November 2021

Human-centric Cybersecurity Research: From Trapping the Bad Guys to Helping the Good Ones

Armin Ziaie Tabari
University of South Florida

Follow this and additional works at: https://digitalcommons.usf.edu/etd

Part of the Computer Sciences Commons

Scholar Commons Citation
Ziaie Tabari, Armin, "Human-centric Cybersecurity Research: From Trapping the Bad Guys to Helping the Good Ones" (2021). USF Tampa Graduate Theses and Dissertations. https://digitalcommons.usf.edu/etd/9277

This Dissertation is brought to you for free and open access by the USF Graduate Theses and Dissertations at Digital Commons @ University of South Florida. It has been accepted for inclusion in USF Tampa Graduate Theses and Dissertations by an authorized administrator of Digital Commons @ University of South Florida. For more information, please contact digitalcommons@usf.edu.
Human-centric Cybersecurity Research: From Trapping
the Bad Guys to Helping the Good Ones

by

Armin Ziaie Tabari

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Xinming Ou, Ph.D.
Jay Ligatti, Ph.D.
Mehran Mozaffari Kermani, Ph.D.
Nasir Ghani, Ph.D.
Raj Rajagopalan, Ph.D.

Date of Approval:
November 8, 2021

Keywords: Internet of Things, Honeypot, Secure Software Development, Anthropology, Ethnography

Copyright © 2021, Armin Ziaie Tabari
Dedication

This dissertation is lovingly dedicated to my family. There is an especially great sense of gratitude for my loving parents, Dr. Aram Ziaee Tabari and Taraneh Badiee Bahnamiri, whose words of encouragement and commitment to tenacity echo in my ears. It would have been impossible for me to complete my doctoral studies without their endless love and encouragement. I appreciate all that you have done for me, and I love you both.

This dissertation is also dedicated to my brothers, Ali and Ramin, who have always been there for me and have supported me throughout the entire process. I am thankful to them, for being my friend, my guidance, and my pillars of strength throughout my journey. I am also grateful to my sister-in-law, Manijeh, for all her kindness and love. I’m grateful for my beloved niece, Vanda for all the love she borough to our lives.

My family has always been by my side through thick and thin and I dedicate this work to them. You are my world.
Acknowledgments

First, I would like to express my sincere gratitude to my advisor, Dr. Xinming Ou for all the support and encouragement during my Ph.D. study. I learned from him how to handle research difficulties and enhance my capabilities to be a better researcher and engineer. He is an excellent researcher, inspiring advisor and a perfect mentor that every graduate student would like to work with.

I would like to thank my dissertation committee members, Dr. Jay Ligatti, Dr. Mehran Mozaffari Kermani, Dr. Nasir Ghani and Dr. Raj Rajagopalan for their guidance and valuable advice toward completion of this work. In addition, I wish to thank Dr. Daniel Lende for his guidance and critical contributions to the success of our ethnographic study.

Additionally, I am so grateful for my love, Farzaneh. She has constantly encouraged, motivated, and supported my every step of the way.

At last, I would like to thank my fellow Argus Lab members for the many memorable times and fruitful discussions.

This work was supported in part by the U.S. National Science Foundation under Grant No. 1801633 and 1638301, U.S. National Security Agency, Cyber Florida and U.S Office of Naval Research.
# Table of Contents

List of Tables .............................................. iv

List of Figures ............................................. v

Abstract .................................................... vi

Chapter 1: Overview ........................................ 1

Chapter 2: MPMFPot – Multi-phased Multi-faceted IoT Honeypot Framework ........................................ 3
  2.1 Introduction ........................................... 3
  2.2 An Introduction to Honeypots ............................ 5
    2.2.1 Honeypots for General Purposes ................. 6
    2.2.2 Honeypots for Internet of Things ............ 8
  2.3 Honeypot Related Research .......................... 10
  2.4 Chapter Summary ..................................... 12

Chapter 3: MPMFPot Framework ............................. 13
  3.1 IoT Honeypot Framework Design – The First Step .... 13
  3.2 Layer 1 – Multi-Phased Multi-Faceted Low-Interaction Ecosystem ........................................... 15
    3.2.1 Honeypot Server Farms ............................. 16
      3.2.1.1 Off-the-shelf Honeypots ................... 18
      3.2.1.2 HoneyCamera ................................ 19
    3.2.2 Honeypot Vetting Infrastructure ................. 21
    3.2.3 Data Analytics Infrastructure .................. 21
  3.3 Layer 2 – Internet of Things Laboratory and ProxyPot .................................................. 22
    3.3.1 Internet of Things Laboratory .................. 22
    3.3.2 ProxyPot .......................................... 24
  3.4 Chapter Summary ..................................... 25

Chapter 4: Multi-faceted and Multi-phased Deployment/Experimentation ......................... 26
  4.1 HoneyShell ............................................. 26
  4.2 HoneyWindowsBox ...................................... 28
  4.3 HoneyCamera .......................................... 28
  4.4 Chapter Summary ..................................... 29

Chapter 5: Analysis of Honeypot Logs – A Clustering Approach .................................. 31
  5.1 Clustering of Captured Commands ...................... 31
    5.1.1 Similarity Metrics ................................. 31
5.1.2 Clustering Approach ........................................... 32
5.2 Identifying Common Patterns Behind Attacker Intentions ........ 34
5.3 Chapter Summary ............................................... 35

Chapter 6: Experimentation and Data Analysis ......................... 36
6.1 HoneyShell ....................................................... 36
6.2 HoneyWindowsBox .............................................. 39
6.3 HoneyCamera .................................................... 40
6.4 Experimentation on the Clustering Algorithm ....................... 44
6.5 Experimentation on the Grouping Algorithm ....................... 47
6.6 Chapter Summary ............................................... 49

Chapter 7: Discussion and Conclusion .................................. 50

Chapter 8: A Co-Creation Models to Improve the Software Development Process 52
8.1 Introduction ..................................................... 52
8.2 Literature Review ............................................... 53

Chapter 9: Research Methods and Context ............................. 56
9.1 Methodology .................................................... 56
9.2 Data Analysis .................................................. 57
9.3 Context ........................................................ 58
  9.3.1 The Company and Its Products ................................ 58
  9.3.2 Development Process ....................................... 59
  9.3.3 Study Participants .......................................... 60
  9.3.4 Research Ethics ............................................. 60

Chapter 10: Live Discovery through Pen-testing during Ethnography .... 62
10.1 Penetration Testing Method Adopted for This Study ............... 62
10.2 Exploring Software Vulnerabilities Found ........................ 64
10.3 Behaviors and Reactions from Developers ........................ 65

Chapter 11: Analyzing the Findings ................................... 71
11.1 Developers Should Not Totally Trust Programming Languages .... 71
11.2 Outsider vs. Insider ........................................... 72
11.3 Thinking as an Attacker, Thinking as a Developer ................ 74

Chapter 12: Conclusion and Future Work ............................... 76

References ............................................................ 77

Appendix A: Copyright Permissions .................................... 90

Appendix B: Institutional Review Boards (IRB) Approval ................ 92

Appendix C: Clusters Identified from Honeypot Logs ................... 93
Appendix D: Coding .......................................................... 95
  D.1 Full Set of Codes .................................................. 95
  D.2 Final Collaborative Phase ......................................... 97
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>List IoT Devices Installed in the Layer 2 of MPMFPot Framework</td>
<td>24</td>
</tr>
<tr>
<td>6.1</td>
<td>Number of Hits Based on Different Honeypot Facets</td>
<td>36</td>
</tr>
<tr>
<td>6.2</td>
<td>Top 10 Username and Password Combinations – HoneyShell</td>
<td>38</td>
</tr>
<tr>
<td>6.3</td>
<td>Categorization of Downloaded Files – HoneyShell</td>
<td>39</td>
</tr>
<tr>
<td>6.4</td>
<td>Top 10 Commands Executed – HoneyShell</td>
<td>40</td>
</tr>
<tr>
<td>6.5</td>
<td>Attack Types Executed – HoneyCamera</td>
<td>42</td>
</tr>
<tr>
<td>6.6</td>
<td>Top 10 Username Used – HoneyCamera</td>
<td>42</td>
</tr>
<tr>
<td>6.7</td>
<td>Top 10 Password Used – HoneyCamera</td>
<td>43</td>
</tr>
</tbody>
</table>
# List of Figures

| Figure 3.1 | MPMFPot Framework Architectural View | 14 |
| Figure 3.2 | Multi-phased Multi-faceted Honeypot Ecosystem | 16 |
| Figure 3.3 | Honeypot Deployment Locations For Server Farms | 18 |
| Figure 3.4 | HoneyCamera Architecture | 20 |
| Figure 3.5 | IoT Laboratory Diagrams | 23 |
| Figure 3.6 | An Overview of ProxyPot Architecture | 25 |
| Figure 6.1 | Hits per Location/Phase – HoneyShell | 37 |
| Figure 6.2 | Top 10 Countries with the Most Connections – HoneyShell | 37 |
| Figure 6.3 | Type of Malwares Captured – HoneyWindowsBox | 41 |
| Figure 6.4 | Top 15 Countries With Most Attacks – HoneyCamera | 43 |
| Figure 6.5 | Cumulative Frequency Distribution | 44 |
| Figure 6.6 | The Number of Unique Commands in Each Cluster | 45 |
| Figure 6.7 | The Total Number of Executed Commands in Each Cluster | 46 |
| Figure 6.8 | State Machine that Capture Attack Patterns | 48 |
| Figure B.1 | IRB Approval | 92 |
Abstract

The issue of cybersecurity has become much more prevalent over the last few years, with a number of widely publicised incidents, hacking attempts and data breaches reaching the news. There is no sign of an abatement in the number of cyber incidents, and it would be wise to reconsider the way cybersecurity is viewed and whether a mindset shift is necessary. Cybersecurity, in general, can be seen as primarily a human problem, and it is for this reason that it requires human solutions and tradeoffs. In order to study this problem, using two perspectives; that of the adversaries and that of the defenders, I investigated human activities in cybersecurity.

The growing number of Internet of Things (IoT) devices makes it imperative to be aware of the real-world threats they face in terms of cybersecurity. While honeypots have been historically used as decoy devices to help researchers/organizations gain a better understanding of the dynamic of threats on a network and their impact, IoT devices pose a unique challenge for this purpose due to the variety of devices and their physical connections. When a honeypot is built in such a way that an attacker is given the impression it represents a real system used by humans and organizations, it will yield useful insights. Identifying these threats requires an understanding of what attackers are looking for, and how they penetrate our network. It will therefore be possible to have a more secure and safe environment. In the first part of this dissertation, I present here a new Internet of Things honeypot framework, called MPMFPot, which can be used to observe real-world attackers’ behavior within a controlled environment. The MPMFPot framework consists of three layers. As part of layer 1, I designed a new approach towards creating a multi-phased, multi-faceted honey-pot ecosystem, which gradually increases the sophistication of honeypots’ interactions with adversaries. In addition, I developed and designed a low interaction honeypot for cameras
that allowed researchers to obtain a deeper understanding of what attackers are targeting. In the second layer, I designed and built a laboratory for Internet of Things (IoT) devices to analyze the adversaries' behavior in greater detail. This goal was achieved by developing and implementing a proxy instance called “ProxyPot” that sits between IoT devices and the external network and helps researchers study the inbound and outbound communication patterns of these devices. The ProxyPot instance was used to enhance the sophistication of the honeypots in the previous layer as well as helping the researchers to better understand IoT attacks in more depth. The third layer, or communication layer, is responsible for connecting multiple laboratories together. I have also created an innovative data analytics method that enables us to identify the goals of adversaries. These honeypots have been active for more than three years now. In each phase, we have been able to collect increasingly sophisticated attack data. In addition, our data analytics point to the fact that the majority of attacks caught in the honeypots show striking similarities to a great extent and can be clustered and grouped to yield a more complete understanding of goals, patterns, and trends of IoT attacks in the wild. In the second part of this dissertation, I conducted an ethnographic study of a software development company using the anthropological research method of participant observation for a period of six months. I worked as a software engineer to complete this effort and took part in all of the development activities as a new employee. During the course of the fieldwork, I applied and exploited the penetration testing methodology for the company and studied the developers’ reactions on the spot. During this task I found 1) security vulnerabilities are sometimes intentionally introduced and/or overlooked due to the difficulty in managing the various stakeholders’ responsibilities in an economic ecosystem, and cannot be simply blamed on developers’ lack of knowledge or skills; 2) accidental vulnerabilities discovered in the pen-testing process produce different reactions in the development team, often times contrary to what a security researcher would predict. The findings of this study illustrate the nuanced nature of the root causes of software vulnerability and the necessity to take into account a significant amount of contextual information in order
to better comprehend how and why software vulnerabilities can develop during software development. Instead of focusing on the competence of the developers or their practices, this research sheds light on the often forgotten human factors that significantly influence the security of software developed by actual companies rather than simply focusing on the deficiencies in developer knowledge or practice. Furthermore, I find that improving the security of software during the development process can be improved through the implementation of a co-creation model, where security experts collaborate with software developers to better identify security concerns and provide tools that are readily applicable within the context of the software development process.
Chapter 1: Overview

