3-23-2017

Framework for Industrial Control System Honeypot Network Traffic Generation

Htein A. Lin

FRAMEWORK FOR INDUSTRIAL CONTROL SYSTEM HONEYPOT NETWORK TRAFFIC GENERATION

THESIS

Htein A. Lin, Captain, USAF

AFIT-ENG-MS-17-M-046

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-17-M-046

FRAMEWORK FOR INDUSTRIAL CONTROL SYSTEM HONEYPOT NETWORK TRAFFIC GENERATION

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 Cyber Operations

Htein A. Lin, B.S.C.S, M.S.M.I.S.

Captain, USAF

March 2017

AFIT-ENG-MS-17-M-046

FRAMEWORK FOR INDUSTRIAL CONTROL SYSTEM HONEYPOT NETWORK TRAFFIC GENERATION

THESIS

Htein A. Lin, B.S.C.S, M.S.M.I.S.
Captain, USAF

Committee Membership:

Lieutenant Colonel Mason J. Rice, Ph.D.
Chair

Mr. Stephen J. Dunlap
Member

Barry E. Mullins, Ph.D., P.E.
Member

Abstract

Defending critical infrastructure assets is an important but extremely difficult and expensive task. Historically, decoys have been used very effectively to distract attackers and in some cases convince an attacker to reveal their attack strategy. Several researchers have proposed the use of honeypots to protect programmable logic controllers, specifically those used to support critical infrastructure. However, most of these honeypot designs are static systems that wait for a would-be attacker. To be effective, honeypot decoys need to be as realistic as possible. This thesis introduces a proof-of-concept honeypot network traffic generator that mimics genuine control systems. Experiments are conducted using a Siemens APOGEE building automation system to provide experimental inputs using single and dual subnet instantiations. A custom designed Distributed Network Traffic Generator is used to generate traffic on a decoy network. Output traces from multiple experimental trials are compared against controlled input traces. Analyzing the results indicate that the proposed traffic generator is successfully able to generate control system network traffic that originate and terminate with the honeypot systems. The generated traffic matched the same number of packets, content, and ordering of the original trace. The trials demonstrated that using a network traffic generator along with honeypots can generate and route control system traffic within a decoy network.

AFIT-ENG-MS-17-M-046

To my family, my wife and children, thank you for allowing me the long hours and nights while working on this research. Your support was crucial in my successful completion of this endeavor.

Acknowledgements

I would like thank Stephen Dunlap for his knowledge, skill and time in developing the research.

I would also like to thank LTC Mason Rice, my advisor, for his crucial guidance and countless hours throughout the thesis process.

Finally, I would like to thank all my classmates for their support and companionship throughout my time here at AFIT.

Htein A. Lin

Table of Contents

Page

Page

List of Figures

List of Tables

List of Acronyms

ALN	Automation Level Network
API	application programming interface
ARP	Address Resolution Protocol
BAS	building automation system
BBMD	BACnet/IP Broadcast Management Device
C2	command and control
COTS	commercial off-the-shelf
CUT	component under test
D-ITG	Distributed Internet Traffic Generator
DNTG	Distributed Network Traffic Generator
FLN	Field level network
GPLv3	GNU General Public License version 3
GUI	graphical user interface
HMI	human-machine interface
HTTP	Hypertext Transfer Protocol
ICS	Industrial Control Systems
IDS	Intrusion Detection System
IP	Internet Protocol

IPS	intrusion prevention system
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IT	Information Technology
MAC	media access control
MLN	Management Level Network
NIC	network interface card
NTG	network traffic generator
OS	Operating System
OT	Operational Technology
PCAP	packet capture file
PID	process identification number
PLC	programmable logic controller
PXCM	programmable controller modulars
SCADA	Supervisory Control and Data Acquisition
SUT	system under test
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

FRAMEWORK FOR INDUSTRIAL CONTROL SYSTEM HONEYPOT NETWORK TRAFFIC GENERATION

I. Introduction

1.1 Background

The United States Ghost Army conducted deception operations in France, Belgium, Luxembourg and Germany during World War II [1]. The military unit's mission was to deceive the enemy and lure German units away from Allied combat units. Engineers set up inflatable armored tanks, aircraft, airfields, tents and motor pools. Other tactics such as looping convoy traffic, deploying military police and posting General and Staff Officers in public places were used to distract Axis resources (e.g., intelligence gathering and combat power) away from real targets. The Ghost Army also played recordings of actual armored and infantry units over loud speakers. Deceptive radio transmissions were broadcast on fabricated networks called "Spoof Radio". Through these actions, the Ghost Army achieved a comprehensive deployment of decoys and deception techniques to overload enemy sensors and intelligence gathering capabilities.

Deception techniques and decoy technologies are employed in cyberspace in the form of honeypots. These systems may be simple (e.g., virtual machine) or complex (e.g., full scale replicas of industrial control systems). Non-full scale Industrial Control Systems (ICS) honeypots do not generate traffic to truly represent a control system. As such, attackers with the ability to passively monitor a target may be able to differentiate an operational system from a honeypot. Network traffic generators may

help with this problem, but they are geared for traditional network performance testing on Information Technology (IT) systems.

1.2 Motivation

This research details some of Operational Technology (OT) challenges, security concerns, threats and vulnerabilities. It highlights threat mitigation through the use of honeypots, similar to the decoys used by the Ghost Army. The implementation of honeypots themselves are often hindered by their own challenges. High-interaction honeypots often have a high financial and managerial cost associated with them, while low-interaction honeypots have reduced authenticity due to the limited amount of services that are emulated [7].

The effectiveness of honeypots often require that they are the target of attacks. OT differ from IT systems in that operations occur on a routine basis even without user interaction. Because of this, an attacker capable of passively monitoring a control network can identify honeypots that lack authentic network traffic. Thus, there is a need for network traffic generators on honeypots, to make these decoy systems look, act and communicate like real control systems.

1.3 Research Goals

The goal of this research is to assess the capability of a network traffic generator to replay trace traffic on honeypot systems. In doing the research, commercial and open-source network traffic generators are reviewed with the following questions asked:

- Does the generated traffic maintain the characteristics of the original trace (traffic matching)?
- Does the generated traffic appear to originate and terminate with the honeypot

systems (honeypot integration)?

- Does the generated traffic traverse the designed network (network routing)?
- Can the network traffic generator send network traffic to and from multiple systems using a single trace input file and without C2 traffic during generation (scalability)?
- Achieving the above four goals, does the use of a network traffic generator on honeypots show control system activity during passive monitoring?

Achieving these goals, it is hypothesized that generating traffic using network traces (modified with honeypot characteristics) can create decoys that mimic real control systems on a network.

1.4 Approach

A framework is developed that consists of both honeypot and network traffic generator platforms. Targetable decoy systems on a network are provided using a honeypot platform (e.g., Honeyd). Deception is provided with network traffic generated through the use of a network traffic generator (e.g., commercial, open-source or custom made Distributed Network Traffic Generator (DNTG)). By combining these two platforms, the framework provides decoy systems with deception capabilities during both active network scans and passive network monitoring.

1.4.1 Protocol Emulation.

Protocol emulation is provided through the use of network traces, which alleviates the need for priori knowledge of protocols and operational configuration of control systems. For this research, network traces are captured from a test environment

consisting of an APOGEE building automation system platform. This trace contains the control systems' routine network activity.

1.4.2 Honeypot Integration.

To meet the purpose of providing network activity for honeypots, a collected trace is used during network traffic generation. However, an unmodified trace contains the characteristics of the control systems as originally captured. A sub-function tool of DNTG is used to search for control system characteristics (e.g., Internet Protocol and Media Access Control addresses) within the Open Systems Interconnection model Layer 2 and Layer 3 packet headers. Matching results are replaced with corresponding honeypot characteristics. The end result is network traffic that bears honeypot characteristics derived from real control systems.

1.4.3 Network Traffic Generation.

A sub-function tool of DNTG provides network traffic replay using network traces and is designed to complement existing honeypot solutions. This tool is run on each instance of honeypot so that network traffic traverses from each decoy system through each connected point in the network. With this distributed design, each honeypot instance generates and receives network traffic.

1.4.4 Experimentation.

All experimental network traffic in this research is generated using DNTG instances on three Honeyd honeypot instances. Each one of these systems represent the corresponding control system devices from the APOGEE platform. An experiment is developed to test network traffic generation of each honeypot using a reference trace. The experiment consists of replaying all the packets within the trace and re-

capturing the generated traffic to be used as a comparison against the original trace. The reference and generated trace are then compared to validate and assess criteria measurements for honeypot-network traffic generation.

1.5 Assumptions and Limitations

This research provides a proof-of-concept framework for generating network traffic on honeypots using network traces.

1.5.1 Limitations of Network Trace-Based Approaches.

Trace-based approaches have limitations in that traffic generation is limited to what was previously recorded. The data contained within the trace packets represent the state of control system devices when the capture was originally made. Replayed data can contradict subsequent system state changes. Additionally, a network traffic generator is only as effective as the duration of the trace recorded. When used in 24/7 operations, the network traffic generator may replay the same trace multiple times resulting in repeated identical packets. A longer duration in the trace will result in fewer repeated packets over a certain period of time.

1.5.2 Network Protocols Involved.

While DNTG was designed to work with any network trace, only traces containing P2 and BACnet protocols were tested. Additionally, DNTG is limited to supporting Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Internet Protocol version 4 (IPv4) traffic.

1.5.3 Limited Configuration Setup.

This research tests network traces based on three control systems (i.e., one human machine interface and two programmable controller modular devices) on two network topologies (i.e., single-subnet and dual-subnet). A production environment may consist of a multitude of varying control systems operating in many different network topologies. Note that proof-of-concept does not simulate these complex environments.

1.5.4 Timing.

Control system operations are based on timed intervals which generate network traffic based on system configurations. Trace timing are based on the timestamps as recorded by a capture device. It is not possible to generate exact packet timing based on the network trace alone without specific knowledge of protocols and control system configurations. This research does not address timing delays added by network devices (e.g., router, switches and network interface cards). As a result the delays added during capture and network traffic generation may create a difference between experimental and original control system timing.

1.5.5 Network Exploitation.

This research covers network exploitation and access, however, the technical details and steps are not discussed. For this research an assumption is made that attackers are capable of exploiting a network and monitoring the traffic.

1.6 Thesis Overview

Chapter II contains background and related research on ICS threats, honeypot technology and network traffic generator products. Chapter III provides a detailed

description of the developed Distributed Network Traffic Generator. Chapter IV explains the experimental design with the results covered in Chapter V. Chapter VI presents research conclusions and future work.

II. Background and Related Research

2.1 Overview

This chapter provides a background on the vulnerabilities and threats that affect industrial control systems. A brief background on previous honeypot research is provided. Criteria for network traffic generation is established and a review of several available commercial and open-source network traffic generators is made.

2.2 Background

Critical infrastructure systems have long been considered immune to network attacks that have plagued traditional IT systems. Historically, process control and Supervisory Control and Data Acquisition (SCADA) systems relied on proprietary hardware, software and isolation for security. A convergence between IT and OT is pushing a move towards open standards based on technologies and protocols such as Ethernet, TCP/Internet Protocol (IP) and other web technologies [4]. According to Gartner, OT is becoming more like IT systems (including their vulnerabilities) [16].

There are many challenges in the OT/IT convergence that make mitigating security issues difficult. OT systems often run the software as installed without updates for 15-20 years, compared to the 3-5 year rigorous life cycle of IT systems [24]. Trends in the evolution of system architecture of SCADA systems since the 1960s show a drastic decline in use of proprietary hardware (from 60% to 2%) and software (from 100% to 30%) [18]. As a result, security policies such as "security through obscurity" used for older-generation OT platforms are no longer applicable to these newer systems [16].

Resource constraints pose a big challenge to control systems. Compared to IT where systems are designed to handle multiple functions and have resources available

to support addition of third-party security applications, OT systems are designed to support specific industrial processes. Adding resources or features may not be possible and these systems often lack the memory or computing resources to support the addition of security. Implementing traditional IT security practices on control systems may cause timing disruptions, negatively impacting performance and availability [24].

Trend Micro [27] published a study covering attacks on external-facing ICS devices and honeypot technology developed to capture threat actors and motivations behind attacks. The document highlighted five tasks conducted by attackers: (i) reconnaissance using free and open sources (e.g., ShodanHQ); (ii) port scanning of an intended IP target and surrounding subnets; (iii) fingerprinting of devices for Operating System (OS) and other identifiable information; (iv) persistence and lateral movement; and (v) data exfiltration.

A report by Idaho National Laboratory [8] highlights security challenges that are often associated with IT networks that are applicable to OT systems through convergence. Critical cyber security issues that need to be addressed in OT include those related to: (i) backdoors and holes in network perimeter; (ii) protocol vulnerabilities; (iii) attacks on field devices; (iv) database attacks; and (v) communications hijacking and 'man-in-the-middle' attacks. The report recommends and provides guidance for developing defense-in-depth strategies as a best practice for controls systems in multi-tier information architecture. In building defense-in-depth for OT, time sensitive requirements (e.g., clock cycles on PLCs) may make proven IT security technologies inappropriate for control systems. Another recommendation is the use of an Intrusion Detection System (IDS), capable of watching traffic and network activity passively without impacting traffic. It functions by comparing data against pre-defined rule sets and attack signatures. While the use of contemporary IDS signatures works for a wide range of attacks, for control networks they are inadequate, and modern IDS

may be blind to attacks on control systems.

2.3 Related Research

2.3.1 Honeypots as Defense-in-Depth Security Measure.

Honeypots can help mitigate the control system threats mentioned earlier. A honeypot is a decoy based intrusion detection technology that attracts hackers and provides the ability to study attacker actions and behaviors [5, 11]. Honeypots can be broken down in two type of systems: (i) low-interaction; and (ii) high-interaction. Low-interaction honeypots emulate services and operating systems, provide limited activity and are generally easy to deploy. High-interaction honeypots use real operating systems, applications and hardware to provide a more realistic environment but are far more difficult to deploy [7].

Prior research demonstrated applications of honeypot emulators and proxy technologies to create hybrid honeypots capable of being used to protect ICS systems. Winn et al. [28] showed that honeypots could be combined with a PLC proxy to provide multiple instances of systems. Winn et al. also provided a means to create low-cost ICS honeypots that are authentic and targetable by using data from the a programmable logic controller (PLC) while maintaining authenticity with unique network identities (e.g., IP address and media access control (MAC) address) that match corresponding honeypot devices. Warner's [26] research focused on developing a framework that automatically configured the emulation behavior by building protocol trees from networked PLC traces (captured network data). Girtz et al. [6] further extended Warner's work by forwarding unknown requests to a PLC proxy to provide a response and update to the protocol tree. The research bolstered the targetability and authenticity of ICS honeypots and the ability to emulate the characteristics and interactions of a PLC.

Successful implementation of honeypot technology could aid existing cyber security by adding defense-in-depth measures to mitigate vulnerabilities. Complimenting conventional security technologies (e.g., firewalls, IDS/intrusion prevention system (IPS) and defense-in-depth techniques) with honeypots can provide early intrusion detection, threat intelligence collection against unknown vectors and provide defenders valuable knowledge and time to address security concerns [22]. However, by design, honeypots do not have authorized activity and are never meant for operational use. As such, any activity on these systems can be considered suspicious [11]. Honeypots function as a litmus test to detect unauthorized access. The downside to the current approach is that honeypots do not actively engage in autonomous network communication. Instead, they rely on interaction with an attacker to generate network activity. This poses an issue because real OT networks have non-stop and recurring traffic flow. Unlike traditional IT systems, where servers may only require interaction with users and can sit idle until requests are made, OT systems are automated and perform operations in the absence of connected users. OT systems communicate status continually. If an attacker were to target a honeypot system, the absence of this OT network traffic would indicate that it was not part of a real control system. As a result, the honeypot would no longer entice the attacker, prompting a new attack target. This reduces the overall effectiveness of a honeypot as a security measure.

2.3.2 Network Traffic Generation.

Some understanding of network architecture is necessary as network traffic visibility depends on the level of compromise. In a switched network, an attacker may only be able to map the network using broadcast messages. The segregation of traffic on a switched network would normally prohibit an attacker from seeing all control

system traffic. Traffic collected would be restricted to the packets originating from (or destined for) a compromised host. For a more skilled attacker, network device exploitation can offer different vantage points that allow for all traffic to be visible. Using Figure 1 as an example, compromising the switch on Subnet 2 could reveal all traffic within that particular subnet. Layer 3 traffic can be seen when compromising the router. Exploiting the Subnet 1 switch or human-machine interface (HMI) would reveal traffic from all control systems that communicate with the HMI. An attacker may be able to isolate active systems from non-active systems (e.g., honeypots) by passively monitoring the traffic after compromising key nodes on the network.

The study of network activity is important for an attacker as it can increase the odds of a successful attack. If an attacker were to focus her resources at the wrong target (e.g., honeypot), the cost of revealing attack information would be detrimental. As such, more vigorous network discovery, reconnaissance and network enumeration actions would most likely occur prior to any actual attack, especially when using a zero-day exploit. At the system level, an attacker could collect traffic data going to and from the compromised host. This would reveal the identity and function of the machine to an attacker. At the network level, a compromised network device provides traffic data from connected systems. With this data, an attacker would be able to identify control systems by observing traffic patterns and packet contents. This would also identify false targets (honeypots) due to lack of any traffic originating from said devices.

It becomes important that honeypots designed for ICS reflect the same data traffic and patterns as the systems they are trying to emulate. The implementation of full network traffic generation would then make honeypots more effective as decoys. This research introduces the use of network traffic generators in conjunction with low-interaction and hybrid honeypots.

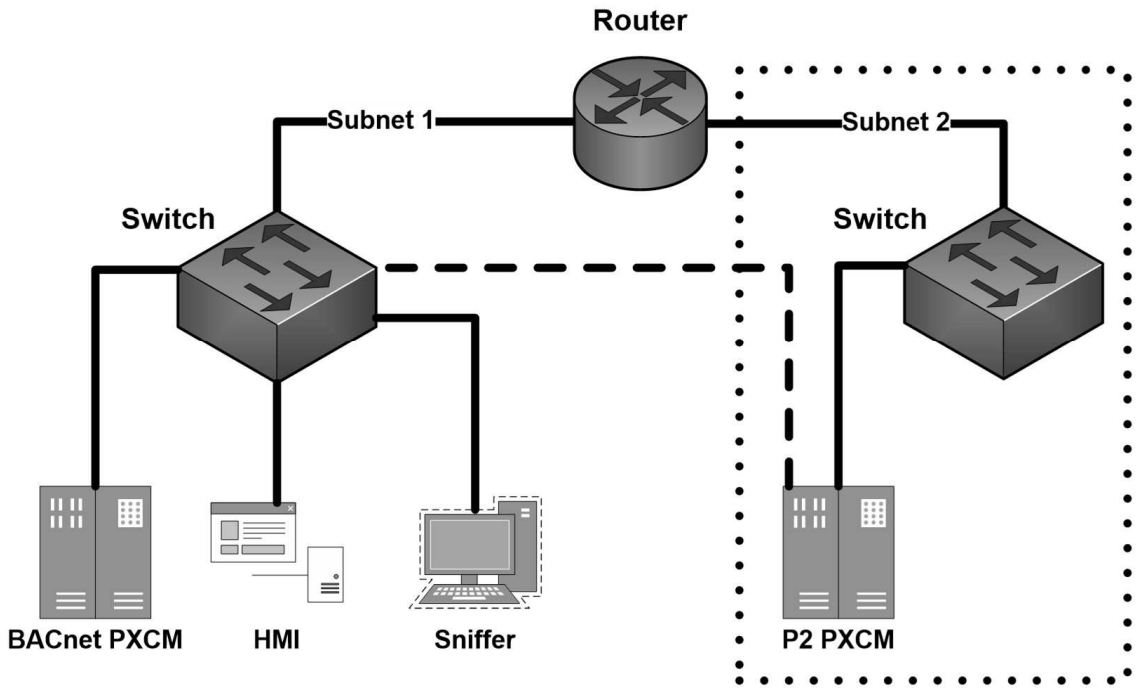

Figure 1. APOGEE platform dual-subnet network design.

To meet the requirements for ICS honeypot-network traffic generation, the criteria in Table 1 were formulated to assess viable network traffic generators.

2.3.3 Network Traffic Generators.

Several commercial off-the-shelf and open-source network testing products were selected as candidates to provide honeypot-network traffic generation. They were evaluated based on product literature review against the criteria from Table 1.

2.3.3.1 Commercial Network Traffic Generators.

Several commercial products were considered including: (i) SolarWinds WAN Killer: Network Traffic Generator; (ii) NetLoad Inc. Stateful Traffic Mix Tester Solution; and (iii) Ixia IxLoad.

SolarWinds WAN Killer: Network Traffic Generator is one of the over 60 network

Table 1. Honeypot-network traffic generator evaluation criteria.

