneyHive Framework

main.js (C2 Sever)

```javascript
1  var cyan = '\x1b[36m%s\x1b[0m';
2  var green = '\x1b[32m%s\x1b[0m';
3  var blue = '\x1b[34m%s\x1b[0m';
4  var yellow = '\x1b[33m%s\x1b[0m';
5  var red = '\x1b[31m%s\x1b[0m';
6  var magenta = '\x1b[35m%s\x1b[0m';
7
8  var net = require('net');
9
10 var fs = require('fs');
11 var buffer = require('buffer');
12 var path = require("path");
13 var fork = require('child_process').fork;
14 const sqlite3 = require('sqlite3').verbose();
15 var exec = require('child_process').exec;
16
17 var encrypt_decrypt = require('./encrypt_decrypt.js');
18 var crypto = require('crypto');
19 var password = 'honeyhive';
20 var algorithm = 'aes-256-cbc';
21
22 var HOST = '0.0.0.0';
23 var PORT = 9830;
24 var resetCounter = 0;
25 var honeydIP = [];
26 var honeyPots = [];
27 var completeTransfers = [];
28 var alerts = [];
29 var srcIPs = {};
```

```
30  var dstIPs = {};

31  var srcPrts = {};

32  var dstPrts = {};

33  var percentage = 0.35;

34

35  var numInteractions = 0

36  var numSnortAlerts = 0;

37  var numPackets = 0;

38  var numSuricataAlerts = 0;

39  var numSuricataTypes = 0;

40  var suricataPackets = 0;

41  var numSnortAlerts_Merged = 0;

42  var numSnortTypes = 0;

43  var numSnortTypesMerged = 0;

44  var snortICount = 0;

45  var snortMCount = 0;

46

47  //Check for DB and create it if it doesn't exists

48  // spawns a child process to check / create DB

49  const database_creator = path.resolve("database_creator.js");

50

51  console.log(green, 'Checking Database FIle\n');

52

53  const params = [];

54  const options = {

55    stdio: [ 'pipe', 'pipe', 'pipe', 'ipc' ]

56  };

57

58  const database_child = fork(database_creator, params, options);

59

60  database_child.on('message', message =>{

61      console.log(green, 'message from Database Child:',  message);
```

```
62 });
63 //////////////////////////////////////////////////////////////////

64

65

66 function execute(command, callback){
67     exec(command, function(error, stdout, stderr){ callback(stdout);
        });
68 };

69

70 function countPackets(output)
71 {
72     numPackets += parseInt(output.split('Number of packets:')[1].
        trim())
73     console.log(green, "Num Packets: " + numPackets);
74 }

75

76 //Assumptions: Honeypots are not emitting malicious traffic / haven'
      t been compromised
77 function alertAnalyzer()
78 {
79   // threshold for number of honeypots that can be interacted with
        before an alert is generated
80   var hpThreshold = (Object.keys(dstIPs).length / (honeyPots.length)
        );

81

82   if(hpThreshold  > percentage)
83   {
84     createAlert('Alert: Multiple Honeypot Interaction Detected!')
85   }

86

87 }

88
```

146

```javascript
89  function createAlert(msg)
90  {
91    console.log(red, msg)
92  }
93
94  function parsePCAP(filename)
95  {
96    /*
97    "C:\Program Files\Wireshark\capinfos.exe" -c C:\Snort\log
       \192.168.1.152_2019-10-14_0939\192.168.1.152_2019-10-14_0939.pcap
98    File name:          C:\Snort\log\192.168.1.152_2019-10-14_0939
       \192.168.1.152_2019-10-14_0939.pcap
99    Number of packets:   1894
100   */
101   var cmd = '"C:\\Program Files\\Wireshark\\capinfos.exe" -c "C:\\
       Snort\\log\\' + filename
102   execute(cmd, countPackets);
103  }
104
105
106  function snortMerge(mergeOutput)
107  {
108    console.log('Starting Snort Parser on Merged File\n');
109    const snort_parser_merge = path.resolve("snort_parser.js");
110
111    const paramsMerge = [resetCounter, 0];
112    const optionsMerge = {
113      stdio: [ 'pipe', 'pipe', 'pipe', 'ipc' ]
114    };
115
116    const snort_child_merge = fork(snort_parser_merge, paramsMerge,
       optionsMerge);
```

```
117
118    snort_child_merge.on('message', message =>{
119        console.log(red, 'message from Snort Child Merged:',  message);
120          snortMCount +=1;
121          message.forEach(function(alert)
122          {
123              if(alert.Count != undefined)
124              {
125                numSnortAlerts_Merged += alert.Count;
126              }
127              numSnortTypesMerged +=1;
128          });
129      });
130
131      snort_child_merge.on('exit', (code) => {
132        snortMCount +=1;
133        console.log("Snort M Child Exited");
134      });
135
136 }
137
138 // looks at interfaces to automatically grab and bind on an IP
139 var os = require('os');
140 var ifaces = os.networkInterfaces();
141 var serverIP;
142
143 Object.keys(ifaces).forEach(function (ifname) {
144 var alias = 0;
145
146    ifaces[ifname].forEach(function (iface) {
147      if ('IPv4' !== iface.family || iface.internal !== false ||
       ifname.includes('VMware')) {
```

```
148        // skip over internal (i.e. 127.0.0.1) and non-ipv4 addresses
149        return;
150      }
151
152      if (alias >= 1) {
153        // this single interface has multiple ipv4 addresses
154        console.log(ifname + ':' + alias, iface.address);
155      } else {
156        // this interface has only one ipv4 adress
157        console.log(ifname, iface.address);
158        serverIP = iface.address;
159      }
160      ++alias;
161    });
162 });
163 console.log(cyan, "Sever IP: " +serverIP);
164
165
166 // Create a server instance, and chain the listen function to it
167 // The function passed to net.createServer() becomes the event
        handler for the 'connection' event
168 // The sock object the callback function receives UNIQUE for each
        connection
169 var server = net.createServer(function(sock) {
170
171    // Add a 'data' event handler to this instance of socket
172    sock.on('data', function(data) {
173      var JSONData = JSON.parse(data);
174
175      /*
176      header = {"Honeyd": honeydIP,
177             "Honeypots": honeypots
```

```
178              "MSG": 'AUTHENTICATE'}
179         */
180          if ( JSONData . MSG  == 'AUTHENTICATE ')
181         {
182            if (! honeydIP . includes ( JSONData . Honeyd ))
183            {
184              honeydIP . push ( JSONData . Honeyd );
185            }
186
187            JSONData . Honeypots . forEach ( function ( pot )
188            {
189                if (! honeyPots . includes ( pot ))
190                {
191                  honeyPots . push ( pot );
192                }
193            });
194
195            console . log ( green ,  "Connected Honeypots ");
196            console . log ( green ,  honeyPots );
197         }
198
199         // receive suricata alert count
200         else if ( JSONData . MSG  == 'SURICATA ')
201         {
202            numSuricataAlerts  +=  JSONData . NumAlerts ;
203            numSuricataTypes  +=  JSONData . NumTypes ;
204            suricataPackets  =  JSONData . NumPackets ;
205            console . log ( green ,  'Suricata Alerts Received: ' +
     numSuricataAlerts + ', ' + numSuricataTypes + ', ' +
     suricataPackets );
206         }
207
```

```
208          // reboot machine for fresh stable state
209          // C2 server should be relaunched automatically at startup
210          else if ( JSONData.MSG  == 'REBOOT')
211          {
212            // deletes all snort log files and then
213            // reboots when the cmd is finished
214            execute("del C:\\Snort\\log\\* /S /F /Q",
215            function(output){
216              execute("shutdown /g /f /t 0", function(){});
217            });
218          }
219
220          else if ( JSONData.MSG  == 'SNORT')
221          {
222            console.log("SNORT Command received, parsing PCAPS\n");
223            // parse unscanned  PCAPS
224            if(!(completeTransfers === undefined || completeTransfers.
    length == 0))
225              {
226
227              // mergecap -w outfile.pcapng dhcp-capture.pcapng imap-1.
    pcapng
228              // have to add all their dirs in front of the filename
    too
229              // then run through Snort
230              var cmd = '"C:\\Program Files\\Wireshark\\mergecap" -F
    pcap -w "C:\\Snort\\log\\merged'+resetCounter+'.pcap"';
231              completeTransfers.forEach(function(filename)
232              {
233                parsePCAP(filename);
234                cmd = cmd + ' ' + "C:\\Snort\\log\\"+filename;
235              });
```

151

```
236
237          console.log('Starting Snort Parser individually\n');
238          const snort_parser = path.resolve("snort_parser.js");
239
240          const params = [resetCounter, 1];
241          const options = {
242            stdio: [ 'pipe', 'pipe', 'pipe', 'ipc' ]
243          };
244
245          const snort_child = fork(snort_parser, params, options);
246
247          snort_child.on('message', message =>{
248            console.log(red, 'message from Snort Child
     individually:',  message);
249              message.forEach(function(alert)
250              {
251                if(alert.Count != undefined)
252                {
253                  numSnortAlerts += alert.Count;
254                }
255                numSnortTypes +=1;
256            });
257          });
258
259          snort_child.on('exit', (code) => {
260            snortICount +=1;
261            console.log("Snort I Child Exited");
262          });
263
264          if(completeTransfers.length > 1)
265          {
266            //function noop(){}
```

```
267            execute ( cmd , snortMerge );
268          }
269        }
270      }
271
272      else if ( JSONData . MSG  == 'RESET ')
273      {
274        // need to add a wait for all the snort processes to end
275        sock . write ( JSON . stringify ({ Interactions : numInteractions ,
    SnortICount : snortICount , Snort : numSnortAlerts , SnortTypes :
    numSnortTypes , SnortMCount : snortMCount , SnortMerged :
    numSnortAlerts_Merged , SnortTypesMerged : numSnortTypesMerged ,
    Packets : numPackets , Suricata : numSuricataAlerts , SuricataTypes :
    numSuricataTypes , suricataPackets : suricataPackets , numHoneypots :
     honeyPots . length , numPCAPs : completeTransfers . length }));
276
277        honeydIP = [];
278        honeyPots = [];
279        completeTransfers = [];
280        alerts = [];
281        srcIPs = {};
282        dstIPs = {};
283        srcPrts = {};
284        dstPrts = {};
285
286        snortICount = 0;
287        snortMCount = 0;
288        numInteractions = 0
289        numSnortAlerts = 0;
290        numPackets = 0;
291        numSuricataAlerts = 0;
292        numSuricataTypes = 0;
```

153

```
293         suricataPackets = 0;

294         numSnortAlerts_Merged =0;

295         numSnortTypes = 0;

296         numSnortTypesMerged =0;

297         resetCounter +=1;

298

299         fs.writeFile("C:\\Snort\\log\\pcaps.txt", '', function(){
      console.log('Snort PCAP File cleared\n')});

300         console.log(yellow, "Reset Received\n");

301       }

302

303     /*

304     header = {"Time": time.strftime("%Y-%m-%d_%H%M"),

305         "Honeyd": honeydIP,

306          "IP": '192.168.72.150',

307         "MSG": 'HEARTBEAT'}

308     */

309      else if(JSONData.MSG  == 'HEARTBEAT')

310      {

311        console.log(green, "Honeypot Heartbeat - Time: " + JSONData
      .Time + " Honeyd: "+ JSONData.honeydIP + " Honeypot IP: " +
      JSONData.IP);

312      }

313

314

315     /*

316     header = {"Time": time.strftime("%Y-%m-%d_%H%M"),

317         "TransLayer": transLayer,

318         "IP_SRC": pckt_src,

319         "IP_DST": pckt_dst,

320         "SPORT": sport,

321         "DPORT": dport,
```

```
322            "MSG": 'ALERT'}
323        */
324         else if(JSONData.MSG  == 'ALERT')
325          {
326            numInteractions +=1;
327            console.log(yellow, "Honeypot interaction detected \n\tTime
    : "+JSONData.Time+
328            "\n\tTransport Protocol: "+JSONData.TransLayer+
329            "\n\tIP SRC: "+JSONData.IP_SRC+
330            "\n\tSRC Port: "+JSONData.SPORT+
331            "\n\tIP DST: "+JSONData.IP_DST+
332             "\n\tDST Port: "+JSONData.DPORT);
333
334            if (JSONData.IP_SRC in srcIPs)
335             {
336               srcIPs[JSONData.IP_SRC] +=1;
337             }
338
339            else {
340               srcIPs[JSONData.IP_SRC] = 1;
341             }
342
343            //--------------------------
344            if (JSONData.SPORT in srcPrts)
345             {
346               srcPrts[JSONData.SPORT] +=1;
347             }
348
349            else {
350               srcPrts[JSONData.SPORT] = 1;
351             }
352
```

155

```
353          //---------------------------
354          if (JSONData.IP_DST in dstIPs)
355          {
356            dstIPs[JSONData.IP_DST] +=1;
357          }
358
359          else {
360            dstIPs[JSONData.IP_DST] = 1;
361          }
362
363          //---------------------------
364          if (JSONData.DPORT in dstPrts)
365          {
366            dstPrts[JSONData.DPORT] +=1;
367          }
368
369          else {
370            dstPrts[JSONData.DPORT] = 1;
371          }
372
373          alerts.push({Time: JSONData.Time, Protocol: JSONData.
       TransLayer, SrcIP: JSONData.IP_SRC, SrcPort:JSONData.SPORT, DstIP
       :JSONData.IP_DST, DstPort:JSONData.DPORT});
374          alertAnalyzer();
375
376       // spawns a child process to check / create DB
377       const database_inserter = path.resolve("database_inserter.
       js");
378
379       console.log(green, 'Adding to Database\n');
380
381       // would need to re-JSON-ify JSONData, so just sending the
```

156

```
382        // original data
383        const params = [data];
384        const options = {
385          stdio: [ 'pipe', 'pipe', 'pipe', 'ipc' ]
386        };

388        const inserter_child = fork(database_inserter, params,
     options);

390        inserter_child.on('message', message =>{
391            console.log(green, 'message from Inserter Child:',
     message);
392        });


394      }



397    // client is sending PCAP, open a new port for transfer state
398    // open a server / port for the file transfer to keep command
399    // data and binary data separate
400    else if(JSONData.MSG  == 'PCAP')
401    {
402      //console.log(green, JSONData.Filename + "\n" + JSONData.
     File_Size + "\n" + JSONData.MD5);
403      //var callbackPort = JSONData.listenPort
404      const transfer_server = path.resolve("transfer_server.js");
405      const params = [JSONData.Filename, JSONData.File_Size,
     JSONData.MD5, JSONData.Port];

407      const options = {
408        stdio: [ 'pipe', 'pipe', 'pipe', 'ipc' ]
409      };
```

```
410
411         const transferChild = fork(transfer_server, params, options)
    ;
412
413         transferChild.on('message', message =>{
414             console.log(green, 'message from transfer child:',
    message + '\n');
415             if(message == 'Download Complete')
416             {
417                 fs.appendFile("C:\\Snort\\log\\pcaps.txt", "C:\\Snort\\
    log\\"+JSONData.Filename+'\n', function (err) {
418                     if (err) throw err;
419                     console.log('PCAP File added to Snort File list \n');
420                 });
421                 completeTransfers.push(JSONData.Filename);
422
423             }
424         });
425       }
426       // end of PCAP
427
428     // end of on data
429     });
430
431     // Add a 'close' event handler to this instance of socket
432     sock.on('close', function(data) {
433         //console.log(yellow, 'CLOSED: ' + sock.remoteAddress +' '+
    sock.remotePort);
434     });
435 });
436
437 server.once('error', function(err) {
```

```
438    if (err.code === 'EADDRINUSE') {

439      console.log('Port Already in Use!: \n');

440      process.exit()

441    }

442 });

443

444 server.listen(PORT, HOST);

445

446 console.log(cyan, 'Server bound on ' + HOST +':'+ PORT);
```

transfer_server.js (Transfer Server)

```
1  const crypto = require ('crypto');
2  const downloadMD5 = crypto.createHash('md5');
3  var net = require('net');
4  var fs = require('fs');
5  var buffer = require('buffer');
6
7  var HOST = '0.0.0.0';
8
9  var params = process.argv;
10 var fileName = String(params.slice(2, 3));
11 var dir = "C:\\Snort\\log"; //\\" + fileName.split('.pcap')[0];
12 var file_size = params.slice(3, 4);
13 var md5 = params.slice(4, 5);
14 var PORT = parseInt(params.slice(5, 6));
15 var downloadedBytes = 0;
16
17
18 // test that args were being received correctly
19 //process.send("arguments: " + params.slice(2, 3) + "\n" + params.
       slice(3, 4) + "\n" + params.slice(4, 5));
20 //process.send("filename: " + params.slice(2, 3));
21
22 var fileStream = fs.createWriteStream(dir + '\\'+ fileName);
23
24
25 var server = net.createServer(function(sock) {
26
27   process.send('Downloading PCAP: \n');
28   downloadedBytes = sock.on('data', function(data) {
29
30     //fs.access(file, fs.constants.W_OK, (err) => {
```