Due to multiple incidents, hacking attempts, and data breaches gaining widespread media attention and public attention over the last few decades, it has become increasingly important to implement a robust cybersecurity strategy. In view of the fact that there is no sign of an abatement in the number of cyber incidents, there is a necessity for us to reconsider the way cybersecurity is viewed and whether a shift in mindset is necessary. There is no question that cybersecurity, in general, is primarily a human problem, and, for this reason, there must be a human solution and tradeoff involved. I conducted research on human activities in cybersecurity using two perspectives: that of the adversaries and that of the defenders, in order to study this problem. My objective was to study the attackers’ behavior by trapping them inside a honeypot and analyzing the commands they executed to learn about their intentions. Additionally, to take a closer look at the defenders, I became a software developer and live among them in order to study the security problem from their native point of view. In my dissertation, I describe these two research projects which I conducted during the course of my PhD. To begin with, in part one, I will describe the MPMFPot framework which was designed to study the attacks against IoT devices. The introduction is in chapter two and the related topics are all discussed. In chapter three, I explore the MPMFPot framework in more detail. In chapter four, I describe the experiments that are carried out within the MPMFPot framework. In Chapter five of my dissertation, I describe the clustering algorithm I developed so that we can analyze how attackers behave and what their intentions are. Chapter six will describe how I conducted the data analysis for this part of study. The conclusion of first part will be presented in the seventh chapter. The second part of this dissertation describes my six-month long embedding study at a small
company that provides security software services. The literature review will be presented in chapter eight. I will explain in chapter nine, how the anthropological method of participant observation was used to gather data, and how data is then analyzed and synthesized using general inductive principles. Chapter ten of this study describes the method of pen-testing I developed and implemented for this study to gain a better understanding of developers’ reactions. The findings from this research are summarized in the eleventh chapter of the dissertation. At the end, chapter twelfth provides a comprehensive analysis of this study.
2.1 Introduction

In recent years, IoT devices have become ubiquitous and essential tools people use every day. The number of Internet-connected devices continues to rise every year, according to a report by IDC. It was estimated that by 2025, there will be at least 41.6 billion IoT devices connected to the Internet [42]. Business Insider projected in their report [43] a 512% increase compared to 2018 (8 billion IoT devices) [43]. The exponential growth raises serious security concerns. For example, many IoT devices have simple vulnerabilities like default username and password as well as open telnet/ssh ports. Usually, these devices are placed in weak or insecure networks, such as those in a home or a public space. In reality, IoT devices are subject to attacks just as much as traditional computing systems, if not more so. New IoT devices could open up new entry points for adversaries and expose the entire network. Around 20% of businesses around the world have experienced at least one IoT-related attack in the past few years, according to [44, 45].

In the past, cyber-attacks have mostly taken the form of data breaches or compromised devices used as spamming or DDoS agents. In general, breaches affect important systems in industry, computer devices, banks, automated vehicles, and smartphones, to name just a few. Moreover, there are a lot of examples where they have caused serious and significant damages. Because IoT devices are now an integral part of most people’s lives, cyber-attacks have become more dangerous because of the widespread use of them. Compared to the past, now many more people are at risk and need to be aware of them. As IoT devices

---

1 Some part of this chapter was published in ACM proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. Permission is included in Appendix A. [1, 2]
become more common, cyber-attacks are likely to change significantly both in terms of reasons and methods. Due to the high level of intimacy IoT devices possess to people’s lives, attacks on them could have much more devastating consequences compared to cyber-attacks in the past. These threats not only affect more people, but they have also expanded in scope. Cyber criminals, for example, can cause unprecedented levels of privacy invasion if they hack into camera devices. These attacks can even endanger people’s lives (imagine an intruder attempting to take control of an autonomous vehicle).

Another factor exacerbating the situation is a pattern in the IoT industry where speed to market overrides security concerns. For example, many IoT devices have simple vulnerabilities like default username and password as well as open telnet/ssh ports. Weak or unsecured networks like those at home or in public places are frequent places that these devices are installed. The exposure to attacks against IoT devices has unfortunately become a reality, if not worse than traditional computing systems. The number of IoT attacks increased significantly in 2017 according to a report by Symantec [46]. They identified 50,000 attacks which had an increase of 600% compared to 2016. In 2021 Kaspersky reported that IoT attacks more than doubled in the first six months of 2021 compared to the six-month period before [47]. In addition, attackers have also improved their skills to make these attacks even more sophisticated with new attacks such as VPNFilter [48], Wicked [49], UPnProxy [50], Hajime [51], Masuta [52] and Mirai [3] botnet. Adversaries are continuously improving their skills to make these forms of attacks even more sophisticated. At present, however, few systematic studies have been conducted on the nature or scope of such attacks in the wild. As of now, most large-scale attacks on IoT devices in the news have been DDoS attacks (e.g., the Mirai attack [3]). Understanding what attackers are doing with IoT devices and what their motives could be is of utmost importance. There have been various approaches used by researchers to study the attackers inside the cyber space environment in the past to find out about new approaches, new tools, etc. Researchers study and identify these problems using live monitoring systems, data forensics, network telescope and honeypots, among other
methods. In this study, one of the main goals was to identify the intentions of humans in a cybersecurity environment and study them. Consequently, we decided to use a honeypot for this study, as using a honeypot will give us the opportunity to indirectly interact with attackers and better understand how they operate in the field by providing them with more information.

2.2 An Introduction to Honeypots

In cyber security, a honeypot is a device set up for the purpose of attracting attack activity. Usually, such systems are Internet-facing devices that either emulate or contain real systems for attackers to target. Since these devices are not intended to serve any other purpose, any access to them would be considered malicious. Security researchers have used honeypots for a long time to understand various types of attacker behavior. Honeypots facilitate researchers’ ability to uncover new methods, tools, and attacks by analyzing data collected by them (network logs, downloaded files, etc.). This allows for the discovery of zero-day vulnerabilities as well as attack trends. As a result of this information, cyber security measures can be improved, especially for organizations with limited resources when it comes to fixing security vulnerabilities.

In the rest of this part, I presents our approach toward a comprehensive experimentation and engineering framework for capturing and analyzing real-world cyber-attacks on IoT devices using honeypots. There are a number of challenges when it comes to creating IoT honeypots and analyzing them for the purpose of producing valuable research data:

1. Various types of IoT devices exist, each of which has unique features that an attacker may wish to access. In order to capture even a small percentage of all IoT devices, it is not feasible to build one honeypot system.

2. At this point, there has not been a deep and systematic understanding of the specific natures of attackers’ activity towards IoT devices, and attackers may have very dif-
ferent focuses. Furthermore, IoT devices offer much more varieties of responses than traditional IT systems due to its interaction with a physical environment. An IoT camera, for example, will need to display some real video to look like a real device. It would take an impressive amount of engineering work to replicate these different types of responses.

3. According to our data, IoT honeypots can collect huge amounts of data, inundating an analyst’s ability to interpret the data and identify actionable intelligence. A challenge for researchers involved in IoT honeypot research is to find interesting data from these huge amounts of data.

We address these challenges through a number of techniques:

1. In order to address the first challenge, we take a multi-faceted approach to the development of IoT honeypots. In order to build a variety of honeypot systems for attackers to target, we adapt off-the-shelf honeypot systems and build some new ones.

2. We adopt a multi-phased approach whereby the sophistication of the emulated responses is increased, as gathered data is analyzed to understand what the attackers might be trying to accomplish in order to overcome the second challenge.

3. By utilizing the speed and convenience of Cosine Similarity and Gaussian Mixture Models (GMMs), we create an clustering algorithm for automatically grouping adversarial activities in an unsupervised way. This provides an opportunity for revealing more stealthy activities that would otherwise be buried in the large amount of background noise.

2.2.1 Honeypots for General Purposes

There are a lot of general-purpose honeypots available, but they’re not specifically targeted towards IoT devices. These honeypots were initially used by researchers to study
the IoT attacks since they are capable of simulating some of the well-known services for IoT devices. There are a number of common honeypots used in general purpose honeynet architecture, and this section identifies them and describes them.

In 2003, Niels Provos developed the open source computer program “HoneyD” [5], which enables users to set up and run as many virtual hosts as they wish on a network, all in one location. Virtual hosts are able to mimic a wide variety of different server types on a computer network, enabling the user to emulate as many kinds of servers as they desire. HoneyD is primarily used for computer security software applications. HoneyD was used by researchers to create low-interaction and scalable honeypots. HoneyD enables the creation of virtual honeypots as well as the ability to integrate physical honeypots with the application. It is able to simulate UDP, TCP, FTP, SMTP, Telnet, IS, POP, and Telnet, as well as simulate various other protocols as well.

Kippo [53] is a Python-based medium-interaction honeypot for SSH. Kippo is capable of logging brute force attacks and the entire shell interaction performed by an attacker. It has a fake filesystem, simulating a Debian Linux server, and that is what the attacker sees on login. The attacker can navigate the system once inside, but he or she cannot effectively do any harm at all.

Cowrie [54] is a medium interaction honeypot designed to log SSH and Telnet interactions performed by an attacker, capturing successful brute force attacks and shell interactions. Cowrie is also capable of serving as a telnet proxy and an SSH proxy for observing the behavior of an attacker on another system. Cowrie was developed as a fork of Kippo. In the event that an attacker logs in, they will be able to explore a simulated Linux shell in which they can run commands and receive realistic looking responses, but these commands will never have the ability to actually be executed outside of the honeypot environment. The reason for this is that this Cowrie “shell” is in fact not the same as the Linux shell in any way. All of the commands and parameters in Cowrie are handled by a Python application.
Dionaea [55] is a low-interaction honeypot solution released as a successor to “Nepenthes”. The intention behind the Dionaea project is to trap malware that exploits vulnerabilities exposed by services that are offered within networks. The Honeypot Dionaea has several features, including: First, Dionaea is based on a modular architecture that implements protocol emulation by using Python as a scripting language. As a second benefit, the most popular protocols implemented in Dionaea are implemented as modules. The protocols that are emulated by Dionaea are SMB, HTTP, FTP, and TFTP. As a third benefit, this tool allows you to use several modules which you may not be aware of, which include MSSQL, MySQL, as well as SIP.

The KFSensor [56] software acts as a honeypot on the Windows operating system. It also acts as an intrusion detection system. It is its job to attract and to detect all the potential attackers in the network. Moreover, it achieves that not only by creating a fake environment which pretends to be a vulnerable one but also by hiding itself as a server, and therefore, not only does it succeed to catch the attacker, but it also helps to know what their reason may be. Because it is particularly designed for the Windows operating system, it comes with a variety of special features that only are available to Windows users. Considering its GUI based console, its low maintenance, and its ease of use.

2.2.2 Honeypots for Internet of Things

When it comes to IoT honeypots, those that provide complete device emulation capabilities are the most versatile. Throughout this section, you will learn about the honeypots designed specifically for Internet of Things devices.

Luo et al. [6] developed a honeypot for Internet of Things devices called IoTCandyJar that enables intelligent interaction. In this intelligent honeypot, which simulates the behavior of IoT devices without the risk of having its interaction compromised, you will have the benefits of both high and low interaction honeypots. It actively scans other IoT devices around the world and sends some part of the received attacks to these devices. It employs ML with
Markov Decision Process to learn the behavior of IoT devices on the Internet and learn which has the best response to extend the attack session.

Honware [7] is a highly interactive honeypot framework which supports emulation of a wide variety of devices without the manufacturers’ hardware being accessed. Honware automatically processes a standard firmware image, logs an attacker’s activity, and records which of their actions lead to a compromise. Honware uses Quick Emulator (QEMU), a tool that enables it to emulate devices to a high degree, and then runs it with a pre-built kernel and the file system settings on the host OS.

ThingPot [8] is an easy-to-implement medium-interaction, scalable, virtual open-source honeypot which simulates the entire IoT platform, as well as all application layer protocols supported by the device. ThingPot simulates the Extensible Messaging and Presence Protocol (XMPP) and Message Queue Telemetry Transport (MQTT) as well as low interaction for HTTP REST traffic. These services can also be run in a virtual environment by using docker containers. In addition to supporting XMPP and REST, the controller node stores data and logs events. With this design, ThingPot was able to mimic a Philips Hue smart light and allow a real attacker to attempt to break into it.