Traffic Matching	**Content Matching:** Generated traffic should match the packet data of the control system the honeypot is emulating. **Extraneous Packets:** Packets that do not match control systems must be avoided during generation (e.g., network traffic generator command and control and synchronization). **Packet Ordering:** Generated traffic should not have out of sequence packets. **Timing Consistency:** Generated traffic should replicate timing from trace data as accurately as possible.
Honeypot Integration	**Honeypot Pairing:** Generated traffic must be received by and sent from the corresponding honeypot. **Honeypot Header Matching:** IP and MAC addresses in generated packet headers should reflect corresponding honeypot systems.
Network Routing	**Distributed Operation:** Traffic generation should originate from multiple points within a network. **Layer 2 Forwarding:** Generated traffic which shares the same characteristics as its corresponding honeypots must not cause network addressing issues (e.g., MAC table conflicts). **Layer 3 Routing:** Generated network traffic must be routable in multi-subnet environments.
Scalability	**Cost:** ICS may consist of systems that are distributed both physically and logically on a network requiring a large number of honeypot instances. A high cost for each network traffic generator instance will make it cost prohibitive to replicate a full control system. **Flexibility:** ICS may consist of many diverse components, often distributed across geographically dispersed environments. To accurately generate control system traffic, network traffic generator instances may need to be installed in multiple locations. The network traffic generator must be configurable to accommodate a large variety of ICS applications.

management tools that comes as part of Engineer's Toolset (priced starting at 1,495 dollars) [23]. The WAN Killer tool lets network administrators generate random traffic on a Wide Area Network. The tool can manipulate packet size, circuit bandwidth and percent of bandwidth utilization with randomly generated data. The tool was designed to simulate network traffic primarily for load testing and does not generate the specific protocols traffic required for ICS honeypots.

NetLoad Inc. Stateful Traffic Mix Tester Solution provides off-the-shelf network processing hardware and software as a single package (5,000 dollars for the 1 Gbps model and 25,000 dollars for the 80 Gbps model) [12, 14]. It provides network testing using TCP/Hypertext Transfer Protocol (HTTP), UDP network traffic generation and packet capture file (PCAP) replay. A particular feature of interest is the PCAP replay. With this PCAP replay feature, packets are generated using PCAP data. It can use packet time stamps mimicking the timing of the original PCAP. NetLoad advertises inter-packet timing within one microsecond [13]. The software identifies bi-directional traffic when using PCAP replay and allows directed traffic from two ports. Other features allows the PCAP replay load to be distributed among the four ports. While the PCAP replay would serve the purpose of replaying captured ICS traffic, the appliance was meant for network testing and does not implement honeypot integration features. The documentation did not indicate a way to load custom software (e.g., honeypot) on the appliance nor does it feature a way to modify PCAP data to match honeypot characteristics.

Ixia IxLoad is a suite of software and hardware that provides network performance testing [9]. Pricing for IxLoad starts at 3,500 dollars per subscription license and 33,750 dollars per perpetual license. The software can be used in a virtual environment loaded on a server or used in conjunction with Ixia's proprietary appliance products. IxLoad delivers a wide variety of fully stateful IT protocols to emulate

web, video, voice, storage, VPN, wireless, infrastructure and encapsulation/security protocols. For unsupported or propriety protocols, it provides TCP/UDP replay traffic options through its Application Replay feature [10]. This feature replays network packet captures and has the ability to create bidirectional traffic flows through unique ports. Options include modifying IP headers and using inter-packet timing from traces. Of the evaluated commercial solutions, IxLoad would best meet the network traffic generator requirement based on product literature and vendor contact. However, IxLoad was designed for network performance testing and not honeypot-network traffic generation. Multiple licenses may be required for implementation on large scale networks, making it cost prohibitive. Further evaluation is needed to determine if it meets all the required criteria for honeypot-network traffic generation (e.g., honeypot integration, traffic matching, network routing and network routing).

2.3.3.2 Open-source Network Traffic Generators.

Three open-source traffic generators were considered: (i) Ostinato; (ii) Distributed Internet Traffic Generator; and (iii) Tcpreplay.

Ostinato is a packet crafter, network traffic generator and analyzer with both graphical user interface (GUI) and Python application programming interface (API) implementations. The software operates in a controller-agent architecture, where the controller performs the command and control (C2) operations and the agents perform the network traffic generation [15]. The software generates traffic on the network using crafted packets or replay data from PCAPs. For the controller-agent architecture to function, Ostinato transmits trace data and device configurations from the controller to the agents during initialization. There were limitations observed during testing in timing implementation. During testing, it was observed that packet transmission was based on packets per second and burst modes. As a result, the generated traffic

did not maintain the original timing of the trace. This limitation was identified using WireShark analysis tools. A final implementation of honeypot-network traffic generator using Ostinato would require modifications to how timing is handled in the software. Another limitation was found in the software's handling of command and control (C2) operations. These operations (e.g., synchronization and state status) are continuously present on the network between the controller and agents. The presence of these identifiable packets on the network would likely alert an attacker that an Ostinato network traffic generator is running.

Distributed Internet Traffic Generator (D-ITG) is a platform capable of producing IPv4 and Internet Protocol version 6 (IPv6) traffic by replicating the workload of current Internet applications [2]. D-ITG generates traffic following stochastic models for packet size and inter-packet timing that mimic supported application-level protocol behavior. Packet size and inter-packet timing can also be loaded from capture files. Network traffic generation is performed between ITGSend (sender component of the platform) and ITGRecv (receiver component of the platform). A C2 connection exists between these two components which controls the traffic generation process for each traffic flow (e.g., port assignments). For large-scale distributed environments, multiple ITGSend instances can be remotely controlled by an D-ITG API. The review of D-ITG indicated that it did not meet the honeypot integration and traffic matching criteria. The platform offered no method to modify the header data to match honeypot characteristics. Traffic is generated one way from the ITGSend to ITGRecv instances. To establish two way traffic generation, multiple instances of ITGSend and ITGRecv are required on each pair of honeypots to perform conversational traffic flow. On large scale implementations, this can be an overwhelming task. Additionally, application layer payload data is ignored and only packet size and inter-packet timing from trace data is used. As such, it was determined that D-ITG

did not meet the honeypot integration and traffic matching criteria.

Tcpreplay, written by Aaron Turner and licensed under GNU General Public License version 3 (GPLv3), is a suite of tools that allows for previously captured traffic to be replayed through various network devices [25]. Tcpreply possesses the ability to generate trace data using recorded timestamps. It also supports bi-directional traffic flow through the use of two interfaces. The software suite also includes a tool to modify packet header information. Tcpreplay was selected as a candidate for a pilot study.

2.4 Chapter Summary

This chapter shows that ICS are vulnerable to attacks, especially more so through the OT/IT convergence. Attacks vectors include network-based attacks that compromise control networks and systems. One method of mitigating threats on OT systems is through the use of honeypots as a defense-in-depth security measure. Strides have been made to bridge the gap between low-interaction and high-interaction honeypots, and provide authentic user interaction with these systems. However, these honeypots still lack the capability to mimic OT systems in autonomous operation. Network traffic generation criteria were defined that highlight what generated control system traffic should emulate. Using these criteria, several commercial and open-source network traffic generators were reviewed.

III. Research Design

3.1 Overview

This chapter provides the considerations for network traffic generation and establishes the test environment to create the network trace needed for the traffic generation. Pilot studies on network traffic (i.e., APOGEE platform), honeypot systems (i.e., Honeyd), and network traffic generation (i.e., Tcpreplay) are made. The results of the pilot studies are used in the design of the honeypot-network traffic generator framework.

3.2 Test Environment

The test environment chosen for this research is based on building automation system (BAS) technology, specifically Siemens APOGEE. While this test environment may not fully replicate complex production environments, it provides the required network traffic traces from operational control systems. The components listed below were used in the test environment:

- Siemens Insight software which provides HMI and engineering workstation functionality.
- Ubuntu VM with Honeyd honeypot and network traffic generator software (Tcpreplay and a custom Distributed Network Traffic Generator).
- Ubiquiti EdgeRouter X router.
- Two Netgear ProSafe Plus switches.
- Two Siemens APOGEE PXC100 programmable controller modulars (PXCM) with input/output modules.

- Field level network (FLN) devices (e.g., sensors, lights and fans).

3.2.1 Design Consideration.

The environment was designed to replicate the APOGEE platform that provides BAS functionality on a single field panel as shown in Figure 2. Each PXCM was mounted side-by-side with wired connections to the FLN devices using the input/output expansion modules. Separate serial connections were made to a Siemens Simatic S7-200 PLC and Siemens 550-833 TEC unit conditioner circuit board. User feedback was provided through liquid-crystal display panels, physical lights and HMI.

3.2.2 Network Topology.

The APOGEE platform consists of three networks: (i) Management Level Network (MLN); (ii) Automation Level Network (ALN); and (iii) FLN. The MLN has servers and client workstations that provide the management controls for the APOGEE automation system [20, 21]. Hardware systems at this level consist of servers and workstations running Siemens Insight or InfoCenter software suites, web accessible terminals, mobile devices and PXCM with MLN functionalities. Communication integration between systems is provided using proprietary and open-standard protocols (e.g., P2, TCP/IP and BACnet). The ALN provides field panel to field panel as well as MLN to FLN communication. Hardware components found at this level consists of PXCM supervisory field panels. These panels can operate in networked or stand-alone configurations and provide control, monitoring and energy management functions to FLN devices. The FLN is the lowest level of the APOGEE building automation network. All end point devices reside at this level and vary depending on application (e.g., terminal equipment controllers and sensor units).

Network implementation for the test environment platform was based on the

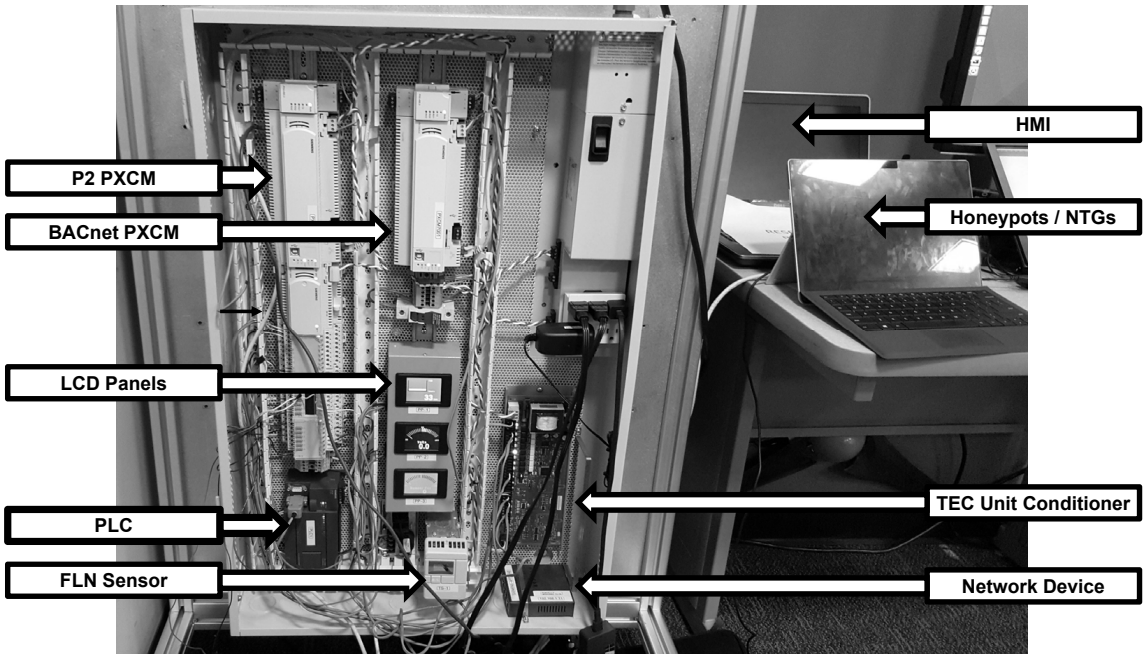

Figure 2. APOGEE platform.

recommended Ethernet single-subnet and multi-subnet configurations from Siemens' APOGEE technical specification sheet (see [19]). The two network topology scenarios considered were: (i) single-subnet; and (ii) dual-subnet. Due to equipment availability, a dual-subnet configuration was used to test the multi-subnet environment. The same P2 PXCM was used for both single-subnet and dual-subnet configurations with minimal wiring (e.g., connection to Subnet 1 or Subnet 2) and configuration changes (e.g., IP reassignment).

Figure 1 shows the APOGEE platform network design used for the test environment. The figure depicts a dual-subnet environment with two different Siemens PXCMs: (i) P2 protocol over TCP/IP; and (ii) BACnet protocol over UDP. A network sniffer was added to a mirrored port on Subnet 1 for network capture and analysis functionality. The single-subnet design represents BAS infrastructure for a single building campus. Within this infrastructure, the three APOGEE network layers belong to a single control system network. The multi-subnet design represents a

multi-building campus infrastructure consisting of BAS sites at multiple buildings. The control networks are distributed with network connectivity provided by individual switches. In this setup, a central router connects the multiple buildings (e.g., switches) to form the BAS network infrastructure.

Honeypots and network traffic generators were used to replicate the APOGEE platform (see Figure 3). A single honeypot/network traffic generator was used to replicate the HMI on Subnet 1. A network sniffer/C2 workstation was added to Subnet 1 and provides: (i) network capture and analysis functionality on one network interface card (NIC) connected to a mirrored port on the switch; and (ii) experimental C2 automation on a second NIC.

A third workstation (containing the PXCM honeypots/network traffic generators) was added and connected to both subnets. A total of three NICs were used to provide: (i) BACnet PXCM honeypot connection to Subnet 1; (ii) P2 PXCM honeypot connection to Subnet 1; and (iii) an additional P2 PXCM honeypot with connection to Subnet 2. Having connections to both subnets for the P2 PXCM/network traffic generator allowed multiple experiments to be conducted without significant changes between runs.

3.3 Pilot Studies

This section highlights the results of the pilot studies.

3.3.1 APOGEE Network Traffic Analysis.

Network communication between the PXCMs and the HMI workstation was analyzed to better understand what an attacker might see on the network. Traffic collection was performed on a Windows based client with open source software (e.g., Wireshark). Multiple collections were made with different settings on each switch

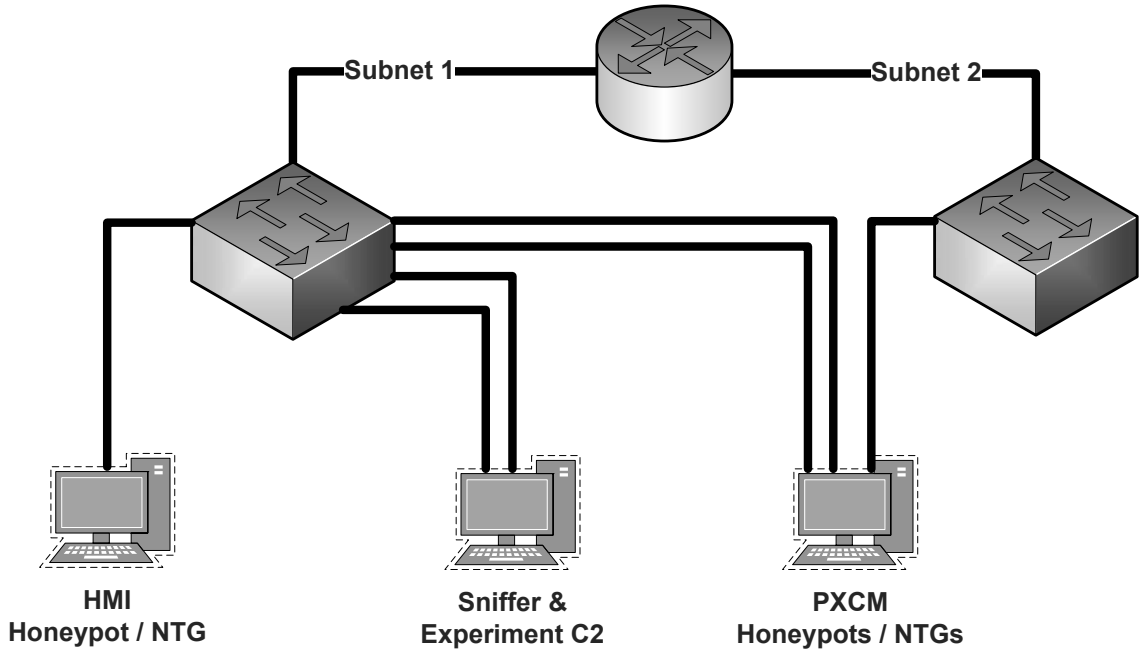

Figure 3. Test platform single and dual-subnet network design.

(e.g., switched and mirrored port configurations). In addition to the switch configurations, network traffic was collected with the control system devices running in normal operation and failed states.

The captures represent traffic that can be observed by an attacker at various levels of network compromise. The switched ports represent what an attacker capable of compromising a node within a network would see on that particular host. While the attacker may be able to see traffic traversing the compromised node, traffic destined for other nodes would not be seen. The mirrored port configuration replicates the access a more skilled attacker may be able gain through various exploits (e.g., media access control address flooding, administrative credential compromise, switch vulnerability exploit and switch misconfiguration). A successful network compromise could reveal all traffic traversing the subnet to an attacker.

Capturing traffic with the sniffer connected to a switched port collected no control traffic between the PXCMs and the Insight HMI workstation. This was expected as

unicast traffic would normally be restricted between the intended source and destination by the switch. When alternating the PXCM and the HMI workstation into failed states, Address Resolution Protocol (ARP) and BACnet/IP Broadcast Management Device (BBMD) broadcast message requests were seen. The analysis confirmed that the system uses unicast traffic for normal operations and broadcast messages to discover nodes. Since the broadcast message are visible without compromising the network, it does provide some information for an attacker to map a control system network.

The next set of traffic collected was conducted using a mirrored port. This resulted in the sniffer capturing all unicast traffic between the PXCMs and the HMI workstation. The capture included the C2 data for the APOGEE platform, to include periodic polling and manual requests. By alternating the PXCM and the HMI workstations into failed states, unicast TCP 3-Way handshake and BBMD messages were observed in addition to the broadcast requests found previously.

3.3.2 Identifying Honeypots.

The honeypots were built using Honeyd Virtual Honeypot [17]. Honeyd is a small daemon that creates virtual hosts running arbitrary services and configurable to appear as though it is running specific operating systems. It is capable of simulating multiple addresses from a single host. Building the honeypot configuration profile included retrieving identity data (e.g., OS fingerprint, MAC address and services) from the control systems using NMAP. The key information gathered was then transferred to the honeypot configuration profile for Honeyd. Note that the efficiency of honeypot software was not tested, it was only tasked with providing targetable nodes on the network. Final implementation of the network traffic generator should work with any other honeypot platform.

For each control system an equal number of honeypot systems were created. The resulting network topology from an active NMAP scan is shown in Figure 4. A honeypot system was created for each control system and provided with a unique IP and MAC address. For experimentation purposes, the honeypots were configured to drop packets from the network traffic generator. In final implementation, non-replay traffic and replay traffic can be segregated and handled by the appropriate honeypot/network traffic generator platform.

A pilot study was performed to demonstrate that the lack of OT traffic on the network can give away the identity of honeypots. NMAP was used to perform an active scan of the test network, which verified the honeypot systems along with the control systems were active and detectable on the network (see Figure 4). It was also used to validate that the characteristics of the honeypots created matched the corresponding APOGEE system. The resulting network topology shown in Figure 4 shows three real control systems (i.e., 10.1.3.2, 10.1.3.3 and 10.1.3.5) and three honeypot systems (i.e., 10.1.3.111, 10.1.3.112 and 10.1.3.104) for a single-subnet configuration. Note that a separate honeypot (i.e., 10.1.4.104) is used on Subnet 2 in a dual-subnet configuration. From an active scan standpoint, the introduction of the three honeypots into the experiment potentially decreases an attacker's target selection success by half (three real control systems out of six control systems). This target selection success rate can be further decreased by introducing additional sets of honeypots.