160

```javascript
31    //   process.send('${file} ${err ? 'is not writable' : 'is
        writable'}');
32    //});
33    downloadedBytes += data.length;
34    fileStream.write(data);
35  });
36
37   sock.on('close', function(data) {
38     process.send('Download Complete');
39     //process.send("Downloaded: "+ downloadedBytes + " bytes");
40     server.close();
41     //md5Verification();
42     process.exit()
43   });
44 });
45
46 server.once('error', function(err) {
47   if (err.code === 'EADDRINUSE') {
48     process.send('Port Already in Use!: \n');
49     process.exit()
50   }
51 });
52
53 server.listen(PORT, HOST);
54 process.send('Transfer Server bound on ' + HOST +':'+ PORT);
```

snort_parser.js (Snort Log Parser)

```
1  // Run Snort on a PCAP
2
3  // Parse Snort alert log, combine like alerts,
4  // and other metada in JSON format (src IP, dst IP, port range)
5  // keep count of occurances
6  // dstIP should be the same since the pcaps are transfered as
      individually for each IP
7  //
8
9  // Send back results to main.js for compilation and monitoring
10 // which will have some threshold for alerting if so much traffic is
       seen
11 // sent to one host, or a smaller amount of traffic sent to multiple
       hosts
12
13 // Snort Alert Example
14 /*
15 [**] [1:10000005:2] NMAP TCP Scan [**]
16 [Priority: 0]
17 10/01-16:22:13.304233 192.168.1.11:61273 -> 192.168.1.150:80
18 TCP TTL:128 TOS:0x0 ID:28818 IpLen:20 DgmLen:52 DF
19 ******S* Seq: 0x5F62CB19  Ack: 0x0  Win: 0xFAF0  TcpLen: 32
20 TCP Options (6) => MSS: 1460 NOP WS: 8 NOP NOP SackOK
21 */
22
23 // Snort Alert Example
24 /*
25 [**] [1:2009582:3] ET SCAN NMAP -sS window 1024 [**]
26 [Classification: Attempted Information Leak] [Priority: 2]
27 10/30-14:41:29.472407 192.168.1.230:43923 -> 192.168.1.150:111
28 TCP TTL:56 TOS:0x0 ID:13589 IpLen:20 DgmLen:44
```

162

```
29  ******S* Seq: 0x15754519  Ack: 0x0  Win: 0x400  TcpLen: 24
30  TCP Options (1) => MSS: 1460
31  [Xref => http://doc.emergingthreats.net/2009582]
32  */
33
34  var fs = require('fs');
35  const readline = require('readline');
36  var exec = require('child_process').exec;
37  var params = process.argv;
38
39  var iteration = parseInt(params.slice(2, 3));
40  var dir = "C:\\Snort\\log\\";
41  var individually = parseInt(params.slice(3, 4));
42
43  var lineCounter = 0;
44  const numLinesSnortAlert = 8;
45  var snortLogResults = [];
46  var found = false;
47  var indx = 0;
48  var lastAlert = "";
49
50  if (!fs.existsSync(dir +  'individually' + iteration))
51  {
52      fs.mkdirSync(dir +  'individually' + iteration);
53      fs.mkdirSync(dir +  'merged' + iteration);
54  }
55
56  function execute(command, callback){
57    exec(command, function(error, stdout, stderr){
58      //process.send(stdout);
59      //process.send(stderr);
60       callback(stdout); });
```

```
61 };
62
63 var noop = function(){}; // do nothing.
64
65 function snortLog(output){
66    process.send(output);
67 }
68
69
70 async function processLineByLine() {
71    const fileStream = fs.createReadStream(dir+'\\alert.ids');
72
73    const rl = readline.createInterface({
74      input: fileStream,
75      crlfDelay: Infinity
76    });
77    // Note: we use the crlfDelay option to recognize all instances of
        CR LF
78    // ('\r\n') in input.txt as a single line break.
79
80    for await (const line of rl) {
81      try{
82        // Type of Alert:[**] [1:10000005:2] NMAP TCP Scan [**]
83        if (lineCounter%numLinesSnortAlert == 0)
84        {
85          var regex = /[\] ]([ 0-9A-z])+[ \[]/g;
86          var alert = line.match(regex);
87          var alertType = alert[1].replace(']', ' ');
88          alertType = alertType.replace('[', ' ');
89          alertType = alertType.trim();
90
91          // checks to see if alert is same as prev, so we can skip
```

```
    searching
92        if(alertType == lastAlert)
93        {
94          snortLogResults[indx].Count +=1;
95        }

96

97        else{
98          found = false;
99          for(var i = 0; i < snortLogResults.length; i++)
100         {
101             if (snortLogResults[i].Type == alertType)
102             {
103                 found = true;
104                 indx = i;
105                 snortLogResults[i].Count +=1;
106                 break;
107             }
108          }

109

110         if (found == false) {
111           snortLogResults.push({Type: alertType, Count: 1,
     StartDate: "", EndDate: "", StartTime: "", EndTime: "", Src: [],
     Dst:[]});
112           found = true;
113           indx = snortLogResults.length-1;
114           lastAlert = alertType;
115         }
116       }
117     }

118

119     // Priority:[Priority: 0]
120     //else if (lineCounter%numLinesSnortAlert == 1)
```

```javascript
121        //{
122            // Each line in input.txt will be successively available
       here as 'line'.
123            //process.send('Line from file: ${line}');
124        //}
125
126        // Timestamp and Src -> Dst '10/01-16:22:13.304233
       192.168.1.11:61273 -> 192.168.1.150:80'
127        else if (lineCounter%numLinesSnortAlert == 2)
128        {
129          var dateSrcDst = line.split(' ');
130          var dateTime = dateSrcDst[0].split('-');
131          var date = dateTime[0];
132          var time = dateTime[1].split('.')[0];
133
134          var src = dateSrcDst[1];
135          var srcSplit = src.split(':');
136          var srcIP = srcSplit[0];
137          var srcPrt = parseInt(srcSplit[1], 10);
138
139
140          var dst = dateSrcDst[3];
141          var dstSplit = dst.split(':');
142          var dstIP = dstSplit[0];
143          var dstPrt = parseInt(dstSplit[1], 10);
144
145          // first entry
146          if(snortLogResults[indx].StartDate == "")
147          {
148            snortLogResults[indx].StartDate = date;
149            snortLogResults[indx].StartTime = time;
150          }
```

```
151

152        else {
153          snortLogResults[indx].EndDate = date;
154          snortLogResults[indx].EndTime = time;
155        }

156

157        if(!snortLogResults[indx].Src.includes(src))
158        {
159          snortLogResults[indx].Src.push(src);
160        }

161

162        if(!snortLogResults[indx].Dst.includes(dst))
163        {
164          snortLogResults[indx].Dst.push(dst);
165        }
166        //process.send(snortLogResults);
167      }

168

169      // TCP TTL:128 TOS:0x0 ID:28818 IpLen:20 DgmLen:52 DF
170      //else if (lineCounter%numLinesSnortAlert == 3)
171      //{
172        // Each line in input.txt will be successively available
      here as 'line'.
173        //process.send('Line from file: ${line}');
174      //}

175

176      // ******S* Seq: 0x5F62CB19  Ack: 0x0  Win: 0xFAF0  TcpLen: 32
177      //else if (lineCounter%numLinesSnortAlert == 4)
178      //{
179        // Each line in input.txt will be successively available
      here as 'line'.
180      //  process.send('Line from file: ${line}');
```

167

```
181        //}
182
183        // TCP Options (6) => MSS: 1460 NOP WS: 8 NOP NOP SackOK
184        else if (lineCounter%numLinesSnortAlert == 5)
185        {
186          // has 5 lines for alert instead of 6 or 7
187          // modify by 2 to catch up
188          if(!(line.includes('TCP Options')))
189          {
190            lineCounter +=2;
191          }
192        }
193
194        else if (lineCounter%numLinesSnortAlert == 6)
195        {
196          // has 6 lines for alert instead of 7
197          // modify by 1 to catch up
198          if(!(line.includes('Xref')))
199          {
200            lineCounter +=1;
201          }
202        }
203
204        // blank line between alerts
205        //else if (lineCounter%numLinesSnortAlert == 7)
206        //{
207
208        //}
209        lineCounter +=1;
210      }
211    process.send(snortLogResults);
212  }
```

```
213  catch ( err )
214  {
215    process . send ( err )
216  }
217  // process . exit ()
218 }
219
220 var cmd = '"C:\\Snort\\bin\\snort.exe" -c "C:\\Snort\\etc\\snort.
       conf" ';
221 if ( individually == 1)
222 {
223   dir = dir + 'individually' + iteration;
224   cmd = cmd + '--pcap-file "C:\\Snort\\log\\pcaps.txt" --pcap-reset
       -l '+ dir;
225 }
226 else {
227   dir = dir + 'merged' + iteration;
228   cmd = cmd + '-r "C:\\Snort\\log\\merged'+iteration+'.pcap" -l '+
       dir;
229
230 }
231
232 execute ( cmd , processLineByLine );
```

honeyB_Agent.py (HoneyB Agent)

```python
1  import subprocess
2  import thread
3  import socket
4  import json
5  import time
6  import os
7  import base64
8  from datetime import datetime, date, time, timedelta
9
10 from scapy.all import *
11
12 # color honey yellow is #a98307
13 port = 9830
14 transferPort = 9831
15 honeyHiveIP = '192.168.1.233'
16 honeydIP = '192.168.1.154'
17 honeypots = ["192.168.1.150", "192.168.1.151", "192.168.1.152"]
18 connections = {}
19 timeOutSeconds = 9000
20 sessionTimeout = timedelta(seconds=timeOutSeconds)
21 autoTransferTimeout = timedelta(seconds=timeOutSeconds)
22 connections_Lock = thread.allocate_lock()
23 honeyd = None
24 devnull = open(os.devnull, 'wb')
25 alertMode = False
26
27 def main():
28     global honeyd
29
30     pcap_monitor_id = thread.start_new_thread(pcapMonitor,  ())
31     #heartbeat_id = thread.start_new_thread(heartbeat,  ())
```

```python
32
33     # I want this in my main console output
34     # once run, it casues blocking
35     print "Scapy Packet Sniffer Engaged"
36     sniff(iface="eth0", prn=processPacket, store=0)
37
38 def pcapMonitor():
39     while True:
40         connections_Lock.acquire()
41         uct = datetime.utcnow()
42         remove = []
43         for ip in connections:
44             if (uct - connections[ip].get("time") >=
        autoTransferTimeout):
45                 transferFile(ip, connections[ip].get('filename'))
46                 remove.append(ip)
47                 print "Automatic Transfer"
48         for i in remove:
49             connections.pop(i)
50         connections_Lock.release()
51         floatSeconds = timeOutSeconds*1.0
52         time.sleep(floatSeconds)
53
54 # method called when a packet is received
55 # parses the packet to identify the dest ip, port, ect.
56 # will implement scan recognition and attacker pivoting
57 def processPacket(packet):
58
59     #automatically named files based on ip, date, and time
60     now = datetime.now()
61     uct = datetime.utcnow()
62
```

```
63    # ignore traffic not to honeypots
64    if packet.haslayer(IP) and (not packet[IP].dst in honeypots) and
      (not packet[IP].src in honeypots) and (not packet[IP].dst ==
      honeydIP) and (not packet[IP].src == honeydIP):
65        None
66
67    # ignore 9830 and 9831 because they are packets we are sending
68    elif packet.haslayer(TCP) and (packet[TCP].dport == port or
      packet[TCP].dport == transferPort or packet[TCP].sport == port or
       packet[TCP].sport == transferPort):
69        None
70
71    # ignores some Ubuntu traffic that is not being filtered out my /
      etc/hosts
72    elif packet.haslayer(UDP) and (packet[UDP].sport == 68 or packet[
      UDP].dport == 68 or packet[UDP].dport == 5353 or packet[UDP].
      dport == 53 or packet[UDP].sport == 53):
73        None
74
75    elif packet.haslayer(UDP) and (packet[UDP].dport == 9830):
76        print "Command Received"
77        print now.strftime("%Y-%m-%d_%H%M")
78        runCommand(packet[UDP].load.strip('\n'))
79
80    # ensures this packet has an IP
81    # log non 9830 and 9831 traffic from c2 server1
82    elif packet.haslayer(IP) and alertMode:
83        pckt_src=packet[IP].src
84        pckt_dst=packet[IP].dst
85        pckt_ttl=packet[IP].ttl
86
87
```

```python
88   #
     #######################################################################

89        # if the honeypot doesnt already have an ongoing pcap log file
90        # and the time since last packet is less than 5 minutes
91        connections_Lock.acquire()

92

93        if any(x in connections for x in[pckt_dst, pckt_src]):
94            ip = ""
95            filename = ""

96

97            # determine which ip was in the connections list
98            if pckt_dst in connections:
99                ip = pckt_dst

100

101            else:
102                ip = pckt_src

103

104            # time difference is less than than session timeout
105            # therefore, keep logging to file
106            # and update its timestamp to the new time
107            # new time - oldtime < allowed time
108            if(uct - connections[ip].get("time") < sessionTimeout):
109                filename = connections[ip].get("filename")
110                connections[ip].update({"time": uct})

111

112            # else there was a timeout so need a new file
113            # there was a connection at one point so the ip
114            # is in our honeypot list, no need to check again
115            # update time of connection to now
116            else:
117                print "Transfering PCAP"
```

173

```
118            #if packet.haslayer(TCP):
119            #   print "Packet: Source IP %s, Port %s\n Dest IP %s,
      Port %s" % (pckt_src, packet[TCP].sport, pckt_dst, packet[TCP].
      dport)
120            #else:
121            #   print "Packet: Source IP %s\n Dest IP %s" % (
      pckt_src, pckt_dst)
122            transferFile(ip, connections[ip].get('filename'))
123
124            # create the new pcap file
125            filename = ip + "_" + now.strftime("%Y-%m-%d_%H%M") +".
      pcap"
126            #print "New filename: " + filename
127            #print connections[ip]
128            connections[ip].update({"time": uct})
129            connections[ip].update({"filename": filename})
130            #print connections[ip]
131            alert(packet, now)
132
133          #appends packet to output file
134          wrpcap(filename, packet, append=True)
135
136
137      # ips not in connected list, but this may be first interaction
138      # with the honeypots. Want to assume first captured packet
      will
139      # be sent to honeypot, but compromised hp could beacon out if
140      # hardware or software make it
141      elif any(x in honeypots for x in[pckt_dst, pckt_src]):
142          ip = ""
143
144          # determine which ip was in the connections list
```

174

```python
145             if pckt_dst in honeypots:
146                 ip = pckt_dst
147             else:
148                 ip = pckt_src
149
150             # create the new pcap file
151             filename = ip + "_" + now.strftime("%Y-%m-%d_%H%M") +".pcap
    "
152             #print "filename: " + filename
153             connections[ip] = {"filename": filename, "time": uct}
154             alert(packet, now)
155
156             #appends packet to output file
157             wrpcap(filename, packet, append=True)
158         connections_Lock.release()
159
160
161 # receive and execute commands
162 def runCommand(msg):
163     global connections
164     global honeyd
165     global alertMode
166
167     print msg
168
169     # transfer all current pcaps
170     if(msg == "TRANSFER"):
171         connections_Lock.acquire()
172         for ip in connections:
173             transferFile(ip, connections[ip].get('filename'))
174         # return all values to 0
175         connections = {}
```

```python
176         connections_Lock.release()
177
178     elif(msg == "RESET"):
179         # transfer all current pcaps
180         connections_Lock.acquire()
181         for ip in connections:
182             transferFile(ip, connections[ip].get('filename'))
183         # return all values to 0
184         connections = {}
185         connections_Lock.release()
186
187         # kill all honeyd
188         if(honeyd != None):
189             subprocess.Popen(["sudo killall honeyd"], stdin=None,
    stdout=devnull, stderr=devnull, shell=True)
190
191         honeyd = None
192         alertMode = False
193
194     # stops program completely
195     elif(msg == "KILL"):
196         # kill all honeyd
197         if(honeyd != None):
198             subprocess.Popen(["sudo killall honeyd"], stdin=None,
    stdout=devnull, stderr=devnull, shell=True)
199
200         # kill this script
201         subprocess.Popen(["sudo killall python"], stdin=None, stdout=
    devnull, stderr=devnull, shell=True)
202
203     # starts just honeyd
204     elif(msg == "START"):
```