IoTPOT is a hybrid honeypot proposed by Pa et al. [9], which simulates the Telnet services for different IoT devices and is primarily concerned with Telnet intrusions. IoTPOT consists of two parts: a Telnet services “frontend” and a sandboxed “backend”. The front-end low-interaction responder in IoTPOT is used to simulate IoT devices by responding to TCP requests, banner interactions, authentication requests, and command requests. A high-interaction virtual environment, called IoTBOX, running on Linux is proposed to analyze attacks, capture malwares, and run malwares across multiple CPU architectures in the back-end environment.

A virtual production honeypot, HoneyIo4 [10], simulates four IoT devices (a camera, a printer, a video game console, and a cash register) with a low level of interaction. Using HoneyIo4, network scanners that perform reconnaissance attacks are fooled by the simulation
of IoT OS fingerprints that this honeypot uses. Due to this fake OS information, the attack is redirected and it becomes unsuccessful.

Conpot [11] has been widely used by researchers as one of the most popular honeypots for the Industrial Control Systems. Conpot is a low interactive server side ICS honeypot designed to be easy to deploy, modify and extend. It is a project developed and maintained by Honeynet Project, and it is used to work on ICS honeypots. The Internet of Things (IoT) covers a wide range of devices and systems like thermostats, electrical components, and appliances that bear a very close similarity to ICS. Conpot provides a suite of protocols that are typically found on ICS networks, throttling their responses to simulate the real system response time.

2.3 Honeypot Related Research

In essence, a honeypot is designed to fool attackers into thinking that they are accessing a real system by making them think they have gained access to it in the first place. The first honeypot was introduced in 2000 [4]. Honeypots can be categorized into two classes: Low-interaction honeypot and high-interaction honeypot. Low-interaction honeypots only emulate some services such as SSH or HTTP, whereas high-interaction honeypots provide a real operating system with lots of vulnerable services [4]. Honeypots are also categorized based on their purpose [57]. Production honeypots help companies mitigate possible risks, and research honeypots provide new information for the research community. In terms of the location of honeypots, there are many options available. The honeypots can be deployed in cloud computing environments (e.g., Amazon and Azure), Demilitarized Zones (DMZ) of enterprise networks, or in production environments. It is important to realize that each of these deployment options has its own advantages and disadvantages. In addition, the type of deployment environment can influence the choice of honeypot that is most appropriate for your deployment.
Alba et al. [12] conducted a survey of existing threats and vulnerabilities on IoT devices. The first time IoT devices were used as a platform for large Internet-scale attack dates back to the summer of 2016, when the French hosting company OVH was targeted with the first wave of Mirai attacks [3]. In the follow-up attack in October 2016, Mirai brought down the Dyn DNS provider which at the time was hosting major companies’ websites including Twitter, Github, Paypal and so on. Wang et al. [13] presented an IoT honeypot called IoTC-Mal, which is a hybrid IoT honeypot framework, includes low-interaction component with Telnet/SSH service and high-interaction vulnerable IoT devices. Another innovative honeypot is the HoneyPLC honeypot. It develops high-interaction honeypots for Programmable Logic Controllers (PLCs) within ICS [14]. Using a multi-component honeypot, Semic and Mrdovic investigated Telnet Mirai attacks. Honeypots are designed to recruit and target attackers by exposing a weak, generic password in the front end of the honeypots. In place of using an emulation file, the front-end is programmed to generate responses based on input from the attacker, with logic defined in the code. Anarudh et al. [16] developed a honeypot model for the main server to shift DoS attacks in IoT networks and to improve the IoT device performance. Hanson et al. [17] extended the concept of the IoT honeypot by presenting a hybrid honeynet system that includes virtual and real devices. In order to analyze traffic and predict the next move of the attackers, the system used machine learning algorithms. Puna et al. [18] proposed IRASSH-T to develop an IoT honeypot that can automatically adapt to new threats. To capture more information about target malware, IRASSH-T uses reinforcement learning algorithms to identify optimal rewards for self-adaptive honeypots that communicate with attackers. The study by Lingenfelter et al. [19] focused on capturing data on IoT botnets by simulating an IoT system through three Cowrie SSH/Telnet honeypots. To facilitate as much traffic as possible, their system sets the prefab command outputs to match those of actual IoT devices, and uses sequence matching connections on ports. Oza et al. [20] presented a deception and authorization mechanism called OAuth to mitigate Man-in-the-Middle (MitM) attacks.
There have also been studies that utilized low-interaction honeypots, high-interaction honeypots separately or together and studied adversaries’ attacks on IoT devices [21, 22, 23, 24].

Compared to the prior work mentioned above, our main contribution is the design, implementation, and deployment of a multi-phased multi-faceted ecosystem that addresses the challenges of capturing useful attack data on IoT devices and study adversaries behaviors in this context. A comprehensive analysis has also been conducted on the captured data logged by our honeypot framework. With the novel clustering approach that we implemented, we were able to group attackers together and study their intentions at the same time.

2.4 Chapter Summary

IoT is an area of interest that has rapidly been growing in recent years, and this chapter discusses why it is important to improve the security of IoT. In order to enhance the security aspect of IoT devices and have a better understanding of attack vectors against them, we suggest a framework for an IoT Honeypot based on a multi-phased multi-faceted approach. To conclude, related research in the field of IoT honeypots is examined in order to gain an understanding of existing solutions, their shortcomings, and their inspiration in developing this research.
Chapter 3: MPMFPot Framework

One-shot deployment of IoT honeypots – simply having boxes running emulated or simulated IoT systems, can only obtain limited attack information. The longer a honeypot can “hook” an attacker on it, the more useful information can be revealed about attacker goal and tactics. The more interested an attacker becomes in a device, the more sophisticated it needs to be to fool them into thinking it is a real device. Due to the rich interaction an IoT device has with its environment, an IoT honeypot must be organized in a way that allows intelligent adaptation to varying types of traffic. The effectiveness of this arms race is measured by how much useful insights can be gleaned for the amount of engineering effort expended. It is our aim to build a carefully designed a framework that has a variety of honeypot devices working in concert with a vetting and analysis infrastructure, enabling us to achieve a high “return on investment.”

3.1 IoT Honeypot Framework Design – The First Step

As we are designing a new IoT honeypot framework, it is important to make sure that this framework is capable of working with virtually any IoT device. We set out to achieve this objective by creating an IoT laboratory that would enable us to implement different IoT devices and, at the same time, understand how they communicate. Having a full laboratory filled with IoT devices could result in a serious break-in risk that needs to be addressed in the framework. In order to be able to react to attacks made by an attacker, we have proposed the addition of another layer on top of the laboratory. IoT devices in the back-end of the designed framework will be protected by this layer which will reduce the number of

2Some part of this chapter was published in ACM proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. Permission is included in Appendix A. [1, 2]
communications between a real attacker and the devices. One of the problems with IoT devices is that there are so many different types of them. As a result, it may be a little difficult if not impossible to collect the majority of them. In our first study, I proposed adding a third layer as a communication layer. In this layer, the honeypot framework can communicate with other IoT laboratories around the world, or it can access the darknet as a data collection source. Having this layer is essential for the framework because if the requested device doesn’t exist in the local IoT lab, the framework is able to acquire the necessary information from other sources and respond to the attack correctly. Figure 3.1 illustrates our first step towards building such a framework. There are three main layers in this design:
1. The first layer of this framework acts as a “low-interaction honeypot” that responds to a simple request from an attacker. Moreover, this layer contains a knowledge database that can store the requests and responses that take place in this framework. It is the main purpose of this layer to intercept the attacks without having to interact with the second layer or the IoT devices themselves.

2. If layer 1 is unable to handle the received request, it will send it to the next layer for further information to be obtained. The second layer is responsible for processing requests and responses relating to the IoT devices in our laboratory that are actually connected to the Internet. This layer is going to contain a proxy that will receive requests coming through the first layer and then determine which device should be able to receive the request. Aside from that, it will send back the responses received from the real devices and pass it back to the first layer, so that it can be stored in the knowledge-base and also show it to the attacker.

3. In the absence of being able to deal with the received request in the previous layers, the framework will share the request with the other IoT laboratories and work towards getting a response from them. The point of this layer is to assist the community in building a network of IoT laboratories and to share their data so that they may identify the 0-day attacks on IoT devices.

In the remainder of this chapter, we’ll describe our process for creating layer 1 and layer 2 of this framework. The implementation of layer 3 has been postponed to future.

3.2 Layer 1 – Multi-Phased Multi-Faceted Low-Interaction Ecosystem

In order to implement the layer 1 of the MPMFPot, we implemented and designed a honeypot ecosystem consisting of three components, outlined in Figure 3.2. In general, the layer one is what we consider to be the low interaction honeypots. There are three distinct components in this design:
Figure 3.2: Multi-phased Multi-faceted Honeypot Ecosystem

1. honeypot server farms (on premise and in the cloud) that include the honeypot instances

2. a vetting system to ensure that adversaries have a hard time detecting the honeypot device is a honeypot

3. an analysis infrastructure used to monitor, collect, and analyze the captured data

3.2.1 Honeypot Server Farms

Honeypot Instances are hosted by honeypot server farms. To create a wide geographic coverage, we use both on-premise servers and cloud instances from AWS [30] and Azure [31] in multiple countries. Figure 3.3 shows the locations of the honeypot instances deployed in our server farms. These locations include Australia, Canada, France, India, Singapore, United Kingdom, Japan, and United States. For the on-premise server farm, we used a PowerEdge R830 with 256GB of RAM, a VMware ESXi server, and a Synology NAS server for hosting the honeypots and storing logs. The ESXi server is running five Fedora instances and two Windows instances. Two Windows servers and four Fedora instances are used to deploy different honeypots. The fifth Fedora instance runs Splunk [32] to support data monitoring and analytics. Splunk is a software platform that enables search, analysis, and visualization of machine-generated data gathered from various sources. Splunk is used for monitoring and
searching through big data. Information can be indexed and correlated within a container so it can be searched, but also generates alerts, reports and visualizations based on the information. For business challenges such as IT management, security and compliance, it can recognize patterns in data, create metrics, and help diagnose problems. Splunk was used in this study to collect and analyze log files generated by honeypots in order to find patterns inside the data. Honeypot instances running on AWS and Azure are either installed on Ubuntu or Windows depending on what type of honeypot is deployed. At this stage of our research, we have only used low-interaction honeypots. To run honeypots on Fedora and Ubuntu instances we utilize Docker containers. Containerization is one of the technologies that has gained popularity, approval and support in the software development industry over the past few years. There is a possibility that software can lose functionality when it is moved between different development environments. This can be prevented by using containers that ensure no such thing will happen. When considering both the advantages of low interaction honeypots and the capabilities of containerization architecture, it is obvious that both of these aspects can be combined to make an effective tool that can detect attacks as soon as possible. Therefore, it was decided to use this architecture along with honeypots as well. Every honeypot has a container with all dependencies, configuration files, and libraries it needs to function without error. A containerization ecosystem known as Docker was used for this purpose. Splunk receives the logs transmitted over the syslog protocol. In the honeypot ecosystem, networking controls are implemented through security groups to ensure that only entities within the honeypot ecosystem can communicate with each other, and external attackers can only access the honeypot devices through the public-facing interfaces.

In light of the fact that different IoT devices have different specifications and configurations, each honeypot must be designed and configured in a unique way. We adopt a “multi-faceted” approach to building the various honeypot instances. We both use off-the-shelf honeypot emulators and adapt them, and build specific emulators from scratch.
3.2.1.1 Off-the-shelf Honeypots

Many popular off-the-shelf honeypots emulate general services and protocols that are not specific to IoT. However since many IoT devices have those services, it is still useful to adapt these existing honeypots for studying IoT attacks. We evaluated various open-source and commercial honeypots and selected three off-the-shelf software to use in the first step: Cowrie [54], Dionaea [55] and KFSensor [56]. In the rest of this section, a brief introduction of these honeypots is presented.

Cowrie is a low-interaction honeypot\(^3\) that attempts to imitate SSH and telnet services to attract adversaries and capture their interaction. In addition to providing a fake file system and fake ssh shell, Cowrie can also capture files from the input. It can log every activity in JSON format for ease of analysis [54]. Considering that many IoT devices still rely on telnet

\(^3\)The Cowrie author uses the term “medium interaction” honeypot; but it falls within the low interaction category based on the definition introduced in Section 2.3.
and SSH for management, Cowrie is a good honeypot candidate for understanding certain aspects of attacks against IoT devices. We run Cowrie on Debian inside a docker container.