Since a more skilled attack may involve passive network monitoring, an assessment of the traffic data is required. To simulate the highest level of compromise on the network, a Wireshark sniffer was placed on a mirrored port of the switch servicing the HMI. With only one-minute worth of traffic data, the three real control systems were identified using Wireshark's endpoints statistics tool (see Figure 5). An attacker capable of viewing network traffic can conclude that the targets of interest

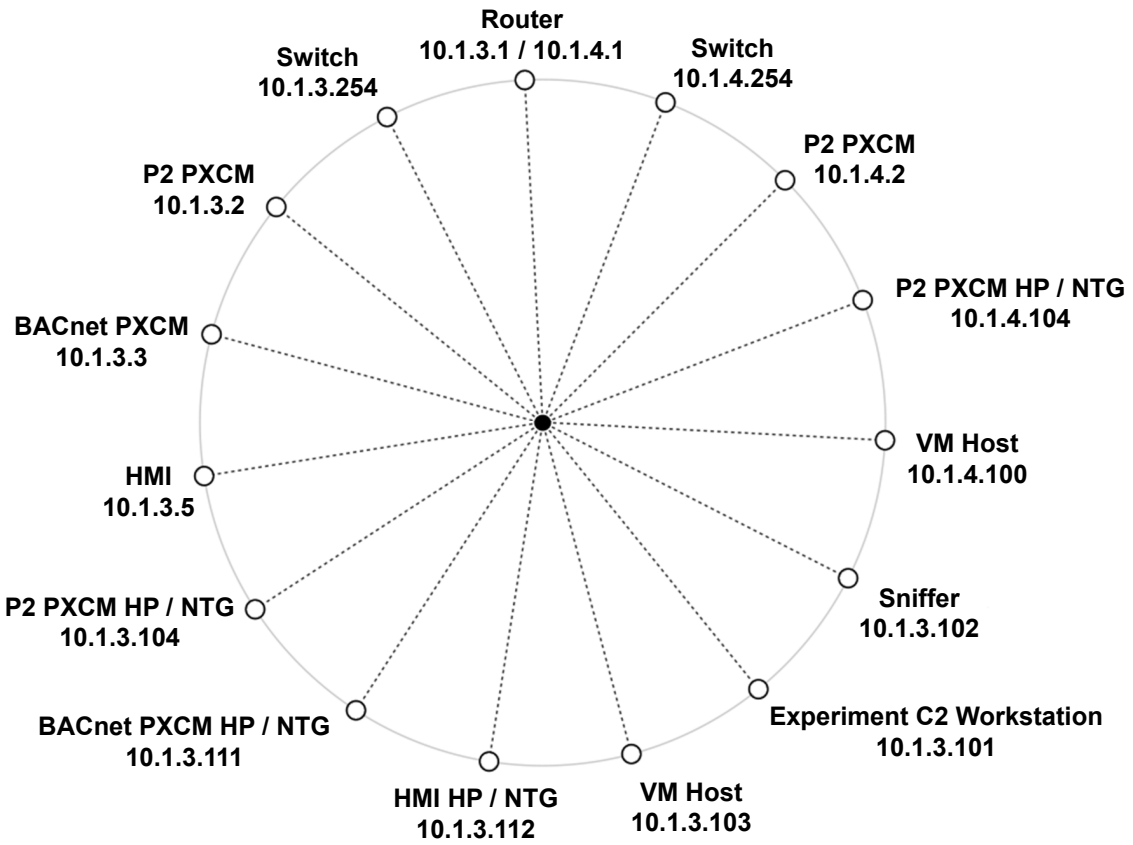

Figure 4. Test network implementation.

have IP addresses 10.1.3.3, 10.1.3.5 and 10.1.4.2 as these are the only systems that are actively transmitting on the network. With this additional reconnaissance, an attacker's target selection success returns to one-hundred percent. While the initial implementation of honeypot systems into the control network looked promising, this pilot study demonstrates that a skilled attacker can easily detect an OT honeypot through passive monitoring.

3.3.3 Tcpreplay.

For the purpose of this pilot study, network captures (single and dual-subnet configurations) were taken from the APOGEE platform. These PCAP files are referenced as the Production Trace. The Tcpreplay suite contains a variety of tools including:

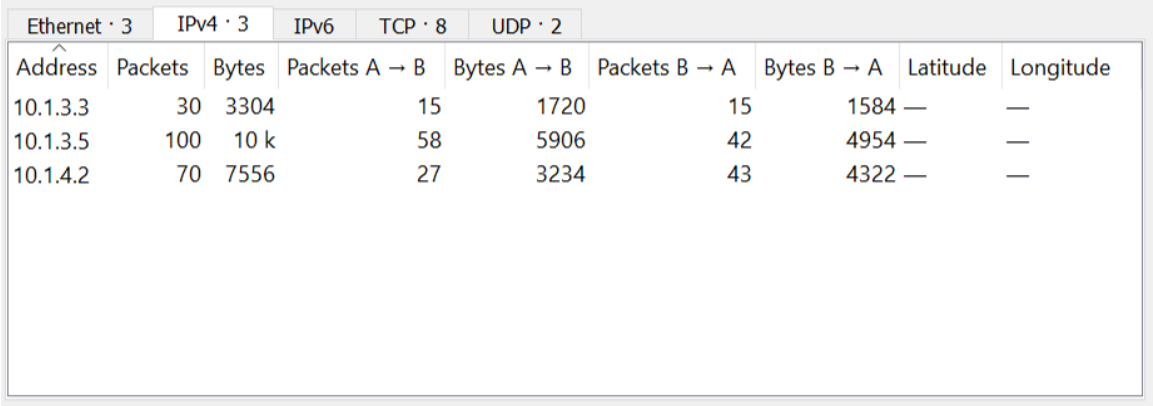

Figure 5. Pre-network traffic generation passive monitoring.

(i) tcpprep; (ii) tcprewrite; and (iii) tcpreplay [25].

Tcpprep is a PCAP pre-processor for tcpreplay and tcprewrite. It identifies and "splits" traffic into two sides within a conversation and assigns a network interface to each packet. It allows tcpreplay to generate network traffic through two NICs (primary/secondary and client/server). While it can emulate two-way traffic through two unique ports, network traffic generation is still limited to a single workstation and two interfaces. This would become a limitation in simulating an ICS environment which would normally involve a significant number of devices that are physically and logically distributed on a network.

Tcprewrite is a PCAP file editor which rewrites TCP/IP and Layer 2 packet headers. It replaces Layer 2 source and destination addresses of packets so they can be processed by the correct device. Operational tests found that while the tool was able to re-write packets it was dependent on tcpprep being able to properly identify conversations. Using a sample capture from the APOGEE platform, the tool failed to modify source and destination addressing of all packets found within.

Tcpreplay was used to generate traffic on the network using the original timing recorded in the Production Trace. A sample capture consisting of one minute of network traffic data was recorded by the sniffer. The results of this capture are shown

using Wireshark endpoints statistics tool in Figure 6. The figure shows that an attacker capable of monitoring the network would see traffic between all six systems (maintaining fifty-percent target selection success probability – three real control systems out of six total systems). Because the real and honeypot systems are generating traffic, an attacker would then have to perform a deeper analysis to determine the targets of interest. This requires more steps, which allows defenders time to detect and mitigate. This demonstrates that by adding decoys (with corresponding network traffic), the chance of target selection failure is also increased.

Once the initial test showed that Tcpreplay was capable of deception in passive scans, a full hour of traffic was generated on Subnet 1. Wireshark was used to re-capture the generated traffic for data collection and analysis on both Subnet 1 and Subnet 2. Various statistical tools found in Wireshark were used to compare the original capture and re-captured packets (referred to as Generated Trace).

Analysis of the traffic capture from Subnet 1 was made first. Using Wireshark's conversation statistics, which displays a list of conversations (traffic between two endpoints), it showed that there were four pairs of conversations. Conversation statistics for the Production Trace are shown in Figure 7 and the Generated Trace in Figure 8. Between the Production Trace and Generated Trace, the number of packets transmitted and total size of the conversations matched. However, there was an average delay of 11.12 ms before the start of a conversation and an increase of 113.65 ms in the duration of the conversations. Tcpreplay timing measurements (using methods described in Section 4.3) showed the difference in the packet interval (inter-packet timing) between the Production Trace and Generated Trace had a mean of 0.07 ms. Overall there was an increase of 132 ms in the total hour duration of the replay.

The input/output graph statistics in Wireshark shows the number of packets transmitted per unit of time over the course of the capture. Visually reviewing the

Ethernet · 5 | IPv4 · 6 | IPv6 | TCP · 17 | UDP · 4

Address	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Latitude	Longitude
10.1.3.3	32	3566	16	1853	16	1713	—	—
10.1.3.5	104	11 k	60	6171	44	5219	—	—
10.1.3.111	30	3304	15	1720	15	1584	—	—
10.1.3.112	103	11 k	42	4992	61	6202	—	—
10.1.4.2	72	7824	28	3366	44	4458	—	—
10.1.4.104	73	7890	46	4482	27	3408	—	—

Figure 6. Post-network traffic generation passive monitoring.

Ethernet · 4 | IPv4 · 4 | IPv6 | TCP · 58 | UDP · 3

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
10.1.3.111	10.1.3.112	1,803	199 k	902	103 k	901	95 k	0.000000000	3592.027962	230	212
10.1.3.111	10.1.3.255	2	120	2	120	0	0	700.733082000	1799.853148	0	0
10.1.3.112	10.1.4.104	4,308	468 k	1590	201 k	2718	266 k	0.289063000	3598.016725	448	591
10.1.3.112	10.1.3.255	6	360	6	360	0	0	700.735001000	2444.526332	1	0

System PCAP

Conversation	Rel Start (s)	Duration (s)
10.1.3.111 <> 10.1.3.112	0.000000000	3592.027962
10.1.3.111 <> 10.1.3.255	700.733082000	1799.853148
10.1.3.112 <> 10.1.4.104	0.289063000	3598.016725
10.1.3.112 <> 10.1.3.255	700.735001000	2444.526332

Figure 7. Tcpreplay conversation statistics (Production Trace).

Ethernet · 4 | IPv4 · 4 | IPv6 | TCP · 58 | UDP · 3

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
10.1.3.111	10.1.3.112	1,803	199 k	902	103 k	901	95 k	0.000000000	3592.159244	230	212
10.1.3.111	10.1.3.255	2	120	2	120	0	0	700.750636000	1799.940574	0	0
10.1.3.112	10.1.4.104	4,308	468 k	1590	201 k	2718	266 k	0.289150000	3598.148088	448	591
10.1.3.112	10.1.3.255	6	360	6	360	0	0	700.750709000	2444.630851	1	0

Generated PCAP

Conversation	Rel Start (s)	Duration (s)
10.1.3.111 <> 10.1.3.112	0.000000000	3592.159244
10.1.3.111 <> 10.1.3.255	700.750636000	1799.940574
10.1.3.112 <> 10.1.4.104	0.289150000	3598.148088
10.1.3.112 <> 10.1.3.255	700.750709000	2444.630851

Figure 8. Tcpreplay conversation statistics (Generated Trace).

patterns shows transmission occurs nearly the same. Figure 9 shows a graph of the packet transmission rates found in sample Production Trace and Generated Trace.

The two graphs are transposed over each other to show that they follow a similar network traffic pattern. Some minor variations in the number of packets sent was observed within the 22-28 minute mark due to possible processing or networking delays. However, it would be difficult for an attacker to determine which pattern represents the real system from this visual representation.

While Tcpreplay statistics looked appealing initially, some limitations were discovered over the course of the pilot study. Sniffing on Subnet 2 revealed that only a limited number of packets destined for the P2 PXCM honeypot were received. This was attributed to port assignments made by the switch in its MAC address table and the way Tcpreplay operates. Since Tcpreplay transmits the generated traffic via an interface on Subnet 1, all the MAC addresses from the Production Trace are entered into the switch's address table and assigned to a particular port on Subnet 1. All network communications (including those outside of the generated traffic) destined for P2 PXCM honeypot would then be sent to this specific port. Occasionally, the MAC tables would refresh with the actual location of the honeypot (via router MAC address). During these periods, packets are forwarded correctly to the router and routed to the correct subnet. This demonstrated a limitation in using Tcpreplay for distributed systems in multi-subnet environments.

If a network topology consists of a single subnet then implementation of Tcpreplay would be a viable solution. It would also require that the honeypots be collocated on the same workstation with Tcpreplay, otherwise multiple associations of non-unique MAC addresses would occur. If any particular honeypot is placed separately from the network traffic generator, an attacker would not see any replayed traffic on that particular host. This also creates a hardware limitation for designs requiring multiple honeypots on the same workstation. The pilot study revealed that Tcpreplay did not meet the honeypot integration, network routing and scalability criteria as detailed in

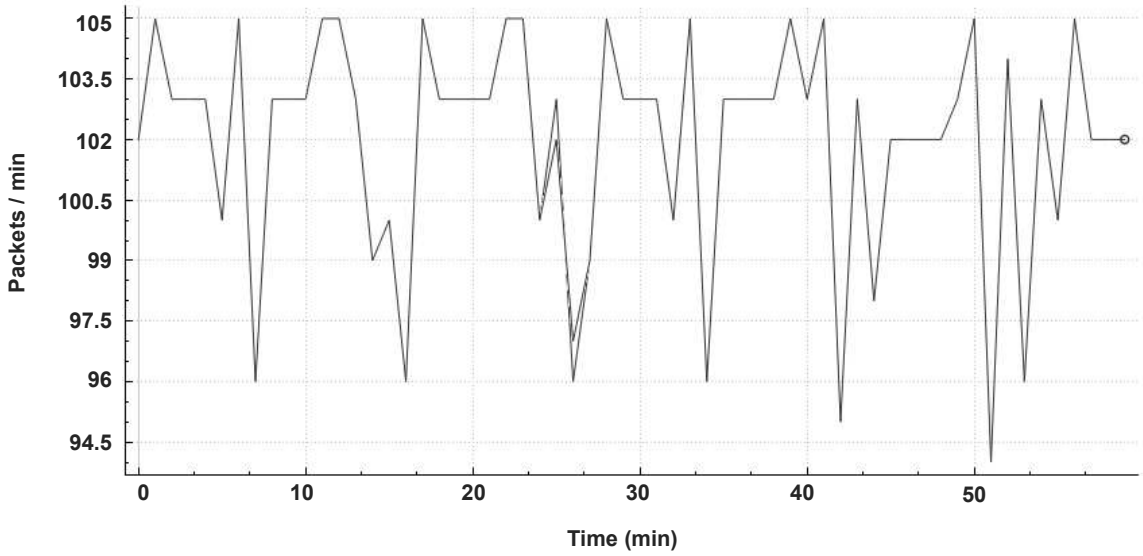

Figure 9. Production Trace and Tcpreply Generated Trace transmission rate.

Table 1.

3.4 Honeypot-Network Traffic Generator Framework

While authentic network traffic can be generated through the use of full scale systems, it is often associated with a very high cost [28]. Low-interaction and hybrid honeypot systems can be used to emulate ICS for a much lower cost. To provide authenticity for these honeypot systems, using protocol specific traffic generation requires significant knowledge base, time and resources to authentically replicate the operations of a genuine system.

An alternate solution involves using packet captures for network traffic generation. Using this method, authenticity is maintained as the data and patterns are derived from real systems. Note that, by using this solution, confidentiality of the traffic is not maintained. Justification for this, however, is that a capable attacker would most likely be able to view the same traffic from real systems in the absence of honeypots. One way to mitigate this and maintain some form of confidentiality is the use of multiple sets of false captures. These false captures could be generated by

real systems running in a fake operational environment. The traffic, while it would appear real, would otherwise be useless operationally. In this scenario, the honeypots and real systems would have distinct traffic, yet they would still present themselves as targets of opportunity for attackers, functioning as decoys.

Design of the honeypot-network traffic generator framework is divided into five sections: (i) production network; (ii) honeypot platform; (iii) honeypot integration; (iv) distributed network traffic generation; and (v) decoy network. A block diagram of this design is shown in Figure 10. The production network provides system characteristics (e.g., OS fingerprint, network addresses and trace data). The honeypot platform provides targetable honeypot systems to a decoy network with matching characteristics and emulated services from the control systems found in a production network. The honeypot platform used was discussed in Section 3.3. Honeypot integration, involves taking trace data (captured from a production network) and modifying headers to match corresponding honeypot system. With honeypot integration completed, the trace data is used by network traffic generators to provide replayed ICS traffic into a decoy network or back into a production network. The replay function is broken down into two different sub-functions to: (i) match trace timing and packet ordering; and (ii) route packets on the network appropriately.

3.5 Distributed Network Traffic Generator

Due to commercial off-the-shelf (COTS) and open-source product limitations, a custom network traffic generator solution was designed. A DNTG platform was created using Python and Scapy following the design criteria from Table 1 with some design characteristics taken from D-ITG (see [3]). DNTG consists of two main functions: (i) DNTG Prep; and (ii) DNTG Replay.

DNTG Prep is a PCAP tool used to modify packet header information to match

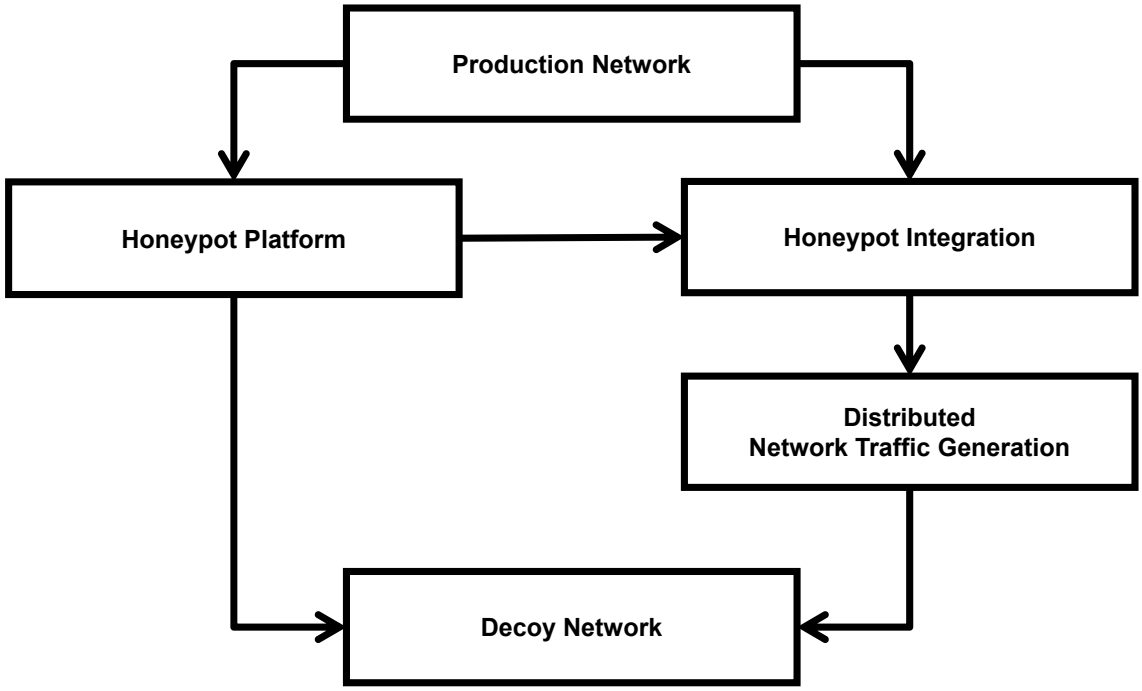

Figure 10. Honeypot-network traffic generator framework.

honeypot characteristics (e.g., IP and MAC addresses). The entire code for DNTG Prep can be found in Appendix 1.1. These characteristics are passed as parameters to DNTG Prep which locates corresponding control system packet data. These packets are then modified with honeypot information.

DNTG Replay was designed to be run on a honeypot device and is intended to operate in a distributed environment. The entire code for DNTG Replay can be found in Appendix 2.1. DNTG Replay operates in two modes: (i) listen mode; and (ii) send mode. In the listening mode, the network traffic generator performs no actions until a packet arrives. Once a packet arrives, it performs a check to see if it is a part of the Production Trace. If a match is found, the appropriate response packet is placed in a queue for network traffic generation. The queued packet is then transmitted based on the inter-packet timing from the Production Trace.

3.5.1 DNTG Prep: Honeypot Integration.

Recall from Figure 1 that main traffic collection occurs on Subnet 1. This is of particular importance because all control systems communicate with the HMI and the control system network traffic would be visible on this subnet. For an attacker, this would be the best collection point for network traffic and ideal target (e.g., HMI or network switch) to compromise. The network traffic was collected using Wireshark on a mirrored port with a duration window of 10 minutes. Wireshark IP filters were used to filter out all traffic except for the APOGEE platform. For the single-subnet, the following IPs were filtered 10.1.3.2, 10.1.3.3 and 10.1.3.5. For the dual-subnet setup, the following IPs were filtered 10.1.4.2, 10.1.3.3 and 10.1.3.5. A separate capture file was made for each subnet configuration.

The Production Trace used for traffic generation contains characteristics of the real control systems. However, the honeypot integration criteria requires that the generated traffic match the honeypot characteristics (e.g., IP and MAC addresses). To meet this criteria, the trace must be modified prior to generation. Preparation of the Production Trace using DNTG Prep was chosen rather than altering these values during traffic generation. This reduces impact to DNTG Replay performance during runtime. Each packet in the capture was overwritten with the desired replacement IP and MAC address of the corresponding control system-honeypot pair. Checksum values were also corrected to maintain packet validity.

Figure 11 depicts the original Production Trace (on top) with addresses reflecting the real control systems. It also depicts the modified Production Trace (bottom portion) with the replacement addresses of the honeypot systems. DNTG Prep tool is used to perform this modification of network traces. The tool is run using command line arguments to pass in the IP and MAC addresses of a control system (CS)-honeypot (HP) pair. Multiple pairs of IP and MAC can be passed in as an argument

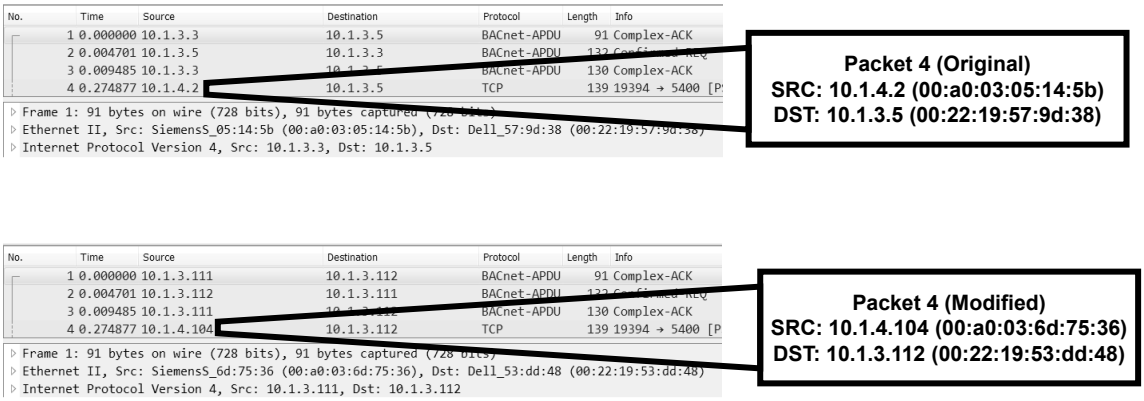

Figure 11. Production Trace modification.

in a single run (shown in Listing III.1).