176

```python
205         # relaunch honeyd
206         if( honeyd == None ):
207             honeyd = subprocess.Popen(["sudo /home/edge/Desktop/
       honeyhive/scripts/startHoney.sh"], stdin=None, stdout=devnull,
       stderr=devnull, shell=True)
208         alertMode = True
209
210         # authenticate
211         authenticate()
212
213     # stops just honeyd
214     elif( msg == "STOP" ):
215         # kill all honeyd
216         if( honeyd != None ):
217             subprocess.Popen(["sudo killall honeyd"], stdin=None,
       stdout=devnull, stderr=devnull, shell=True)
218         honeyd = None
219         alertMode = False
220
221     elif( msg == "REBOOT" ):
222         subprocess.call(["sudo", "reboot"])
223
224 def heartbeat():
225     # header information to send server
226     # python dictonary
227     header = {"Time": time.strftime("%Y-%m-%d_%H%M"),
228         "Honeyd": honeydIP,
229         "IP": '192.168.72.150',
230         "MSG": 'HEARTBEAT'}
231
232     # converts dictionary to json format
233     jsonHeader = json.dumps(header)
```

```
234
235    try:
236        # attempts to connect to the server and tell it to open a
       transfer port
237        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
238        server_address = (honeyHiveIP, port)
239        sock.connect(server_address)
240
241        sock.sendall(jsonHeader)
242
243    except socket.error, e:
244        print "Error creating Heartbeat socket: %s" %e
245
246    #clean-up actions that must be executed under all circumstances
247    finally:
248        sock.close()
249
250 def authenticate():
251
252    # header information to send server
253    # python dictonary
254    header = {"Honeyd": honeydIP,
255        "Honeypots": honeypots,
256       "MSG": 'AUTHENTICATE'}
257
258    # converts dictionary to json format
259    jsonHeader = json.dumps(header)
260
261    try:
262        # attempts to connect to the server and tell it to open a
       transfer port
263        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```python
264         server_address = ( honeyHiveIP , port )
265         sock . connect ( server_address )
266
267         sock . sendall ( jsonHeader )
268
269     except socket . error , e :
270         print " Error creating Authenticate socket : %s " %e
271
272     #clean-up actions that must be executed under all circumstances
273     finally :
274         sock . close ()
275
276
277 def alert ( packet , time ) :
278     sport = 0
279     dport = 0
280     transLayer = 'TCP '
281     if packet . haslayer ( TCP ) :
282         sport=packet [ TCP ] . sport
283         dport=packet [ TCP ] . dport
284
285     elif packet . haslayer ( UDP ) :
286         sport=packet [ UDP ] . sport
287         dport=packet [ UDP ] . dport
288         transLayer = 'UDP '
289
290     elif packet . haslayer ( ICMP ) :
291         transLayer = 'ICMP '
292
293     else :
294         transLayer = 'Other '
295
```

```python
296    pckt_src=packet[IP].src
297    pckt_dst=packet[IP].dst
298    pckt_ttl=packet[IP].ttl
299
300    # header information to send server
301    # python dictonary
302    header = {"Time": time.strftime("%Y-%m-%d_%H%M"),
303        "TransLayer": transLayer,
304      "IP_SRC": pckt_src,
305      "IP_DST": pckt_dst,
306      "SPORT": sport,
307      "DPORT": dport,
308      "MSG": 'ALERT'}
309
310    # converts dictionary to json format
311    jsonHeader = json.dumps(header)
312
313    try:
314      # attempts to connect to the server and tell it to open a
    transfer port
315      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
316      server_address = (honeyHiveIP, port)
317      sock.connect(server_address)
318
319      sock.sendall(jsonHeader)
320
321    except socket.error, e:
322      print "Error creating Alert socket: %s" %e
323
324    #clean-up actions that must be executed under all circumstances
325    finally:
326      sock.close()
```

```python
327
328
329  #sends transfer file command to the server
330  def transferFile(ip, filename):
331
332      # run command and return output
333      hashPcap = subprocess.check_output(["md5sum", filename])
334      # header information to send server
335      # python dictonary
336      header = {"Honeyd": honeydIP,
337               "IP": ip,
338               "Filename": filename,
339               "File_Size": os.path.getsize(filename),
340               "MD5": hashPcap.split(' ')[0],
341               "MSG": 'PCAP',
342               "Port": transferPort}
343
344      # converts dictionary to json format
345      jsonHeader = json.dumps(header)
346
347      try:
348          # attempts to connect to the server and tell it to open a
      transfer port
349          sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
350          server_address = (honeyHiveIP, port)
351          sock.connect(server_address)
352
353          sock.sendall(jsonHeader)
354
355      except socket.error, e:
356          print "Error creating Transfer Server socket: %s" %e
357
```

```python
358    #clean-up actions that must be executed under all circumstances
359    finally:
360        sock.close()
361
362    # allows time for the C2 server to receive and start the transfer
        server
363    # on the requested port
364    time.sleep(3)
365
366    try:
367        transSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
368        transfer_address = (honeyHiveIP, transferPort)
369        transSock.connect(transfer_address)
370
371        # reads and sends the file to the server
372        f = open (filename, "rb")
373        l = f.read(1024)
374
375        while (l):
376            # encrypt data before sending
377            transSock.send(l)
378            l = f.read(1024)
379
380        f.close()
381        subprocess.call(['rm', '-f', filename])
382
383    except socket.error, e:
384        print "Error creating Transfer File socket: %s" %e
385
386    #clean-up actions that must be executed under all circumstances
387    finally:
388        transSock.close()
```

```
389
390 main ()
```

Appendix B. Honeyd Configuration File

iotHoneyd.conf

```
1 create default

2 set default default tcp action block

3 set default default udp action block

4 set default default icmp action block

5

6 create titacamera

7 set titacamera personality "Linux 2.3.28-33"

8 set titacamera default tcp action reset

9 add titacamera tcp port 80 "TitaCamera/camera_web.sh"

10 add titacamera tcp port 554 open

11 add titacamera tcp port 49152 open

12 add titacamera udp port 443 filtered

13 add titacamera udp port 990 filtered

14 add titacamera udp port 1900 filtered

15 add titacamera udp port 1901 filtered

16 add titacamera udp port 3702 open

17 add titacamera udp port 16896 filtered

18 add titacamera udp port 18676 filtered

19 add titacamera udp port 19956 filtered

20 add titacamera udp port 22986 filtered

21 add titacamera udp port 30697 filtered

22 add titacamera udp port 32772 filtered

23 add titacamera udp port 32777 filtered

24

25 create proliphixthermostat

26 set proliphixthermostat personality "D-Link Print Server"
```

```
27 set proliphixthermostat default tcp action reset
28 #Regular Thermostat Interface when Internet is available
29 #add proliphixthermostat tcp port 80
       "ProliphixThermostat/thermostat_web.sh"
30 #Thermostat Interface when no internet is available
31 add proliphixthermostat tcp port 80
       "ProliphixThermostat/thermostat_web_nointernet.sh"
32
33 create ezoutlet
34 set ezoutlet personality "IBM OS/2 Warp 4.0"
35 set ezoutlet default tcp action reset
36 add ezoutlet tcp port 80 "ezOutlet/outlet_web.sh"
37
38 set titacamera ethernet "7C:DD:90:B0:22:82"
39 bind 192.168.45.150 titacamera
40 #dhcp titacamera on eth0
41 set proliphixthermostat ethernet "00:11:49:00:62:46"
42 bind 192.168.45.151 proliphixthermostat
43 #dhcp proliphixthermostat on eth0
44 set ezoutlet ethernet "00:03:EA:0E:11:67"
45 bind 192.168.45.152 ezoutlet
46 #dhcp ezoutlet on eth0
```

Appendix C. suricataConnect.py

suricataConnect.py

```python
1  import subprocess
2  import socket
3  import json
4  import os
5
6  os.sys.path.append('/usr/local/bin/scapy')
7  from scapy.all import *
8
9  # color honey yellow is #a98307
10 port = 9830
11 honeyHiveIP = '192.168.1.233'
12 suricataIP = '192.168.1.231'
13 attacker = '192.168.1.230'
14 suricataRunning = False
15 packetRXStart = 0
16 packetTXStart = 0
17 totalPackets = 0
18 devnull = open(os.devnull, 'wb')
19
20 def main():
21     print "Scapy Packet Sniffer Engaged"
22     sniff(iface="eth0", prn=processPacket, store=0)
23
24 def processPacket(packet):
25     global totalPackets
26     if packet.haslayer(IP) and (packet[IP].src == attacker or packet[
       IP].dst == attacker):
27         totalPackets += 1
28     #running on mirrored port so make sure it's for this IP
```

```
29    if packet.haslayer(UDP) and packet[UDP].dport == 9830 and packet[
      IP].dst == suricataIP:

30        print "Command Received"

31        runCommand(packet[UDP].load.strip('\n'))

32

33 # receive and execute commands

34 def runCommand(msg):

35    global suricataRunning

36    global packetRXStart, packetTXStart

37    global totalPackets

38

39    logGrep = subprocess.Popen(['sudo', 'grep', '-i', 'scan', '/var/
      log/suricata/fast.log'], stdout=subprocess.PIPE)

40

41    wc = subprocess.Popen(['wc', '-l'], stdin=logGrep.stdout, stdout=
      subprocess.PIPE)

42    out, err = wc.communicate()

43    numAlerts = int(out)

44

45    typeCountGrep = subprocess.Popen(['sudo', 'grep', '-i', 'scan', '
      /var/log/suricata/fast.log'], stdout=subprocess.PIPE)

46

47    cutRuleSig = subprocess.Popen(['cut', '-f', '4', '-d', " "],
      stdin=typeCountGrep.stdout, stdout=subprocess.PIPE)

48    sort = subprocess.Popen(['sort'], stdin=cutRuleSig.stdout, stdout
      =subprocess.PIPE)

49    uniq = subprocess.Popen(['uniq'], stdin=sort.stdout, stdout=
      subprocess.PIPE)

50    wcTypes = subprocess.Popen(['wc', '-l'], stdin=uniq.stdout,
      stdout=subprocess.PIPE)

51    out2, err2 = wcTypes.communicate()

52    numAlertTypes = int(out2)
```

187

```python
53
54     print msg
55
56     # transfer current scan alert count and clear the log
57     if(msg == "TRANSFER"):
58         # packets sent and received after run complete
59         countRXEnd = subprocess.Popen(['cat', '/sys/class/net/eth0/
       statistics/rx_packets'], stdout=subprocess.PIPE, close_fds=True)
60         packetRXEnd, err = countRXEnd.communicate()
61         countTXEnd = subprocess.Popen(['cat', '/sys/class/net/eth0/
       statistics/tx_packets'], stdout=subprocess.PIPE, close_fds=True)
62         packetTXEnd, err = countTXEnd.communicate()
63
64         packetCount = int(packetRXEnd) + int(packetTXEnd) - int(
       packetRXStart) - int(packetTXStart)
65
66         # send alert count
67         suricata(numAlerts, numAlertTypes, totalPackets)
68
69         print "Alerts: "+ str(numAlerts) + str(numAlertTypes) + str(
       packetCount)
70
71         # clear log file
72         os.system("sudo rm /var/log/suricata/fast.log")
73         subprocess.call(['sudo', 'touch', '/var/log/suricata/fast.log'
       ])
74
75     elif(msg == "RESET"):
76         # packets sent and received after run complete
77         countRXEnd = subprocess.Popen(['cat', '/sys/class/net/eth0/
       statistics/rx_packets'], stdout=subprocess.PIPE, close_fds=True)
78         packetRXEnd, err = countRXEnd.communicate()
```

```python
79      countTXEnd = subprocess.Popen(['cat', '/sys/class/net/eth0/
    statistics/tx_packets'], stdout=subprocess.PIPE, close_fds=True)
80      packetTXEnd, err = countTXEnd.communicate()
81      packetCount = int(packetRXEnd) + int(packetTXEnd) - int(
    packetRXStart) - int(packetTXStart)
82
83      # send alert count
84      suricata(numAlerts, numAlertTypes, totalPackets)
85
86      print "Alerts: "+ str(numAlerts) + str(numAlertTypes) + str(
    packetCount)
87      # finds and kills suricata based on PID
88      if(suricataRunning):
89          subprocess.Popen(['sudo ./killSuricata.sh'], stdout=
    subprocess.PIPE, shell=True)
90          suricataRunning = False
91
92      # clear log file
93      os.system("sudo rm /var/log/suricata/fast.log")
94      subprocess.call(['sudo', 'touch', '/var/log/suricata/fast.log'
    ])
95
96  # stops program completely
97  elif(msg == "KILL"):
98      if(suricataRunning):
99          subprocess.Popen(['sudo ./killSuricata.sh'], stdout=
    subprocess.PIPE, shell=True)
100
101     # kill this script
102     subprocess.Popen(["sudo killall python"], stdin=None, stdout=
    devnull, stderr=devnull, shell=True)
103
```

```
104    # starts just suricata
105    elif(msg == "START"):
106        countRXStart = subprocess.Popen(['cat', '/sys/class/net/eth0/
       statistics/rx_packets'], stdout=subprocess.PIPE, close_fds=True)
107        packetRXStart, err = countRXStart.communicate()
108        countTXStart = subprocess.Popen(['cat', '/sys/class/net/eth0/
       statistics/tx_packets'], stdout=subprocess.PIPE, close_fds=True)
109        packetTXStart, err = countTXStart.communicate()
110
111        # launch suricata
112        # sudo suricata -c /etc/suricata/suricata.yaml -i eth0
113        if(not suricataRunning):
114            #subprocess.Popen(["sudo "], stdin=None, stdout=devnull,
       stderr=devnull, shell=True)
115            subprocess.Popen(["sudo suricata -c /etc/suricata/suricata.
       yaml -i eth0"], stdin=None, stdout=devnull, stderr=devnull, shell
       =True)
116            suricataRunning = True
117
118    # stops just suricata
119    elif(msg == "STOP"):
120        # kill suricata
121        # sudo kill $(ps aux | grep '[s]udo suricata -c /etc/suricata/
       suricata.yaml -i eth0' | awk '{print $2}')
122        if(suricataRunning):
123            subprocess.Popen(['sudo ./killSuricata.sh'], stdout=
       subprocess.PIPE, shell=True)
124            suricataRunning = False
125
126    elif(msg == "REBOOT"):
127        subprocess.call(["sudo", "reboot"])
128
```

```python
129  def suricata ( numAlerts , numTypes , numPackets ) :
130      # header information to send server
131      # python dictonary
132      header = { "NumAlerts" : numAlerts ,
133                 "NumTypes" : numTypes ,
134                 "NumPackets" : numPackets ,
135                 "MSG" : 'SURICATA' }
136
137      # converts dictionary to json format
138      jsonHeader = json . dumps ( header )
139
140      try :
141          sock = socket . socket ( socket . AF_INET , socket . SOCK_STREAM )
142
143          # creates a socket for the connection to the server
144          server_address = ( honeyHiveIP , port )
145
146          # attempts to connect to the server
147          sock . connect ( server_address )
148          sock . sendall ( jsonHeader )
149
150      except socket . error , e :
151          print "Error creating Suricata socket : %s" %e
152
153      finally :
154          # closes connection
155          sock . close ()
156
157  main ()
```