Dionaea is a low-interaction honeypot that emulates various vulnerable protocols commonly found in a Windows system. It was released in 2013 and is useful for trapping malware that exploits vulnerabilities [55]. The main function of this honeypot is to capture malicious files, like worms, that are sent by adversaries. In Dionaea, various protocols can be simulated, including HTTP, MYSQL, SMB, MSSQL, FTP, and MQTT. All detected events are stored in a SQLite database or in JSON format. We run Dionaea on Debian inside a docker container.

KFSensor is a commercial Intrusion Detection System (IDS) that acts as a honeypot to attract and record potential adversaries’ activities. It runs on Windows. KFSensor draws adversaries’ attention from the real systems to itself, providing valuable information for both research and operations. KFSensor is also capable of managing the system remotely, easy integration with other IDSs like Snort [33], and emulating Windows network protocols [56]. Due to Windows’ large footprint as an IoT operating system, both Dionaea and KFSensor can shed light on attacks on IoT devices. In our server farms, KFSensor is installed in Windows VMs.

3.2.1.2 HoneyCamera

Cameras have become an interesting target for IoT devices in the past few years. It is evident that along with the growth in the number of IoT cameras installed, there is also a rise in the number of attacks against them. As a result, the camera was selected as the more specific IoT honeypot [25, 26, 27]. Therefore, in order to detect attacks against these devices, we created a honeypot for IoT cameras and named it HoneyCamera. Figure 3.4 illustrates the honeypot’s architecture. Honeycamera is a low-interaction honeypot for D-Link IoT cameras. One of the devices that have gained a lot of attention recently is the D-Link cameras, which are popular IoT devices. We studied a D-Link camera and carefully
examined its responses to various types of inputs. Honeycamera uses basic authentication for login and repeatedly plays a few seconds’ real video as a fake video stream from the emulated camera device. In addition, we constructed six different pages that emulated the various features of this IoT camera, such as password changing, reading network information, and adding new users. As we observe the adversaries’ behaviors as they attempt to exploit these features, we can gain a deeper understanding of what their intentions may be. We also developed a fake firmware upload service that would let us capture and analyze attack tools and exploits. The adversaries will use this feature in the hope of uploading their malicious code as firmware into a vulnerable camera device. Then we can study the tools they have uploaded by storing them and analysing them. Honeycamera records all activities in JSON format. HoneyCamera is implemented in Python3, runs in Clear Linux [34] that in turn runs inside a docker container.

Figure 3.4: HoneyCamera Architecture
3.2.2 Honeypot Vetting Infrastructure

A honeypot is valuable only as long as it remains undetectable, i.e., unknown to the attacker as a fake system. This is inherently a hard task since honeypots (especially low-interaction ones) will inevitably fail to demonstrate some observable features only a real system can possess, or present ones a real system will never show. An important goal of the vetting process is to find any leaks of information that could identify the device as a honeypot, and mitigate such leaks accordingly. The server farms in the cloud are used to test various fingerprinting techniques to make sure our honeypots cannot be detected easily. We used manual and automatic fingerprinting methods (e.g., Metasploit [35]). We used Shodan [36], an IoT search engine that can be used to search for IoT devices on the Internet. Shodan provides information such as service banners and metadata, and a honeyscore in the range from 0 to 1 (1 indicates honeypot while 0 means real system). This score provides a preliminary insight into how good the honeypot impersonates a real device. We also use Censys [37], another IoT search engine, to help analyze our honeypot instances to make sure they look like the real ones they imitate. Furthermore, and most importantly, fingerprinting approaches of attackers can be identified based on the data captured inside honeypots. Using this insight, we design mitigation solutions that make such fingerprinting ineffective. This is part of our multiphased honeypot design, which will be explained in more depth in chapter 4.

3.2.3 Data Analytics Infrastructure

In order to be successful, two aspects of a honeypot system are equally important: 1) how the honeypot software is developed and implemented; and 2) how the captured data is analyzed. To manage and analyze logs from the honeypot devices, we use Splunk [32]. Splunk provides a tool for creating various queries using its domain-specific language that can be used to achieve various analysis purposes in this work. Splunk is used to analyze all the log files collected from our honeypots. To extract valuable information from the collected logs, we developed a Splunk app. Some example analyses done by the app are identifying
the combinations of username and password used by attackers, analyzing locations of the attacks, detecting the most and least frequent commands executed during attack sessions, analyzing downloaded files and sending them directly to VirusTotal [38], storing the results and checking attackers’ IPs through DShield [39] and AbuseIPDB [40], and so on. These are only some of the most important features that were put in this log management component. In addition, Splunk can collect and visualize data in real time, streamline investigations, search logs dynamically, and take advantage of AI and machine learning embedded in it.

3.3 Layer 2 – Internet of Things Laboratory and ProxyPot

Towards implementation layer 2, the IoT laboratory is first implemented as part of the MPMFPot framework. Furthermore, we designed and implemented the ProxyPot as well. The remainder of this section will describe each of these approaches in detail.

3.3.1 Internet of Things Laboratory

The Internet-of-Things-Laboratory (IoT-Lab) at the University of South Florida has been designed as an innovative laboratory environment. In this lab, the main purpose is to examine the communication of the different IoT devices from the security point of view, as well as to assist the MPMFPot framework in terms of being able to produce meaningful responses for the first layer. A number of devices have been purchased for this laboratory. In order to connect to the internet, IoT devices either use a WIFI connection or network access through a local area network. Figure 3.5a and 3.5b represents a view of the IoT-Lab diagram and environment.

A total of 13 devices have been selected for the first stage of implementation. Listed in table 3.1 are these devices and their connection to the network.

Icons made by “Freepik” from www.flaticon.com and “fabrik” from thenounproject.com/fabrikicons/
Figure 3.5: IoT Laboratory Diagrams
Table 3.1: List IoT Devices Installed in the Layer 2 of MPMFPot Framework

<table>
<thead>
<tr>
<th>#</th>
<th>Device Name</th>
<th>MAC Address</th>
<th>Internet Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nest Camera Indoor</td>
<td>18:B4:30:5E:C1:59</td>
<td>WIFI</td>
</tr>
<tr>
<td>2</td>
<td>Nest Camera Outdoor</td>
<td>18:B4:30:E4:60:B7</td>
<td>WIFI</td>
</tr>
<tr>
<td>3</td>
<td>TP_Link Smart Outlet 1</td>
<td>50:C7:BF:78:D1:1D</td>
<td>WIFI</td>
</tr>
<tr>
<td>4</td>
<td>TP_Link Smart Outlet 2</td>
<td>50:C7:BF:78:E3:02</td>
<td>WIFI</td>
</tr>
<tr>
<td>5</td>
<td>Nest Smoke Detector</td>
<td>18:B4:30:9D:25:F5</td>
<td>WIFI</td>
</tr>
<tr>
<td>6</td>
<td>Amazon Echo</td>
<td>4C:EF:C0:5D:76:9D</td>
<td>WIFI</td>
</tr>
<tr>
<td>7</td>
<td>Samsung SmartThings</td>
<td>D0:52:A8:A0:6C:B4</td>
<td>LAN</td>
</tr>
<tr>
<td>8</td>
<td>Wink Hub</td>
<td>00:21:CC:4D:53:EA</td>
<td>LAN</td>
</tr>
<tr>
<td>9</td>
<td>LaMetric_LM7879</td>
<td>88:83:5D:FB:83:9D</td>
<td>WIFI</td>
</tr>
<tr>
<td>10</td>
<td>D-link Camera</td>
<td>B0:C5:54:3F:DD:B1</td>
<td>LAN</td>
</tr>
<tr>
<td>13</td>
<td>Philips Hue</td>
<td>00:17:88:27:D8:D1</td>
<td>LAN</td>
</tr>
</tbody>
</table>

3.3.2 ProxyPot

IoT Lab was designed and developed with the primary goal of studying the communication of each device that is integrated into it. These communication knowledge will aid us to better understand how the legitimate activities look like when the objective is to access an Internet of Things device. Also, the second layer of MPMFPot plays a very important role in the whole framework when it comes to interaction with the IoT devices in our laboratory and providing meaningful responses to the first layer. This way, the framework is able to speak with many more cyber-threat actors. First, I designed and implemented a proxy instance on top of the lab, called it ProxyPot, and enables it to communicate with various lab devices. In short, ProxyPot is a proxy instance that sits between an IoT device and a network gateway, capturing all traffic between these devices. A diagram of the ProxyPot architecture is shown in Figure 3.6. The proxypot application has been written in Python and installed within a Raspberry Pi3 model B in our IoT-Lab. I have designed several modules specifically for this device. The first is a scanning module which does a scan of the network using the Nmap back-end which enables the software to identify the devices that are connected to the network automatically. The second component is a Man-in-the-Middle module.
uniquely designed for the ProxyPot. This module allows the ProxyPot to put itself in the way between the gateway and the target device and to monitor all the traffic that passes between them. In addition, the device also has a capability of capturing traffic. Through the use of T-Shark, ProxyPot is able to capture the traffic in the *pcap* format and convert it into the JSON format for further analysis. After that, ProxyPot uses the report module to send the JSON files to the analysis infrastructure and store them there. During nearly a year, I had the ProxyPot running in our lab. The first result of this data analysis was used to add more information into the HoneyCamera’s second phase installation (section 4.3 provides more information).

3.4 Chapter Summary

It was discussed in this section how we designed and implemented a honeypot framework that is effective for IoT devices. Because IoT devices create a variety of challenges on this topic, I propose the “MPMFPot – Multi-phased Multi-faceted – IoT honeypot framework” that addresses these challenges. A detailed explanation of the different components of this architecture is provided in this section. As well as utilizing well-known off-the-shelf honeypots, I have designed the *HoneyCamera* and *ProxyPot* to meet the requirements of this framework. Detailed information about these two instances is provided in this section.
Chapter 4: Multi-faceted and Multi-phased Deployment/Experimentation

In the first layer, we use a multi-phased approach to introduce sophistication into how our honeypots respond to attacker traffic, based on traffic collected previously. In the first phase, we simply deploy the honeypot at hand and receive attack traffic. From this point forward, the honeypot ecosystem collects data, and that data will be analyzed in order to create the subsequent phases defined by what attackers seem to be looking for, and we can emulate those responses accordingly. We go through multiple iterations until we are satisfied with the insights we gained and the attacker’s behaviors. The insights from the previous phase are used to drive the creation of more sophisticated low-interaction honeypots. We present this multi-phased process from three facets that our honeypots attempt to capture about IoT attacks: attacks through login service to obtain a command shell, windows service attacks resulting in malware download, and IoT camera attacks.

4.1 HoneyShell

We use the “Cowrie” honeypots for emulating vulnerable IoT devices over SSH (port 22) and telnet (port 23). Cowrie can be configured to emulate different types of operating systems. A popular Linux distribution for IoT devices is busybox [41]. Therefore, we configure our Cowrie honeypots to emulate busybox. Three Cowrie honeypots were created for the three phases.

- During Phase 1, an initial version of cowrie is deployed with minimal changes to the original code. This step was designed to begin collecting data that would be used in

---

5Some part of this chapter was published in ACM proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. Permission is included in Appendix A. [1, 2]
the next step and identify any information leakage. Every possible combination of usernames and passwords are accepted by the honeypot at this stage. We deployed four honeypot instances – two on-premise and two in the cloud (Singapore, United states).

- In *Phase 2*, honeypot instances were deployed on-premises after six months of testing the phase 1 infrastructure. Our honeypot instances are filled with more data as we fix bugs. We selected the top 30 username/password combinations that executed at least one command after logging in as the authentication credentials for our honeypot. We gathered this information from our analysis component. The honeypot will display login failure messages for any other combination of username and password. A further modification of the emulation mechanism is that it is configured in such a way that attackers are provided with more meaningful responses, such as adding new usernames and file systems to the configuration. In addition, we emulated new commands and added them to the honeypot configuration files in order to attract more activity. We also analyzed phase 1 logs for fingerprinting techniques used by attackers, in order to handle them properly in phase 2. Examples include *file* command’s response being added to the honeypot configuration.

- In *Phase 3*, the key part is using all of the information that has been collected so far to create a more sophisticated honeypot. The purpose of this step was to design a honeypot instance that could attract a real human (attacker) into it. Therefore, a complex password was generated, and only one possible login combination was possible. Due to the complexity of the password, a successful login indicated it was probably a real hacker, and therefore extremely valuable information could be gathered. The honeypot filesystem was replaced by a cloned version of the operational system’s filesystem. All confidential information is replaced with fake information, so in case a hacker successfully logged into the honeypot, it is not exposing any real data.
4.2 HoneyWindowsBox

Using “Dionaea”, we emulate IoT devices running on Windows. The majority of these attacks result in malware being downloaded on the device. It would require some additional work to further emulate the downloaded malware’s behavior inside a honeypot, so this is reserved for future work. In this work, we use phase 2 of this honeypot to apply our vetting system to ensure they are not easily identifiable as honeypots.