Listing III.1. DNTG Prep: Command line arguments.

```
DNTG-Prep.py <input.pcap> <output.pcap> <CS1 IP> <HP1 IP> <CS1 MAC> <HP1
    MAC> <CS2 IP> <HP2 IP> <CS2 MAC> <HP2 MAC>
```

The tool processes each packet by performing a search and replace of the Layer 2 and Layer 3 header fields (shown in Listing III.2). The entire code for DNTG Prep can be found in Appendix 1.1. The Bash script used for the experiment can be found in Appendix 1.2. After the modification of the header fields, Wireshark statistical tools are used to validate that the values were changed properly.

Listing III.2. DNTG Prep: Find and replace.

```
if p.haslayer(IP):
        for i in range(0, num_argv):
                if i % 4 == 0:
                        if p[IP].src == sys.argv[i+3]:
                                p[IP].src = sys.argv[i+4]
                        if p.src == sys.argv[i+5]:
                                p.src = sys.argv[i+6]
                        if p[IP].dst == sys.argv[i+3]:
                                p[IP].dst = sys.argv[i+4]
```

```
                    if p.dst == sys.argv[i+5]:
                            p.dst = sys.argv[i+6]
        del p[IP].chksum
        if p.haslayer(TCP):
                del p[TCP].chksum
        if p.haslayer(UDP):
                del p[UDP].chksum
```

3.5.2 DNTG Replay: Distributed Operation.

DNTG Replay is designed to run on each honeypot. The tool requires arguments to be passed when run to identify the: (i) IP and MAC address of the honeypot (used to match relevant packets in the Production Trace); (ii) gateway router MAC address (used for correction of routed packets); and (iii) interface to use for listening and transmit operations. The command line argument used to run the tool is shown in Listing III.3. During operation, each DNTG instance is responsible for handling network traffic generation for its particular honeypot.

Listing III.3. DNTG Replay: Command line arguments.

```
DNTG-Replay.py <Production Trace> <HP IP> <HP MAC> <GW MAC> <interface>
```

3.5.3 DNTG Replay: Traffic Matching.

Synchronization is important for multiple DNTG instances to operate in a distributed environment. However, no extraneous packets are allowed during network traffic generation per criteria from Table 1. Without C2 traffic, synchronization is dependent on the generated packets. If a receiving network traffic generator is not in the correct listening state, it may inadvertently miss incoming packets. It is especially crucial during initialization that each DNTG is in the correct state. To perform the

initial synchronization, a master DNTG is selected based on the first packet within the Production Trace and the honeypot IP (shown in Listing III.4).

Listing III.4. DNTG Replay: Master node selection.

```
master = pkts[0][IP].src
i_am_master = (master == sys.argv[2])
```

The DNTG that possesses the first packet's originating IP source becomes the master and all others enter an immediate listening mode upon initialization. The master DNTG creates a list (e.g., nodes) of all non-master DNTG instances which is later used for initial syncronization (shown in Listing III.5).

Listing III.5. DNTG Replay: Non-master nodes list.

```
if i_am_master:
        if not p[IP].src == master and not p[IP].src in nodes:
                nodes.append(p[IP].src)
```

The master DNTG sends a periodic UDP packet as a heartbeat to the other DNTG instances. When this heartbeat is received, a response is sent back to the master to indicate that the receiving DNTG is active and ready. That specific DNTG is then removed from the list (e.g., nodes) and added to another list (e.g., nodes_ready). The master DNTG sends out a UDP start packet once it receives heartbeat responses from all DNTG instances. The start packet signals each network traffic generator to enter the appropriate state to transmit (or listen) based on the Production Trace. No additional or future synchronization packets are required outside this initial period. The code that handles the transmission of the heartbeat and start packets is shown in Listing III.6. The code that handles the receipt and response to these two packets are shown in Listing III.7.

Listing III.6. DNTG Replay: Initial syncronization request.

```
if i_am_master:
        while not synced:
                for ip in nodes:
                        syncpkt=IP(src=sys.argv[2],dst=ip)/UDP(sport
                            =10000,dport=10000)/Raw(load='HEARTBEAT')
                        send(syncpkt, iface=sys.argv[5]) # send L3
                            packet
                        time.sleep(.1)
                        if len(nodes) == 0:
                                synced = True
                                break
        for ip in nodes_ready:
                syncpkt=IP(src=sys.argv[2],dst=ip)/UDP(sport=10000,dport
                    =10000)/Raw(load='START')
                send(syncpkt, iface=sys.argv[5]) # send L3 packet
                time.sleep(.1)
        time.sleep(1)
        has_started = True
        start_conversations(conversations)
```

Listing III.7. DNTG Replay: Initial syncronization response.

```
if not has_started:
        if pkt.haslayer(UDP) and pkt.haslayer(IP) and sys.argv[2]==pkt[
            IP].dst:
                if pkt[UDP].dport == 10000:
                        if pkt[Raw].load == 'HEARTBEAT':
                                syncpkt=IP(src=pkt[IP].dst,dst=pkt[IP].
                                    src)/UDP(sport=10000,dport=10000)/
                                    Raw(load='TAEBTRAEH')
                                send(syncpkt, iface=sys.argv[5])
                        elif pkt[Raw].load == 'TAEBTRAEH':
                                if pkt[IP].src in nodes:
```

```
                    nodes.remove(pkt[IP].src)
                if not pkt[IP].src in nodes_ready:
                    nodes_ready.append(pkt[IP].src)
            if pkt[Raw].load == 'START':
                    has_started = True
                    start_conversations(
                        conversations)
```

Each DNTG uses an identical Production Trace and operates in an asynchronous nature. By removing control traffic during generation each DNTG is responsible for keeping track of the current location in the Production Trace and re-synchronizing based on received packets. As each DNTG receive and send replay packets, it resynchronizes index location within the Production Trace. To an attacker these C2 operations are transparent as they are performed locally and not on the network. Only Production Trace data is transmitted by each DNTG during network traffic generation. In addition, because no separate C2 data is sent during generation operations, traffic which might impact network performance is avoided.

DNTG achieves synchronization using a sniff handler which listens and tracks incoming packets (shown in Listing III.8). Relevant packets are identified by a condition statement using the passed-in arguments when DNTG Replay was run (e.g., elif pkt.haslayer(IP) and sys.argv[2]==pkt[IP].dst). This check ensures that the received packet's destination IP matches that of the honeypot. DNTG then checks if that packet belongs to any conversations from the Production Trace (e.g., if not c.finished and c.belongs(pkt)). If the packet does belong to a conversation, it generates a call to a function (e.g., c.get_responses(pkt)) that searches for a matching request packet and its corresponding response packets from the Production Trace.

Listing III.8. DNTG Replay: Listen mode.

```
elif pkt.haslayer(IP) and sys.argv[2]==pkt[IP].dst:
```

```
        n = time.time()
        send_time = n
        for c in conversations:
                if not c.finished and c.belongs(pkt):
                        index, resps = c.get_responses(pkt)
                        if not index is None:
                                req = c.get_packet(index)
                                t = req.time
                                delay = 0
                                for resp in resps:
                                        delay += (resp.time - t - t_cal)
                                        send_time = n + delay
                                        send_q.put((send_time,
                                            PicklablePacket(resp)))
                                        t = resp.time
                                # If we made it here, there's no point
                                    checking the other conversations
                                break
```

Consistency is required to replicate the original system as accurately as possible. Autonomous ICS network traffic consists of routine/polling messages that are sent and received on timed intervals. The DNTG should replicate these intervals (inter-packet timing from Production Trace) and not use transmission methods such as packets per second or burst modes. Two methods were considered to handle the timing: (i) calculate the time between the current queued packet and the first packet of the conversation; and (ii) use the inter-packet timing. The first method implements a catch up algorithm to ensure that processing and network delays do not add to the overall length of the Production Trace generation (start to finish). However, significant changes in inter-packet timing were observed. A second method uses inter-packet timing to generate packets without a catch up algorithm. This second method

was chosen for implementation and as a result, the overall length of a Generated Trace is longer than the Production Trace.

Timing calculations are performed using the sniff handler (see Listing III.8). The timestamp of a matched packet from the Production Trace (i.e., t = req.time) and the system time (i.e., n = time.time) is stored. For each response a delay is calculated (i.e., delay += (resp.time - t - t_cal)). This takes the difference between the timestamps of the response and request packets (inter-packet time). A calibration variable (i.e., t_cal) is present in the code which allows for adjustments to be made in this calculation. The send time (i.e., send_time = n + delay) for each response packet is calculated using the original system time and adds the inter-packet time. The response packets are then queued for network traffic generation (i.e., send_q.put((send_time, PicklablePakcet(resp)))).

The second portion of the synchronization code is found in Conversation.py which provides the Conversation class (the code in its entirety can be found in Appendix 2.2). Synchronization of the Production Trace is performed whenever a packet is received and responses are queried (i.e., get_response). To check if a received packet passed-in by the sniff handler matches the Production Trace packets, the: (i) time to live; (ii) checksum; and (iii) padding are set the same (shown in Listing III.9). Modification of these three fields are required to account for network induced changes to the generated packets. A comparison is then made between the received packet and packets from the Production Trace conversation (i.e., if self.packets[i][IP] == packet[IP]). Synchronization is then completed by setting a index variable (i.e., self.last_pkt = i) to the index of the matching packet. By setting this index variable ensures that as each subsequent packet in a conversation is received, the DNTG Replay tool does not have to search from the start and instead continues from the last recorded index. This function is also responsible for finding subsequent response packets from the

Production Trace to be returned to the sniff handler.

Listing III.9. DNTG Replay: Conversation class synchronization.

```
def get_responses(self, packet): # returns index of request, and list of
    response packets
        resp = []
        index = None
        i = (self.last_pkt + 1) % len(self.packets)
        while not i == self.last_pkt:
                packet[IP].ttl = self.packets[i][IP].ttl
                packet[IP].chksum = self.packets[i][IP].chksum
                if packet.haslayer(Padding) and self.packets[i].haslayer
                    (Padding):
                        packet[Padding] = self.packets[i][Padding]
                if self.packets[i][IP] == packet[IP]:
                        index = i
                        self.last_pkt = i
                        i = (i + 1) % len(self.packets)
                        self.finished = self.last_pkt == (len(self.
                            packets) - 1) # check at receipt
                        while self.packets[i][IP].src == packet[IP].dst
                            and not self.finished:
                                resp.append(self.packets[i])
                                self.last_pkt = i
                                i = (i + 1) % len(self.packets)
                        self.finished = self.last_pkt == (len(self.
                            packets) - 1) # check at send
                        return index, resp
                else:
                        i = (i + 1) % len(self.packets)
        # resp is [] if this happens
        return index, resp
```

To meet the packet ordering criteria, DNTG was designed to handle multiple conversations simultaneously. DNTG parses the Production Trace and separates the packets into conversations. All traffic that occur to (and from) an IP pair are identified as a single conversation to limit the amount of data that need to be processed by each DNTG. As such, a conversation is defined as traffic flow between two unique IP source and destination pair. Using multi-threading, each conversation is processed independently without waiting on one another. By allowing a system pair to communicate independently from the entire Production Trace packet order, it allows unique conversations to intermingle. This adds variability to the overall capture while maintaining packet ordering within each conversation. Figure 12 shows packets from a Production Trace, and Figure 13 shows how those packets would be segmented into conversations.

Conversations are created when DNTG Replay is first initialized and packets are read in the from Production Trace (shown in Listing III.10)

Listing III.10. DNTG Replay: Conversation creation.

```
for p in pkts:
        found = False
        for c in conversations:
                if c.belongs(p):
                        found = True
                        c.add_packet(p)
        if not found:
                con = Conversation(p, sys.argv[2], sys.argv[3], sys.argv
                    [4])
                conversations.append(con)
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.1.4.104	10.1.3.112	TCP	60	52810→5033 [ACK] Seq=1 Ac
2	0.019935	10.1.4.104	10.1.3.112	TCP	60	39482→5402 [ACK] Seq=1 Ac
3	1.708573	10.1.3.112	10.1.3.111	BACnet-APDU	67	Confirmed-REQ readPrope
4	1.792864	10.1.3.111	10.1.3.112	BACnet-APDU	91	Complex-ACK readPrope
5	1.797696	10.1.3.112	10.1.3.111	BACnet-APDU	132	Confirmed-REQ confirmed
6	1.802435	10.1.4.104	10.1.3.112	TCP	139	19394→5400 [PSH, ACK] Seq
7	1.807349	10.1.3.111	10.1.3.112	BACnet-APDU	130	Complex-ACK confirmed
8	1.831848	10.1.3.112	10.1.4.104	TCP	144	5400→19394 [PSH, ACK] Seq

Figure 12. Production Trace sample.

Conversation: 10.1.4.104 <> 10.1.3.112

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.1.4.104	10.1.3.112	TCP	60	52810→5033 [ACK] Seq=1 Ac
2	0.019935	10.1.4.104	10.1.3.112	TCP	60	39482→5402 [ACK] Seq=1 Ac
3	1.802435	10.1.4.104	10.1.3.112	TCP	139	19394→5400 [PSH, ACK] Seq
4	1.831848	10.1.3.112	10.1.4.104	TCP	144	5400→19394 [PSH, ACK] Seq

Conversation: 10.1.4.111 <> 10.1.3.112

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.1.3.112	10.1.3.111	BACnet-APDU	67	Confirmed-REQ readPrope
2	0.084291	10.1.3.111	10.1.3.112	BACnet-APDU	91	Complex-ACK readPrope
3	0.089123	10.1.3.112	10.1.3.111	BACnet-APDU	132	Confirmed-REQ confirmed
4	0.098776	10.1.3.111	10.1.3.112	BACnet-APDU	130	Complex-ACK confirmed

Figure 13. Sample conversations.

3.5.4 DNTG Replay: Network Routing.

The network routing criteria focuses on DNTG instances ability to operate in a distributed environment (e.g., network and physical locations). Each DNTG must be able to use real network connections and relay the generated traffic just as real control systems. Achieving this requires proper Layer 2 forwarding and Layer 3 routing of generated traffic.

The Layer 2 forwarding requires that the traffic generated (which shares MAC addresses from corresponding honeypots) must not cause network addressing issues. To ensure that no Layer 2 networking issues occur, network traffic generators that share IP and MAC address with a honeypot are collocated on the same workstation.

Proper Layer 3 routing ensures that DNTG instances can operate in a multi-

subnet environment (Figure 14 shows an example dual-subnet topology). Recall that

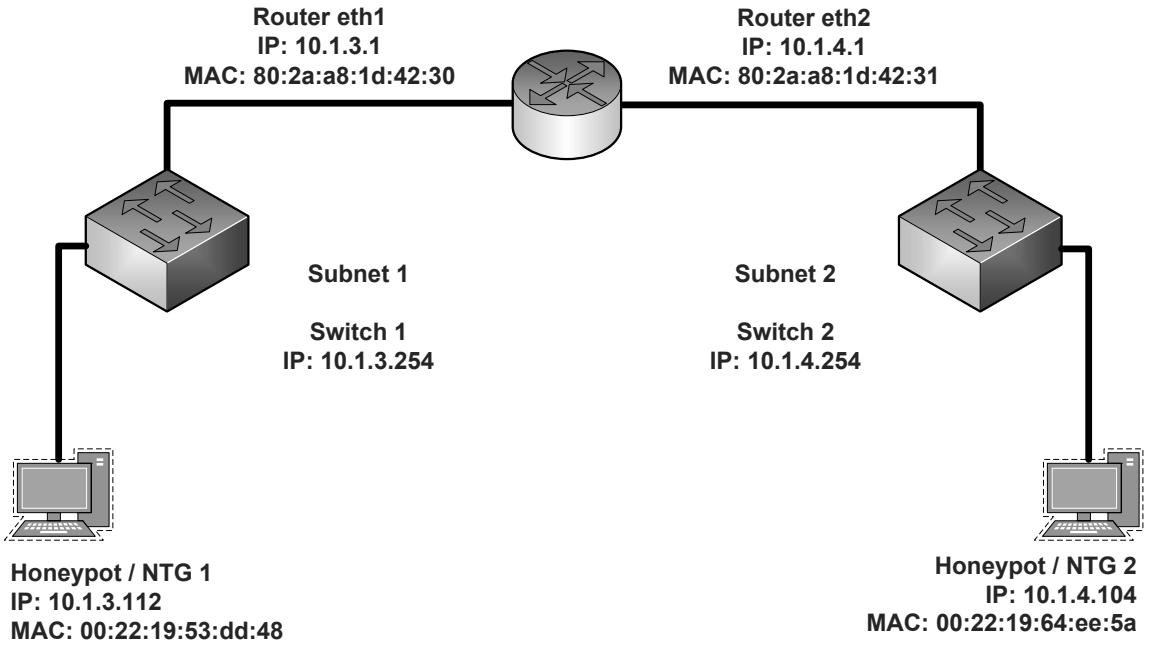

Figure 14. Example dual-subnet environment.

the Production Trace was captured on Subnet 1 and all traffic from outside subnets would have recorded the MAC address of the Router eth1 as the source (shown in Figures 14 and 15). Problems can arise in traffic generation when trying to generate traffic from devices on Subnet 2 with trace data captured on Subnet 1. Using packet seven from Figure 15 as an example, the header contains incorrect address values: (i) source is MAC of Router eth1 (80:2a:a8:1d:42:30); and (ii) destination is MAC of network traffic generator (NTG) 1 (00:22:19:53:dd:48). DNTG identifies these instances and modifies the header with the correct values: (i) new source is MAC of NTG 2 (00:22:19:64:ee:5a); and (ii) new destination is MAC of Router eth2 (80:2a:a8:1d:42:31). These corrections are performed during runtime to avoid customizing the Production Trace for each individual DNTG instance.

The Conversation class found in Conversation.py handles the correction of routed packets (shown in Listing III.11). The DNTG command line arguments of honeypot

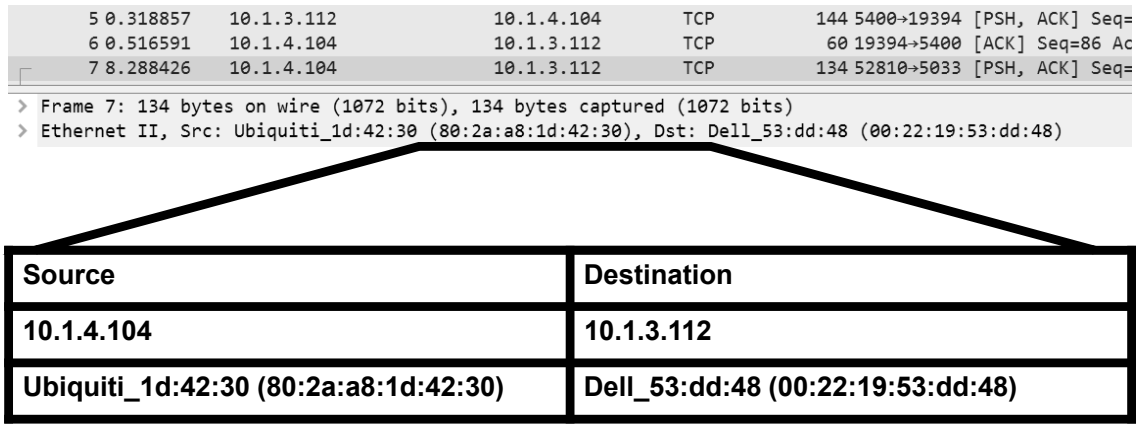

Figure 15. Example MAC addressing issue.

IP, honeypot MAC and router gateway MAC address are used when creating conversations from the Production Trace. These arguments are set to the following variables my_ip, my_mac and gw_mac respectfully. When adding packets to the conversation, the function fixPacket is called. Routed packets and their associated gateway MAC addresses need modification for proper network traffic generation. A check is made to verify if the IP addresses matches between the honeypot and receive packet (i.e., if(packet[IP].src == self.my_ip and packet.src !=self.my_mac)). This also identifies if the MAC address is different. For packets that meet this criteria, the remainder of the code sets the correct MAC addresses (i.e., packet.dst = self.gw_mac and packet.src = self.my_mac) and resets the checksums.

Listing III.11. DNTG Replay: Conversation class - MAC address correction.