Appendix D. runExperiment.py

runExperiment.py

```python
1  import subprocess
2  import sys
3  import socket
4  import json
5  import time as t
6  import random
7  import os
8  from datetime import datetime, date, time, timedelta
9
10 devnull = open(os.devnull, 'wb')
11
12 rasPi = '192.168.1.231'
13
14 honeyd1 = '192.168.1.154'
15 honeyd2 = '192.168.1.164'
16 honeyd3 = '192.168.1.174'
17 honeyd4 = '192.168.1.184'
18
19 port = 9830
20 honeyHiveIP = '192.168.1.233'
21 network = '192.168.1.150-192'
22
23 camera = ['192.168.1.190', '192.168.1.150', '192.168.1.170', '
       192.168.1.160']
24 thermostat = ['192.168.1.191', '192.168.1.151', '192.168.1.171', '
       192.168.1.161']
25 outlet =  ['192.168.1.192', '192.168.1.152', '192.168.1.172', '
       192.168.1.162']
26
```

```python
27 order = []
28 experimentFile = 'experiment.txt'
29 expRunNum = 0
30 numRuns = 30
31
32 def main():
33     global order
34
35     # checks to see if a file with the run order exists and makes
       sure it's not empty
36     if os.path.exists(experimentFile) and os.path.getsize(
       experimentFile) > 0:
37         f = open(experimentFile, 'r')
38         for line in f:
39             value = line.split()
40             lambdaFunc = None
41             print value
42             # address of the functions can change between runs
43             # better to just check the name than store the address
       value
44             if value[0] == 'nmapScan':
45                 lambdaFunc = nmapScan
46             elif value[0] == 'baseline':
47                 lambdaFunc = baseline
48             elif value[0] == 'wget':
49                 lambdaFunc = wget
50
51             # appends all experiments in the file to be run
52             order.append({"Lambda": lambdaFunc, "Name": value[0], "
       HoneyPots": int(value[1]), "Level": int(value[2])})
53         f.close()
54     else:
```

```python
    for i in range(numRuns):
        #runExperiment(i+1)
        #order = []
        randomizeOrder()
    random.shuffle(order)
    f = open(experimentFile, 'a')
    for run in order:
        f.write(run['Name'] + ' ' + str(run['HoneyPots']) + ' ' +
str(run['Level']) + '\n')
    f.close()

    # removes very last newline '\n' so that there isn't a blank
line at the end of the file
    fT = open(experimentFile, 'rb+')
    fT.seek(-1, os.SEEK_END)
    fT.truncate()
    fT.close()
runExperiment(1)

def runExperiment(runNumber):
    global expRunNum

    for run in order:
        # starts the specified number of honeypots, and suricata
        print "Sending Honeypot Starts"
        sendCmd(run["HoneyPots"], "START\n")
        sendCmd(run["HoneyPots"], "START\n")
        sendCmd(run["HoneyPots"], "START\n")

        # allows enough time for everything to reach a stable state
        # takes about 30 seconds for suricata to startup on the
rasberry pi
```

```
84          t.sleep (40.0)
85
86          # packets sent and received before run
87          # cat /sys/class/net/enp2s0/statistics/rx_packets
88          # RX number of packets received
89          # TX number of packets transmitted
90          # need to cat both for full picture
91          countRXStart = subprocess.Popen(['cat', '/sys/class/net/
      enp2s0/statistics/rx_packets'], stdout=subprocess.PIPE, close_fds
      =True)
92          packetRXStart, err = countRXStart.communicate()
93          countTXStart = subprocess.Popen(['cat', '/sys/class/net/
      enp2s0/statistics/tx_packets'], stdout=subprocess.PIPE, close_fds
      =True)
94          packetTXStart, err = countTXStart.communicate()
95
96          #print packetCountStart
97
98          start = datetime.now()
99          print "Start Time:" + start.strftime("%Y-%m-%d_%H%M")
100         print "nmapScan" + " Level: " + str(run["Level"]) + '
      Honeypots: ' + str(run["HoneyPots"])
101
102         # runs the specified test with corresponding level
103         run["Lambda"](run["Level"], run["HoneyPots"])
104
105         # packets sent and received after run complete
106         countRXEnd = subprocess.Popen(['cat', '/sys/class/net/enp2s0
      /statistics/rx_packets'], stdout=subprocess.PIPE, close_fds=True)
107         packetRXEnd, err = countRXEnd.communicate()
108         countTXEnd = subprocess.Popen(['cat', '/sys/class/net/enp2s0
      /statistics/tx_packets'], stdout=subprocess.PIPE, close_fds=True)
```

```python
109        packetTXEnd, err = countTXEnd.communicate()
110
111        #print packetCountEnd
112        packetCount = int(packetRXEnd) + int(packetTXEnd) - int(
       packetRXStart) - int(packetTXStart)
113        print 'Packets Sent and Received: '+ str(packetCount)
114
115        end = datetime.now()
116        print "End Time:" + end.strftime("%Y-%m-%d_%H%M")
117        elapsedTime = (end - start).total_seconds()
118        print "Elapsed Time:" + str(elapsedTime)
119
120        print "Sending Honeypot Resets"
121
122        # stops all honeypots and forces transfer of pcap, stops
       suricata
123        sendCmd(run["HoneyPots"], "RESET\n")
124        sendCmd(run["HoneyPots"], "RESET\n")
125        sendCmd(run["HoneyPots"], "RESET\n")
126
127        # time required for honeypots and suricata to send all data
       to c2 server
128        t.sleep(45.0)
129
130        sendSnort()
131
132        # allows time for Snort to parse PCAP files
133        t.sleep(90.0)
134
135        # tells the C2 server to reset and send gathered stats to
       this script
136        res = sendReset()
```

```
137
138         # write gathered results to corresponding csv file
139         writeResults(run, res, packetCount, elapsedTime)
140
141   #removes run from experimentFile since it successfully completed
142         lines = open(experimentFile).readlines()
143         with open(experimentFile, 'w') as f:
144             f.writelines(lines[1:])
145         f.close()
146
147         # Reboots all machines for a clean stable start state
148         print "Sending Honeypot Reboots"
149         sendCmd(run["HoneyPots"], "REBOOT\n")
150         sendCmd(run["HoneyPots"], "REBOOT\n")
151         sendCmd(run["HoneyPots"], "REBOOT\n")
152
153         sendReboot()
154
155         # allows plenty of time for all devices to reboot
156         # and startup scripts before proceeding to next iteration
157         # win 10 VM takes longest (30 seconds for reboot, another 15
      for c2 server startup - 45s total)
158         # sometimes can take longer and crash script....
159         t.sleep(45.0)
160
161         # makes sure c2 servere is up and running berfore starting
      next iteration
162         sendStart()
163         expRunNum += 1
164
165
166 # randomizes the order in which all 36 tests are run
```

```
167 def randomizeOrder():
168     # inital testing
169     #test = [{"Lambda": wget, "Name": 'wget'}]
170
171     test = [{"Lambda": nmapScan, "Name": 'nmapScan'}] #, {"Lambda":
        baseline, "Name": 'baseline'}] #, {"Lambda": wget, "Name": 'wget
        '}]
172     for i in range(4):
173         numHps = i;
174         for testType in test:
175             for level in range(4):
176                 order.append({"Lambda": testType["Lambda"], "Name":
        testType['Name'], "HoneyPots": numHps, "Level": level})
177     #random.shuffle(order)
178
179
180 # tests to see if alerts are generated for different kinds of scans
181 # Levels are the different scan types:
182     # nmap 192.168.1.0/24 -sT
183     # nmap 192.168.1.0/24 -A
184     # sudo nmap 192.168.1.0/24 --max-hostgroup 1 --randomize-hosts -
        f 8
185 def nmapScan(scanType, numHps):
186     global expRunNum
187     if (scanType == 0):
188         print "Control Group: sleep(1341)"
189         t.sleep(1341)
190     elif (scanType == 1):
191         print "nmap ipLst.txt -sT -Pn"
192         subprocess.call(['nmap', '-iL', 'ipLst.txt', '-sT', '-Pn', '
        -oX', 'run'+str(expRunNum)+'.xml'], stdin=None, stdout=None,
        stderr=None, shell=False)
```

198

```
193    elif (scanType == 2):
194        print "nmap ipLst.txt -A -Pn"
195        subprocess.call(['nmap', '-iL', 'ipLst.txt', '-A', '-Pn', '-
       oX', 'run'+str(expRunNum)+'.xml'], stdin=None, stdout=None,
       stderr=None, shell=False)
196    elif (scanType == 3):
197        print "sudo nmap ipLst.txt --scan-delay 1075ms --randomize-
       hosts -f 8 -Pn"
198        subprocess.call(['sudo', 'nmap', '-iL', 'ipLst.txt', '--scan
       -delay', '1075ms', '--randomize-hosts', '-f', '8', '-Pn', '-oX',
       'run'+str(expRunNum)+'.xml'],
199            stdin=None, stdout=None, stderr=None, shell=False)
200
201
202
203 # tests to see if an alert is generated for malicious wgets that
       modify IoT device
204 # Levels are the different typse of devices:
205    # TITAThink Camera 192.168.1.1[5-8]0
206    # Prolophix Thermostat 192.168.1.1[5-8]1
207    # ez-Outlet 2 Power Outlet 192.168.1.1[5-8]2
208 def wget(iotDevice, numHps):
209    if (iotDevice == 0):
210        print "Control Group: sleep(30)"
211        t.sleep(30)
212    elif (iotDevice == 1):
213        for ip in range (numHps + 1):
214            subprocess.call(['wget', '--user', 'admin', '--password'
       , 'admin', '--tries', '1', '--timeout', '1', 'http://'+camera[ip
       ]+'/form/deleteStorageAllApply?lang=en'],
215                stdin=None, stdout=None, stderr=None, shell=False)
216        print "wget  TITAThink Cameras"
```

```python
217         elif (iotDevice == 2):
218             print "wget Prolophix Thermostats"
219             for ip in range (numHps + 1):
220                 subprocess.call(['wget', '--user', 'admin', '--password',
        'admin', '--tries', '1', '--timeout', '1', 'http://'+thermostat[
        ip]+'/index.shtml'],
221                     stdin=None, stdout=None, stderr=None, shell=False)
222         elif (iotDevice == 3):
223             print "wget ez-Outlets"
224             for ip in range(numHps + 1):
225                 subprocess.call(['wget', '--tries', '1', '--timeout', '1'
        , 'http://'+outlet[ip]+'/invert.cgi'],
226                     stdin=None, stdout=None, stderr=None, shell=False)
227
228
229 # tests to see if alerts are genereated with no alert traffic
230 # levels: 20s, 660s, 6000s - to match length of each scan (from
        pilot studies)
231 def baseline(runTime, numHps):
232     if (runTime == 1):
233         print "Baseline 30s"
234         t.sleep(30.0)
235     elif (runTime == 2):
236         print "Baseline 660s"
237         t.sleep(660.0)
238     else:
239         print "Baseline 1500s"
240         t.sleep(1500.0)
241
242
243 def sendCmd(numHPs, cmd):
244     # 1 non-mirrored
```

```python
245     if (numHPs == 1):
246         hp1 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
247         hp1.sendto(cmd, (honeyd1, port))
248         hp1.close()
249
250     # 1 non-mirrored, 1 mirrored
251     elif (numHPs == 2):
252         hp1 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
253         hp1.sendto(cmd, (honeyd1, port))
254         hp1.close()
255
256         hp3 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
257         hp3.sendto(cmd, (honeyd3, port))
258         hp3.close()
259
260     # 2 non-mirrored, 1 mirroed
261     elif (numHPs == 3):
262         hp1 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
263         hp1.sendto(cmd, (honeyd1, port))
264         hp1.close()
265
266         hp2 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
267         hp2.sendto(cmd, (honeyd2, port))
268         hp2.close()
269
270         hp3 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
271         hp3.sendto(cmd, (honeyd3, port))
272         hp3.close()
273
274     # 2 non-mirrored, 2 mirrored
275     elif (numHPs == 4):
276         hp1 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```python
277          hp1.sendto(cmd, (honeyd1, port))
278          hp1.close()
279
280          hp2 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
281          hp2.sendto(cmd, (honeyd2, port))
282          hp2.close()
283
284          hp3 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
285          hp3.sendto(cmd, (honeyd3, port))
286          hp3.close()
287
288          hp4 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
289          hp4.sendto(cmd, (honeyd4, port))
290          hp4.close()
291
292      # starts suricata on raspberry pi
293      ras = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
294      ras.sendto(cmd, (rasPi, port))
295      ras.close()
296
297
298  def sendStart():
299      print "START sent to C2 Server"
300      while True:
301          try:
302              sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
303
304              # creates a socket for the connection to the server
305              server_address = (honeyHiveIP, port)
306
307              # attempts to connect to the server
308              sock.connect(server_address)
```

```python
309
310          # header information to send server
311          # python dictonary
312          header = {"MSG": 'START'}
313
314          # converts dictionary to json format
315          jsonHeader = json.dumps(header)
316
317          # sends all data
318          sock.sendall(jsonHeader)
319
320      except socket.error, e:
321          print "Error creating Start Socket: %s" %e
322          continue
323      break
324      sock.close()
325
326
327  def sendSnort():
328      print "SNORT sent to C2 Server"
329      while True:
330          try:
331              sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
332
333              # creates a socket for the connection to the server
334              server_address = (honeyHiveIP, port)
335
336              # attempts to connect to the server
337              sock.connect(server_address)
338
339              # header information to send server
340              # python dictonary
```

```python
            header = {"MSG": 'SNORT'}

            # converts dictionary to json format
            jsonHeader = json.dumps(header)

            # sends all data
            sock.sendall(jsonHeader)

        except socket.error, e:
            print "Error creating Snort Socket: %s" %e
            continue
        break
        sock.close()

def sendReboot():
    print "REBOOT Sent to C2 Server"
    while True:
        try:
            sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

            # creates a socket for the connection to the server
            server_address = (honeyHiveIP, port)

            # attempts to connect to the server
            sock.connect(server_address)

            # header information to send server
            # python dictonary
            header = {"MSG": 'REBOOT'}

            # converts dictionary to json format
            jsonHeader = json.dumps(header)
```

```
373
374             # sends all data
375             sock.sendall(jsonHeader)
376
377         except socket.error, e:
378             print "Error creating Reboot Socket: %s" %e
379             continue
380         break
381         sock.close()
382
383
384 def sendReset():
385     print "RESET sent to C2 Server"
386     while True:
387       try:
388         sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
389
390         # creates a socket for the connection to the server
391         server_address = (honeyHiveIP, port)
392
393         # attempts to connect to the server
394         sock.connect(server_address)
395
396         # header information to send server
397         # python dictonary
398         header = {"MSG": 'RESET'}
399
400         # converts dictionary to json format
401         jsonHeader = json.dumps(header)
402
403         # sends all data
404         sock.sendall(jsonHeader)
```

```python
405          #{Interactions: numInteractions, Snort: numSnortAlerts,
     Packets: numPackets}
406          res = sock.recv(4096)
407
408      except socket.error, e:
409          print "Error creating Reset Socket: %s" %e
410          continue
411      break
412    sock.close()
413
414    return json.loads(res)
415
416 def writeResults(runInfo, results, packetCount, elapsedTime):
417
418     print "Writing Results"
419     # add suricata packet count and suricata num types of alerts
     count
420     output = str(runInfo['Level']) + ',' + str(runInfo['HoneyPots'])
      + ','  + str(results['numHoneypots']) + ',' + str(results['
     Interactions']) + ',' + str(results['numPCAPs']) + ',' +  str(
     results['SnortICount']) + ',' +  str(results['SnortMCount']) + ',
     ' +  str(results['Snort']) + ',' + str(results['SnortTypes']) + '
     ,' + str(results['SnortMerged']) + ','  + str(results['
     SnortTypesMerged']) + ','  + str(results['Suricata']) + ',' + str
     (results['SuricataTypes']) + ',' + str(packetCount) + ',' + str(
     results['Packets']) + ',' + str(results['suricataPackets']) + ','
      + str(elapsedTime) + '\n'
421     print "Output: " + output
422
423     #nmap
424     if (runInfo['Name'] == 'nmapScan'):
425         nmap_file = open('nmap.csv', mode='a')
```

206

```python
426        nmap_file.write(output)
427        nmap_file.close()
428

429    #baseline
430    elif (runInfo['Name'] == 'baseline'):
431        baseline_file = open('baseline.csv', mode='a')
432        baseline_file.write(output)
433        baseline_file.close()
434

435    # wget
436    else:
437        wget_file = open('wget.csv', mode='a')
438        wget_file.write(output)
439        wget_file.close()
440

441 main()
```

Appendix E. Experiment Results

Table 13. Experiment Results

Trial	*ST*	*HP*	HHI	**SAI**	**STI**	**SAM**	**STM**	**SuA**	**SuT**	HHP	SuP	**% HHP**	**% SuP**	AP	ET
1	*NIDS Avoidance*	*0*	0	0	0	0	0	31	8	0	18104	0%	32%	56978	2365
2	*Control Group*	*6*	0	0	0	0	0	0	0	0	96	0%	7%	1436	1341
3	*NIDS Avoidance*	*0*	0	0	0	0	0	30	6	0	15447	0%	27%	57756	2365
4	*TCP Connect*	*0*	0	0	0	0	0	31	8	0	3892	0%	12%	31239	312
5	*TCP Connect*	*6*	6	35	5	33	7	40	8	10706	9110	33%	28%	32030	129
6	*Aggressive*	*3*	3	0	0	0	0	105	12	13508	31709	2%	4%	893271	1093
7	*Aggressive*	*0*	0	0	0	0	0	125	12	0	32153	0%	4%	842658	679
8	*Control Group*	*3*	0	0	0	0	0	0	0	0	89	0%	7%	1218	1341
9	*Aggressive*	*0*	0	0	0	0	0	115	14	0	31803	0%	4%	857500	1086
10	*Aggressive*	*6*	5	0	0	0	0	124	12	13456	19050	7%	11%	181337	1829
11	*NIDS Avoidance*	*3*	3	0	0	0	0	28	5	7942	13030	10%	16%	79264	6459
12	*TCP Connect*	*6*	6	27	5	25	7	30	6	8490	5569	28%	18%	30590	35
13	*Aggressive*	*3*	3	61	7	69	9	107	11	11647	31299	2%	5%	673715	1163
14	*Aggressive*	*0*	0	0	0	0	0	112	12	0	32658	0%	4%	727012	1593
15	*TCP Connect*	*6*	6	32	5	33	8	44	8	10434	9281	33%	29%	31725	139
16	*TCP Connect*	*6*	6	33	5	28	7	38	8	10209	7751	33%	25%	30671	37
17	*NIDS Avoidance*	*9*	9	24	4	33	7	63	8	32051	36139	32%	36%	100940	2367
18	*Control Group*	*6*	0	0	0	0	0	0	0	0	95	0%	7%	1437	1341