- In Phase 1, a default version of Dionaea was deployed in the cloud. To identify the weak point of Dionaea, we used the cloud infrastructure as a test bed. AWS France hosts an instance of this honeypot. This instance was detected quickly as a honeypot by our vetting system. Nevertheless, it continued to capture automated malicious activities, which helped us create phase 2.

- During Phase 2, various services were broken down into two different combinations. The first honeypot provides FTP, HTTP, and HTTPS, whereas the second only provides SMB and MSSQL. These two versions were deployed across three locations (India, Canada, and on-premise). In our vetting system, these IP addresses appeared as real systems. We enhanced the HoneyWindowsBox by introducing the KFSensor into the ecosystem to add more coverage into our honeypot. As for the locations, we chose Paris and on-premise, and each instance was vetted.

4.3 HoneyCamera

The “HoneyCamera” was the last one to be implemented. During this step, we will simulate the behavior of an IoT device more specifically. We chose the D-Link camera from our IoT-Lab for this study. ProxyPot was used to study the camera’s communication with the outside world. HoneyCamera was designed based on the data collected from the ProxyPot. By implementing the first version, we began collecting logs against it. Using the collected data, we identified possible weaknesses within the application. In the second
phase, we improved the application and also added some new vulnerabilities we observed adversaries trying to exploit in the previous phase. To increase the sophistication of this honeypot, a HoneyShell instance was also added to the HoneyCamera.

- **Phase 1**, involved the deployment of three honeypots. The two instances in Sydney and Paris only had port 8080 open, while the one in London had port 80. The first two honeypots were used to emulate D-Link DCS-5020L and the other one to imitate D-Link DCS-5030L camera. Instances of this type are configured in such a way that they provide as much information as an interaction-based honeypot can. These instances of HoneyCameras were identified as real IoT devices by our vetting system. Data collected in this phase indicated that attackers were also trying to exploit known vulnerabilities related to the IoT cameras.

- **Phase 2**, We discovered 6 vulnerabilities that attackers attempted to exploit inside HoneyCamera from the data collected in Phase 1. The most common bug was Authentication Information leakage. These vulnerabilities were carefully studied, and we incorporated the corresponding responses into HoneyCamera instances. Additionally, the IoT cameras are equipped with a telnet/SSH port for remote configuration and diagnostic purposes. In order to replicate these types of activities, we combined our HoneyShell and HoneyCamera and deployed them as single instances into the on-premise and cloud (Tokyo) infrastructures. Using HoneyCamera and HoneyShell, we were able to identify attacker behavior that involves both Unix command-line and camera-specific commands.

### 4.4 Chapter Summary

The aim of this chapter is to describe the implementation and experimentation of the multi-faceted and multi-phased IoT honeypot ecosystem. There are three components implemented by this framework: Honeyshell, HoneyWindowsBox, and HoneyCamera. All of
them are deployed according to multiple phases. In the first phase, all we need to do is deploy the honeypot and wait for the attack traffic to come in. The second phase goes through many iterations until we are satisfied with the insights we gained and the attackers’ behaviors elicited. Later on, in the third phase, these insights are used to create even more advanced low-interaction honeypots.
Chapter 5: Analysis of Honeypot Logs – A Clustering Approach

For the unique nature of IoT devices’ communication and the various types of commands, it can be difficult to discover new or unknown cyber attacks against these devices. To differentiate between different malicious actors such as Bot actors and Zero-day attacks, we need to better understand how commands cause program execution. One key observation from our data is that the honeypot instances collect huge amount of attack activities, but most of these activities belong to a few categories. Activities in the same category show similarity among one another. This inspires us to design an unsupervised approach using clustering, so that we can group similar attacks together to make the attackers’ intentions clear.

We adopt a distance-based clustering method, which utilizes “cosine similarity” and the unsupervised learning algorithm “Gaussian Mixture Model (GMM)” [29] to calculate the distances between different commands executed in the honeypot and perform clustering based on this metric. We then identify “actors” (represented as unique IP addresses) that share similar commands according to the clustering results, and group the actors based on this similarity. The attacker intentions then emerge from those groupings. In the rest of this chapter we describe the clustering and grouping algorithms and the intuitions behind them.

5.1 Clustering of Captured Commands

5.1.1 Similarity Metrics

Our honeypots captured large numbers of commands through SSH login sessions. We used cosine similarity as the metric for determining how similar two commands are [28]. It measures the cosine of the angle between two vectors in a multidimensional space. In this
context, the two vectors are arrays containing the word counts of two commands executed inside a honeypot. A smaller angle means a higher similarity. Using the Euclidean dot product formula, the cosine of two non-zero vectors \( \mathbf{A} \) and \( \mathbf{B} \) can be found through the following equation.

\[
\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\|\|\mathbf{B}\| \cos \theta
\]  

(5.1)

where \( \theta \) is the measure of the angle between \( \mathbf{A} \) and \( \mathbf{B} \) in a high-dimensional space. The similarity is then calculated as:

\[
\text{similarity}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{p} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}
\]  

(5.2)

where \( A_i \) and \( B_i \) are components of vector \( \mathbf{A} \) and \( \mathbf{B} \) respectively. The values are between 0 and 1. A cosine value of 0 means that the two vectors are at 90 degrees to each other (orthogonal) and have no match. The closer the cosine value to 1, the smaller the angle and the greater the match between the two vectors. As an example, the cosine similarity between the following two commands is 0.6249.

“\texttt{cat /proc/cpuinfo — grep name — cut -f2 -d: — uniq -c}”

“\texttt{cat /proc/cpuinfo — grep name — head -n 1 — awk \{print $4,$5,$6,$7,$8,$9;\}}”

5.1.2 Clustering Approach

Researchers have been struggling for the past decades to devise mechanisms to detect cyber-security threats. This period of effort resulted in a number of novel approach such as rule-based, signature-based, and supervised Machine Learning (ML) algorithms that were developed to detect intrusions that have already been encountered and classified as such. The reality is that new unknown threats go undetected as they are often misclassified by those techniques and are often referred to as zero-day attacks or zero-day threats. Therefore, we used a soft clustering method known as Gaussian Mixture Models (GMM), which are probabilistic models for representing normally distributed subpopulations within an overall
population. It is a form of unsupervised learning. First, we extract all executed commands from the HoneyShell logs. We calculate cosine similarity metrics between the unique commands, and then used the Gaussian Mixture Model to create the clusters where similar commands are clustered together.

We examined the created clusters carefully and identified the objective(s) behind each cluster at a higher level. Some commands had multiple subcommands — the adversaries executed them all together in a single composite command. Such composite commands may be clustered with other commands that share some characteristics, but not all of them. For this reason, a cluster may be labeled with multiple objectives, but not every command in the cluster demonstrates all the objectives. Here are a few examples of how clusters’ objectives (or goals) look like

- Cluster 7 includes commands such as `free -m` and `free -h`. These commands display information about how much physical memory and swap memory is present, as well as how much free and used memory is available. We identify the objective as “System Intelligence.”

- Cluster 17 includes `lspci grep VGA`. The adversaries are trying to obtain information pertaining to the GPU. We identify the objective as “GPU intelligence.”

- Cluster 24 consists of `cat /proc/cpuinfo`. This command attempts to extract information about the CPU cluster; we thus named the objective “CPU Intelligence.”

- One command included in Cluster 21 is `git clone https://github.com/robertdavidgraham/masscan.git`. Masscan is an Internet-scale port scanner. According to the author, it is capable of scanning the entire Internet within 5 minutes, sending 10 million packets per second, from a single machine. The cluster also includes `wget -c http://222.186.139.216:9960/chongfu.sh`, which the VirusTotal report indicates that the file contains a Shell Downloader. These data led us to identify the objectives as “Pivot point,” “Malicious Installation,” and “Resource Capture /Extraction.”
- Cluster 37 includes the command `/etc/init.d/iptables stop`, which indicates that an attacker tried to disable the firewall. As a result, we identified the objective as “Stop Services.”

We went through all the unique commands in each cluster to identify the objectives behind those commands. Appendix C provides an exhibit of all the clusters from our analysis, along with all the objectives identified in each cluster.

5.2 Identifying Common Patterns Behind Attacker Intentions

Our next step is to use the clustering results to help us identify the intentions behind the malicious actors. For simplicity, we identify a malicious actor as a remote source IP address identified in the honeypot log. We wanted to find out whether different actors exhibit similar behaviors through shared command clusters. The intuition is that if two actors’ commands fall into a number of the same command clusters, the shared clusters then represent a pattern of behaviors that likely pursue the same type of objectives. By identifying such shared command clusters, we can identify common patterns behind attacker intentions.

We first find all the pair-wise overlaps of command cluster IDs between any two actors (IP addresses). Two actors do not have to share the exact same commands to have overlap, as long as the commands belong to the same cluster as identified in the process described in Section 5.1.2. For an overlap to count as a pattern, it needs to be shared by at least three actors, and has a minimum of ten different clusters. For each pattern, we also associate it with the actors that manifest it, i.e., the IP addresses demonstrate the commands belonging to all the clusters in the pattern. We use the term “group” to refer to these actors (IP addresses) that share that pattern. Some actors may be associated with multiple groups, i.e., they demonstrate multiple patterns in their recorded behaviors. If an attacker shares the same pattern, their corresponding actions have the same intentions, even though the specific techniques and tools may be different. Using this approach, we could determine attack trends and intentions. As soon as a new vulnerability is known in the wild, adversaries
will try to take advantage of it as soon as possible and target as many victims as they can. Thus these new activities will likely form a pattern observable from the honeypots. Finding these patterns and the associated malicious actors could allow defenders to determine if the attackers might launch the next steps of their attacks, and take actions accordingly.

5.3 Chapter Summary

The purpose of this chapter is to introduce a new approach for identifying the distribution of similar attack commands inside a honeypot by using clustering methods. Cosine similarity and Gaussian mixtures were used to group executed commands into 50 clusters. Using these information, we identified the purposes of each cluster and labelled them accordingly. It should be noted that there are some clusters that have more than one label. This may be due to the fact that similar commands may run for different reasons. Also, we used these clusters to group attack actors (based on the source IP address). Every group represented a higher level purpose from the outset until the end.
A total number of 22,629,347 hits were captured by our honeypot ecosystem over a period of three years. As shown in Table 6.1, HoneyShell attracted the most hits. This information is described in detail in the rest of this section.

Table 6.1: Number of Hits Based on Different Honeypot Facets

<table>
<thead>
<tr>
<th>Honeypot</th>
<th>Up Time</th>
<th># of Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>HoneyShell</td>
<td>12 months</td>
<td>17,343,412</td>
</tr>
<tr>
<td>HoneyWindowsBox</td>
<td>7 months</td>
<td>1,618,906</td>
</tr>
<tr>
<td>HoneyCamera</td>
<td>25 months</td>
<td>3,667,029</td>
</tr>
</tbody>
</table>

In the following sections, I present the results from the experimentation of the multi-phased honeypot evolution as described in Section 4. The analysis presented therein is based on data collected in the last phase in each experiment\(^7\).

### 6.1 HoneyShell

Cowrie honeypots were able to capture the largest portion of the hits during this period. Figure 6.1 represents the number of hits based on locations and phases. It is notable that the on-premise phase 2 honeypot captured more hits in 6 months’ time than the on-premise phase 1 honeypot did in a year, clearly showing the effectiveness of the multi-phased approach. Figure 6.2 shows that the majority of connections came from China, Ireland and the United Kingdom.

\(^6\)Some part of this chapter was published in ACM proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. Permission is included in Appendix A. [1, 2]

\(^7\)In the discussion I sometimes mention data from earlier phases for the purpose of comparison.
Figure 6.1: Hits per Location/Phase – HoneyShell

Figure 6.2: Top 10 Countries with the Most Connections – HoneyShell
Furthermore, statistics shows that 15% of the total number of hits belong to successful logins. Most of these logins used random combinations of username and password which shows that automated scripts were used to find the correct authentications blindly. Table 6.2 represents the top 10 username/password combinations that were used by attackers. The information seems to indicate that attackers commonly look for high-value user with a weak password. However, by looking into the database, some other combinations such as “university/florida”, “root/university” and “university/student” were found inside the on-premise honeypot (inside a university) which indicates that attackers were aware of the organization’s nature. and tried to customize their attacks based on that.