```
def __init__(self, packet, my_ip, my_mac, gw_mac):
        self.packets = []
        self.last_pkt = -1
        self.my_ip = my_ip
        self.my_mac = my_mac
        self.gw_mac = gw_mac
        self.src = packet[IP].src
        self.dst = packet[IP].dst
```

```
        self.add_packet(packet)
        self.finished = False

def add_packet(self, packet):
        p = self.fixPacket(packet)
        self.packets.append(p)

def fixPacket(self, packet):
        if packet.haslayer(IP):
                if(packet[IP].src == self.my_ip and packet.src != self.
                    my_mac):
                        packet.dst = self.gw_mac
                        packet.src = self.my_mac
                        del packet[IP].chksum # Reset IP checksum
                        if packet.haslayer(TCP): # put check, otherwise
                            may error
                                del packet[TCP].chksum # Reset TCP
                                    checksum
                        if packet.haslayer(UDP): # put check, otherwise
                            may error
                                del packet[UDP].chksum # Reset UDP
                                    checksum
        return packet
```

3.6 Design Summary

This chapter details the APOGEE test environment that generates the trace used for experimentation. An analysis of this trace is made. A pilot study using Tcpreplay highlights some design considerations needed for a network traffic generator. The honeypot-network traffic generator framework and the Distributed Network Traffic

Generator is introduced. New features added with DNTG over other network traffic generators include:

- A prep tool used to modify the Production Trace with honeypot characteristics.
- A replay tool that runs asynchronously from each instance in a distributed configuration.
- Transmission of synchronization packets only during initialization.
- Network traffic generation without control traffic in a distributed environment.
- Hardware address correction for routed packets.

IV. Research Methodology

4.1 Goals

This chapter focuses on evaluating the developed DNTG. The evaluation goals are derived from the honeypot-network traffic generator criteria set in Table 1 and discussed in Section 2.3. The following questions are addressed during experimentation:

- Does the generated traffic maintain the characteristics of the original trace (traffic matching)?
- Does the generated traffic appear to originate and terminate with the honeypot systems (honeypot integration)?
- Does the generated traffic traverse the designed network (network routing)?
- Can the network traffic generator send network traffic to and from multiple systems using a single trace input file and without C2 traffic during generation (scalability)?
- Achieving the above four goals, does the use of a network traffic generator show control system activity on honeypots during passive monitoring?

4.2 Approach

Multiple experimental trials are conducted to evaluate the DNTG implementation's ability to meet the network traffic generation criteria from Table 1. Each experimental trial consist of generating network traffic from a 10 minute Production Trace. The different trials are conducted using an automated script that alternates the operating environments (single and dual-subnet).

4.3 System Parameters

This section discusses the system parameters of the experiment (see Figure 16). The system under test (SUT) is the overarching honeypot-network traffic generator framework, with specific focus on the Distributed Network Traffic Generator. The component under test (CUT) are the individual honeypot-DNTG instances within the system. A combination of two workstations containing three CUT instances: (i) HMI; (ii) BACnet PXCM; and (iii) P2 PXCM are used in the experiment to emulate the APOGEE platform. Each CUT instance generates the Generated Trace which provides the metrics for evaluating the system.

4.3.1 Computing Parameters.

While the computing parameters differ for each CUT instance, the same hardware is used for all experimental trials. Note that the PXCM CUT workstation has three instances: (i) P2 PXCM on Subnet 1; (ii) P2 PXCM on Subnet 2; and (iii) BACnet on Subnet 1. To maintain consistency the same brand and model NIC is used for the two P2 PXCM instances. The first P2 PXCM instance establishes a connection to Subnet 1. The second P2 PXCM instance establishes a connection to Subnet 2 using a different NIC, IP and MAC address configurations. The experiment network topology is shown in Figure 17.

The workstation configuration and software information for the HMI CUT is listed in Table 2, PXCM CUT in Table 3, sniffer and experiment C2 system in Table 4 and network devices and their connections in Table 5.

4.3.2 Workload Parameters.

The workload parameters for the SUT are the Production Trace and Network Configuration. Using the APOGEE platform, two set of production captures are

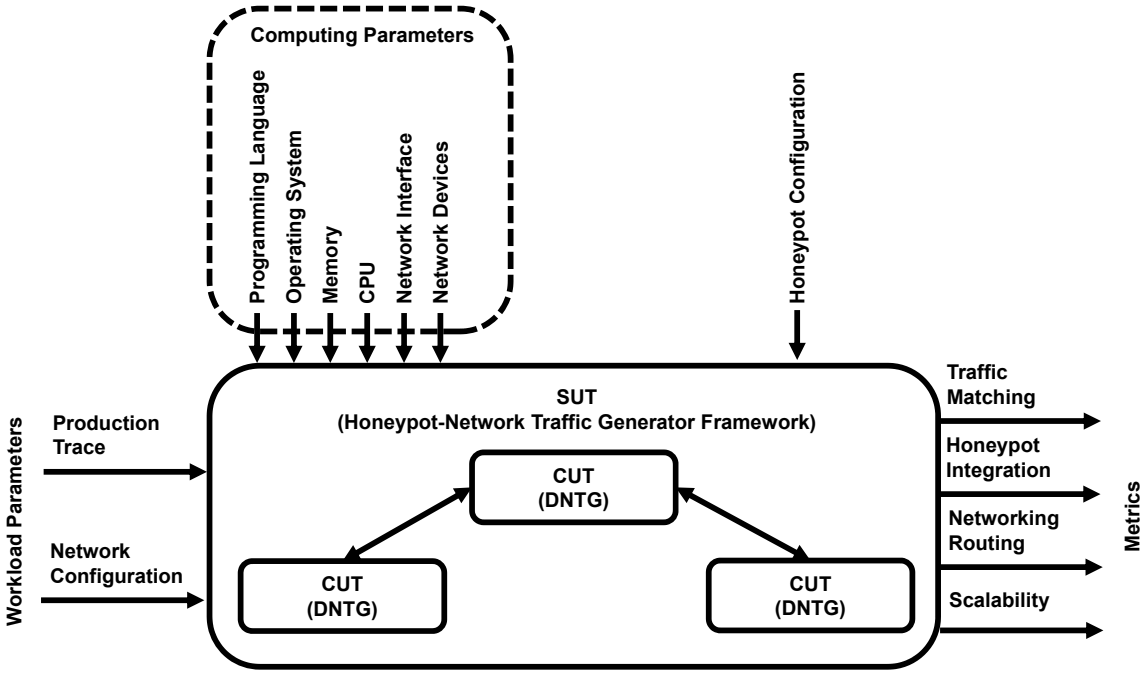

Figure 16. Experiment system parameters.

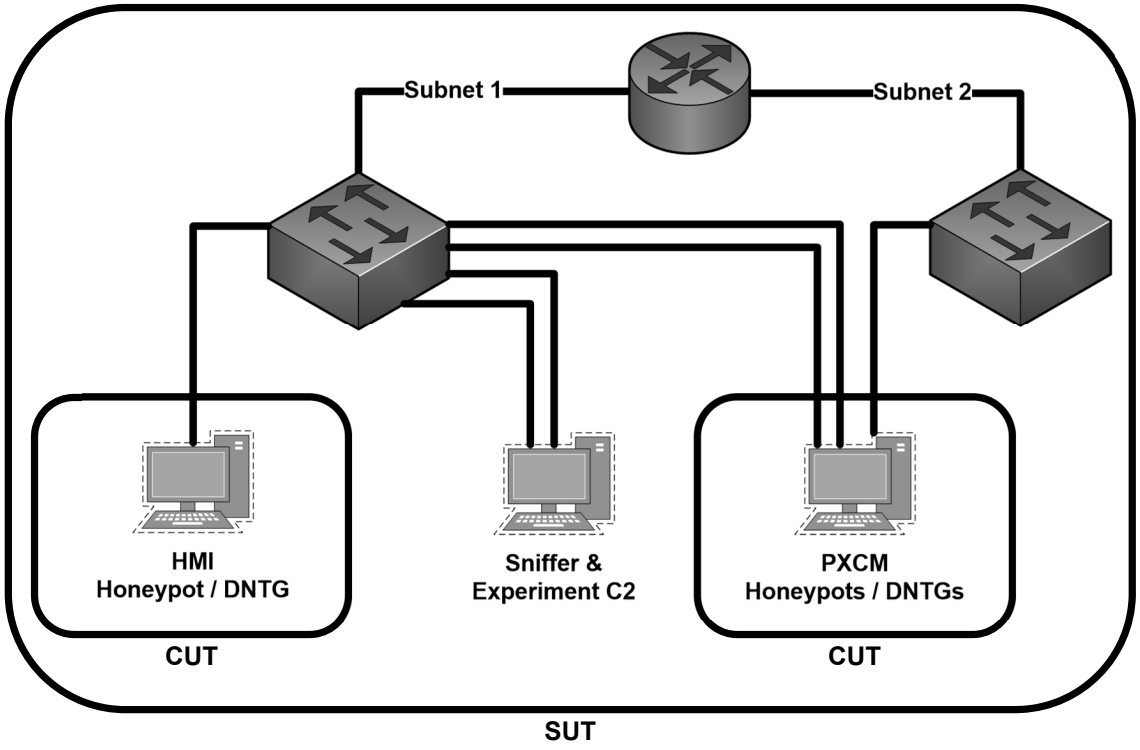

Figure 17. Experiment network topology.

Table 2. HMI CUT workstation.

Dell Latitude E6320	
4 processor cores	Ubuntu 16.04 LTS
4 GB RAM	Honeyd V1.6d
Intel 82579LM NIC	Python 2.7.12
	Scapy 2.20
	DNTG Replay

Table 3. PXCM CUT workstation.

Dell Latitude E6520	
8 processor cores	Ubuntu 16.04 LTS
8 GB RAM	Honeyd V1.6d
Intel 82579LM NIC	Python 2.7.12
plugable USB2-E100 USB Ethernet adapter	Scapy 2.20
plugable USB2-E100 USB Ethernet adapter	DNTG Replay

Table 4. Sniffer and experiment C2 workstation.

Dell Precision M4500	
4 processor cores	Ubuntu 16.04 LTS
8 GB RAM	Python 2.7.12
Intel 82577LM NIC	Scapy 2.20
SMC 2209 USB Ethernet adapter	tcpdump

Table 5. Experiment network.

Hardware and associated connections (from Figure 17)	
Ubiquiti EdgeRouter X router	eth0: Netgear ProSafe 108E
	eth1: Netgear ProSafe GS108E
Netgear ProSafe GS108E	eth0: HMI HP/DNTG
	eth1: PXCM HP/DNTG
	eth2: PXCM HP/DNTG
	eth3: Experiment C2
	eth7 (mirrored): Sniffer
Netgear ProSafe 108E	eth0: PXCM HP/DNTG

performed for the single-subnet and dual-subnet configurations. The two Production Traces contain different IP and MAC addresses for the two P2 PXCM CUT instances. The same IP and MAC addresses are provided for the HMI CUT workstation and BACnet PXCM CUT instance. The packets contained in the two traces

differ as capture is performed at different periods of time. For experimentation, the network configuration is alternated between single-subnet and dual-subnet using the appropriate Production Trace.

4.3.3 Metrics.

This section outlines the metrics used to validate the SUT against the criteria from Table 1.

4.3.3.1 Traffic matching.

Table 6. Traffic matching metrics.

Content Matching	**Packet Bytes:** Generated packets match Production Trace.
Extraneous Packets	**Quantity of Packets:** The number of generated packets match Production Trace.
Packet Ordering	**Packet Order:** Generated conversations match Production Trace conversations.
Timing Consistency	**Δ Inter-Packet Time:** Generated traffic timing patterns match Production Trace.

Table 6 describes the metrics used to evaluate DNTG against the traffic matching criteria. Content matching is evaluated based on a direct comparison of corresponding packets between the Generated Trace and Production Trace. Two packets are determined to match if both packets contain the same bytes. A trial is considered successful if every packet in the Generated Trace matches the corresponding packet in the Production Trace.

The Generated Trace is determined to have no extraneous packets if it contains the same quantity of packets as the Production Trace. A trial is considered successful if the quantity and content of packets in the Generated Trace matches the Production Trace.

Packet ordering is determined to match if the packet order of a Generated Trace conversations matches the packet order of the corresponding Production Trace conversations. A trial is considered successful if every packet in the Generated Trace and the Production Trace is in the correct order.

Timing consistency is measured by comparing the inter-packet timing of packets from the Generated Trace to the inter-packet timing of packets from the Production Trace. The inter-packet time from the Generated Trace ($GIPT_n$) is

$$GIPT_n = GT_n - GT_{n-1} \tag{1}$$

where GT_n is the time of packet n from the Generated Trace.

The inter-packet time from the Production Trace ($PIPT_n$) is

$$PIPT_n = PT_n - PT_{n-1} \tag{2}$$

where PT_n is the time of packet n from the Production Trace.

The difference in the inter-packet time (ΔIPT_n) between corresponding packets within the two traces is calculated using

$$\Delta IPT_n = ABS(GIPT_n - PIPT_n) \tag{3}$$

A Wilcoxon rank-sum test is a non-parametric statistical test used to compare the distribution of $GIPT$ values to the distribution of $PIPT$ values for each Generated

Trace. A significance level of 0.05 is used to determine if the two distributions are statistically similar. The trial is considered a success if the Wilcoxon rank-sum test returns a p-value greater than 0.05.

4.3.3.2 Honeypot integration.

Honeypot pairing is evaluated based on a direct comparison of corresponding packets between the Generated Trace and Production Trace. DNTG was designed to only generate traffic if it starts a conversation or is responding to a received packet. In addition, DNTG is hosted on the same honeypot workstation and uses matching IP and MAC addresses in the trace headers. Honeypot pairing is determined to be a match if packets are sent and received from each honeypot workstation. The experiment is considered successful if every packet in the Generated Trace and corresponding Production Trace passes the content matching and extraneous packets criteria.

To validate the honeypot header matching, an NMAP scan is used to obtain the IP and MAC address of each honeypot. This information is then compared to the Production Trace and Generated Traces to validate that the generated traffic matches the honeypots. A trial is considered successful if every packet header matches the intended honeypot information.

4.3.3.3 Network routing.

Multiple DNTG instances can be run at various locations in a network. A trial is considered successful if every packet in the Generated Trace and corresponding Production Trace passes the traffic matching criteria. The ability to successfully pass the content matching, extraneous packets and packet ordering criteria in a single-subnet environment demonstrates that the experiment successfully met the Layer 2 forwarding criteria. The ability to successfully pass the content matching, extraneous

packets and packet order criteria in a dual-subnet environment demonstrates that the experiment successfully met the Layer 3 routing criteria. Passing both Layer 2 forwarding and Layer 3 routing criteria demonstrates that the experiment successfully met the distributed operation criteria.

4.3.3.4 Scalability.

The scalability of a network traffic generator is based primarily on the design, implementation and pricing of a final implementation. The author of this research did not consider the actual costs (in terms of dollars), considering that most instantiations would require at least some engineering effort to determine proper placement of the traffic generators. While the DNTG is designed to be as flexible as possible, the experiment is limited to the Siemens APOGEE system and different implementations are left for future work.

4.4 Experimental Setup

Experimental trials are conducted using automated scripts from a C2 workstation. Each CUT is configured on the network and provided a copy of: (i) DNTG Replay tool; and (ii) two Production Traces. The usage of the DNTG Replay tool was discussed in Section 3.5.

4.4.1 Production Trace Preparation.

A Bash script (i.e., Prep-PCAP.sh) is used to prepare the production trace for network traffic generation (note that the entire script is found in Appendix 1.2). This script requires modification of the honeypot characteristics (shown in Listing IV.1). The script simplifies the command line argument structure that is used to run DNTG Prep. It executes DNTG Prep for the two Production Traces (i.e., single-subnet and

dual-subnet). The modified values are then copied over to a Bash script (i.e., Run-Experiments.sh) and sets the workload parameters for the experimental trials.

Listing IV.1. Prep-PCAP.sh: Honeypot characteristic modification.

```
# Change values based on HP characteristics (Copy to Run-Experiments.sh)
PXCM2IP3SUB='10.1.3.2' # P2 PXCM single-subnet
PXCM2MAC='00:a0:03:04:d9:d0'
HP20IP3SUB='10.1.3.20'
HP20MAC3SUB='00:a0:03:17:81:94'

PXCM2IP4SUB='10.1.4.2' # P2 PXCM dual-subnet
HP20IP4SUB='10.1.4.20'
HP20MAC4SUB='00:a0:03:3e:59:6f'

PXCM3IP='10.1.3.3' # BACnet PXCM
PXCM3MAC='00:a0:03:05:14:5b'
HP30IP='10.1.3.30'
HP30MAC='00:a0:03:bb:3f:10'

APGSVR5IP='10.1.3.5' # HMI
APGSVR5MAC='00:22:19:57:9d:38'
HP50IP='10.1.3.50'
HP50MAC='00:22:19:ce:95:7b'
```

4.4.2 Command and Control.

The sniffer and C2 workstation provides the centralized control for all experimental trials. This is accomplished through the use of SSH. A SSH certificate is created on each CUT workstation and provides the C2 system the ability to connect without login and password authentication.

A Bash script (i.e., Run-Experiments.sh) is used to automate the multiple exper-

imental trials (the entire script is found in Appendix 3.1). The script is run with command line arguments containing a folder for trial results storage and a comma-separated values file to track trial runs. The output can be piped into a log file if desired. The command line execution for the script is shown in Listing IV.2.

Listing IV.2. Run-Experiments.sh: Command line arguments.

```
Run-Experiments.sh folder tracker.csv >> Log.txt
```

The workload parameters (i.e., single-subnet, dual-subnet and Production Trace) are alternated in between trial runs (shown in Listing IV.3). This is accomplished by using a list of values (i.e., EXP_LIST). An appropriate call to the experiment function (i.e., Call-Exp) is made based on the list value. Each iteration of an experimental trial is logged into a file.

Listing IV.3. Run-Experiments.sh: Workload parameter selection.

```
while [ $COUNTER2 -le ${#EXP_LIST[@]} ]
do
        RUNINDEX=$(($COUNTER2-1))
        ((COUNTER2++))

        TEST=${EXP_LIST[$RUNINDEX]}
        if [ $TEST == 0 ]
        then
                TEST_TYPE='Apogee-1-Subnet'
                Call-Exp $COUNTER $ApogeePCAP1SUB $HP20IP3SUB
                    $HP20MAC3SUB $SUB3GW enx8cae4cfe406d
        elif [ $TEST == 1 ]
        then
                TEST_TYPE='Apogee-2-Subnet'
                Call-Exp $COUNTER $ApogeePCAP2SUB $HP20IP4SUB
                    $HP20MAC4SUB $SUB4GW enx8cae4cfe51f6
```

```
        fi


        # Next
        ((COUNTER++))
done
```

On the C2 workstation a function (i.e., Call-Exp) runs tcpdump to capture the generated network traffic. For experimental purposes, only CUT generated network traffic is of interest and all others are ignored. Filters are used to isolate the honeypot IP addresses, remove network generated packets (i.e., broadcast, multicast and ARP) and ignore pre-generation packets (i.e., heartbeat and start). The initial synchronization is ignored as the metrics comparison is made on network traffic during generation.

Listing IV.4. Run-Experiments.sh: Call-Exp function - Generated Trace output.

```
sudo tcpdump -i eno1 -nnvvXSs 0 -w ~/Desktop/Experiment/$FOLDER/Exp-$1.
    pcap -U ip host $HP20IP3SUB or $HP20IP4SUB or $HP30IP or $HP50IP and
     not dst port 10000 and not broadcast and not multicast and not arp>
     /dev/null &
tdpid=$!
```

For the PXCM CUT instances, the function (i.e., Call-Exp) receives the single or dual-subnet values (i.e., P2 PXCM HP IP, HP MAC, GW MAC address and NIC) after the workload parameter selection. The values for the BACnet and HMI CUT instances are pulled from the honeypot configuration for these two systems. DNTG-Replay is executed on the P2 PXCM, BACnet PXCM and HMI honeypots using SSH connection from the C2 workstation. Each SSH connection's process identification number (PID) is stored to track DNTG-Replay completion. Listing IV.5 shows an example SSH command issued to the PXCM CUT (P2 PXCM instance).

Listing IV.5. Run-Experiments.sh: Call-Exp function - DNTG Replay run on PXCM CUT.

```
COMMAND="sudo python ~/Desktop/Experiment/DNTG-Replay.py ~/Desktop/
    Experiment/$2 $3 $4 $5 $6"
ssh $C2HP2030IP "$COMMAND" > $FOLDER/Exp-$1-HP20-Log.txt &
ntg20=$!
```

The SSH connection is terminated when each CUT reaches the end of the Production Trace and completes the execution of DNTG Replay. This results in the removal of the SSH PID. Once all SSH PIDs are terminated, the experimental trial is concluded and the next iteration is run.

During an experimental trial, if a CUT fails to generate traffic then all other CUTs will also stop generating traffic. This is due to DNTG's intended design and its reliance on received packets to find responses to transmit. Given that the Production Trace is 10 minutes in length, a 12 minutes timeout is set for each trial. Any trial that reaches this threshold is marked as a failed experiment.

The two checks made to conclude an experimental trial is shown in Listing IV.6.

Listing IV.6. Run-Experiments.sh: Call-Exp function - end of experimental trial.