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
19	Aggressive	6	5	93	8	87	10	54	9	18912	12077	7%	5%	267195	1089
20	NIDS Avoidance	6	6	12	4	16	5	62	8	19988	36159	23%	41%	87887	2365
21	Aggressive	9	8	67	6	62	8	53	7	16992	12077	52%	37%	32767	1096
22	Aggressive	0	0	0	0	0	0	112	12	0	31907	0%	5%	702602	698
23	Aggressive	3	3	55	7	55	9	111	12	14269	31707	2%	4%	712468	838
24	Aggressive	0	0	0	0	0	0	117	12	0	32710	0%	5%	699017	622
25	NIDS Avoidance	3	3	0	0	0	0	29	6	13444	18133	19%	26%	69043	2366
26	TCP Connect	3	3	21	5	21	7	31	8	2413	2120	6%	6%	37184	404
27	NIDS Avoidance	6	6	0	0	0	0	65	8	19797	36120	23%	41%	87561	2367
28	Control Group	0	0	0	0	0	0	0	0	0	173	0%	17%	1045	1341
29	Aggressive	3	3	0	0	0	0	109	12	13053	31143	2%	4%	721156	742
30	TCP Connect	9	9	13	5	16	5	45	10	5732	9613	15%	24%	39352	288
31	Control Group	9	0	0	0	0	0	0	0	0	93	0%	8%	1164	1341
32	TCP Connect	3	3	8	5	9	5	31	8	2381	3588	9%	14%	25923	79
33	NIDS Avoidance	9	9	17	5	19	7	66	8	21364	36139	21%	35%	101819	2523
34	Control Group	3	0	0	0	0	0	0	0	0	80	0%	7%	1096	1341
35	Control Group	3	0	0	0	0	0	0	0	0	65	0%	6%	1035	1341
36	Aggressive	9	8	126	8	118	8	77	9	22445	12080	60%	32%	37418	1108
37	TCP Connect	3	3	6	4	8	5	28	8	2607	2516	7%	7%	35691	398
38	TCP Connect	3	3	8	5	8	5	20	6	2319	1802	9%	7%	24792	47

Table 13 continued from previous page

Trial	*ST*	*HP*	HHI	**SAI**	**STI**	**SAM**	**STM**	**SuA**	**SuT**	HHP	SuP	**% HHP**	**% SuP**	AP	ET
39	*Control Group*	*9*	0	0	0	0	0	0	0	0	96	0%	9%	1051	1341
40	*TCP Connect*	*6*	6	0	0	0	0	43	8	0	12008	0%	29%	41541	1325
41	*Aggressive*	*9*	6	220	9	204	9	67	7	30154	12663	83%	35%	36358	1346
42	*Control Group*	*9*	0	0	0	0	0	0	0	0	98	0%	8%	1173	1341
43	*NIDS Avoidance*	*9*	9	24	4	35	7	65	8	32047	36187	32%	36%	100513	2364
44	*Aggressive*	*0*	0	0	0	0	0	113	12	0	32960	0%	3%	1001189	1149
45	*Control Group*	*0*	0	0	0	0	0	0	0	0	81	0%	5%	1630	1341
46	*Control Group*	*6*	0	0	0	0	0	0	0	0	97	0%	8%	1200	1341
47	*TCP Connect*	*3*	3	9	5	9	5	23	8	2665	3932	10%	14%	27946	180
48	*TCP Connect*	*9*	9	0	0	0	0	40	8	17267	7562	47%	21%	36457	25
49	*Aggressive*	*0*	0	0	0	0	0	115	11	0	32110	0%	3%	990706	950
50	*NIDS Avoidance*	*9*	9	24	4	33	7	68	8	32056	36234	32%	36%	100504	2365
51	*TCP Connect*	*0*	0	0	0	0	0	29	8	0	6685	0%	24%	28329	321
52	*NIDS Avoidance*	*0*	0	0	0	0	0	28	6	0	18122	0%	32%	57333	2367
53	*Control Group*	*3*	0	0	0	0	0	0	0	0	70	0%	5%	1281	1341
54	*Aggressive*	*9*	8	174	7	155	9	54	7	37572	12096	97%	31%	38909	1111
55	*TCP Connect*	*3*	3	7	5	8	5	26	6	2457	4213	8%	14%	30038	252
56	*TCP Connect*	*9*	9	54	5	42	7	35	8	16239	6219	45%	17%	36342	23
57	*Control Group*	*6*	0	0	0	0	0	0	0	0	90	0%	6%	1591	1341
58	*NIDS Avoidance*	*0*	0	0	0	0	0	33	6	0	18200	0%	31%	58469	2367

Table 13 continued from previous page

Trial	*ST*	*HP*	HHI	**SAI**	**STI**	**SAM**	**STM**	**SuA**	**SuT**	HHP	SuP	**% HHP**	**% SuP**	AP	ET
59	*TCP Connect*	*0*	0	0	0	0	0	33	8	0	4873	0%	16%	30904	366
60	*Aggressive*	*6*	5	69	7	66	9	36	9	11084	22560	3%	6%	385004	1828
61	*TCP Connect*	*3*	3	0	0	0	0	21	7	10031	7352	19%	14%	51586	2677
62	*Control Group*	*6*	0	0	0	0	0	0	0	0	95	0%	9%	1117	1341
63	*TCP Connect*	*0*	0	0	0	0	0	25	8	0	6123	0%	26%	23572	247
64	*Aggressive*	*9*	8	78	7	74	9	132	10	9304	18645	21%	42%	44115	1817
65	*TCP Connect*	*6*	6	27	5	24	7	34	8	8321	6698	27%	21%	31299	46
66	*Control Group*	*9*	0	0	0	0	0	0	0	0	80	0%	6%	1261	1341
67	*TCP Connect*	*9*	9	55	5	42	7	37	8	16691	8561	46%	24%	36386	32
68	*TCP Connect*	*0*	0	0	0	0	0	16	5	0	4564	0%	15%	31116	409
69	*TCP Connect*	*3*	3	7	5	8	5	16	6	2369	2076	7%	6%	34185	330
70	*Control Group*	*6*	0	0	0	0	0	0	0	0	95	0%	8%	1188	1341
71	*Control Group*	*9*	0	0	0	0	0	0	0	0	95	0%	8%	1211	1341
72	*TCP Connect*	*9*	9	0	0	0	0	12	6	16174	6893	44%	19%	36607	22
73	*Control Group*	*6*	0	0	0	0	0	0	0	0	109	0%	10%	1123	1341
74	*Aggressive*	*6*	5	103	7	99	9	119	10	24344	17919	6%	5%	382701	1826
75	*Control Group*	*9*	0	0	0	0	0	0	0	0	84	0%	7%	1216	1341
76	*Aggressive*	*9*	8	0	0	0	0	125	11	14966	18280	36%	44%	41339	1806
77	*Control Group*	*6*	0	0	0	0	0	0	0	0	110	0%	8%	1460	1341
78	*NIDS Avoidance*	*6*	6	12	4	17	5	62	8	19992	36290	23%	41%	88185	2366

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
79	TCP Connect	0	0	0	0	0	0	31	8	0	5310	0%	18%	29927	321
80	Aggressive	6	5	129	9	123	9	26	9	18030	17390	4%	3%	514449	1089
81	TCP Connect	6	6	3	1	3	1	42	8	577	10136	2%	32%	32128	154
82	NIDS Avoidance	6	6	108	6	112	7	67	8	19822	36111	23%	41%	87978	2365
83	TCP Connect	3	3	7	4	9	6	29	8	2634	2217	7%	6%	36804	471
84	Control Group	6	0	0	0	0	0	0	0	0	90	0%	8%	1114	1341
85	Control Group	0	0	0	0	0	0	0	0	0	79	0%	7%	1057	1341
86	TCP Connect	3	3	36	5	35	5	26	8	2523	2045	7%	6%	35480	382
87	Aggressive	9	8	137	8	122	10	124	12	37932	17753	96%	45%	39660	1802
88	Aggressive	9	8	156	9	137	10	120	10	29815	18369	70%	43%	42311	1806
89	Control Group	9	0	0	0	0	0	0	0	0	74	0%	7%	1026	1341
90	Control Group	9	0	0	0	0	0	0	0	0	196	0%	20%	1001	1341
91	TCP Connect	9	9	61	5	39	7	35	8	15533	6876	43%	19%	36470	20
92	NIDS Avoidance	6	6	12	4	18	5	63	8	19817	36112	23%	41%	87659	2367
93	Aggressive	6	5	104	7	104	9	118	10	23544	17856	5%	4%	507274	1829
94	TCP Connect	3	3	8	5	9	5	31	8	2608	4324	9%	14%	30338	295
95	NIDS Avoidance	9	9	24	4	33	7	59	8	32071	36129	32%	36%	100046	2365
96	Aggressive	3	3	43	7	46	9	107	12	12392	31865	1%	3%	1120935	932
97	Aggressive	6	5	58	7	59	9	92	8	13785	48851	41%	147%	33270	408
98	Aggressive	3	3	47	7	48	9	103	12	11984	31445	1%	3%	1121648	953

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
99	NIDS Avoidance	3	3	12	4	16	5	30	6	13457	18121	19%	26%	69068	2367
100	TCP Connect	3	3	9	5	11	7	33	8	2642	2661	8%	8%	35200	375
101	Aggressive	9	8	206	10	178	11	123	10	35124	16630	108%	51%	32566	1870
102	NIDS Avoidance	6	6	12	4	16	5	69	8	19816	36170	22%	41%	88593	2363
103	TCP Connect	6	6	26	5	25	7	41	8	8405	8325	26%	26%	32046	179
104	NIDS Avoidance	3	3	12	4	14	5	31	8	11941	17841	17%	25%	70457	2367
105	Control Group	6	0	0	0	0	0	0	0	0	91	0%	6%	1551	1341
106	Control Group	6	0	0	0	0	0	0	0	0	91	0%	5%	1688	1341
107	NIDS Avoidance	6	6	12	4	15	5	23	8	19817	36292	22%	41%	88230	2367
108	TCP Connect	6	6	23	5	20	7	39	8	6603	7154	20%	22%	32983	146
109	NIDS Avoidance	6	6	121	4	124	5	64	8	19821	36117	23%	41%	87948	2366
110	NIDS Avoidance	3	3	12	4	17	5	32	8	13458	18123	19%	26%	69950	2366
111	TCP Connect	0	0	0	0	0	0	6	6	0	21267	0%	63%	33801	547
112	Control Group	3	0	0	0	0	0	0	0	0	85	0%	6%	1384	1341
113	Control Group	3	0	0	0	0	0	0	0	0	174	0%	10%	1710	1341
114	TCP Connect	6	6	10	5	10	5	32	6	2387	7857	8%	25%	31182	101
115	NIDS Avoidance	9	9	24	4	32	7	63	8	32064	36118	32%	36%	100385	2367
116	Control Group	9	0	0	0	0	0	0	0	0	76	0%	7%	1071	1341
117	Control Group	0	0	0	0	0	0	0	0	0	82	0%	6%	1466	1341
118	TCP Connect	9	9	13	5	16	7	48	10	5785	7542	15%	19%	39194	280

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
119	Aggressive	6	5	108	7	108	9	234	10	22964	15869	7%	5%	324561	555
120	Control Group	0	0	0	0	0	0	0	0	0	67	0%	6%	1210	1341
121	NIDS Avoidance	6	6	0	0	0	0	68	8	19998	36136	23%	41%	87976	2366
122	TCP Connect	6	6	24	5	22	7	45	8	6286	9568	19%	29%	33472	203
123	Aggressive	6	5	50	7	47	9	269	12	8637	35068	2%	8%	413867	439
124	NIDS Avoidance	9	9	24	4	34	7	61	8	32091	36125	32%	36%	100790	2367
125	Aggressive	0	0	0	0	0	0	116	12	0	31918	0%	3%	1095314	718
126	Control Group	6	0	0	0	0	0	0	0	0	76	0%	7%	1107	1341
127	TCP Connect	0	0	0	0	0	0	30	8	0	6440	0%	25%	25899	252
128	Control Group	9	0	0	0	0	0	0	0	0	92	0%	7%	1360	1341
129	Control Group	9	0	0	0	0	0	0	0	0	96	0%	8%	1200	1341
130	Aggressive	0	0	0	0	0	0	118	12	0	32203	0%	3%	1000779	721
131	Control Group	6	0	0	0	0	0	0	0	0	94	0%	8%	1185	1341
132	NIDS Avoidance	9	9	24	4	34	7	64	8	31938	36658	31%	36%	102266	2368
133	NIDS Avoidance	9	9	0	0	0	0	67	8	31902	35737	31%	35%	101576	2364
134	TCP Connect	0	0	0	0	0	0	29	8	0	4409	0%	15%	30325	336
135	Aggressive	6	5	108	7	110	9	229	12	24171	15611	67%	43%	36188	530
136	Aggressive	0	0	0	0	0	0	103	11	0	30912	0%	5%	626567	1019
137	NIDS Avoidance	0	0	0	0	0	0	27	5	0	17984	0%	31%	58084	2369
138	Aggressive	0	0	0	0	0	0	121	12	0	31342	0%	5%	617717	604

Table 13 continued from previous page

Trial	ST	HP	HHI	**SAI**	**STI**	**SAM**	**STM**	**SuA**	**SuT**	HHP	SuP	**% HHP**	**% SuP**	AP	ET
139	Control Group	9	0	0	0	0	0	0	0	0	96	0%	4%	2441	1341
140	TCP Connect	0	0	0	0	0	0	29	8	0	4563	0%	19%	23664	221
141	TCP Connect	6	6	0	0	0	0	30	8	8454	5505	27%	18%	30797	51
142	Control Group	3	0	0	0	0	0	0	0	0	98	0%	6%	1681	1341
143	NIDS Avoidance	0	0	0	0	0	0	31	6	0	18153	0%	31%	57711	2366
144	NIDS Avoidance	3	3	12	4	17	5	33	6	12073	18218	17%	26%	70010	2369
145	Control Group	9	0	0	0	0	0	0	0	0	89	0%	5%	1641	1341
146	NIDS Avoidance	0	0	0	0	0	0	7	5	0	18218	0%	31%	57985	2366
147	NIDS Avoidance	6	6	32	5	36	5	66	8	19891	36207	22%	41%	89170	2367
148	TCP Connect	9	9	17	4	14	6	34	8	5507	6119	15%	17%	36474	21
149	NIDS Avoidance	0	0	0	0	0	0	27	6	0	18118	0%	31%	57733	2365
150	NIDS Avoidance	6	6	12	4	15	5	64	8	19808	36135	22%	41%	88909	2368
151	Aggressive	3	3	0	0	0	0	118	11	11018	30300	2%	5%	637042	666
152	TCP Connect	0	0	0	0	0	0	26	8	0	5026	0%	19%	25994	245
153	Aggressive	6	5	101	7	100	9	121	10	24777	17704	83%	59%	29869	1795
154	NIDS Avoidance	9	9	129	6	140	7	66	8	32106	36144	32%	36%	101266	2365
155	NIDS Avoidance	6	6	8	2	10	3	62	8	13939	36494	14%	38%	96538	7855
156	TCP Connect	6	6	23	5	25	7	35	8	8337	5158	25%	16%	33002	188
157	Control Group	6	0	0	0	0	0	0	0	0	80	0%	5%	1703	1341
158	Aggressive	0	0	0	0	0	0	122	14	0	32463	0%	5%	629445	1035

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
159	TCP Connect	9	9	55	5	40	7	38	8	17989	7239	49%	20%	36566	25
160	NIDS Avoidance	3	3	12	4	16	5	31	6	13178	18013	19%	26%	70151	2369
161	Aggressive	3	3	28	5	27	7	113	12	10649	31231	2%	5%	641598	1095
162	Control Group	0	0	0	0	0	0	0	0	0	81	0%	4%	2128	1341
163	NIDS Avoidance	3	3	12	4	17	5	32	6	12056	18171	17%	26%	70605	2366
164	Control Group	9	0	0	0	0	0	0	0	0	94	0%	5%	1815	1341
165	Control Group	3	0	0	0	0	0	0	0	0	140	0%	6%	2354	1341
166	Control Group	3	0	0	0	0	0	0	0	0	82	0%	7%	1163	1341
167	Control Group	0	0	0	0	0	0	0	0	0	66	0%	5%	1404	1341
168	Control Group	3	0	0	0	0	0	0	0	0	89	0%	8%	1099	1341
169	Control Group	3	0	0	0	0	0	0	0	0	77	0%	7%	1114	1341
170	Aggressive	3	3	47	7	47	9	42	11	10987	30930	2%	5%	640906	653
171	Aggressive	3	3	0	0	0	0	120	12	12157	32081	2%	5%	641763	655
172	Aggressive	9	8	146	7	121	9	123	12	38851	17979	95%	44%	40997	1810
173	NIDS Avoidance	0	0	0	0	0	0	31	6	0	18042	0%	31%	58041	2367
174	NIDS Avoidance	3	3	3	1	4	2	0	0	7461	18246	11%	26%	70969	4842
175	Control Group	9	0	0	0	0	0	0	0	0	191	0%	14%	1348	1341
176	Aggressive	0	0	0	0	0	0	121	12	0	32360	0%	5%	643431	701
177	TCP Connect	3	3	8	4	10	5	23	6	4535	2192	14%	7%	32258	493
178	NIDS Avoidance	3	3	12	4	17	5	34	6	13252	18704	19%	26%	70878	2367