Table 6.2: Top 10 Username and Password Combinations – HoneyShell

<table>
<thead>
<tr>
<th>Username/Password</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin / 1234</td>
<td>975729</td>
</tr>
<tr>
<td>root / (empty)</td>
<td>167869</td>
</tr>
<tr>
<td>admin / (empty)</td>
<td>82018</td>
</tr>
<tr>
<td>0 / (empty)</td>
<td>62140</td>
</tr>
<tr>
<td>(empty) / root</td>
<td>52780</td>
</tr>
<tr>
<td>1234 / 1234</td>
<td>50305</td>
</tr>
<tr>
<td>admin / admin</td>
<td>39349</td>
</tr>
<tr>
<td>admin / 1234567890</td>
<td>12444</td>
</tr>
<tr>
<td>root / admin</td>
<td>10359</td>
</tr>
</tbody>
</table>

In addition, only 314,112 (13%) unique sessions were detected with at least one successful command execution inside the honeypots. This result indicates that only a small portion of the attacks executed their next step, and the rest (87%) solely tried to find the correct username/password combination. A total number of 236 unique files were downloaded into honeypots. 46% of the downloaded files belong to three honeypots inside the university, and the other 54% were found in the honeypot in Singapore. Table 6.3 demonstrates categorization of the captured malicious files by Cowrie. VirusTotal flagged all these files as malicious. DoS/DDoS executables were the most downloaded ones inside honeypots. Attackers tried
to use these honeypots as a part of their botnets. IRCBot/Mirai and Shelldownloader were the second most downloaded files. It shows that Mirai, which was first introduced in 2016, is still an active botnet and has been trying to add more devices to itself ever since. Shelldownloader tried to download various formats of files that can be run in different operating systems’ architectures like x86, arm, i686 and mips. It should be highlighted that since adversaries were trying to gain access in their first attempt, they would run all the executable files. SSH scanner, mass scan and DNS Poisoning are categorized in the “Others” section of Table 6.3.

Table 6.3: Categorization of Downloaded Files – HoneyShell

<table>
<thead>
<tr>
<th>Malicious Files Campaign</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dos/DDos</td>
<td>59</td>
</tr>
<tr>
<td>IRCBot/Mirai</td>
<td>40</td>
</tr>
<tr>
<td>SHELLDownloader</td>
<td>40</td>
</tr>
<tr>
<td>BACKDOOR</td>
<td>36</td>
</tr>
<tr>
<td>CoinMiner</td>
<td>31</td>
</tr>
<tr>
<td>Others</td>
<td>30</td>
</tr>
</tbody>
</table>

Besides downloading files, attackers tried to run different commands. Table 6.4 shows the top 10 commands executed with their occurrence number.

6.2 HoneyWindowsBox

Dionaea was representing a vulnerable Windows operating system. Most of the connections came from the United States followed by China and Brazil. During the usage of Dionaea, 43 unique files were captured in our on-premise infrastructure. Type of malwares observed by our HoneyWindowsBox is represented in figure 6.3. HTTP was the protocol used the most by attackers. FTP and smb were also used to download malicious files. In addition, a noticeable amount of SIP communication was found in the process of examination. SIP is mostly used by VoIP technology, and like other services, it suffers from common vul-
nerabilities such as buffer overflow and code injection. Collected data from these honeypots was used to create a more realistic file system for other honeypots.

KFSensor is an IDS-based honeypot. It listens to all ports and tries to create a proper response for each request it receives. The information gathered from this honeypot was also used to create a better environment and file system for Dionaea.

6.3 HoneyCamera

Six IoT camera devices were emulated using HoneyCamera. Figure 6.4 shows that most attacks captured inside the on-premise HoneyCamera came from Chile. Several malicious files attempt to be installed in these honeypots. These were mainly coin-miner and Mirai (variants) files. Analyzing the captured logs reveals that this honeypot attracted many attacks specifically targeting IoT cameras. Here are some examples:

- The first attack found was camera credential brute-force (/?action=stream/snapshot.cgi-?
user=[USERNAME]&pwd=[PASSWORD]&count=0). On this attack, adversaries...
tried to find a correct combination of username and password to get access to the video streaming service.

- The second attack found was trying to exploit CVE-2018-9995 vulnerability. This vulnerability allows attackers to bypass credential via a “Cookie: uid=admin” header and get access to the camera (/device.rsp?opt=\&cmd=list).

- A list of more attacks can be found in Table 6.5. D-Link, Foscam, Hikvision, Netwave and AIVI were only some of the targeted cameras found from the data collected from these honeypots.

In addition, attackers mostly (92%) used GET protocol to communicate with the honeypots, 5% used POST method. The rest 3% used other methods such as CONNECT, HEAD, PUT, etc. Table 6.6 and table 6.7 represent the top 10 username and password that were used by attackers to login into the on-premise HoneyCamera.
We intentionally crafted the HoneyCamera vulnerability to reveal the username and password for the login pages. We instrumented the vulnerable page such that a successful exploit will reveal the username and password as an image inside the HTML page, indistinguishable to humans’ eyes from the effect of the real vulnerability. Based on the analysis of the log files, 29 IP addresses exploited this vulnerability and successfully logged into the Honeycamera web console and explored it. The pattern of the user’s movements between different web pages and the fact that the username and password were only visible to humans’ eyes indi-
Figure 6.4: Top 15 Countries With Most Attacks – HoneyCamera

Table 6.7: Top 10 Password Used – HoneyCamera

<table>
<thead>
<tr>
<th>Password</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin</td>
<td>1280</td>
</tr>
<tr>
<td>8hYTSUFk</td>
<td>150</td>
</tr>
<tr>
<td>password</td>
<td>116</td>
</tr>
<tr>
<td>123456</td>
<td>70</td>
</tr>
<tr>
<td>admin1</td>
<td>65</td>
</tr>
<tr>
<td>1234</td>
<td>65</td>
</tr>
<tr>
<td>admin123</td>
<td>64</td>
</tr>
<tr>
<td>12345</td>
<td>63</td>
</tr>
<tr>
<td>password1</td>
<td>60</td>
</tr>
</tbody>
</table>

cate that these activities likely were performed by a real person as opposed to an automated program.
6.4 Experimentation on the Clustering Algorithm

In order to identify the attacker’s intentions, we began by extracting all of the commands executed from HoneyShell’s logs. This experiment was conducted using Singapore Honeyshell logs. The total number of unique commands found in this process was 526. After applying the algorithm outlined in chapter 5, 50 clusters have been generated. Figure 6.6 shows the distribution of the number of unique commands that occur in each cluster. Figure 6.7 shows the total number of commands executed in each cluster. As can be seen on figure 1, the majority of commands executed (99.7%) belong to only six clusters [18, 25, 26, 27, 39, 40]. There are two major attacks that are grouped together in these clusters: Fingerprinting and Mirai and its variants. Figure 6.5 represent the Cumulative Frequency Distribution for the number of commands executed in each cluster.

Figure 6.5: Cumulative Frequency Distribution
Figure 6.6: The Number of Unique Commands in Each Cluster
Figure 6.7: The Total Number of Executed Commands in Each Cluster
6.5 Experimentation on the Grouping Algorithm

In Section 5.2, we described our approach to identify attacker patterns and group the adversaries together based on those patterns. As a result of this process, 84 different patterns/groups were identified. Examining the command clusters and the concrete commands in each group, reveals how the adversaries’ attack patterns are arranged.

As a high-level strategy, we classified the attack commands into three categories: 1) Fingerprinting, 2) Malicious Activities, and 3) Miscellaneous. Activities related to fingerprinting aim to identify the resources on a target, such as the number of CPUs, whether the target has GPUs, HoneyPot fingerprinting activities, etc. As a result of these details, adversaries select their candidate for the next step of their attacks. The next steps may result in the installation of malicious software if the target returns a satisfactory result. Our analysis shows the presence of a large amount of malware and coin-miners installed at that time. Sufficiently advanced bots, such as Mirai and its variants, begin their activities after a successful login into the target. Malicious Activity is the second high-level category, which includes the commands that attempt installing malicious programs in the honeypot without fingerprinting. Other commands executed inside our HoneyShell are defined as Miscellaneous. This includes stopping services, creating pivot points, scanning the network, and so on.

We created a state machine (Fig 6.8) that defines the possible transitions from one goal to another, based on manual inspections of the patterns identified above. The state machine could be used to forecast the goals of an attacker in the future. We provide an example below to illustrate how we utilize the patterns to create the state machine. We grouped 90 IP addresses in group 5. Among the clusters shared by these 90 IP addresses, there were 25, 17, 23, 39, 35, 46, 5, 9, 30 and 24. The concrete commands from these clusters include the following (not based on time order).

• `uname -a`
In light of these data and after analyzing the clusters and commands, we abstract this pattern as the following: **Fingerprinting** → **System Intelligence** → **Malicious Installation**. In particular, *uname -a* and *echo ' ' > /var/log/messages* belong to **System Intelligence**, which is part of **Fingerprinting**. The other commands fall into the category of **Malicious Installation**.
6.6 Chapter Summary

This chapter illustrates the different data analytics for each facet based on our “Data Analytics Infrastructure”. Some of these details include the number of hits, the top command executed, the locations of the top attackers, etc. This chapter also includes experiments on grouping and clustering.
Chapter 7: Discussion and Conclusion

Analyzing the data from our IoT honeypot ecosystem in several phases yielded some interesting results. As it turns out, IoT devices are under heavy attack by automated tools and bots. The Mirai and its variants are still active, looking for targets to add to their arsenal. Additionally, the increasing sophistication in the data we collected in each phase proves that the multi-faceted multi-phased approach is a useful approach for designing an IoT honeypot ecosystem to study and identify unknown novel attacks. This was further supported by human activities we captured in HoneyCamera as presented in section 6.3. The vast majority of data captured in our ecosystem was bot-related. This makes it difficult to detect unknown and stealthy attacks. Our clustering algorithm provides the insight that by utilizing a syntax-based similarity metrics we can group the most executed commands together, providing important insights for understanding the background noise. In addition, our grouping algorithm attempts to identify the various intentions of the attackers based on their commands as they show up in the various clusters. A future direction is to further research the granularity of such intentions to visualize more fine-grained steps of attackers’ mode of operations.

The MPMFPot is a multi-layered, multi-phased and multi-faceted approach to building a honeypot ecosystem for the Internet of Things presented in this part. As part of the implementation process, two new instances were designed and implemented, HoneyCamera, a new honeypot that is low-interaction and focused on camera devices, and Proxypot, a proxy instance that captures the traffic between IoT devices and the gateway. Analysis on the information captured during this work shows that adversaries generally look for vulnerable IoT devices to exploit them. Moreover, the results indicate that a more realistic
and well-configured low-interaction honeypot will attract more attacks in the same network compared to a honeypot that is poorly configured. According to HoneyCamera’s log files, IoT camera devices have become an interesting target for attackers in recent years. A number of different vulnerabilities were found from this process.
Chapter 8: A Co-Creation Models to Improve the Software Development Process

8.1 Introduction

A major part of ever-present software vulnerabilities can be attributed to human factors, with substantial research devoted to this area [59, 60, 61, 62, 63, 64, 65, 66, 67, 68]. These Researchers have utilized various approaches in past efforts, such as surveys, interviews, experiments, examining code artifacts and studying competition results. It is also understood that software insecurity has a fundamental economic basis [69], and industry often seems reluctant to give code security the same priority as other business considerations, such as the speed of time to market and richness of features. Therefore, it is crucial to recognize that software security (in)security is not solely driven by the developers’ knowledge and skills and the coding languages and environments that they use, but also by the various incentives in the market and within the organization. It is therefore essential to study secure software development within the context of where it happens, i.e., within software companies, to have a real impact. Incentive structures, organizational relationships, work flow, and other factors impact human behaviors in specific contexts; this specificity is hard to replicate in controlled settings and to assess using standardized surveys or interviews [70, 71]. Replication research also requires understanding these structures and relationships in the first place, which close proximity observation facilitates [72, 73, 74, 75].

In a recent study, Sundaramurthy et al. [76, 77] demonstrated that participant observation, an anthropological research method [78, 79], allows researchers to gain a better under-
standing of the challenges faced by security analysts in security operations centers (SOCs). The embedding in SOCs also allowed the researchers to develop non-technical and technical interventions to improve their work processes and environments by addressing specific pain points within them. Following the success of that study, we conducted an ethnographic study at a software company utilizing the same method of participant observation. An anthropologist trained two PhD students in computer science in qualitative research methods, and they spent 1.5 years taking part in the fieldwork. Researchers performed normal business activities throughout the study period, including coding and participating in meetings, while observing and studying software (in)security issues through analysis of historical details (code repositories and ticketing system records), pen-testing the developed software and observing how developers and managers address vulnerabilities. After collecting their observations, the fieldworkers shared their insights with a larger group of researchers, including an anthropologist and two computer science professors who collaborated on this study. Research team weekly meetings allowed the discussion of emerging data, as well as the identifying of specific research topics and areas for further data collection and analysis.