```
while true
do
        # Check for hanged experiments
        CURRENTTIME=$(date +"%s")
        ELAPSEDTIME=$(($CURRENTTIME - $STARTTIME))
        if [ $ELAPSEDTIME -gt $TIMEOUT ] # put a timer on experiment, if
            exceeds timeout, kill
        then
                sudo kill -9 $ntg20 $ntg30 $ntg50
                TEST_STATUS='FAILED'
        fi
```

```
        # Stop TCPDump condition: all NTGs are not running
        if ! ps -p $ntg20 $ntg30 $ntg50 > /dev/null
        then
                sudo pkill -TERM -P $tdpid
                break
        fi
done
```

4.4.3 Analysis.

The outputs (i.e., Generated Traces from the trials) of the SUT are captured on the experiment network and provide the sample points used as a comparison against the reference inputs (i.e., Production Traces of the single-subnet and dual-subnet configurations). The analysis of the research focuses on answering the questions:

- Does the generated traffic maintain the characteristics of the original trace (traffic matching)?
- Does the generated traffic appear to originate and terminate with the honeypot systems (honeypot integration)?
- Does the generated traffic traverse the designed network (network routing)?
- Can the Distributed Network Traffic Generator send network traffic to and from multiple systems using a single trace input file and without C2 traffic during generation (scalability)?
- Achieving the above four goals, does the use of a network traffic generator show control system activity on honeypots during passive monitoring?

4.4.3.1 Computational Analysis.

To answer the traffic matching question, computations on the outputs of the SUT are made. Each Generated Trace is compared against the reference Production Trace using an analysis tool designed to find differences between the two traces. The tool consists of three components: (i) an analysis tool; (ii) a class file (i.e., Analysis.py) that supports the tool; and (iii) Bash script used to run the tool.

The analysis tool (i.e., Analyze.py) performs a comparison of two input traces (i.e., Production Trace and Generated Trace). This tool makes a comparison of the trace lengths to determine experimental failures and timing calculations. The entire code for the tool is found in Appendix 4.1. The code for the class is found in Appendix 4.2.

The analysis tool first verifies that the number of packets, packet data and order within the Production Trace and Generated Trace match (shown in Listing IV.7 and IV.8). This check is done by: (i) breaking down the trace into conversations; (ii) verifying the number of conversations match; and (iii) packets in the conversations match to include packet order and byte data.

Listing IV.7. Analyze.py: Packet match validation.

```
if not len(training_convos) == len(sample_convos):
        test_pass = False


for t_convo in training_convos:
        p = t_convo.get_packet(0)
        found = False
        for s_convo in sample_convos:
                if s_convo.belongs(p):
                        found = True
                        if not s_convo.equals(t_convo):
                                test_pass = False
```

```
                        break
        if not found:
                test_pass = False
```

Listing IV.8. Analysis.py: Packet match validation.

```
def belongs(self, packet):
        if packet.haslayer(IP):
                if self.src == packet[IP].src:
                        return self.dst == packet[IP].dst
                elif self.src == packet[IP].dst:
                        return self.dst == packet[IP].src
        return False


def p_equals(self, p1, p2):
        p1.ttl = p2.ttl
        p1.chksum = p2.chksum
        if p1.haslayer(Padding) and p2.haslayer(Padding):
                p1[Padding] = p2[Padding]
        return p2[IP] == p1[IP]


def equals(self, convo):
        pcks1 = convo.packets
        pcks2 = self.packets
        is_equal = True
        if not len(pcks1) == len(pcks2):
                is_equal = False
        else:
                for i in range(0, len(pcks1)):
                        if not self.p_equals(pcks1[i], pcks2[i]):
                                is_equal = False
        return is_equal
```

Once content of the traces (i.e., Production Trace and Generated Trace) are validated as being identical, calculations on the timing are made (shown in Listings IV.9 and IV.10). The analysis tool takes each inter-packet timestamps from the Production Trace (i.e., control), Generated Trace (i.e., sample) and calculates the difference between the two (i.e., delta). The values are returned as a list and written to an output file.

Listing IV.9. Analyze.py: Timing calculations.

```
for t_convo in training_convos:
        p = t_convo.get_packet(0)
        for s_convo in sample_convos:
                if s_convo.belongs(p):
                        ctrl_int, sample_int, int_delta = s_convo.
                            delta_time(t_convo)
        csvfile = a_name+"-DPI.csv"
        with open(csvfile, "a") as output:
                for i in range(0, len(ctrl_int)):
                output.write(str(ctrl_int[i]) + "," + str(sample_int[i
                    ]) + "," + str(int_delta[i]) + "\n")
                time.sleep(.02)
```

Listing IV.10. Analysis.py: Timing calculations.

```
def delta_time(self, convo):
        pcks1 = convo.packets
        pcks2 = self.packets
        ctrl_int = []
        sample_int = []
        int_delta = []
        if len(pcks1) == len(pcks2):
                for i in range(0, len(pcks1)-1):
                        ctrl = pcks1[i+1].time - pcks1[i].time
```

```
                        sample = pcks2[i+1].time - pcks2[i].time
                        delta = sample - ctrl
                        ctrl_int.append(ctrl)
                        sample_int.append(sample)
                        int_delta.append(delta)
        return ctrl_int, sample_int, int_delta
```

To help automate the analysis of the multiple experimental trace outputs, a Bash script (i.e., Run-Analyze.sh) is used. The entire script can be found in Appendix 4.3 and sample shown in Listing IV.11.

Listing IV.11. Run-Analyze.sh: Running analysis on experimental outputs.

```
for ((i=1;i<178;i+=1))
do
        python Analyze.py 1Sub1/Production.pcap 1Sub/Exp\ \($i\).pcap 1
            Sub >> 1Sub.log
        sleep 10
done
```

4.4.3.2 Observational Analysis.

Observational analysis is used to measure the four criteria: (i) honeypot integration; (ii) network routing; (iii) scalability; and (iv) network traffic generation during passive monitoring. Each CUT is a combination pair of honeypot and network traffic generator. Honeypot modified Production Traces are provided as inputs and tested within a defined experiment network designed to replicate a real control system network (i.e., APOGEE platform). By design, the successful operation of the DNTG Replay tool on each CUT within the SUT validates whether if the outputs pass or fail the chapter questions.

4.5 Chapter Summary

In summary, this chapter describes the methodology used to measure the honeypot-network traffic generator framework. The system parameters establish the inputs and outputs used to test DNTG CUT. An experiment is developed for this research using automated and remote scripts. Experimental trials are developed with the objective of answering the chapter questions. The C2 workstation reiterates multiple experimental trials on each CUT to create the Generated Traces. The Generated Traces are then used against the reference Production Traces for metrics analysis.

V. Results and Analysis

5.1 Overview

Experimental trials were conducted over a 65 hour time period. 177 iterations were conducted for each environment for a total of 354 experimental trials. Outputs provided over 375,000 sample packets for analysis and were evaluated against the specified criteria from Table 1.

5.2 Traffic Matching

All 354 trials achieved a 100% match between the Production Trace and Generated Trace for the following metrics: (i) packet bytes; (ii) quantity of packets; and (iii) packet ordering. The success rate for the criteria determined by these metrics is shown in Table 7.

Table 7. Traffic matching criteria success rate.

	Trials	Single-Subnet	Dual-Subnet
Content Matching	**354**	**100%**	**100%**
Extraneous Packets	**354**	**100%**	**100%**
Packet Ordering	**354**	**100%**	**100%**
Timing Consistency	**354**	**0%**	**0%**

Timing consistency was evaluated by using a Wilcoxon rank-sum test. This test compares the distribution of $GIPT$ from each of the 354 Generated Traces to the distribution of $PIPT$ from the Production Trace. The Wilcoxon rank-sum test returned a p-value less than 0.05, for all 354 Generated Trace distributions. Therefore, the generated traffic timing did not match the production traffic.

Table 8 shows the ΔIPT summary for all 354 trials. There was a mean difference of 2.07 ms (single-subnet) and 2.26 ms (dual-subnet) between the Production Trace and generated traffic. Further examination shows that the generated traffic timing

Table 8. ΔIPT (ms) summary.

	Mean	Min	Max	SD
Single-Subnet	**2.07**	**0.00**	**95.78**	**1.59**
Dual-Subnet	**2.26**	**0.00**	**98.93**	**1.68**

was consistent between trials (Wilcoxon rank-sum test). While the generated traffic timing did not match the Production Trace, the generated traffic for each trial maintained consistency between trials.

Figure 18 shows a boxplot of the two subnet configurations with similar timing. Figure 19 shows an overlay of both the Production Trace (line) and a sample Generated Trace (points). It is scaled to show a more detailed view for intervals between packets 800 and 820 in Figure 20. Visually, it is difficult to distinguish between the real system and the honeypot system.

5.3 Honeypot Integration

All 354 trials resulted in a 100% match between the Production Trace and Generated Trace for the following criteria: (i) content matching; (ii) extraneous packets; and (iii) packet ordering. Meeting these three criteria demonstrates that traffic was generated to (and from) the honeypot-network traffic generator pair. In addition, because the generated traffic matched the Production Trace (validated against honeypots using NMAP), the honeypot header matching criteria is also met. The results of the criteria are shown in Table 9. Figure 21, shows an NMAP active scan and a Wireshark passive monitoring of the decoy network used for the experiments.

Table 9. Honeypot integration criteria pass rate.

	Trials	Single-Subnet	Dual-Subnet
Honeypot Pairing	**354**	**100%**	**100%**
Honeypot Header Matching	**354**	**100%**	**100%**

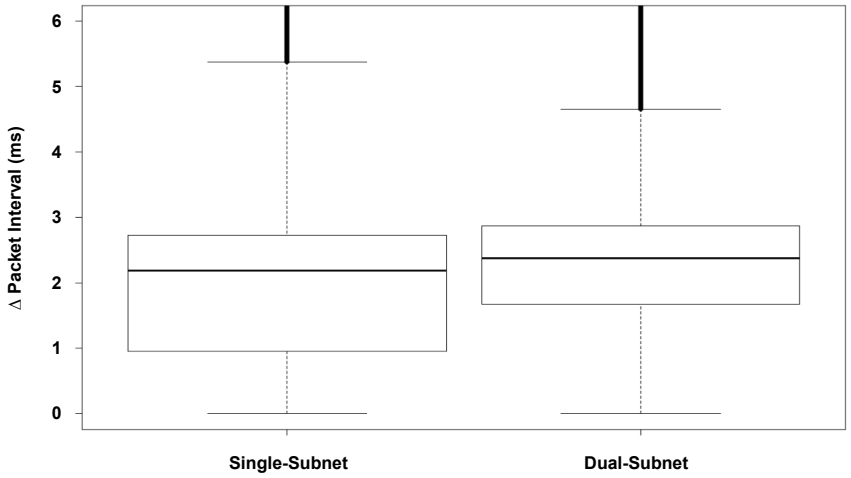

Figure 18. Difference in timing between Production Trace and Generated Trace intervals.

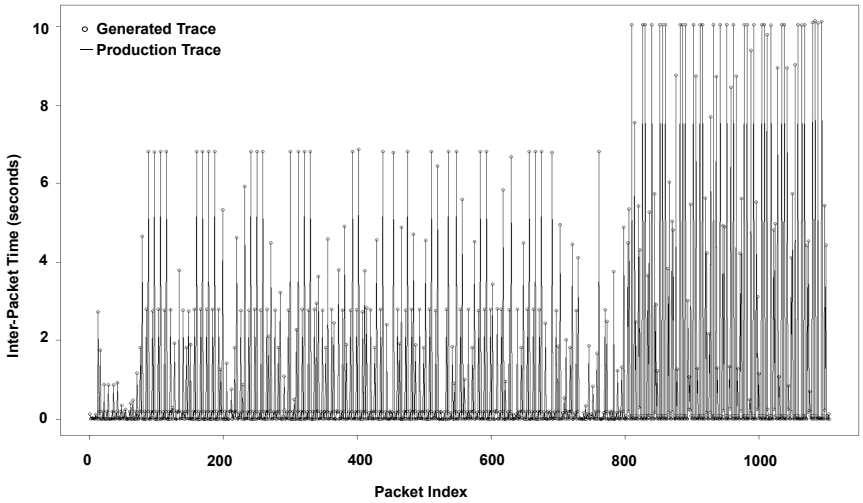

Figure 19. Traffic timing pattern.

5.4 Network Routing

All 354 trials achieved a 100% match between the Production Trace and Generated trace for the following criteria: (i) content matching; (ii) extraneous packets; and (iii) packet ordering. Meeting these three criteria demonstrates that traffic was generated to and from multiple instances of DNTG located in two different network configurations. It demonstrates that Layer 2 forwarding and Layer 3 routing was successfully accomplished and validates that DNTG met the distributed operation

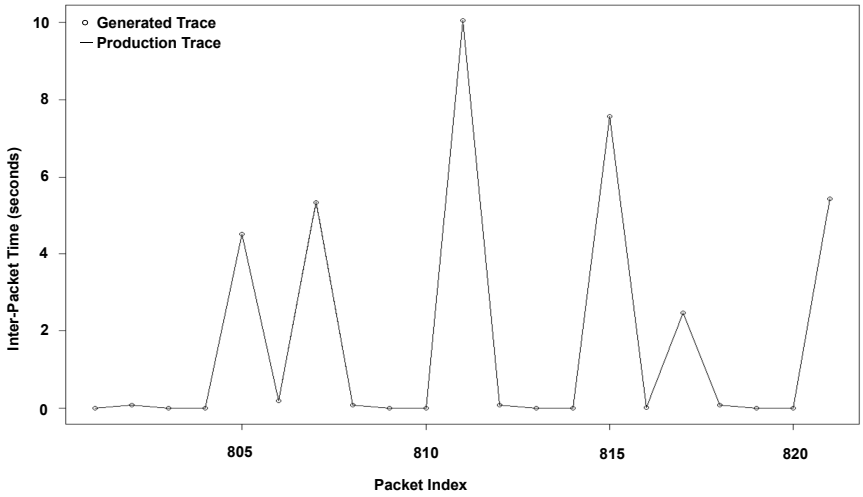

Figure 20. Detailed traffic timing pattern.

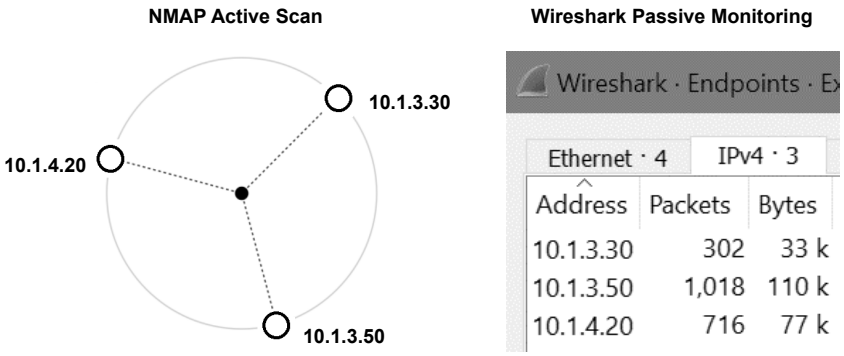

Figure 21. Active and passive network mapping.

criteria. The results for the network routing criteria are shown in Table 10.

Table 10. Network routing criteria pass rate.

	Trials	Single-Subnet	Dual-Subnet
Distributed Operation	354	100%	100%
Layer 2 Forwarding	354	100%	100%
Layer 3 Routing	354	100%	100%

5.5 Scalability

The DNTG was designed to be cost effective and flexible, however, future work is required to evaluate this criteria for commercial implementation. The research did

show that under the given experiment environment, all trials were able to complete the network traffic generation objective successfully.

5.6 Chapter Summary

The results of the experiment showed that the use of DNTG in the honeypot-network traffic generator framework met some of goals set as part of this research:

- Does the generated traffic maintain the characteristics of the original trace (traffic matching)? Packet order and byte data of the trace were maintained in the generated traffic. The results, however, did show that there were differences in the timing of the generated traffic from the reference input trace.

- Does the generated traffic appear to originate and terminate with the honeypot systems (honeypot integration)? The generated traffic headers contained the characteristics of the honeypot systems and network traffic generation originated and terminated with each honeypot system.

- Does the generated traffic traverse the designed network (network routing)? The SUT was able successfully generate and route all trace packets between multiple CUTs. All generated traffic was observed on the network devices concerned.

- Can the network traffic generator send network traffic to and from multiple systems using a single trace input file and without C2 traffic during generation (scalability)? The SUT was able to successfully generate traffic on multiple CUTs within the experimental network topologies.

- Achieving the above four goals, does the use of a network traffic generator show control system activity on honeypots during passive monitoring? The SUT demonstrated that through the use of a honeypot-network traffic generation

framework, both system and network traffic were detected during active scans and passive network monitoring activities.

VI. Conclusions and Recommendations

6.1 Research Conclusions

For this research, a framework consisting of a honeypot platform and a network traffic generator platform was introduced. To accomplish network traffic generation, a custom made Distributed Network Traffic Generator was developed. A DNTG Prep tool was developed to modify Production Trace with honeypot characteristics to integrate with the honeypot platform. A DNTG Replay tool was created, distributed, and operated asynchronously on each honeypot system. Network traffic generation was accomplished through this tool by monitoring for trace data and transmitting subsequent packets from trace.

Experimentation was conducted to validate the framework and DNTG using the following metrics: (i) traffic matching; (ii) honeypot integration; (iii) network routing; and (iv) scalability. While packet data and order matched between the reference inputs (i.e., Production Trace) and the experiment outputs (i.e., Generated Trace), there were timing discrepancies (mean difference of 2.07 ms for single-subnet and 2.26 ms for dual-subnet experiments). Despite the discrepancy in timing, the results showed that DNTG was able to successfully replicate control system traffic on honeypots and route through two different network topologies. Visually, it was difficult to distinguish between the real and generated traffic patterns without detailed analysis. Honeypot integration and network routing were successfully demonstrated through 100 percent successful trial runs. While DNTG was designed for scalability, in the context of this research, it was only tested using inputs provided from the APOGEE platform. Testing for complex network environments and other control systems were left for future work.

6.2 Research Contributions

This research showed that a honeypot-network traffic generation framework can be used as defense-in-depth security measure to protect ICS. While high-interaction honeypots can replicate whole systems, they often come with high cost and management requirements. Different solutions using low-interaction and hybrid designs offer authenticity when interactions are made. However for OT systems, autonomous and continuous network traffic is required. Using the developed DNTG tools provide control system traffic on honeypot systems. This was demonstrated during passive network monitoring and in the experimental trials. The presence of this traffic helps ensure that honeypot systems are not easily identified during network discovery, enumeration and attack.

New features added with DNTG over other network traffic generators include: (i) modification of trace data; (ii) asynchronously operation in a distributed configuration; (iii) transmission of synchronization packets only during initialization; (iv) synchronization during network traffic generation using only trace data; and (v) hardware address correction for network routing.

6.3 Recommendations for Future Work

6.3.1 Timing.

The DNTG tool, showed promise by successfully replicating control system traffic in a distributed environment. While the results indicated that the Production Trace and Generated Trace did not have the same timing, multiple runs showed consistency in DNTG outputs. This indicated that optimization of the software code with the objective to reduce the delta in the timing is possible. While a calibration variable (i.e., t_cal) was added to the code, it was not used during experimentation. This

calibration variable allows adjustment to the delay before a packet is transmitted during network traffic generation. Further study of network topologies, configuration and network device performance is needed to effectively implement this calibration value.

Improvements on the design of DNTG Replay could produce better timing results. Currently, DNTG Replay performs operations (e.g., trace search for matching packets and packet queuing for generation) only when a packet is received. Modifications to the code could include pre-processing trace packets for the listening and transmit operations. This would reduce processing delays incurred during runtime and lessen the timing delta of subsequent packets transmitted.

6.3.2 Limitations of Using a Trace-based Approach.

The duration of trace data impacts the effectiveness of network traffic generation and its ability to provide authenticity for honeypot systems. For the timespan of a trace, network traffic may look authentic. However, past this time period, the same packets are be generated if a different trace is not used for reiterated runs of DNTG. Repeated traffic (e.g., TCP) is automatically highlighted as retransmissions by tools like Wireshark. Future work may include implementing the ability to alternate Production Traces or modify packet data and values to maintain uniqueness during multiple iterations.

In addition, the use of trace data does not account for real-time system changes. State changes made during operation may contradict traffic data generated by DNTG. This would be a challenging task to implement, as it would involve knowledge of protocol specifics for different control systems. One possible method is to use live production data to update the network traffic generator. Future work is required to evaluate possible solutions to this challenge.

6.4 Concluding Thoughts

Honeypots can be used as a security measure to protect industrial control systems. The effectiveness of a honeypot is dependent on the ability to entice an attacker to select it as a target. Using network traffic generators as a deception technique is similar to the concept employed by Ghost Army using realistic sound recordings, operational movements and signal transmissions. The addition of network traffic on honeypots helps build a realistic decoy control system environment. This research demonstrated that honeypot identification during target selection can be made difficult by introducing network traffic generation.

Appendix A. DNTG Prep

1.1 DNTG-Prep.py

DNTG tool used to prepare a trace file by modifying Layer 2 and Layer 3 header values. This tool is run on the C2 workstation.