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
179	Aggressive	9	8	176	7	149	10	124	10	30504	18477	69%	42%	44339	1809
180	TCP Connect	9	9	21	5	17	7	36	8	5538	6557	15%	18%	36719	21
181	Aggressive	0	0	0	0	0	0	122	12	0	32033	0%	5%	643158	593
182	Aggressive	6	5	0	0	0	0	53	7	23947	12114	97%	49%	24802	1854
183	TCP Connect	9	9	57	5	42	7	40	8	16630	7488	45%	20%	36608	26
184	NIDS Avoidance	0	0	0	0	0	0	28	6	0	18170	0%	29%	63306	3548
185	Aggressive	3	3	23	5	23	7	122	12	7402	32247	1%	5%	634299	1902
186	Aggressive	9	8	219	7	176	9	184	12	30316	18705	68%	42%	44587	1819
187	TCP Connect	9	9	65	5	44	7	38	8	19160	7960	52%	21%	37108	34
188	Control Group	9	0	0	0	0	0	0	0	0	93	0%	6%	1639	1341
189	Control Group	3	0	0	0	0	0	0	0	0	67	0%	4%	1630	1341
190	NIDS Avoidance	6	6	12	4	14	5	62	8	19824	36130	22%	41%	88777	2366
191	Aggressive	6	5	47	6	50	8	55	7	12240	11964	54%	53%	22766	942
192	Control Group	3	0	0	0	0	0	0	0	0	73	0%	4%	1680	1341
193	Control Group	6	0	0	0	0	0	0	0	0	94	0%	6%	1686	1341
194	NIDS Avoidance	0	0	0	0	0	0	6	2	0	18268	0%	32%	57911	2365
195	TCP Connect	9	9	94	7	88	7	62	8	24616	13503	54%	29%	45843	1747
196	TCP Connect	9	9	28	5	25	7	30	8	8641	6581	24%	18%	36486	27
197	Aggressive	6	5	0	0	0	0	222	10	0	15258	0%	45%	34217	577
198	Aggressive	3	1	126	9	120	9	105	12	8073	31319	1%	6%	552845	910

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
199	Control Group	6	0	0	0	0	0	0	0	0	667	0%	20%	3260	1341
200	TCP Connect	6	6	22	5	25	7	45	8	8132	10500	25%	33%	32216	169
201	NIDS Avoidance	6	6	108	6	114	7	65	8	19828	36232	22%	41%	89449	2369
202	Control Group	6	0	0	0	0	0	0	0	0	99	0%	5%	1876	1341
203	TCP Connect	3	3	9	4	9	4	31	8	2490	2312	7%	6%	36686	386
204	TCP Connect	0	0	0	0	0	0	21	8	0	4125	0%	12%	33202	467
205	NIDS Avoidance	6	6	0	0	0	0	71	8	18208	36181	18%	36%	100925	8985
206	Aggressive	9	8	184	7	157	9	65	7	36488	12802	97%	34%	37611	1106
207	Aggressive	6	5	104	8	95	8	88	9	17461	11975	72%	49%	24233	953
208	Control Group	0	0	0	0	0	0	0	0	0	82	0%	5%	1750	1341
209	NIDS Avoidance	9	9	30	5	39	8	61	8	31890	36047	31%	36%	101483	2369
210	TCP Connect	0	0	0	0	0	0	32	10	0	5200	0%	18%	29611	350
211	NIDS Avoidance	6	6	12	4	14	5	67	8	19935	35877	22%	40%	89865	2368
212	Aggressive	9	8	157	7	128	9	62	7	33662	12771	95%	36%	35500	1116
213	Aggressive	3	3	24	5	25	7	121	11	7981	31656	1%	5%	640899	1198
214	TCP Connect	0	0	0	0	0	0	27	8	0	4763	0%	15%	31533	369
215	TCP Connect	9	9	10	5	10	5	36	8	2376	7241	6%	20%	36809	22
216	Aggressive	3	3	43	7	44	9	112	12	12380	31724	2%	5%	626577	1156
217	Aggressive	3	3	39	7	42	9	108	12	11487	31105	2%	5%	635276	790
218	Control Group	0	0	0	0	0	0	0	0	0	66	0%	4%	1784	1341

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
219	Aggressive	3	3	91	7	89	9	43	11	11601	30455	2%	5%	635512	1226
220	NIDS Avoidance	9	9	45	5	56	7	67	8	31991	35993	31%	35%	102228	2396
221	Control Group	6	0	0	0	0	0	0	0	0	93	0%	3%	2763	1341
222	Aggressive	6	5	119	7	121	9	241	12	24515	16528	72%	48%	34137	520
223	TCP Connect	0	0	0	0	0	0	28	8	0	4164	0%	15%	27524	232
224	NIDS Avoidance	6	6	12	3	13	4	56	9	11827	29122	12%	30%	95912	7710
225	NIDS Avoidance	0	0	0	0	0	0	31	6	0	18251	0%	31%	58184	2367
226	Aggressive	9	8	110	9	84	9	127	10	30828	17955	89%	52%	34637	1803
227	Control Group	9	0	0	0	0	0	0	0	0	415	0%	13%	3174	1341
228	NIDS Avoidance	0	0	0	0	0	0	30	6	0	18100	0%	31%	58233	2365
229	Aggressive	9	8	131	7	113	10	123	10	32154	16534	96%	49%	33417	1066
230	TCP Connect	9	9	14	5	12	5	41	8	2890	6663	8%	18%	36512	22
231	TCP Connect	0	0	0	0	0	0	24	8	0	2524	0%	7%	34101	547
232	Aggressive	3	3	41	6	46	9	107	12	11240	31468	2%	5%	631313	843
233	TCP Connect	0	0	0	0	0	0	17	6	0	4072	0%	13%	30692	361
234	Aggressive	3	3	42	7	48	9	113	12	11541	32069	2%	5%	634563	1833
235	Control Group	9	0	0	0	0	0	0	0	0	32345	0%	1247%	2593	1341
236	TCP Connect	6	6	33	5	35	7	37	8	10589	8003	33%	25%	32002	137
237	Control Group	0	0	0	0	0	0	0	0	0	77	0%	5%	1668	1341
238	Control Group	9	0	0	0	0	0	0	0	0	88	0%	5%	1670	1341

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
239	Control Group	0	0	0	0	0	0	0	0	0	77	0%	4%	1714	1341
240	Control Group	6	0	0	0	0	0	0	0	0	189	0%	10%	1882	1341
241	TCP Connect	3	3	4	2	5	3	33	8	1985	2635	6%	8%	33381	331
242	NIDS Avoidance	0	0	0	0	0	0	30	6	0	18125	0%	31%	57950	2368
243	Aggressive	0	0	0	0	0	0	119	12	0	31340	0%	5%	595363	657
244	Control Group	3	0	0	0	0	0	0	0	0	73	0%	4%	1856	1341
245	Control Group	0	0	0	0	0	0	0	0	0	76	0%	4%	1721	1341
246	Aggressive	9	8	93	7	88	9	26	7	21261	12184	63%	36%	33586	1102
247	Aggressive	0	0	0	0	0	0	113	12	0	32255	0%	5%	612987	556
248	Aggressive	9	8	134	8	123	11	129	11	23659	18491	65%	50%	36619	1803
249	Aggressive	3	3	44	7	50	9	118	12	10722	33033	2%	5%	628083	969
250	TCP Connect	6	6	30	8	32	8	15	8	10666	44201	23%	94%	47052	4293
251	NIDS Avoidance	0	0	0	0	0	0	30	6	0	18157	0%	31%	58055	2367
252	NIDS Avoidance	0	0	0	0	0	0	8	4	0	36276	0%	62%	58487	2369
253	TCP Connect	0	0	0	0	0	0	20	7	0	4096	0%	13%	30987	345
254	Control Group	3	0	0	0	0	0	0	0	0	78	0%	3%	2343	1341
255	NIDS Avoidance	0	0	0	0	0	0	32	8	0	18145	0%	31%	58453	2369
256	TCP Connect	6	6	30	7	28	7	11	5	8162	22640	24%	67%	33560	126
257	Control Group	3	0	0	0	0	0	0	0	0	80	0%	4%	1838	1341
258	Control Group	0	0	0	0	0	0	0	0	0	82	0%	4%	1919	1341

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
259	Aggressive	9	8	153	7	134	9	125	10	38937	18166	93%	43%	42059	1815
260	TCP Connect	0	0	0	0	0	0	24	8	0	4333	0%	12%	34864	710
261	Control Group	6	0	0	0	0	0	0	0	0	94	0%	5%	2027	1341
262	Aggressive	3	3	70	7	73	7	113	11	11198	31090	2%	5%	623431	647
263	NIDS Avoidance	3	3	12	4	16	5	31	6	13468	18135	17%	23%	77784	2367
264	TCP Connect	0	0	0	0	0	0	51	8	0	3966	0%	14%	29352	403
265	Control Group	0	0	0	0	0	0	0	0	0	76	0%	4%	1918	1341
266	Aggressive	3	3	0	0	0	0	100	12	10279	31317	2%	5%	641550	639
267	NIDS Avoidance	3	3	3	1	10	4	26	5	7091	18248	8%	21%	85346	6785
268	Aggressive	9	8	130	7	118	10	112	10	37829	17290	93%	42%	40685	1801
269	Aggressive	0	0	0	0	0	0	113	12	0	30741	0%	5%	634409	991
270	TCP Connect	9	9	9	4	9	4	41	8	3000	6372	8%	17%	38513	20
271	Aggressive	6	5	70	7	73	10	28	8	10369	8125	31%	24%	33185	3742
272	NIDS Avoidance	3	3	21	5	22	6	32	6	13465	18143	18%	24%	76634	2367
273	Control Group	6	0	0	0	0	0	0	0	0	84	0%	5%	1831	1341
274	NIDS Avoidance	6	6	21	5	25	6	62	8	19986	36280	21%	38%	96653	2366
275	Aggressive	6	5	82	7	76	9	123	10	22437	17137	70%	53%	32080	1796
276	NIDS Avoidance	3	3	12	4	16	5	36	8	13375	18266	17%	24%	77461	2369
277	NIDS Avoidance	3	3	12	4	16	5	33	8	13344	18078	17%	24%	76664	2369
278	TCP Connect	9	9	35	5	30	7	42	8	10420	7431	27%	19%	38563	22

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
279	Aggressive	3	0	60	7	64	9	101	12	13963	31282	2%	5%	635043	643
280	Control Group	3	0	0	0	0	0	0	0	0	81	0%	5%	1624	1341
281	Aggressive	9	8	114	6	92	9	73	7	24621	12493	94%	48%	26233	938
282	Aggressive	9	8	231	7	202	9	189	10	40885	18035	93%	41%	43828	1820
283	NIDS Avoidance	6	6	0	0	0	0	57	8	21550	36430	21%	36%	100883	4659
284	TCP Connect	6	6	21	5	22	7	43	8	6104	9148	17%	26%	35026	203
285	Control Group	0	0	0	0	0	0	0	0	0	66	0%	4%	1700	1341
286	TCP Connect	0	0	0	0	0	0	16	6	0	3637	0%	12%	30859	508
287	TCP Connect	3	3	8	5	8	5	16	6	2414	1195	9%	4%	27226	60
288	TCP Connect	6	6	26	5	24	7	34	8	8376	6666	26%	20%	32762	39
289	Aggressive	3	3	40	7	46	9	103	12	11202	30472	2%	5%	639607	855
290	TCP Connect	3	3	0	0	0	0	28	8	0	3873	0%	12%	31268	315
291	TCP Connect	3	0	26	5	37	7	4	4	14062	10565	29%	22%	48180	2727
292	NIDS Avoidance	9	9	12	4	14	5	65	8	18087	36178	17%	34%	107584	2364
293	Control Group	3	0	0	0	0	0	0	0	0	69	0%	4%	1663	1341
294	Aggressive	6	5	46	7	45	9	236	10	8961	15931	27%	48%	33465	406
295	NIDS Avoidance	9	9	24	4	33	7	60	8	32040	36261	30%	34%	107230	2368
296	Control Group	0	0	0	0	0	0	0	0	0	655	0%	19%	3416	1341
297	TCP Connect	6	6	15	5	19	7	46	8	5862	8239	16%	22%	37122	431
298	TCP Connect	0	0	0	0	0	0	24	8	0	3904	0%	15%	26627	293

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
299	NIDS Avoidance	0	0	0	0	0	0	29	6	0	18051	0%	28%	63930	2367
300	Control Group	9	0	0	0	0	0	0	0	0	96	0%	5%	1802	1341
301	Aggressive	3	3	40	7	43	7	105	12	11784	31123	2%	5%	622328	814
302	Control Group	9	0	0	0	0	0	0	0	0	95	0%	5%	2006	1341
303	NIDS Avoidance	0	0	0	0	0	0	28	6	0	18111	0%	28%	64609	2368
304	NIDS Avoidance	3	3	12	4	15	5	11	5	13374	36263	17%	47%	76766	2366
305	NIDS Avoidance	9	9	24	4	35	7	59	8	31378	36098	28%	32%	113753	5503
306	Aggressive	6	5	99	7	88	9	47	7	20432	12197	82%	49%	25016	1059
307	Control Group	0	0	0	0	0	0	0	0	0	67	0%	4%	1759	1341
308	Control Group	3	0	0	0	0	0	0	0	0	85	0%	4%	1944	1341
309	TCP Connect	3	3	18	5	16	7	0	0	6146	3578	22%	13%	27547	65
310	Aggressive	0	0	0	0	0	0	112	12	0	31399	0%	5%	628763	614
311	Control Group	6	0	0	0	0	0	0	0	0	97	0%	5%	1895	1341
312	Control Group	6	0	0	0	0	0	0	0	0	74	0%	4%	1889	1341
313	Control Group	3	0	0	0	0	0	0	0	0	168	0%	9%	1826	1341
314	TCP Connect	3	3	8	5	8	5	22	6	2241	2545	8%	9%	26881	41
315	Control Group	9	0	0	0	0	0	0	0	0	99	0%	5%	1925	1341
316	Aggressive	0	0	0	0	0	0	105	12	0	31669	0%	5%	614810	849
317	NIDS Avoidance	6	6	12	4	17	5	65	8	19993	36001	21%	38%	95888	2369
318	Control Group	3	0	0	0	0	0	0	0	0	79	0%	4%	1888	1341

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
319	NIDS Avoidance	3	3	90	6	96	7	34	8	13433	18197	18%	24%	76571	2367
320	NIDS Avoidance	9	9	24	4	34	7	66	8	32035	36181	29%	33%	108858	2368
321	Control Group	0	0	0	0	0	0	0	0	0	80	0%	4%	2039	1341
322	NIDS Avoidance	6	6	32	5	36	5	20	8	15965	36234	17%	38%	95469	2365
323	Aggressive	9	8	188	8	166	10	129	11	41038	17657	93%	40%	44184	1821
324	Control Group	6	0	0	0	0	0	0	0	0	93	0%	4%	2394	1341
325	NIDS Avoidance	9	9	24	4	34	7	62	8	32021	36011	29%	33%	108974	2365
326	NIDS Avoidance	6	6	12	4	18	5	23	8	16006	72494	16%	75%	97299	2368
327	TCP Connect	9	9	19	5	16	7	39	8	5310	11010	12%	26%	42974	414
328	Aggressive	6	5	108	8	99	10	121	11	26021	17738	83%	57%	31384	1800
329	Control Group	0	0	0	0	0	0	0	0	0	82	0%	5%	1726	1341
330	Aggressive	6	5	0	0	0	0	54	7	20217	12029	82%	49%	24647	1058
331	NIDS Avoidance	3	3	0	0	0	0	29	6	13495	18172	18%	24%	76744	2367
332	Aggressive	6	5	115	7	118	9	270	10	20471	37090	5%	9%	402106	435
333	TCP Connect	3	3	10	4	13	5	32	8	4387	3487	14%	11%	31504	290
334	Aggressive	0	0	0	0	0	0	39	10	0	34635	0%	6%	623664	608
335	NIDS Avoidance	6	6	12	4	17	5	59	8	20028	36223	21%	38%	95552	2368
336	NIDS Avoidance	3	3	20	4	29	7	27	6	8568	18721	10%	23%	82583	5964
337	Control Group	9	0	0	0	0	0	0	0	0	103	0%	6%	1634	1341
338	Aggressive	9	8	194	9	175	9	83	9	34898	12110	96%	33%	36407	1099