The information provided in this part is only based on the author’s analysis of his field notes. This part also describes the emerging themes and patterns in the research. I discuss several explanations for the developers’ inconsistent narratives, as well as their reactions to the security concerns, based on these theme. In addition, the dissertation’s author is referred to as a “researcher” throughout this part.

8.2 Literature Review

Assal and Chiasson [59, 60] utilized interviews and surveys to explore the interplay between developers and software security processes. Their research found that developers were motivated to develop secure code, but were often hindered by a mismanaged organizational process. The authors advocated looking beyond developers and examining broader organizational factors that may impact the security of the developed software. Our work is one
such attempt, utilizing an extensive ethnographic study in a software company. Many of our findings confirmed the analysis results from Assal and Chiasson’s work. Our work also revealed some deeper insights into the reason of software (in)security, as well as a co-creation model that can help address them.

Ruef et al. [61] and Votipka, et al. [62] conducted a series of studies based on data collected from the Build It, Break It, Fix It (BIBIFI) contests. A number of patterns of developer mistakes leading to vulnerabilities were analyzed. Our work examined the software development process in a real company. Our in-depth ethnographic study is complementary to the analysis based on large-scale competition data. One possible cross-over between the two types of studies is that one can use the insights from one to drive the analysis in the other. For example, an observed real-world phenomenon that has significant security impact could be replicated in the BIBIFI contest to further examine a hypothesis on a much larger and more diverse population.

Oorschot and Wurster [63] posited that developers have different skills which often do not include security and suggest that the focus should be on those who design APIs, because it is unrealistic to expect all developers be taught sufficient security. We raise a similar question in our paper from our ethnographic data, regarding how much security knowledge developers can realistically master, and whether a co-creation model where security experts and developers closely collaborate would be a more effective approach.

Green and Smith [64] discussed that developers are not the problem for insecure code. The focus should be on creating more developer-friendly and developer-centric approaches and supporting them when they are dealing with the security tasks. Our ethnographic data supports this conclusion. Moreover, our fieldwork resulted in a co-creation model that could be part of a solution to provide the needed support to developers for writing more secure code.

In addition to the works mentioned above, the research community has explored this area through a number of angles. Oliveira et al. [65] conducted surveys to understand de-
velopers’ attitudes toward security which leads to understanding that APIs and tools can be improved significantly. Votipka et al. [66] performed semi-structured interviews to compare how hackers and testers find vulnerabilities. Acar et al. [67] studied whether different documentation resources influence the security of programmers’ code. Krombholz et al. [80] performed a qualitative study with 30 users (18 end-users and 12 administrators) on user mental models of HTTPS. Stransky et al. [81] designed an online platform to conduct online secure-programming studies with remote developer participants. Naiakshina et al. [82] conducted a qualitative study with 20 computer science students and investigated how and why they failed with regards to secure password storage. Gorski et al. [83] designed a controlled online experiment with 53 participants to study the effectiveness of API-integrated security advice. Their study showed that 73% of the participants who received the security advice fixed their insecure code. Acar et al. [68] conducted an online study and evaluated five cryptographic APIs with GitHub Python developers about the usability of the crypto APIs. In this study, they reported the simpler interfaces is not good enough and those crypto libraries should also offer a broad range of common tasks support and provide accessible documentation with secure, easy-to-use code example. There has also been research that studied and characterized different aspects of software bugs [84, 85, 86]. These studies focused on the quality of bug reports and found that important information was often missing in bug reports which made it harder to reproduce and fix them.

Going beyond secure software development, research into other aspects of usable security has also revealed the importance of incorporating broader stake holders’ perspectives in thinking about security solutions [87, 88]. Haney et al. studied the role of cybersecurity advocates within organizations [89, 90, 91, 92]. Much of the findings in that line of research echoes ours, in particular the importance of co-creating security solutions with relevant stake holders.
Chapter 9: Research Methods and Context

9.1 Methodology

A participant observation method was primarily used in this study [78, 79]. Anthropologists have developed this method as a way to study human behavior and cultures through participating in daily activities and observing people’s behaviors over a long period of time (usually more than a year). In addition to providing insights into the subjects’ activities, knowledge, and habits, these activities enable researchers to gain a solid understanding of a particular culture. Through adapting this approach to the context of working within a software company, we can examine in depth the complexity of the software development process, the various incentive structures among stakeholders that influence human behavior, and the tight coupling between technical and human factors that affect software security.

In this research, the participant observers were two computer science PhD students, each of whom underwent systematic training in qualitative research method under the guidance of the anthropologist on our research team. Being CS students and possessing a substantial amount of security knowledge enabled them to get quickly immersed into the company’s software development process and start observing practices that might have an impact on the software products’ security. Being inside the company enabled them to observe both contemporary events as they unfolded, as well as past events studied through ticketing systems and checking the relevant code in the repositories. The students’ role in the company – working as if they were an employee of the company – helped with two important assets of our research. First, their daily interactions with the developers while doing regular on-the-job tasks provided a unique angle to observe the subjects’ authentic behaviors as they performed their job duties. Second, they not only acted as passive observers but as advocates
of software security inside the company. This approach enabled the team to observe how the various stakeholders reacted to discoveries of security vulnerabilities, providing valuable insights into why those vulnerabilities were introduced in the first place and the constraints under which they could be fixed (or not).

Each researcher worked at the company 20 hours a week, spread across three week-days. The author of this dissertation worked there for six months, while the other researcher worked there for a year. The researchers were not paid directly by the company. However, the company provided both financial and in-kind contributions to this research. In general, the researchers' tasks included debugging existing implementations to find bugs' root causes, writing code fixes or implementing new features, performing code reviews, and software quality assurance. The researchers took field notes about their observations, including both security issues found in the software and everyday interactions with developers and other employees involved in the development process. Notes had two forms: descriptive and insightful. Descriptive notes were intended to be as informative as possible, avoiding personal judgments or opinions. Insightful notes aimed to capture “ah-ha” moments and provide reflective analysis of the situations experienced by the observers.

9.2 Data Analysis

To derive research insights from the raw notes, we applied the general inductive approach [93], augmented by specific techniques for qualitative data analysis [94]. The initial step was to find patterns that emerged directly from the data themselves. In this research, the process happened via weekly meetings of the larger research team including both the fieldworkers and the professors, where comparisons could be made across researchers, discussions could address both the human and technical dimensions of software development in a company, and plans made for further exploration of interesting topics. Identifying themes and links between ideas proved central to the inductive analysis, as well as developing contextual analysis around key examples. Data analysis continued through the coding of field
notes based on identified themes. These codes included themes related to software security, human elements of the work, important explanatory concepts that emerged during the research, and data linked to the key examples. A more detailed description of the coding process as well as the “codebook” can be found in the Appendix D. Research meetings then shifted to further developing our joint understanding of the data and identifying ways to explain the observed patterns, as well as potential solutions to how human and technical factors combined to shape (in)security.

It is important to highlight two unique aspects of our participant observation approach. First, participant observation is often a solo affair in the social sciences; having two embedded researchers permitted the examination of the company from two different but complementary perspectives. The researchers were assigned different tasks, had slightly different hours at the company, and developed relationships with company personnel at different points of time. This dual approach to participant observation increases the robustness and validity of the data from this research. Second, the research team consisted of experts in engineering and social science. This multidisciplinary team participated with the embedded researchers in developing the analysis over months, permitting the identification of themes and ideas that crosscut disciplines and had both theoretical and applied dimensions. This team-based approach to both data collection and analysis is a significant contribution to how this type of research can be done effectively.

9.3  Context

9.3.1 The Company and Its Products

At the company, the researchers worked in the same space as four other developers, four support engineers, two network engineers, one customer-facing on-boarding specialist, the CTO, a marketing and sales manager, and other staff. The researchers’ work focused on two products: a solution for controlling network access and a solution for allowing users to securely access networks remotely. The solutions configured third-party network devices
(e.g., routers and access-points), enforced operator-defined access-control policies, and managed remediation flows. Typical customers were medium- and large-size organizations, and common users were IT staff who managed the organizations’ networks. Organization end users attempting to connect to its network were prompted first by a captive portal that asked for credentials. Once authenticated, they were asked to remediate any issues that prevented them from complying with policy, e.g., they might be required to download and run a client-side monitoring agent and update their anti-virus software.

9.3.2 Development Process

The company followed general agile development principles. The development team held a scrum meeting every morning that lasted 15-30 minutes. In this meeting, each developer briefly commented about any progress accomplished or roadblocks encountered the day before and discussed the plan-of-work for the current day. This was an opportunity for developers and managers to give and receive feedback from each other. The meeting was led by the dev team lead. The CTO was usually in the room but did not lead the meeting.

Work was organized, prioritized, assigned, and tracked using ticketing and code management systems. In general, tickets were generated by developers, support techs, or customer-facing specialists, ranked in prioritization meetings held by the dev team lead and CTO, and assigned and tracked by the dev team lead. After implementation, tasks were moved into the peer-review stage in which other developers (often more experienced ones) reviewed any code changes, added pending tasks if necessary, and finally approved merge requests. After code changes were approved by all reviewers, tickets were reassigned for quality assurance and integration testing, which was often done by both developers and support/customer-facing specialists. When all tests had been passed, tickets were marked as “done” and merged into the code repository’s development branch. When the set of target features for a release had been implemented, the team lead created a release candidate branch. Every release can-
date was tested in-house one last time before being finally moved into release and installed on customer environments.

9.3.3 Study Participants

The main participants in the study were the four software engineers on the development team where the student researchers were embedded. The dev team lead was an experienced developer who had been at the company long-term and written many parts of the system. Two of the other developers had been with the company for several years and another had recently joined. One developer specialized in front-end development and two were full-stack developers. The researchers also interacted with other personnel at the company, including the CTO, via company meetings, work communications, and everyday activities such as breaks and lunches where people often “talked shop” in informal ways.

9.3.4 Research Ethics

In our research, the employees of the company (developers, support techs, and managers) were considered human subjects. The study was reviewed and approved by the Institutional Review Board (IRB). Researchers explained the study goals to participants and obtained verbal informed consent from participants. Field notes were anonymized, as well as discussions during weekly research meetings. This part of the dissertation follows that same anonymization approach. Throughout this part, I use the term application under study to refer to a specific application in the company’s product suite. In addition, I anonymized all product-specific terms in the following chapters as well.

One ethical dilemma that emerged during the research was what to do when security vulnerabilities were discovered. Given ethical standards among cybersecurity professionals, we made the decision to present these discoveries to the software development team. This process proved crucial to the further development of the research. Rather than simply observing what happened while continuing to work at the company, the researchers raised
these security concerns, and where directed, actively worked on addressing them. This active engagement led the research team to a co-creation model, where research, programming, and security were all ongoing parts of what happened during the fieldwork.
Chapter 10: Live Discovery through Pen-testing during Ethnography

This chapter describes how the author combined vulnerability discovery with participant observation of developers’ behaviors and reactions. Unlike the other part of this research (chapter 4 [58]), none of the pen-testing discovered vulnerabilities were intentionally introduced by developers, and as such the author had the opportunity to observe the unfolding of developers’ reactions with the discovery of a totally unknown problem. It also provided the opportunity for the author to intervene in a way that resulted in a co-creation model, in which security experts work jointly with developers to improve code security issues and prevent them from happening in future code.

10.1 Penetration Testing Method Adopted for This Study

Penetration testing, also known as pen-testing or ethical hacking, is an authorized simulated cyber-attack process against a computer system to reveal security flaws. The goal is to identify weaknesses which might provide a passage for unauthorized users to gain access and alter the integrity of the computer system. Pen-testing can be categorized as:

1. Black-box penetration testing: Auditors in a black-box test take on the role of an average hacker and have no intimate knowledge of the target system. Neither architecture diagram nor source code is provided to testers that is not publicly accessible. A black-box penetration test identifies the vulnerabilities in a system that can be exploited from outside the network.

---

10Some part of this chapter was published in proceedings of the Sixteenth Symposium on Usable Privacy and Security. Permission is included in Appendix A. [58]
2. White-box penetration testing: White-box testing is referred to by several names, including clear-box testing, open-box testing, auxiliary testing and logic-driven testing. It is the exact opposite of black-box testing: auditors are given access to the source code, architecture documents, etc. White-box testing is the most time consuming type of penetration testing due to the difficulty of sifting through the massive amount of data available to identify potential weaknesses.