```
#!/usr/bin/env python
# PCAP Re-Write
# Author: Htein Lin
# Function: Re-writes the IP and MAC addresses based on matching
    parameters passed to the program as arguments.  Handles UDP, TCP,
    and ARP packets.
# Usage example: sudo python PCAP-PREP.py input.pcap output.pcap
    10.1.3.5 10.1.3.205 00:22:19:57:9d:38 00:22:19:1e:a3:25 10.1.3.3
    10.1.3.201 00:a0:03:05:14:5b 00:a0:03:f8:62:9f 10.1.4.2 10.1.4.201
    00:a0:03:04:d9:d0 00:a0:03:2a:7e:5f
import logging, sys, time
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *

num_argv = (len(sys.argv) - 3) # calculate the number of arguments
if (len(sys.argv) < 7) or (num_argv % 4 != 0): # check to make sure a
    minimum set of IP and MAC addresses are entered and that it is a 4
    argument pair
        print 'Usage: sudo python pcap-rewrite.py <input.pcap> <output.
            pcap> <old ip> <new ip> <old mac> <new mac> ... <old ip> <
            new ip> <old mac> <new mac>'
        sys.exit(1)
start_time = time.time() # Mark start time
print '[+] Process started at: ' + time.ctime()
infile = sys.argv[1] # Read the packets from input PCAP file
pkts = rdpcap(infile)
```

```
for i in range(0, num_argv): # Prompt the arguments that will be
    processed
        if i % 4 == 0: # set index at start of each 4 pair set of
            addresses
                print '[!] The IP: ' + sys.argv[i+3] + ' & MAC: ' + sys.
                    argv[i+5] + ' will be overwritten with IP: ' + sys.
                    argv[i+4] + ' & MAC: ' + sys.argv[i+6]
# Rewrite the packets with the new addresses
# Delete the IP, UDP, and TCP checksums so that they are recalculated
# First IP pair arguments start at index 3 + 4, MAC pair starts at index
    5 + 6 and so forth
for p in pkts:
        if p.haslayer(IP): # Handle packets with IP
                for i in range(0, num_argv):
                        if i % 4 == 0: # check at start of each 4 pair
                            set of addresses
                                if p[IP].src == sys.argv[i+3]: # Replace
                                    SRC IP
                                        p[IP].src = sys.argv[i+4]
                                if p.src == sys.argv[i+5]: # Replace SRC
                                    MAC
                                        p.src = sys.argv[i+6]
                                if p[IP].dst == sys.argv[i+3]: # Replace
                                    DST IP
                                        p[IP].dst = sys.argv[i+4]
                                if p.dst == sys.argv[i+5]: # Replace DST
                                    MAC
                                        p.dst = sys.argv[i+6]
                del p[IP].chksum # Reset IP checksum
                if p.haslayer(TCP): # put check, otherwise may error
                        del p[TCP].chksum # Reset TCP checksum
                if p.haslayer(UDP): # put check, otherwise may error
```

```
                        del p[UDP].chksum # Reset UDP checksum
outfile = sys.argv[2] # Write the packets out to a new PCAP file
wrpcap(outfile, pkts)
stop_time = time.time() # Mark stop time
print '[+] Processing completed at: ' + time.ctime()
print '[+] Total time to finish: ' + str(stop_time - start_time) + '
    seconds'
```

1.2 Prep-PCAP.sh

Bash script is used to call DNTG-Prep.py and run on the C2 workstation. This script requires modification with honeypot characteristics.

```
#!/bin/bash

# Change values based on HP characteristics (Copy to Run-Experiments.sh)
PXCM2IP3SUB='10.1.3.2' # P2 PXCM single-subnet
PXCM2MAC='00:a0:03:04:d9:d0'
HP20IP3SUB='10.1.3.20'
HP20MAC3SUB='00:a0:03:17:81:94'

PXCM2IP4SUB='10.1.4.2' # P2 PXCM dual-subnet
HP20IP4SUB='10.1.4.20'
HP20MAC4SUB='00:a0:03:3e:59:6f'

PXCM3IP='10.1.3.3' # BACnet PXCM
PXCM3MAC='00:a0:03:05:14:5b'
HP30IP='10.1.3.30'
HP30MAC='00:a0:03:bb:3f:10'

APGSVR5IP='10.1.3.5' # HMI
APGSVR5MAC='00:22:19:57:9d:38'
```

```
HP50IP='10.1.3.50'
HP50MAC='00:22:19:ce:95:7b'


# Apogee
python DNTG-Prep.py CS-Apogee-3Sub-Raw.pcap NTG-Apogee-3Sub.pcap
    $APGSVR5IP $HP50IP $APGSVR5MAC $HP50MAC $PXCM3IP $HP30IP $PXCM3MAC
    $HP30MAC $PXCM2IP3SUB $HP20IP3SUB $PXCM2MAC $HP20MAC3SUB
python DNTG-Prep.py CS-Apogee-4Sub-Raw.pcap NTG-Apogee-4Sub.pcap
    $APGSVR5IP $HP50IP $APGSVR5MAC $HP50MAC $PXCM3IP $HP30IP $PXCM3MAC
    $HP30MAC $PXCM2IP4SUB $HP20IP4SUB $PXCM2MAC $HP20MAC4SUB
```

Appendix B. DNTG Replay

2.1 DNTG-Replay.py

DNTG tool used to generate network traffic using a trace file. This tool is run on CUT workstations.

```
#!/usr/bin/env python
import logging, sys
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *
from Conversation import Conversation
from threading import Thread
from multiprocessing import Pool, Process, Queue
from sys import exit

t_cal = 0.0 # Offset to subtract from all times before sending
conversations = []
first_pkt_time = None
send_q = Queue()
nodes = []
nodes_ready = []
synced = False
has_started = False

class PicklablePacket:
        """A container for scapy packets that can be pickled (in
            contrast
        to scapy packets themselves)."""
        def __init__(self, pkt):
                self.contents = bytes(pkt)
                self.time = pkt.time
```

```
        def __call__(self):
                """Get the original scapy packet."""
                pkt = Ether(self.contents)
                pkt.time = self.time
                return pkt


def main():
        global has_started, conversations, first_pkt_time, nodes,
           nodes_ready, synced #, send_q
        # check to make sure the arguments are correct
        # sys.argv[0] = NTG.py
        # sys.argv[1] = <input.pcap>
        # sys.argv[2] = <HP IP>
        # sys.argv[3] = <HP MAC>
        # sys.argv[4] = <GW MAC>
        # sys.argv[5] = <iface>
        if (len(sys.argv) < 6):
                print 'Usage: python NTG.py <input.pcap> <HP IP> <HP MAC
                   > <GW MAC> <iface>'
                sys.exit(1)
        pkts = rdpcap(sys.argv[1]) # read PCAP file
        first_pkt_time = pkts[0].time # Set initial capture time
        master = pkts[0][IP].src
        i_am_master = (master == sys.argv[2])
        # Assign all packets to individual conversations
        for p in pkts:
                found = False
                for c in conversations:
                        if c.belongs(p):
                                found = True
                                c.add_packet(p)
                if not found:
```

```
                con = Conversation(p, sys.argv[2], sys.argv[3],
                    sys.argv[4])
                conversations.append(con)
        if i_am_master:
                if not p[IP].src == master and not p[IP].src in
                    nodes:
                        nodes.append(p[IP].src)
if i_am_master:
        while not synced:
                for ip in nodes:
                        print 'Sending HEARTBEAT to', ip
                        syncpkt=IP(src=sys.argv[2],dst=ip)/UDP(
                            sport=10000,dport=10000)/Raw(load='
                            HEARTBEAT')
                        send(syncpkt, iface=sys.argv[5]) # send
                            L3 packet
                        time.sleep(.1)
                        if len(nodes) == 0:
                                synced = True
                                break
        for ip in nodes_ready:
                print 'Sending START to', ip
                syncpkt=IP(src=sys.argv[2],dst=ip)/UDP(sport
                    =10000,dport=10000)/Raw(load='START')
                send(syncpkt, iface=sys.argv[5]) # send L3
                    packet
                time.sleep(.1)
        time.sleep(1)
        has_started = True
        print 'Starting conversations'
        start_conversations(conversations)
```

```
def start_conversations(conversations):
        #TODO: move this
        # Delay 10 seconds before first send packet
        delay = 10.0
        st = time.time() + delay

        # Check each conversation to see if it needs started (src == sys
            .argv[2])
        for c in conversations:
                pckts = c.get_responses_index(0)
                for p in pckts:
                        # If I am the client
                        if p[IP].src == sys.argv[2]:
                                pt = (p.time - first_pkt_time)
                                send_q.put((st + pt, PicklablePacket(p))
                                    )

def sniff_hndl(pkt):
        global has_started, nodes, nodes_ready
        # This check is for slaves
        if not has_started:
                if pkt.haslayer(UDP) and pkt.haslayer(IP) and sys.argv
                    [2]==pkt[IP].dst:
                        if pkt[UDP].dport == 10000:
                                if pkt[Raw].load == 'HEARTBEAT':
                                        print 'Received HEARTBEAT from',
                                            pkt[IP].src
                                        syncpkt=IP(src=pkt[IP].dst,dst=
                                            pkt[IP].src)/UDP(sport
                                            =10000,dport=10000)/Raw(load
                                            ='TAEBTRAEH')
                                        print 'Sent TAEBTRAEH to', pkt[
```

```
                                        IP].dst
                                    send(syncpkt, iface=sys.argv[5])
                            elif pkt[Raw].load == 'TAEBTRAEH':
                                    print 'Received TAEBTRAEH from',
                                        pkt[IP].src
                                    if pkt[IP].src in nodes:
                                            nodes.remove(pkt[IP].src
                                                )
                                    if not pkt[IP].src in
                                        nodes_ready:
                                            nodes_ready.append(pkt[
                                                IP].src)
                            if pkt[Raw].load == 'START':
                                    print 'Received START from', pkt
                                        [IP].src
                                    has_started = True
                                    start_conversations(
                                        conversations)
    elif pkt.haslayer(IP) and sys.argv[2]==pkt[IP].dst:
            n = time.time()
            send_time = n
            for c in conversations:
                    if not c.finished and c.belongs(pkt):
                            print "RECEIVED packet from: %s with id:
                                %s" % (pkt[IP].src, pkt[IP].id)
                            index, resps = c.get_responses(pkt)
                            if not index is None:
                                    req = c.get_packet(index)
                                    t = req.time
                                    delay = 0
                                    for resp in resps:
                                            delay += (resp.time - t
```

```
                                                - t_cal)
                                        send_time = n + delay
                                        send_q.put((send_time,
                                            PicklablePacket(resp
                                            )))
                                        t = resp.time
                                # If we made it here, there's no point
                                    checking the other conversations
                                break
                finished = True
                for c in conversations:
                        if not c.finished and c.HPbelongs(sys.argv[2]):
                                finished = False
                if finished:
                        if time.time() <= send_time:
                                time.sleep(send_time-time.time()+1)
                        exit(0)

def sendPacket(responsePkt, socket=None): # Transmit packet
        if socket is None:
                sendp(responsePkt, iface=sys.argv[5]) # send packet
        else:
                socket.send(responsePkt)
        print "SENT packet to: %s with id: %s" %(responsePkt[IP].dst,
            responsePkt[IP].id)

def packet_sender_t(send_q):
        s_packets = []
        socket = conf.L2socket(iface=sys.argv[5])
        while True:
                if not send_q.empty():
                        p = send_q.get()
```

```
                        s_packets.append(p)
                curtime = time.time()
                for t, pickle in s_packets:
                        p = pickle()
                        if curtime >= t:
                                sendPacket(p, socket)
                                s_packets.remove((t, pickle))
                                break

def startsniff():
        sniff(prn=sniff_hndl, store=0, iface=sys.argv[5])

if __name__ == '__main__':
        send_p = Pool(1, packet_sender_t, (send_q, ))
        m_thread = Thread( target=main, args=() )
        sniff_thread = Thread( target=startsniff )
        m_thread.start()
        sniff_thread.start()
        m_thread.join()
        sniff_thread.join()
```

2.2 Conversation.py

Conversation.py contains the class Conversation and is called by DNTG-Replay.py. This is part of the DNTG Replay tool and is run on CUT workstations.

```
from scapy.all import *
import time

class Conversation:
        def __init__(self, packet, my_ip, my_mac, gw_mac):
                self.packets = []
```

```
        self.last_pkt = -1
        self.my_ip = my_ip
        self.my_mac = my_mac
        self.gw_mac = gw_mac
        self.src = packet[IP].src
        self.dst = packet[IP].dst
        self.add_packet(packet)
        self.finished = False

def add_packet(self, packet):
        p = self.fixPacket(packet)
        self.packets.append(p)

def fixPacket(self, packet):
        if packet.haslayer(IP):
                if(packet[IP].src == self.my_ip and packet.src
                   != self.my_mac):
                        packet.dst = self.gw_mac
                        packet.src = self.my_mac
                        del packet[IP].chksum # Reset IP
                            checksum
                        if packet.haslayer(TCP): # put check,
                            otherwise may error
                                del packet[TCP].chksum # Reset
                                    TCP checksum
                        if packet.haslayer(UDP): # put check,
                            otherwise may error
                                del packet[UDP].chksum # Reset
                                    UDP checksum
        return packet

def get_packet(self, index):
```

```
        if index < len(self.packets):
                return self.packets[index]
        return None

def get_first_time(self):
        return self.packets[0].time

def HPbelongs(self, IP):
        if self.src == IP or self.dst == IP:
                return True
        else:
                return False

def belongs(self, packet):
        if packet.haslayer(IP):
                if self.src == packet[IP].src:
                        return self.dst == packet[IP].dst
                elif self.src == packet[IP].dst:
                        return self.dst == packet[IP].src
        return False

def print_packets(self):
        index = 0
        for p in self.packets:
                print index, ': ', p[Raw].load.encode('hex')
                index += 1

def get_responses(self, packet): # returns index of request, and
    list of response packets
        resp = []
        index = None
        i = (self.last_pkt + 1) % len(self.packets)
```

```
while not i == self.last_pkt:
        packet[IP].ttl = self.packets[i][IP].ttl
        packet[IP].chksum = self.packets[i][IP].chksum
        if packet.haslayer(Padding) and self.packets[i].
            haslayer(Padding):
                packet[Padding] = self.packets[i][
                    Padding]
        if self.packets[i][IP] == packet[IP]:
                index = i
                self.last_pkt = i
                print 'Pointer moved to packet: ', self.
                    last_pkt+1 # debug
                i = (i + 1) % len(self.packets)
                self.finished = self.last_pkt == (len(
                    self.packets) - 1) # check at
                    receipt
                while self.packets[i][IP].src == packet[
                    IP].dst and not self.finished:
                        print "QUEUED packet to: %s with
                             id: %s" %(self.packets[i][
                            IP].dst,self.packets[i][IP].
                            id) # debug
                        resp.append(self.packets[i])
                        self.last_pkt = i
                        i = (i + 1) % len(self.packets)
                print 'Pointer moved to packet:', self.
                    last_pkt+1 # debug
                self.finished = self.last_pkt == (len(
                    self.packets) - 1) # check at send
                print 'End of conversation? ', self.
                    finished, '\n' # debug
                return index, resp
```

```
                else:
                        i = (i + 1) % len(self.packets)
        # resp is [] if this happens
        return index, resp


def get_responses_index(self, index): # returns index of request
    , and list of response packets
        resp = []
        resp.append(self.packets[index]) # put in first packet
            in response
        i = (index + 1) % len(self.packets) # check for next
            packets
        print 'Pointer moved to packet: ', index+1 # debug
        self.finished = self.last_pkt == (len(self.packets) - 1)
             # check at receipt
        while self.packets[i][IP].src == self.packets[index][IP
            ].src and not self.finished: # while the next packet
             is same source
                print "QUEUED packet to: %s with id: %s" %(self.
                    packets[i][IP].dst,self.packets[i][IP].id)
                resp.append(self.packets[i])
                self.last_pkt = i
                i = (i + 1) % len(self.packets)
        print 'Pointer moved to packet:', self.last_pkt+1 #
            debug
        self.finished = self.last_pkt == (len(self.packets) - 1)
             # check at send
        print 'End of conversation? ', self.finished, '\n' #
            debug
        return resp
```

Appendix C. Experiment

3.1 Run-Experiments.sh

Bash script is used to automate the experimental trials. This script requires modification with honeypot characteristics and is run on the C2 workstation.

```
#!/bin/bash
# Usage: ./Run-Experiments.sh folder($1) tracker.csv($2) >> Log.txt
# $1 is the folder for the trial outputs
# $2 is csv file for the trial tracker

TIMEOUT=720 # Timeout timer in seconds, if experiment exceeds this timer
    then something went wrong

# Change values based on HP characteristics (Copy to Run-Experiments.sh)
PXCM2IP3SUB='10.1.3.2' # P2 PXCM single-subnet
PXCM2MAC='00:a0:03:04:d9:d0'
HP20IP3SUB='10.1.3.20'
HP20MAC3SUB='00:a0:03:17:81:94'

PXCM2IP4SUB='10.1.4.2' # P2 PXCM dual-subnet
HP20IP4SUB='10.1.4.20'
HP20MAC4SUB='00:a0:03:3e:59:6f'

PXCM3IP='10.1.3.3' # BACnet PXCM
PXCM3MAC='00:a0:03:05:14:5b'
HP30IP='10.1.3.30'
HP30MAC='00:a0:03:bb:3f:10'

APGSVR5IP='10.1.3.5' # HMI
APGSVR5MAC='00:22:19:57:9d:38'
HP50IP='10.1.3.50'
```

```
HP50MAC='00:22:19:ce:95:7b'

# IP addresses of HP hosts for C2 workstation to SSH into
C2HP2030IP='10.1.3.119'
C2HP50IP='10.1.3.120'

# Change values based on input PCAP files
# Apogee
ApogeePCAP1SUB='NTG-Apogee-3Sub.pcap'
ApogeePCAP2SUB='NTG-Apogee-4Sub.pcap'

# DO NOT CHANGE VALUES BELOW
SUB3GW='80:2a:a8:1d:42:30' # GW MAC of 10.1.3.0/24
SUB4GW='80:2a:a8:1d:42:31' # GW MAC of 10.1.4.0/24

FOLDER=$1
TEST_LOG="$1" '/' "$2"

function Call-Exp {
  # Initialize experiment
  STARTTIME=$(date +"%s")
  TEST_STATUS='Success'

  # Start capture
  echo "sudo tcpdump -i eno1 -nnvvXSs 0 -w ~/Desktop/Experiment/$FOLDER/
      Exp-$1.pcap -U ip host $HP20IP3SUB or $HP20IP4SUB or $HP30IP or
      $HP50IP and not dst port 10000 and not broadcast and not multicast
       and not arp> /dev/null"
  sudo tcpdump -i eno1 -nnvvXSs 0 -w ~/Desktop/Experiment/$FOLDER/Exp-$1
      .pcap -U ip host $HP20IP3SUB or $HP20IP4SUB or $HP30IP or $HP50IP
      and not dst port 10000 and not broadcast and not multicast and not
       arp> /dev/null &
```

```
tdpid=$!
echo TCPDump PID is $tdpid

# Start NTG (10.1.3.20/10.1.4.20)
# $1 is experiment number
# $2 is input pcap file
# $3 is NTG IP (10.1.3.20 or 10.1.4.20)
# $4 is 10.1.3.20 MAC (honeyd) or 10.1.4.20 MAC (honeyd)
# $5 is 10.1.3.0/24 GW MAC (80:2a:a8:1d:42:30) or 10.1.4.0/24 GW MAC
    (80:2a:a8:1d:42:31)
# $6 is 10.1.3.20 interface (enx8cae4cfe406d) or 10.1.4.20 interface (
    enx8cae4cfe51f6)
COMMAND="sudo python ~/Desktop/Experiment/NTG-Replay.py ~/Desktop/
    Experiment/$2 $3 $4 $5 $6"
echo "ssh $C2HP2030IP $COMMAND > $FOLDER/Exp-$1-HP20-Log.txt"
ssh $C2HP2030IP "$COMMAND" > $FOLDER/Exp-$1-HP20-Log.txt &
ntg20=$!
echo HP20 PID is $ntg20

# Start NTG (10.1.3.30)
COMMAND="sudo python ~/Desktop/Experiment/NTG-Replay.py ~/Desktop/
    Experiment/$2 $HP30IP $HP30MAC $SUB3GW eno1"
echo "ssh $C2HP2030IP $COMMAND > $FOLDER/Exp-$1-HP30-Log.txt"
ssh $C2HP2030IP "$COMMAND" > $FOLDER/Exp-$1-HP30-Log.txt &
ntg30=$!
echo HP30 PID is $ntg30

# Start NTG (10.1.3.50)
COMMAND="sudo python ~/Desktop/Experiment/NTG-Replay.py ~/Desktop/
    Experiment/$2 $HP50IP $HP50MAC $SUB3GW eno1"
echo "ssh $C2HP50IP $COMMAND > $FOLDER/Exp-$1-HP50-Log.txt"
ssh $C2HP50IP "$COMMAND" > $FOLDER/Exp-$1-HP50-Log.txt &
```

```
  ntg50=$!
  echo HP50 PID is $ntg50

  # While TCPDump is running
  while true
  do
    # Check for hanged experiments
    CURRENTTIME=$(date +"%s")
    ELAPSEDTIME=$(($CURRENTTIME - $STARTTIME))
    if [ $ELAPSEDTIME -gt $TIMEOUT ] # put a timer on experiment, if
        exceeds timeout, kill
    then
      echo "TEST FAILED --- SOMETIME WENT WRONG --- KILLING PROCESSES"
      sudo kill -9 $ntg20 $ntg30 $ntg50
      TEST_STATUS='FAILED'
    fi

    # Stop TCPDump condition: all NTGs are not running
    if ! ps -p $ntg20 $ntg30 $ntg50 > /dev/null
    then
      echo "KILLING TCPDump"
      echo ""
      sudo pkill -TERM -P $tdpid
      break
    fi
  done
  sleep 30 # Wait 30 seconds in between experiments to let network
      settle
}

# Initialize program
OLDIFS=$IFS
```

```
IFS=, # Set internal field separator to comma

[ ! -f $TEST_LOG ] && { echo "$TEST_LOG file not found"; exit 99; } #
    Experiments log: error cond

COUNTER=1
while read TEST_NUM TEST_NAME TEST_TYPE TEST_STATUS TEST_TIME # Find
    last entry in log
do
  ((COUNTER++))
done < $TEST_LOG

# Run experiments until limit is reached
COUNTER2=$COUNTER
echo "COUNTER IS AT" $COUNTER2
EXP_LIST=(0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
    1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
    1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
    1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
    1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
    1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 10 1
     0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
     0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
     0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 10 1 0 1
    0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
    0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
    0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1)

while [ $COUNTER2 -le ${#EXP_LIST[@]} ]
do
  echo "RUNNING EXPERIMENT $COUNTER2"
  RUNINDEX=$(($COUNTER2-1))
```

```
((COUNTER2++))

# Run experiments
# $1 is experiment number
# $2 is input pcap file
# $3 is NTG IP (10.1.3.20 or 10.1.4.20)
# $4 is 10.1.3.20 MAC (honeyd) or 10.1.4.20 MAC (honeyd)
# $5 is 10.1.3.0/24 GW MAC (80:2a:a8:1d:42:30) or 10.1.4.0/24 GW MAC
    (80:2a:a8:1d:42:31)
# $6 is 10.1.3.20 interface (enx8cae4cfe406d) or 10.1.4.20 interface (
    enx8cae4cfe51f6)

TEST=${EXP_LIST[$RUNINDEX]}
if [ $TEST == 0 ]
then
  TEST_TYPE='Apogee-1-Subnet'
  echo "--- Apogee-1-Subnet Experiment: Test Type 0"
  Call-Exp $COUNTER $ApogeePCAP1SUB $HP20IP3SUB $HP20MAC3SUB $SUB3GW
      enx8cae4cfe406d
elif [ $TEST == 1 ]
then
  TEST_TYPE='Apogee-2-Subnet'
  echo "--- Apogee-2-Subnet Experiment: Test Type 1"
  Call-Exp $COUNTER $ApogeePCAP2SUB $HP20IP4SUB $HP20MAC4SUB $SUB4GW
      enx8cae4cfe51f6
else
  echo "Exp-Out-Of-Bounds"
  echo "--- Experiment out of bounds"
fi

# Write entry into experiments log
echo "$COUNTER, Experiment-$COUNTER.pcap, $TEST_STATUS, $TEST_TYPE,
```

```
        $TEST_TIME" >> $TEST_LOG


    # Next
    ((COUNTER++))
done


IFS=$OLDIFS # Reset internal field separator
```

Appendix D. Analysis

4.1 Analyze.py

Analysis tool used to analyze two input traces. This script is run on the C2 workstation.