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
339	Aggressive	0	0	0	0	0	0	106	12	0	31205	0%	5%	634197	610
340	Aggressive	9	8	136	7	119	9	55	7	32316	12068	95%	35%	34152	1107
341	NIDS Avoidance	0	0	0	0	0	0	28	6	0	18122	0%	28%	63660	2365
342	Aggressive	3	3	43	9	48	9	100	12	10602	30874	2%	5%	636944	660
343	TCP Connect	6	6	37	5	30	7	37	8	10525	8991	32%	27%	33078	34
344	NIDS Avoidance	9	9	20	4	35	7	68	8	26008	36143	24%	34%	107648	2369
345	Aggressive	0	0	0	0	0	0	113	12	0	31063	0%	5%	627136	593
346	NIDS Avoidance	9	9	131	4	137	7	60	8	32048	36141	30%	34%	107316	2366
347	NIDS Avoidance	3	3	12	4	14	5	28	6	13447	18548	15%	21%	88230	2367
348	NIDS Avoidance	6	6	13	5	18	6	59	8	12304	36518	11%	33%	112233	11102
349	NIDS Avoidance	3	3	0	0	0	0	30	6	0	18614	0%	17%	112058	5860
350	Control Group	9	0	0	0	0	0	0	0	0	195	0%	10%	1972	1341
351	Control Group	9	0	0	0	0	0	0	0	0	93	0%	4%	2368	1341
352	TCP Connect	3	3	18	5	20	5	22	8	6054	3768	19%	12%	31188	335
353	NIDS Avoidance	3	3	19	5	24	5	27	5	9310	18073	11%	21%	86161	4077
354	Control Group	0	0	0	0	0	0	0	0	0	83	0%	5%	1821	1341
355	TCP Connect	9	9	8	5	8	5	41	8	2053	7648	5%	20%	38907	20
356	Aggressive	0	0	0	0	0	0	116	12	0	31734	0%	5%	624185	616
357	TCP Connect	9	9	0	0	0	0	40	8	0	7076	0%	18%	38669	26
358	NIDS Avoidance	9	6	32	5	31	5	63	8	24826	35919	23%	33%	107681	2367

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
359	TCP Connect	3	3	10	5	14	7	27	10	4436	3843	14%	12%	31436	291
360	Aggressive	6	5	96	7	94	9	60	7	20159	12622	8%	5%	244766	1133
361	TCP Connect	3	3	9	4	12	5	27	8	4394	2310	15%	8%	29712	220
362	Control Group	9	0	0	0	0	0	0	0	0	97	0%	5%	1793	1341
363	TCP Connect	9	9	40	5	42	7	37	6	16204	9634	37%	22%	43952	1133
364	TCP Connect	6	6	64	7	63	7	112	8	9671	12918	19%	26%	49775	4317
365	NIDS Avoidance	0	0	0	0	0	0	32	6	0	18100	0%	27%	66522	2367
366	TCP Connect	3	3	4	1	6	3	24	8	5880	4023	11%	7%	55036	3398
367	Control Group	0	0	0	0	0	0	0	0	0	79	0%	4%	2185	1341
368	Control Group	3	0	0	0	0	0	0	0	0	81	0%	5%	1660	1341
369	NIDS Avoidance	6	6	12	4	15	5	25	6	16003	36208	17%	38%	95336	2367
370	TCP Connect	0	0	0	0	0	0	22	6	0	6548	0%	24%	27640	397
371	Aggressive	0	0	0	0	0	0	111	12	0	31316	0%	5%	628280	617
372	NIDS Avoidance	0	0	0	0	0	0	13	5	0	49444	0%	77%	64362	2367
373	Aggressive	3	3	41	7	47	9	101	11	10525	31215	2%	5%	633644	634
374	Aggressive	6	5	112	7	109	9	121	12	24876	17773	69%	49%	36026	1798
375	Control Group	0	0	0	0	0	0	0	0	0	77	0%	5%	1680	1341
376	Aggressive	9	8	153	7	137	9	129	12	28284	18672	67%	44%	42108	1811
377	Control Group	6	0	0	0	0	0	0	0	0	78	0%	3%	2959	1341
378	TCP Connect	9	9	69	7	61	8	48	8	22333	12944	44%	25%	51282	1197

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
379	Aggressive	9	8	13	5	16	7	63	7	5135	12788	15%	36%	35075	1113
380	NIDS Avoidance	6	6	12	4	17	5	64	8	19975	36099	21%	38%	95653	2365
381	NIDS Avoidance	9	9	17	5	20	5	58	8	17417	36140	16%	33%	110377	2470
382	TCP Connect	9	9	67	5	68	8	52	8	21808	13154	41%	25%	52637	1289
383	Control Group	3	0	0	0	0	0	0	0	0	84	0%	4%	2054	1341
384	TCP Connect	6	6	0	0	0	0	56	8	15843	13828	31%	27%	50786	2402
385	NIDS Avoidance	0	0	0	0	0	0	31	8	0	18069	0%	28%	64507	2365
386	Control Group	0	0	0	0	0	0	0	0	0	79	0%	4%	1965	1341
387	NIDS Avoidance	9	9	16	4	19	5	63	8	14146	36176	13%	33%	110131	2368
388	Control Group	6	0	0	0	0	0	0	0	0	93	0%	5%	1837	1341
389	TCP Connect	6	6	16	5	14	5	57	10	4065	9457	11%	25%	37141	389
390	Aggressive	0	0	0	0	0	0	111	12	0	31238	0%	2%	1269081	664
391	Aggressive	6	5	62	7	58	9	123	10	15636	17798	39%	45%	39801	1798
392	TCP Connect	6	6	43	7	35	7	37	8	12264	8118	37%	25%	33134	62
393	Control Group	3	0	0	0	0	0	0	0	0	67	0%	3%	2202	1341
394	TCP Connect	0	0	0	0	0	0	0	0	0	6240	0%	25%	24754	261
395	TCP Connect	6	6	28	5	24	7	28	6	7907	5451	24%	16%	33117	63
396	Aggressive	3	3	42	7	44	9	0	0	10654	30983	1%	2%	1290814	697
397	NIDS Avoidance	3	3	29	6	32	7	32	6	12307	18299	16%	23%	78976	4020
398	TCP Connect	0	0	0	0	0	0	27	7	0	4954	0%	15%	32433	853

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
399	Aggressive	0	0	0	0	0	0	111	12	0	32373	0%	2%	1330966	791
400	Aggressive	3	3	41	7	46	7	101	12	10886	29789	1%	2%	1368807	784
401	TCP Connect	0	0	0	0	0	0	18	6	0	4046	0%	17%	24177	243
402	Aggressive	6	5	41	7	41	7	117	13	9866	15540	7%	11%	139521	1850
403	Control Group	9	0	0	0	0	0	0	0	0	392	0%	19%	2104	1341
404	NIDS Avoidance	3	3	0	0	0	0	30	6	13196	17701	18%	24%	74274	2367
405	NIDS Avoidance	6	6	16	4	19	5	60	8	19288	35979	20%	37%	97429	5004
406	NIDS Avoidance	0	0	0	0	0	0	30	6	0	17680	0%	29%	60017	2367
407	Control Group	6	0	0	0	0	0	0	0	0	271	0%	15%	1852	1341
408	Control Group	3	0	0	0	0	0	0	0	0	65	0%	3%	2257	1341
409	NIDS Avoidance	6	6	18	5	22	6	65	8	13949	35215	15%	38%	92160	2364
410	NIDS Avoidance	3	3	12	4	15	5	34	8	13443	17860	18%	25%	72896	2366
411	TCP Connect	9	9	78	7	47	8	45	10	16941	6340	43%	16%	39820	71
412	NIDS Avoidance	0	0	0	0	0	0	29	8	0	17714	0%	30%	59644	2364
413	Aggressive	6	5	82	7	78	9	126	10	22631	17099	4%	3%	617511	1827
414	NIDS Avoidance	9	9	33	5	41	8	66	8	32040	35917	30%	34%	106518	2361
415	TCP Connect	0	0	0	0	0	0	20	8	0	3560	0%	15%	23808	249
416	Control Group	3	0	0	0	0	0	0	0	0	82	0%	5%	1716	1341
417	TCP Connect	9	9	0	0	0	0	38	8	18274	11916	39%	26%	46571	1926
418	Control Group	0	0	0	0	0	0	0	0	0	80	0%	5%	1725	1341

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
419	TCP Connect	3	3	33	5	22	5	30	5	6497	4413	21%	14%	30685	438
420	Control Group	9	0	0	0	0	0	0	0	0	112	0%	8%	1442	1341
421	NIDS Avoidance	6	6	12	4	14	5	67	8	13982	35230	15%	39%	91372	2367
422	TCP Connect	6	6	7	4	7	4	47	8	2333	9318	7%	27%	34092	291
423	NIDS Avoidance	0	0	0	0	0	0	28	6	0	17868	0%	30%	60313	2363
424	Control Group	6	0	0	0	0	0	0	0	0	93	0%	5%	1920	1341
425	Aggressive	9	8	0	0	0	0	127	12	40262	15467	95%	37%	42275	1815
426	Aggressive	9	8	132	9	115	10	46	7	31240	12512	96%	39%	32389	1099
427	Aggressive	6	5	65	6	58	8	51	7	17467	11541	80%	53%	21869	956
428	TCP Connect	0	0	0	0	0	0	17	5	0	6302	0%	20%	30743	709
429	NIDS Avoidance	6	6	21	5	23	5	63	8	14513	35542	13%	32%	112434	4904
430	TCP Connect	9	9	60	5	40	7	35	8	18381	6910	47%	18%	38861	22
431	NIDS Avoidance	3	3	7	3	10	4	28	6	2128	17759	2%	17%	102732	12076
432	TCP Connect	3	3	10	5	11	5	20	6	4206	2022	14%	7%	31038	391
433	TCP Connect	6	6	28	5	30	7	45	8	7174	11116	22%	34%	32913	204
434	Control Group	0	0	0	0	0	0	0	0	0	80	0%	5%	1748	1341
435	NIDS Avoidance	9	9	131	4	137	7	62	8	32068	36130	30%	34%	106204	2361
436	NIDS Avoidance	3	3	12	4	16	5	0	0	13165	0	18%	0%	72645	2368
437	TCP Connect	9	9	9	3	12	4	40	6	4264	11456	9%	25%	45205	447
438	TCP Connect	9	9	64	5	50	7	40	8	19618	9842	43%	22%	45758	1173

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
439	NIDS Avoidance	9	9	24	4	35	7	63	8	32070	35431	30%	33%	107572	2364
440	Control Group	3	0	0	0	0	0	0	0	0	74	0%	4%	1764	1341
441	NIDS Avoidance	9	9	20	4	20	5	22	6	20031	35028	19%	33%	107602	2369
442	Control Group	0	0	0	0	0	0	0	0	0	80	0%	4%	2096	1341
443	Control Group	0	0	0	0	0	0	0	0	0	66	0%	4%	1704	1341
444	TCP Connect	0	0	0	0	0	0	29	7	0	5686	0%	21%	27613	363
445	NIDS Avoidance	0	0	0	0	0	0	27	5	0	17012	0%	27%	64172	2365
446	Control Group	3	0	0	0	0	0	0	0	0	261	0%	14%	1837	1341
447	NIDS Avoidance	0	0	0	0	0	0	34	8	0	16999	0%	27%	63788	2369
448	TCP Connect	3	3	9	6	10	6	16	6	2360	1656	8%	5%	31020	267
449	Control Group	6	0	0	0	0	0	0	0	0	407	0%	18%	2313	1341
450	Control Group	0	0	0	0	0	0	0	0	0	77	0%	5%	1627	1341
451	Aggressive	0	0	0	0	0	0	114	12	0	29840	0%	5%	616091	608
452	NIDS Avoidance	9	9	0	0	0	0	62	8	32003	34564	30%	32%	107252	2363
453	NIDS Avoidance	9	9	24	4	33	7	62	8	32383	35168	30%	33%	107745	2366
454	Aggressive	6	5	51	9	49	9	239	10	9973	13414	35%	47%	28845	400
455	TCP Connect	9	9	66	7	61	9	45	8	19490	9039	39%	18%	49478	2956
456	Aggressive	9	8	45	7	46	9	244	12	8123	12856	19%	30%	42822	545
457	Aggressive	0	0	0	0	0	0	115	12	0	13926	0%	4%	328201	554
458	TCP Connect	0	0	0	0	0	0	24	8	0	5795	0%	27%	21736	51

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
459	TCP Connect	6	6	30	5	28	7	33	8	8437	7180	26%	22%	32494	21
460	Aggressive	3	3	36	6	44	9	105	12	10791	8698	4%	3%	249548	576
461	TCP Connect	3	3	7	5	8	5	0	0	2757	2426	9%	8%	29896	206
462	NIDS Avoidance	3	3	12	4	16	5	30	5	13365	17607	17%	23%	77673	2369
463	TCP Connect	6	6	13	5	11	5	0	0	4178	0	13%	0%	32501	25
464	Aggressive	0	0	0	0	0	0	133	12	0	8757	0%	3%	250691	579
465	TCP Connect	3	0	69	7	77	9	20	8	20917	1962	66%	6%	31710	349
466	Control Group	3	0	0	0	0	0	0	0	0	86	0%	5%	1777	1341
467	Control Group	0	0	0	0	0	0	0	0	0	73	0%	4%	1805	1341
468	NIDS Avoidance	9	9	24	4	33	7	65	8	30137	36170	27%	33%	110085	2361
469	NIDS Avoidance	3	3	32	5	37	5	33	6	13309	18103	17%	24%	76477	2367
470	Control Group	6	0	0	0	0	0	0	0	0	89	0%	4%	1985	1341
471	Control Group	3	0	0	0	0	0	0	0	0	69	0%	5%	1435	1341
472	Control Group	9	0	0	0	0	0	0	0	0	95	0%	7%	1291	1341
473	NIDS Avoidance	0	0	0	0	0	0	30	6	0	18158	0%	29%	63496	2365
474	Control Group	0	0	0	0	0	0	0	0	0	75	0%	6%	1290	1341
475	Aggressive	3	3	50	7	45	9	193	12	12677	9238	4%	3%	322296	1622
476	NIDS Avoidance	9	9	0	0	0	0	62	8	32065	36238	30%	34%	106835	2361
477	NIDS Avoidance	0	0	0	0	0	0	30	6	0	18168	0%	28%	65130	2364
478	Aggressive	0	0	0	0	0	0	112	12	0	9159	0%	1%	639420	606

Table 13 continued from previous page

Trial	ST	HP	HHI	SAI	STI	SAM	STM	SuA	SuT	HHP	SuP	% HHP	% SuP	AP	ET
479	Control Group	9	0	0	0	0	0	0	0	0	83	0%	7%	1192	1341
480	NIDS Avoidance	3	3	12	4	15	5	29	5	13465	18132	18%	24%	75482	2365

Appendix F. permutation_test.py

permutation_test.py

```python
1  import os
2  import numpy
3  import random
4
5  numPermutations = 900000
6  experimentFile = 'results.csv'
7
8  set1 = []
9  set2 = []
10 combined = []
11 mean1 = 0
12 mean2 = 0
13 difference = 0
14 numExceedDif = 0
15
16 # checks to see if a file with the run order exists and makes sure
       it's not empty
17 if os.path.exists(experimentFile) and os.path.getsize(experimentFile
       ) > 0:
18     f = open(experimentFile, 'r')
19     for line in f:
20         value = line.split(',')
21         lambdaFunc = None
22
23         #print value[0]
24         set1.append(float(value[0]))
25         #print value[1]
26         set2.append(float(value[1].rstrip()))
27
```

```python
28      f.close()
29      #print 'set1'
30      #print set1
31      #print 'set2'
32      #print set2
33
34      mean1 = numpy.average(set1)
35      print 'Mean1: ' + str(mean1)
36      mean2 = numpy.average(set2)
37      print 'Mean2: ' + str(mean2)
38      difference = abs(mean1 - mean2)
39      print "Difference: " + str(difference)
40      combined = set1 + set2
41      #print combined
42
43      for x in range(numPermutations):
44          random.shuffle(combined)
45
46          sample1Avg = numpy.average(combined[:len(combined)//2])
47          sample2Avg = numpy.average(combined[len(combined)//2:])
48          if abs(sample1Avg - sample2Avg) > difference:
49              numExceedDif +=1
50      result = numExceedDif * 1.0 / numPermutations
51      print result
52
53 else:
54    print "Error, file not found!"
```

Bibliography

1. K. Lueth, "State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating," 2018. [Online]. Available: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/ [Accessed: 10 Jan 2020]

2. B. Schneier, "The Internet of Things Is Wildly Insecure — And Often Unpatchable," 2014. [Online]. Available: https://www.wired.com/2014/01/theres-no-good-way-to-patch-the-internet-of-things-and-thats-a-huge-problem/ [Accessed: 10 Jan 2020]

3. K. Rawlinson, "HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack," 2014. [Online]. Available: https://www8.hp.com/us/en/hp-news/press-release.html?id=1744676 [Accessed: 10 Jan 2020]

4. M. Patton, E. Gross, R. Chinn, S. Forbis, L. Walker, and H. Chen, "Uninvited connections: A study of Vulnerable Devices on the Internet of Things (IoT)," in *Proceedings - 2014 IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014*, 2014, pp. 232–235.

5. N. Provos, "Developments of the Honeyd Virtual Honeypot." [Online]. Available: http://www.honeyd.org/ [Accessed: 10 Jan 2020]

6. L. A. Stafira, "Examining Effectiveness of Web-Based Internet of Things Honeypots," Master's thesis, Air Force Institue of Technology, 2019. [Online]. Available: https://scholar.afit.edu/cgi/viewcontent.cgi?article=3285&context=etd [Accessed: 10 Jan 2020]

7. K. Zetter, "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/ [Accessed: 10 Jan 2020]

8. C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and other botnets," *IEEE Computer*, vol. 50, no. 7, pp. 80–84, 2017.

9. W. Stallings, "Introduction to Network-Based Intrusion Detection Systems," 2007. [Online]. Available: http://www.informit.com/articles/article.aspx?p=782118 [Accessed: 10 Jan 2020]

10. N. Provos, "Honeyd Release 1.5c." [Online]. Available: http://www.honeyd.org/release.php?version=1.5c [Accessed: 10 Jan 2020]

11. DataSoft, "Honeyd 1.6d GitHub." [Online]. Available: https://github.com/DataSoft/Honeyd [Accessed: 10 Jan 2020]

12. H. Suo, J. Wan, C. Zou, and J. Liu, "Security in the Internet of Things: A Review," *International Conference on Computer Science and Electronics Engineering, ICCSEE 2012*, vol. 3, pp. 648–651, 2012.

13. T. Salman and R. Jain, "A Survey of Protocols and Standards for Internet of Things," *Advanced Computing and Communications*, vol. 1, no. 1, pp. 1–20, 2017.

14. J. S. Lee, Y. W. Su, and C. C. Shen, "A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi," *IECON Proceedings (Industrial Electronics Conference)*, pp. 46–51, 2007.

15. Nest Support, "How to add your Nest thermostat to the Nest app," 2019. [Online]. Available: https://nest.com/support/article/How-do-I-pair-my-Nest-Learning-Thermostat-with-my-Nest-Account#section-4 [Accessed: 10 Jan 2020]

16. A. Stachowicz, "ZigBee Wireless Networks," 2010. [Online]. Available: http://zigbee.pbworks.com/w/page/25465049/ZigBee [Accessed: 10 Jan 2020]

17. P. McDermott-Wells, "What is Bluetooth?" *Potentials, IEEE*, vol. 23, no. 5, pp. 33–35, 2005.

18. Shodan, "Shodan." [Online]. Available: https://www.shodan.io/ [Accessed: 10 Jan 2020]

19. A. Acien, A. Nieto, G. Fernandez, and J. Lopez, "A comprehensive methodology for deploying IoT honeypots," *15th International Conference on Trust, Privacy and Security in Digital Business (TrustBus)*, vol. L, no. ii, pp. 229–243, 2018.

20. T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, "Handling a trillion (unfixable) flaws on a billion devices," *Proceedings of the 14th ACM Workshop on Hot Topics in Networks - HotNets-XIV*, pp. 1–7, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2834050.2834095 [Accessed: 10 Jan 2020]

21. M. Ward, "Smart meters can be hacked to cut power bills," p. 1, 2014. [Online]. Available: http://www.bbc.co.uk/news/technology-29643276 [Accessed: 10 Jan 2020]

22. Y. M. P. Pa, S. Suzuki, K. Yoshioka, and T. Matsumoto, "IoTPOT: Analysing the Rise of IoT Compromises," *USENIX Workshop on Offensive Technologies*, 2015. [Online]. Available: https://www.usenix.org/system/files/conference/woot15/woot15-paper-pa.pdf [Accessed: 10 Jan 2020]

23. A. G. Manzanares, "HoneyIo4 The construction of a virtual, low-interaction IoT Honeypot," Ph.D. dissertation, Universitat Politecnica de Catalunya, 2017. [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2117/108166/Alejandro_Guerra_Manzanares.pdf [Accessed: 10 Jan 2020]

24. L. Spitzner, *Honeypots: Tracking Hackers.* Boston: Pearson Education, 2002.

25. N. Provos, "A Virtual Honeypot Framework," *Proceedings of the 13th USENIX Security Symposium*, pp. 1–14, 2004.

26. I. Mokube and M. Adams, "Honeypots: concepts, approaches, and challenges," *Proceedings of the 45th ACM Southeast Conference. ACMSE 07*, pp. 321–326, 2007.

27. A. Barfar and S. Mohammadi, "Honeypots: Intrusion Deception," *ISSA Journal*, no. January 2007, pp. 28–31, 2007. [Online]. Available: http://www.researchgate.net/publication/ 228854989_Honeypots_intrusion_deception\%5Cnhttps://dev.issa.org/Library/ Journals/2007/June/BarfarandMohammadi-Honeypots-Intrusiondeception.pdf [Accessed: 10 Jan 2020]

28. J. Guarnizo, A. Tambe, S. S. Bhunia, M. Ochoa, N. Tippenhauer, A. Shabtai, and Y. Elovici, "SIPHON: Towards Scalable High-Interaction Physical Honeypots," in *Proceedings of the 3rd ACM Workshop on Cyber-Physical System Security*. ACM, 2017, pp. 57–68. [Online]. Available: http://arxiv.org/abs/1701.02446 [Accessed: 10 Jan 2020]

29. Cymmetria, "The Crossed Swords wargame: Catching NATO red teams with cyber deception," 2017. [Online]. Available: https://cymmetria.com/blog/ nato-crossed-swords-exercise/ [Accessed: 10 Jan 2020]

30. Cymmetria, "Introduction to cyber deception," 2016. [Online]. Available: https://cymmetria.com/an-introduction-to-cyber-deception/ [Accessed: 10 Jan 2020]

31. N. Provos and T. Hols, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*, 1st ed. Boston: Pearson Education, 2008.

32. CyberEdge Group, "2017 Cyberthreat Defense Report," 2017. [Online]. Available: https://cyber-edge.com/wp-content/uploads/2017/03/ CyberEdge-2017-CDR-report.pdf [Accessed: 10-01-2019]

33. Pluralsight, "JavaScript.com." [Online]. Available: https://www.javascript.com/ [Accessed: 10 Jan 2020]

34. OpenJS Foundation, "Node.js." [Online]. Available: https://nodejs.org/en/ [Accessed: 10 Jan 2020]

35. Electron, "Electron — Build cross platform desktop apps with JavaScript, HTML, and CSS." [Online]. Available: https://electronjs.org/ [Accessed: 10 Jan 2020]

36. Python Software Foundation, "Python." [Online]. Available: https://www.python.org/ [Accessed: 10 Jan 2020]

37. Python Software Foundation, "pip · PyPI." [Online]. Available: https://pypi.org/project/pip/ [Accessed: 10 Jan 2020]

38. G. F. Lyon, "Nmap: the Network Mapper - Free Security Scanner." [Online]. Available: https://nmap.org/ [Accessed: 10 Jan 2020]

39. Wireshark, "Wireshark." [Online]. Available: https://www.wireshark.org/ [Accessed: 10 Jan 2020]

40. TCPDump, "Tcpdump/Libpcap public repository." [Online]. Available: https://www.tcpdump.org/ [Accessed: 10 Jan 2020]

41. TCPDump, "Manpage of TCPDump." [Online]. Available: https://www.tcpdump.org/manpages/tcpdump.1.html [Accessed: 10 Jan 2020]

42. Wireshark, "tshark - The Wireshark Network Analyzer 3.0.3." [Online]. Available: https://www.wireshark.org/docs/man-pages/tshark.htmll [Accessed: 10 Jan 2020]

43. VMware, Inc., "VMware – Official Site." [Online]. Available: https://www.vmware.com/ [Accessed: 10 Jan 2020]

44. Docker Inc., "Enterprise Container Platform — Docker." [Online]. Available: https://www.docker.com/ [Accessed: 10 Jan 2020]

45. D. Sever and T. Kisasondi, "Efficiency and security of docker based honeypot systems," *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018 - Proceedings*, pp. 1167–1173, 2018.

46. Offensive Security, "Exploit Database - Exploits for Penetration Testers, Researchers, and Ethical Hackers." [Online]. Available: https://www.exploit-db.com/ [Accessed: 10 Jan 2020]

47. HoneyNet Project, "Conpot." [Online]. Available: http://conpot.org/ [Accessed: 10 Jan 2020]

48. C. Kreibich and J. Crowcroft, "Honeycomb Creating Intrusion Detection Signatures Using Honeypots," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, pp. 51–56, 2004.

49. The Bro Project, "The Zeek Network Security Monitor." [Online]. Available: https://www.zeek.org/ [Accessed: 10 Jan 2020]

238

50. Cisco, "Snort - Network Intrusion Detection and Prevention System." [Online]. Available: https://www.snort.org/ [Accessed: 10 Jan 2020]

51. J. Kloet, "A Honeypot Based Worm Alerting System," 2005. [Online]. Available: https://www.sans.org/reading-room/whitepapers/detection/honeypot-based-worm-alerting-system-1563 [Accessed: 10 Jan 2020]

52. SolarWinds Worldwide, "Kiwi Syslog Server," 2019. [Online]. Available: https://www.kiwisyslog.com/kiwi-syslog-server [Accessed: 10 Jan 2020]

53. Lavenya and K. Kaur, "HoneyComb: Enhancement to Honeypot Log Management," *International Journal of Engineering Research and Technology (IJERT)*, vol. 1, no. 6, pp. 1–6, 2012.

54. D. Ramirez, J. I. Uribe, L. Francaviglia, P. Romero-Gomez, A. Fontcuberta i Morral, and F. Jaramillo, "IoTCandyJar: Towards an Intelligent-Interaction Honeypot for IoT Devices," *Journal of Materials Chemistry C*, vol. 6, no. 23, pp. 6216–6221, 2017. [Online]. Available: http://xlink.rsc.org/?DOI=C8TC01582A [Accessed: 10 Jan 2020]

55. W. Y. Chin, E. P. Markatos, S. Antonatos, and S. Ioannidis, "HoneyLab: Large-Scale Honeypot Deployment and Resource Sharing," *NSS 2009 - Network and System Security*, no. September 2014, pp. 381–388, 2009.

56. P. Krishnaprasad, "Capturing attacks on IoT devices with a multi-purpose IoT honeypot," Ph.D. dissertation, Indian Institute of Technology Kanpur, 2017. [Online]. Available: https://security.cse.iitk.ac.in/sites/default/files/15111021.pdf [Accessed: 10 Jan 2020]

57. M. Wang, J. Santillan, and F. Kuipers, "ThingPot: an interactive Internet-of-Things honeypot," *arXiv preprint arXiv:1807.04114*, 2018. [Online]. Available: https://arxiv.org/pdf/1807.04114.pdf [Accessed: 10 Jan 2020]

58. Bloomberg, "Company Overview of Cymmetria Inc." 2019. [Online]. Available: https://www.bloomberg.com/research/stocks/private/snapshot.asp?privcapid=305449525 [Accessed: 10 Jan 2020]

59. Cymmetria, "Cymmetria," 2019. [Online]. Available: https://cymmetria.com/ [Accessed: 10 Jan 2020]

60. Cymmetria, "Honeycomb GitHub," 2019. [Online]. Available: https://github.com/Cymmetria/honeycomb [Accessed: 10 Jan 2020]

61. Cymmetria, "MazeRunner Product Whitepaper," 2016. [Online]. Available: https://www.cymmetria.com/wp-content/uploads/2017/10/MazeRunner-Product-White-Paper.pdf [Accessed: 10 Jan 2020]

239

62. Cymmetria, "MazeRunner Product Whitepaper," 2018. [Online]. Available: https://cymmetria.com/white-paper/cymmetrias-mazerunner-product-whitepaper/ [Accessed: 10 Jan 2020]

63. Cymmetria, "MazeRunner User Guide Community Edition," 2018. [Online]. Available: https://webcdn.cymmetria.com/wp-content/uploads/2018/02/MazeRunner-User-Guide-for-v1.10.0.pdf [Accessed: 10 Jan 2020]

64. Cymmetria, "Introducing ActiveSOC," 2016. [Online]. Available: https://cymmetria.com/white-paper/introducing-activesoc-whitepaper/ [Accessed: 10 Jan 2020]

65. Cymmetria, "Catching APT3 with cyber deception Three case studies," 2018. [Online]. Available: https://cymmetria.com/white-paper/catching-apt3/ [Accessed: 10 Jan 2020]

66. Cymmetria, "Unveiling Patchwork – the Copy-Paste Apt," 2016. [Online]. Available: https://s3-us-west-2.amazonaws.com/cymmetria-blog/public/Unveiling_Patchwork.pdf [Accessed: 10 Jan 2020]

67. M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238. [Online]. Available: https://static.usenix.org/publications/library/proceedings/lisa99/full_papers/roesch/roesch.pdf [Accessed: 10 Jan 2020]

68. Open Information Security Foundation, "Suricata — Open Source IDS / IPS / NSM engine." [Online]. Available: https://suricata-ids.org/ [Accessed: 10 Jan 2020]

69. Scapy Project, "Scapy." [Online]. Available: https://scapy.net/ [Accessed: 10 Jan 2020]

70. "About SQLite." [Online]. Available: https://www.sqlite.org/about.html [Accessed: 10 Jan 2020]

71. Cisco, "Snort Rules and IDS Software Download." [Online]. Available: https://www.snort.org/downloads [Accessed: 10 Jan 2020]

72. B. McNeese, "Anderson-Darling Test for Normality," 2011. [Online]. Available: http://www.spcforexcel.com/knowledge/basic-statistics/anderson-darling-test-for-normality [Accessed: 10 Jan 2020]

73. J. Wilber, "Permutation Test: Visual Explanation," 2019. [Online]. Available: https://www.jwilber.me/permutationtest/ [Accessed: 10 Jan 2020]

74. T. Leeper, "Permutation Tests," 2013. [Online]. Available: https://thomasleeper.com/Rcourse/Tutorials/permutationtests.html [Accessed: 10 Jan 2020]

75. A. Downey, "Probably Overthinking It: There is only one test!" 2011. [Online]. Available: http://allendowney.blogspot.com/2011/05/there-is-only-one-test.html [Accessed: 10 Jan 2020]

76. Python Software Foundation, "Sunsetting Python 2 — Python.org." [Online]. Available: https://www.python.org/doc/sunset-python-2/ [Accessed: 10 Jan 2020]

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

HoneyHive – A Network Intrusion Detection System Framework Utilizing Distributed Internet of Things Honeypot Sensors

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Madison, Zachary D, Capt, USAF

5d. PROJECT NUMBER

19G437

5e. TASK NUMBER

5f. WORK UNIT NUMBER

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

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

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-20-M-038

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

Joseph A. Misher
Department of Homeland Security
Cyber Physical Division, Federal Protective Service
800 North Capitol Street NW, Washington D.C. 20001
COMM 202-658-8806
Email: Joseph.misher@hq.dhs.gov

10. SPONSOR/MONITOR'S ACRONYM(S)

DHS

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

Exploding over the past decade, the number of Internet of Things (IoT) devices connected to the Internet jumped from 3.8 billion in 2015 to 17.8 billion in 2018. Because so many IoT devices remain upatched, unmonitored, and left on, they have become a tantalizing target for attackers to gain network access or add another device to their botnet. HoneyHive is a framework that uses distributed IoT honeypots as Network Intrusion Detection Systems (NIDS) sensors that beacon back to a centralized Command and Control (C2) server. The tests in this experiment involve four types of scans and four levels of active honeypots against the HoneyHive framework and a traditional NIDS on the simulated test network. This research successfully created a framework of distributed network intrusion detection IoT honeypot sensors that capture traffic, create alerts, and beacon back to a central C2 server. The HoneyHive framework successfully detected intrusions that traditional NIDS cannot through the use of distributed IoT honeypot sensors and packet capture aggregation.

15. SUBJECT TERMS

Cyber Security, Network Security, Network Monitoring, Command and Control, HoneyHive, Honeypot, IoT, NIDS, Honeyd, Snort, Suricata, Nmap, Node.js, Python, SQLite, Scapy, PCAP, VMware, Docker, Honeytokens, Cyber Deception

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. B. E. Mullins, AFIT/ENG
U	U	U	UU	260	19b. TELEPHONE NUMBER *(include area code)* (937)-255-3636 x7979; Barry.Mullins@afit.edu

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18