```
#!/usr/bin/env python

import logging, sys, csv
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *
from Analysis import Analysis
from sys import exit

training_convos = []
sample_convos = []

def main():

        global training_convos, sample_convos
        ctrl_int = []
        sample_int = []
        int_delta = []
        output_list = []

        # check to make sure the arguments are correct
        if (len(sys.argv) < 3):
                print 'Usage: python <training.pcap> <sample.pcap> <
                    analysis name>'
                sys.exit(1)

        a_name = sys.argv[3]
```

```
ctrl_pkts = rdpcap(sys.argv[1]) # read control PCAP file

# Assign all packets to individual conversations
for p in ctrl_pkts:
        found = False
        for c in training_convos:
                if c.belongs(p):
                        found = True
                        c.add_packet(p)

        if not found:
                con = Analysis(p)
                training_convos.append(con)

sample_pkts = rdpcap(sys.argv[2]) # read sample PCAP file

# Assign all packets to individual conversations
for p in sample_pkts:
        found = False
        for c in sample_convos:
                if c.belongs(p):
                        found = True
                        c.add_packet(p)

        if not found:
                con = Analysis(p)
                sample_convos.append(con)

test_pass = True

if not len(training_convos) == len(sample_convos):
        test_pass = False
```

```
        print 'Unequal number of conversations\n'

print "\n--=== Running Packet Match Analysis ===--"
for t_convo in training_convos:
        p = t_convo.get_packet(0)
        print "Processing conversation: %s > %s" %(p[IP].src, p[
            IP].dst)
        found = False
        for s_convo in sample_convos:
                if s_convo.belongs(p):
                        found = True
                        if not s_convo.equals(t_convo):
                                print 'Conversations are not
                                    equal'
                                test_pass = False
                        break
        if not found:
                print 'Control conversation is not in Sample
                    conversations'
                test_pass = False
if test_pass:
        print "All Control and Sample conversations match."

print "\n--=== Running Packet Interval Analysis ===--"
for t_convo in training_convos:
        p = t_convo.get_packet(0)
        for s_convo in sample_convos:
                if s_convo.belongs(p):
                        ctrl_int, sample_int, int_delta =
                            s_convo.delta_time(t_convo)
        csvfile = a_name+"-DPI.csv"
        with open(csvfile, "a") as output:
```

```
                for i in range(0, len(ctrl_int)):
                        #print "Index: %d, Control: %.6f, Sample
                            : %.6f, Delta: %.6f" % (i, ctrl_int[
                            i], sample_int[i], int_delta[i])
                        output.write(str(ctrl_int[i]) + "," +
                            str(sample_int[i]) + "," + str(
                            int_delta[i]) + "\n")
                        time.sleep(.02)
print "Results outputted to "+csvfile

print "\n---=== Running Delta Response Interval Analysis ===---"
for t_convo in training_convos:
        p = t_convo.get_packet(0)
        for s_convo in sample_convos:
                if s_convo.belongs(p):
                        ctrl_int, sample_int, int_delta =
                            s_convo.delta_response(t_convo)
        csvfile = a_name+"-DResp.csv"
        with open(csvfile, "a") as output:
                for i in range(0, len(ctrl_int)):
                        #print "Index: %d, Control: %.6f, Sample
                            : %.6f, Delta: %.6f" % (i, ctrl_int[
                            i], sample_int[i], int_delta[i])
                        output.write(str(ctrl_int[i]) + "," +
                            str(sample_int[i]) + "," + str(
                            int_delta[i]) + "\n")
                        time.sleep(.02)
print "Results outputted to "+csvfile

print "\n---=== Running Delta Request Interval Analysis ===---"
for t_convo in training_convos:
        p = t_convo.get_packet(0)
```

```
        for s_convo in sample_convos:
                if s_convo.belongs(p):
                        ctrl_int, sample_int, int_delta =
                            s_convo.delta_request(t_convo)
        csvfile = a_name+"-DReq.csv"
        with open(csvfile, "a") as output:
                for i in range(0, len(ctrl_int)):
                        #print "Index: %d, Control: %.6f, Sample
                            : %.6f, Delta: %.6f" % (i, ctrl_int[
                            i], sample_int[i], int_delta[i])
                        output.write(str(ctrl_int[i]) + "," +
                            str(sample_int[i]) + "," + str(
                            int_delta[i]) + "\n")
                        time.sleep(.02)
print "Results outputted to "+csvfile

csvfile = a_name+"-CLength.csv"
with open(csvfile, "a") as output:
        print "\n---=== Running Conversation Length Analysis
            ===---"
        for t_convo in training_convos:
                p = t_convo.get_packet(0)
                for s_convo in sample_convos:
                        if s_convo.belongs(p):
                                ctrl, sample, delta = s_convo.
                                    convo_length(t_convo)
                                print "Length of coversation: %s
                                    <> %s" %(p[IP].src, p[IP].
                                    dst)
                                print "Control: %.6f, Sample:
                                    %.6f, Delta: %.6f\n" % (ctrl
                                    , sample, delta)
```

```
                                        output.write(str(ctrl) + "," +
                                            str(sample) + "," + str(
                                            delta) + "\n")
                                        time.sleep(.02)


        csvfile = a_name+"-PLength.csv"
        with open(csvfile, "a") as output:
                print "Overall PCAP Length"
                ctrl_end = ctrl_pkts[len(ctrl_pkts)-1].time
                ctrl_start = ctrl_pkts[0].time
                ctrl = ctrl_end-ctrl_start
                sample_end = sample_pkts[len(sample_pkts)-1].time
                sample_start = sample_pkts[0].time
                sample = sample_end-sample_start
                delta = sample-ctrl
                print "Control: %s, Sample: %s, Delta: %s" % (ctrl,
                    sample, delta)
                #print ctrl_pkts.show()
                output.write(str(ctrl) + "," +  str(sample) + "," + str(
                    delta) + "\n")
                time.sleep(.02)


        print "\n---=== Analysis Summary ===---"
        if test_pass:
                print 'ALL TESTS PASSED!'
        else:
                print 'FAILED: NOT ALL TESTS COMPLETED SUCCESSFULLY!'

if __name__ == '__main__':
        main()
```

4.2 Analysis.py

Analysis.py contains the class Analysis and is called by Analyze.py. This is part of the Analysis tool and is run on the C2 workstation.

```
from scapy.all import *
import time

class Analysis:
        def __init__(self, packet):
                self.packets = []
                self.last_pkt = 0
                self.src = packet[IP].src
                self.dst = packet[IP].dst
                self.add_packet(packet)
                self.finished = False

        def add_packet(self, packet):
                self.packets.append(packet)

        def get_packet(self, index):
                if index < len(self.packets):
                        return self.packets[index]
                return None

        def belongs(self, packet):
                if packet.haslayer(IP):
                        if self.src == packet[IP].src:
                                return self.dst == packet[IP].dst
                        elif self.src == packet[IP].dst:
                                return self.dst == packet[IP].src
                return False
```

```
def p_equals(self, p1, p2):
        p1.ttl = p2.ttl
        p1.chksum = p2.chksum
        if p1.haslayer(Padding) and p2.haslayer(Padding):
                p1[Padding] = p2[Padding]
        return p2[IP] == p1[IP]

def equals(self, convo):
        pcks1 = convo.packets
        pcks2 = self.packets
        is_equal = True
        if not len(pcks1) == len(pcks2):
                print 'The number of packets does not match.'
                is_equal = False
        else:
                for i in range(0, len(pcks1)):
                        if not self.p_equals(pcks1[i], pcks2[i])
                            :
                                is_equal = False
                                print 'Packet', i+1, 'does not
                                    match.'
        return is_equal

def delta_time(self, convo):
        pcks1 = convo.packets
        pcks2 = self.packets
        ctrl_int = []
        sample_int = []
        int_delta = []
        if not len(pcks1) == len(pcks2):
                print 'The number of packets between Control and
                     Sample does not match'
```

```
            else:
                    for i in range(0, len(pcks1)-1):
                                ctrl = pcks1[i+1].time - pcks1[i
                                    ].time
                                sample = pcks2[i+1].time - pcks2
                                    [i].time
                                delta = sample - ctrl
                                ctrl_int.append(ctrl)
                                sample_int.append(sample)
                                int_delta.append(delta)
            return ctrl_int, sample_int, int_delta


    def delta_response(self, convo):
            t_pcks = convo.packets
            s_pcks = self.packets
            ctrl_int = []
            sample_int = []
            int_delta = []
            debug = 0
            if not len(s_pcks) == len(t_pcks):
                    print 'The number of packets between Control and
                        Sample does not match'
            else:
                    for i in range(0, len(t_pcks)):
                            if self.is_req(t_pcks[i]):
                                    for j in range(i+1, len(t_pcks))
                                        :
                                            if self.is_resp(t_pcks[i
                                                ], t_pcks[j]):
                                                ctrl = t_pcks[j
                                                    ].time -
                                                    t_pcks[i].
```

```
                                                    time
                                                sample = s_pcks[
                                                    j].time -
                                                    s_pcks[i].
                                                    time
                                                delta = sample -
                                                     ctrl
                                                ctrl_int.append(
                                                    ctrl)
                                                sample_int.
                                                    append(
                                                    sample)
                                                int_delta.append
                                                    (delta)
                                                debug += 1
                                                break
        return ctrl_int, sample_int, int_delta

def delta_request(self, convo):
        t_pcks = convo.packets
        s_pcks = self.packets
        ctrl_int = []
        sample_int = []
        int_delta = []
        debug = 0
        if not len(s_pcks) == len(t_pcks):
                print 'The number of packets between Control and
                    Sample does not match'
        else:
                for i in range(0, len(t_pcks)):
                        if self.is_req(t_pcks[i]):
                                for j in range(i+1, len(t_pcks))
```

```
                                        :
                                            if self.is_next_req(
                                                t_pcks[i], t_pcks[j
                                                ]):
                                                    ctrl = t_pcks[j
                                                        ].time -
                                                        t_pcks[i].
                                                        time
                                                    sample = s_pcks[
                                                        j].time -
                                                        s_pcks[i].
                                                        time
                                                    delta = sample -
                                                         ctrl
                                                    ctrl_int.append(
                                                        ctrl)
                                                    sample_int.
                                                        append(
                                                        sample)
                                                    int_delta.append
                                                        (delta)
                                                    debug += 1
                                                    break
            return ctrl_int, sample_int, int_delta


    def convo_length(self, convo):
            pcks1 = convo.packets
            pcks2 = self.packets
            ctrl = 0
            sample = 0
            delta = 0
            if not len(pcks1) == len(pcks2):
```

```
                print 'The number of packets between Control and
                    Sample does not match'
            else:
                ctrl = pcks1[(len(pcks1)-1)].time - pcks1[0].
                    time
                sample = pcks2[(len(pcks2)-1)].time - pcks2[0].
                    time
                delta = sample - ctrl
            return ctrl, sample, delta
```

4.3 Run-Analyze.sh

Bash script is used to automate the analysis of the Generated Traces. This script is run on the C2 workstation.

```
#!/bin/bash
for ((i=1;i<178;i+=1))
do
  echo "python Analyze.py 1Sub/Production.pcap 1Sub/Exp\ \($i\).pcap 1
      Sub >> 1Sub.log"
  python Analyze.py 1Sub1/Production.pcap 1Sub/Exp\ \($i\).pcap 1Sub >>
      1Sub.log
  sleep 10
done

for ((i=1;i<178;i+=1))
do
  echo "python Analyze.py 2Sub/Production.pcap 2Sub/Exp\ \($i\).pcap 2
      Sub >> 2Sub.log"
  python Analyze.py 2Sub/Production.pcap 2Sub/Exp\ \($i\).pcap 2Sub >> 2
      Sub.log
  sleep 10
```

done

Bibliography

1. Armed Forces History Museum, World War IIs US Ghost Army, Largo, Florida (www.armedforcesmuseum.com/world-war-iis-us-ghost-army), 2014.

2. A. Botta, W. Donato, A. Dainotti, S. Avallone and A. Pescapé, D-ITG 2.8.1 Manual, COMputer for Interaction and CommunicationS Group, Department of Electrical Engeering and Information Technologies, University of Napoli "Federico II", Naples, Italy (www.grid.unina.it/software/ITG/manual), 2013.

3. A. Botta, A. Dainotti and A. Pescap, A tool for the generation of realistic network workload for emerging networking scenarios, *Computer Networks*, vol. 56(15), pp. 3531-3547, 2012.

4. E. Byres and J. Lowe, The myths and facts behind cyber security risks for industrial control systems, *Proceedings of the VDE Kongress*, vol. 116, pp. 213-218, 2004.

5. L. Even, IDFAQ: What is a Honeypot? SANS Institute, Bethesda, Maryland, 2000.

6. K. Girtz, B. Mullins, M. Rice and J. Lopez, Practical application layer emulation in industrial control system honeypots, in *Critical Infrastructure Protection X*, M. Rice and S. Shenoi (Eds.), Springer, Cham, Switzerland, pp. 83-98, 2016.

7. J. Harrison, Honeypots: The sweet spot in network security, *Computerworld* (www.computerworld.com/article/2573345/security0/honeypots--the-sweet-spot-in-network-security.html), 2003.

8. Idaho National Laboratory, Control Systems Cyber Security: Defense in Depth Strategies, External Report INL/EXT-06-11478, Idaho Falls, Idaho (www.energy.gov/sites/prod/files/oeprod/DocumentsandMedia/Defense_in_Depth_Strategies.pdf), 2006.

9. Ixia, IxLoad Overview–Converged Multiplay Service Validation, Data Sheet (Revision U), document no. 915–1710–01–2161, Calabasas, California.

10. Ixia, IxLoad Application Replay, Data Sheet (Revision D), document no. 915–1744–01, Calabasas, California, 2013.

11. I. Mokube and M. Adams, Honeypots: Concepts, approaches, and challenges, *Proceedings of the ACM Forty-Fifth Annual Southeast Regional Conference*, pp. 321–326, 2007.

12. NetLoad Inc., Test Traffic Solution, Danville, California (www.netloadinc.com/manuals/NetLoadInc_Products.pdf), 2014.

13. NetLoad Inc., Stateful Traffic Mix Tester Solutions, Danville, California (www.netloadinc.com/manuals/NetLoad_Inc_Brief.pdf).

14. NetLoad Inc., User Guide for NetLoad Product Series (Revision 8.9), Danville, California (www.netloadinc.com/manuals/NetLoadInc._Startup_Guide.pdf), 2016.

15. Ostinato, Ostinato User Guide (www.gitbook.com/book/pstavirs/ostinato-user-guide/details), 2016.

16. C. Pettey and R. van der Meulen, Gartner Says the World of IT and Operational Technology Are Converging, Gartner, Stamford, Connecticut (www.gartner.com/newsroom/id/1590814), 2011.

17. N. Provos, A virtual honeypot framework, *Proceedings of the Thirteenth USENIX Security Symposium*, article no. 1, 2004.

18. W. Shaw, *Cybersecurity for SCADA Systems*, PennWell Corporation, Tulsa, Oklahoma, 2006.

19. Siemens, APOGEE Building Level Network on TCP/IP, Technical Specification Sheet (Revision 1), document no. 149–967, Buffalo Grove, Illinois, 2003.

20. Siemens, Siemens Apogee Scalable Brochure, document no. 611–056, Buffalo Grove, Illinois, 2012.

21. Siemens, Siemens 2011 Apogee Brochure (Revision 8), document no. 153–301 P10, Buffalo Grove, Illinois, 2011.

22. S. Smith, Catching flies: A guide to the various flavors of honeypots, *SANS Institute InfoSec Reading Room*, SANS Institute, Bethesda, Maryland, 2015

23. SolarWinds Worldwide, LLC, Network Traffic Generator with Engineer's Toolset, Austin, Texas (www.solarwinds.com/engineers-toolset/wan-killer), 2016.

24. K. Stouffer, S. Lightman, V. Pillitteri, M. Abrams and A. Hahn, Guide to Industrial Control Systems (ICS) Security, NIST Special Publication 800-82, (Revision 2), National Institute of Standards and Technology, Gaithersburg, Maryland, 2015.

25. Tcpreplay, Wiki (http://tcpreplay.synfin.net/wiki), 2014.

26. P. Warner, Automatic Configuration of Programmable Logic Controller Emulators, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2015.

27. K. Wilhoit, The SCADA That Didn't Cry Wolf, Who's Really Attacking Your ICS Equipment? (Part 2), Trend Micro, Cupertino, California (www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-the-scada-that-didnt-cry-wolf.pdf), 2013.

28. M. Winn, M. Rice, S. Dunlap, J. Lopez and B. Mullins, Constructing cost-effective and targetable industrial control system honeypots for production networks, *International Journal of Critical Infrastructure Protection*, vol. 10, pp. 47-58, 2015.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
23–03–2017	Master's Thesis	Jun 2015 — Mar 2017

4. TITLE AND SUBTITLE

Framework for Industrial Control System Honeypot Network Traffic Generation

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Lin, Htein A., Captain, USAF

5d. PROJECT NUMBER

17G310

5e. TASK NUMBER

5f. WORK UNIT NUMBER

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

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

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-17-M-046

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

Department of Homeland Security ICS-CERT
POC: Neil Hershfield, DHS ICS-CERT Technical Lead
ATTN: NPPD/CSC/NCSD/US-CERT Mailstop: 0635
245 Murray Lane, SW, Bldg 410, Washington, DC 20528
Email: ics-cert@dhs.gov phone: 1-877-776-7585

10. SPONSOR/MONITOR'S ACRONYM(S)

DHS ICS-CERT

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

Defending critical infrastructure assets is an important but extremely difficult and expensive task. Historically, decoys have been used very effectively to distract attackers and in some cases convince an attacker to reveal their attack strategy. Several researchers have proposed the use of honeypots to protect programmable logic controllers, specifically those used to support critical infrastructure. However, most of these honeypot designs are static systems that wait for a would-be attacker. To be effective, honeypot decoys need to be as realistic as possible. This paper introduces a proof-of-concept honeypot network traffic generator that mimics genuine control systems. Experiments are conducted using a Siemens APOGEE building automation system for single and dual subnet instantiations. Results indicate that the proposed traffic generator is capable of honeypot integration, traffic matching and routing within the decoy building automation network.

15. SUBJECT TERMS

Honeypot, network traffic generation, industrial control systems

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. Barry E. Mullins, AFIT/ENG
U	U	U	U	129	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, x7979; barry.mullins@afit.edu