3. Gray-box penetration testing: is in between the two previous categories. Grey-box testers look at a system from the point of view of a user who may have more access to the system than a black-box auditors does. In gray-box pen-testing, the auditor is typically familiar with a network’s internals; they may have access to the network’s design and architecture documents.

There are multiple software pen-testing methodologies, including the Open Web Application Security Project (OWASP) [95], Open Source Security Testing Methodology Manual (OSSTMM) [96], NIST SP 800-115 [97], Penetration Testing Execution Standard (PTES) [98], and Information System Security Assessment Framework (ISSAF) [99]. Results vary based on the way the process is performed.

Two of the company’s products were picked by the author for further study. By acquiring some information and insights about the applications, the researcher started to apply a customized penetration testing methodology. The basic information such as the software workflow, authentication information, and so on was captured by talking with developers and the support team. Both products were designed to work on a web platform, so OWASP’s top ten security vulnerabilities [95] were chosen for the testing. At the same time as developing the pen-testing process, the researcher worked to gain the developers’ trust. This process required building rapport, an understanding with research informants, by participating in daily tasks and getting to know individuals who worked there. The role of a security pen-tester also needed accurate planning and time management. “Code injection” was selected for the first vulnerability to be tested. It is one of the well-known vulnerabilities which
allows attackers to inject malicious codes into a computer system and change the course of execution. The result of a successful injection can potentially be catastrophic. Before we describe the pen-testing findings and developers’ reactions, we first briefly introduce the three types of vulnerabilities found.

10.2 Exploring Software Vulnerabilities Found

1. Cross-site Scripting (XSS): generally found on web platforms. Attackers typically use web applications to inject malicious codes into the application which can be viewed by other users. There are three types of XSS: stored or persistent XSS, reflected XSS, and DOM-based XSS. In stored XSS malicious code is stored permanently on the server side of the web application. Reflected XSS is typically delivered to victims from other routes such as e-mails and bounced back by the vulnerable web application. In DOM-Based XSS or type-0 XSS attackers can modify the Document Object Model (DOM) environment and inject malicious code.

2. HTML Injection: similar to XSS. However, instead of inserting malicious scripts, the attacker can inject valid HTML tags and modify the content of the target website. These vulnerabilities are also categorized as stored HTML injection and reflected HTML injection. The main difference between these two types is that the stored HTML injection is permanently stored inside the server side and executed every time a user accesses the vulnerable page, whereas the reflected HTML injection payload must be delivered to each victim separately (typically delivered via another route, such as email or malicious links on another website) and it is not permanently stored on the server.

3. Shellcode Injection: also known as shellcode upload, a type of web vulnerability that allows an attacker to inject malicious code into a system and provide the attacker a shell on the system. This vulnerability lets an attacker take full control of the server and technically works as a backdoor on that server.
4. Remote Code Execution: is a category of software security flaws/vulnerabilities. Remote code execution (RCE) vulnerabilities will facilitate the execution of any code of a malicious actor’s choice on a remote machine via LAN, WAN, or internet. RCE falls under the broader category of arbitrary code execution (ACE) vulnerabilities.

10.3 Behaviors and Reactions from Developers

In the first day of pen-testing, I discovered an stored-XSS vulnerability in the application being studied. The vulnerability was brought up to the developer team, and a proof of concept was provided for why it was significant. While they showed interest in the finding, since the vulnerability was in a 3rd-party application integrated with the company software, their first reaction was to hope the problem had been fixed by the 3rd-party. One participant said:

“This vulnerability belongs to our 3rd-party application, and we did not develop this part. It is better to upgrade the software and see if we will still have the issue”

They also mentioned that it would be more interesting if I could find any vulnerability inside part of the company’s code. Thus, they expressed interest in security, but saw solving this problem as the responsibility of an outside group even though the application formed part of the company’s software.

In the next round of testing, I tried other parts of the software to see if there were any other vulnerabilities. Multiple XSS vulnerabilities were found. Developers were both excited and concerned about the findings. They said things like,

“If they want to test more, it seems that they will find more things inside our software”

and
“We tried to minimize our bugs, but it seems something is wrong.”

Once again, the third-party issue came up:

“We are using Angular, and I thought we shouldn’t have the XSS. Angular should take care of this issue.”

A Remote Code Execution vulnerability was also found within the application. Attackers were able to execute arbitrary code remotely on the same third-party application. Only knowing the address of the platform is enough for attackers to execute their code. I provided a proof-of-concept to the team. Due to the fact that this vulnerability found in the same 3rd-party application, one of the developer said again:

“…again this is on the X application. I hated to use this software inside our application. We should create our own module for the Y task. Right now, just try to upgrade it.”

On the same day, I found another vulnerability in the same application; this time it was shellcode injection. The vulnerability allowed attackers to inject their customized shellcode into a valid file and upload it into the server and get backdoor access with a powerful user’s account on the server. The attacker must be someone who already had a regular account inside the application web platform. The finding was interesting to the researcher team and the developers for different reasons. For the research team, this was a critical vulnerability – customers should never have escalated access to the server. They should only be able to perform some limited commands on the OS such as changing the network IP address. This essentially allows their customers to jail-break out of the sandbox set up on the server. Developers on the other hand were more interested in understanding how access had been gained. They said things like,

“Interesting! Could you show us how you got the access?”
“What is your user’s privilege?”

At the same time, they discussed the risk in the context of the product,

“because we already ship the OS to the clients with everything inside it, it’s kind of okay! They have the box already. ...We do not have any important information on it.”

During the subsequent discussion with the developers, I asked, “Do you have any hard-coded password or credentials?” The answer was “yes.” This hard-coded credential would allow a customer who successfully exploited the shellcode injection vulnerability to see other customers’ information. Faced with this fact, the developers indicated that this vulnerability was bigger than they originally thought, and that they should take imminent action on it. However, that did not come to fruition at the next group meeting, where they continued to talk about these vulnerabilities. Based on this discussion, the research team inferred that there were other factors that affected the developers’ actions. In fact they said

“fixing the vulnerability has additional impacts and may cause some problems for other parts of the application or customers.”

They also downplayed the significance of the shellcode injection vulnerability, and said that the team should focus on developing new features. They added,

“If we want to fix every bug in our system, we will be out of business very soon.”

Contrary to the researcher team’s initial hope, our intervention effort to fix the discovered security vulnerabilities proved ineffective in the context of the company’s overall functioning. This moment helped me realize that I needed to rethink how security researchers engage developers to create positive change.

It started by simply offering to work on the issue and building the tools and libraries for them to prevent XSS. The bug was then fixed by me. One challenge I faced was that
the application was uniquely designed and would only accept specific types of input entry for the various fields. As a result, it was not possible for me to utilize standard input sanitization solutions, e.g., one that removes all special characters, because that would break the application. Working at the company and interacting with the developers helped me to understand this uniqueness and come up with a customized solution. It was a number of specially designed regular expressions that enforce the proper formats for the various types of fields. I included the application of these regular expressions in a standalone Javascript file that can be invoked at the front-end pages. It turned out that the company’s existing code already contained a similar mechanism for checking other properties of front-end input fields, e.g., if a field is empty. I only needed to extend this mechanism to include the regular expression checks I designed for preventing XSS. Developers could then simply invoke these checks in the same manner they had been doing for the other types of checks. This allowed for easy integration of the security check into existing code with minimum change, and was readily accepted by the development team. For back-end input sanitization, I first tried to apply standard OWASP sanitization functions for Java, which was the language the back-end was written in. However, due to the uniqueness of the formatting requirement of the application, those standard checks were blocking some legitimate inputs. Thus I needed to customize those OWASP functions to work properly with the application fields’ requirements. During the fixing process, another XSS was found in the application. When the researcher brought up the problem to the development team this time, they accepted it very fast. A new ticket was created and the researcher was asked to fix the issue as soon as possible: “...Go ahead and fix this bug as well.”

This example highlights the importance of “being there” for security experts to drive positive change for secure coding. The researcher was able to accomplish this in this case due to two factors: 1) he understood the company’s existing code and designed an effective security check that minimized disruption; 2) he provided the needed security expertise in designing the proper checks using regular expressions and the customization of the OWASP
functions, and this expertise was delivered through code artifacts that were readily applicable within the existing software workflow. Both factors were important for this success.

I tried later to bring up the shellcode injection vulnerability once more in a discussion and tried to convince them to start fixing the issue, but the suggestion was turned down. One of the developers responded,

“It’s somewhere in our backlog. We didn’t do anything about it, and no one has found that exploit so far. So we are safe.”

This comment matched similar instances where developers reacted as though if there were no problems at present, the vulnerabilities might not be an issue that needed urgent attention. The research team considered a possible explanation why the shellcode injection vulnerability was not treated as urgently as the XSS. Exploiting the shellcode injection vulnerability would require a rogue player that can be held accountable (a customer’s IT staff member who possessed the regular account access to the server). This may have alleviated the concern on the company’s liability resulting from this vulnerability.

Later on in the research, an HTML injection was found inside a newly developed part of the code. Like the XSS, the issue was brought up to the developers. Initially, developers mentioned,

“Angular should cover it and not allow the HTML tag in the code! It seems it does not.”

During the next group meeting, they recognized that they had omitted security issues previously. They said,

“When we discussed the development of the page, we talked about everything except security and XSS problems. They didn’t come to our mind.”

At this time, the researcher thought that the developers would ask him to fix this issue like in the XSS case, but they started to fix it by themselves and did not ask the researcher
for any help. Most interestingly, the developers created correct solutions to fix the HTML injection vulnerabilities, based on the way the researcher solved the XSS problem. This showed that the developers learned from the researcher how to create security fixes within their code base, without being explicitly taught so. They learned by simply observing the code artifacts created by our researcher.

This is an example that illustrated the importance for security experts to be in the development environment and “co-create” security solutions with the developers. The difference in the developers’ reactions in this case, compared to earlier ones, pushed the researchers to realize that co-creation happens more in the moment, rather than trying to retroactively fix things. Our earlier interventions mainly focused on fixing vulnerabilities found in code written in the past. We had success in getting some fixed (by the researcher). Whereas in this case, the developers took their own actions and fixed the bugs using the knowledge and tools provided to them by the researcher. This shows that if security professionals are present and part of the team when a product is in the process of being designed and implemented, their views are more likely to be taken into account when decisions about what to do are being made. It was also the researcher’s feeling that the quickness with which the development team accepted his suggestion to fix this issue was related to the increased level of trust he enjoyed from the development team at this point in the research progress.
Chapter 11: Analyzing the Findings

After initially finding the first bug in the pen-testing process, the researcher assumed that developers did not know about these security problems, and lack of security knowledge led them to write code with the vulnerabilities. After working with them on various tasks, he realized that they actually possessed quite a bit of security knowledge. As our research progressed, group discussions and analysis of field notes highlighted some non-intuitive reasons for developers’ behaviors. This indicates that there were other significant factors in causing these vulnerabilities. I outline these factors in the rest of this section.

11.1 Developers Should Not Totally Trust Programming Languages

One of the important conversations that the researcher had with the developers was that the developers believed that the programming language/framework should take care of some vulnerabilities by default.

“...Angular should take care of this vulnerability...”

In this case and according to Angular documents [100], the Angular engine could handle most of the XSS and HTML injection attack scenarios by sanitizing the input fields. Angular documents also mentioned that developers need to take care of backend servers to make sure injection vulnerabilities are not introduced there. After analyzing our field notes carefully, we found that the developers believed (incorrectly) that Angular could handle all XSS and HTML injection vulnerabilities.

In the past decade, programming languages and frameworks have been doing a great job in creating built-in security measures to prevent accidental mistakes by developers, but they
still do not offer a comprehensive security solution. From the developers’ point of view, it is
clear that these languages/frameworks do facilitate programming. This poses some questions
to the security and programming language community:

- How can developers know accurately where they can rely on language/framework and
  where they must rely on their own code to achieve a security property?
- Can this be communicated in a way that does not require sophisticated knowledge on
  all possible ways attack could happen?

11.2 Outsider vs. Insider

In the past, I worked as a pen-tester for four security consulting companies in three
countries for four years. In my experience, the pen-testers were not incorporated into the
development team. The developers might only receive a document with discovered vulner-
abilities and statements about what they should or should not do. This appeared to be a
common industry approach to software security pen-testing [101]. The problem was that
the security pen-testers did not understand how much workload the developers had, nor
the actual reasons for the vulnerabilities. As a result, this approach did not often lead to
the desired changes in the development process, but set up an outsider/insider dynamic,
where developers felt the need to defend what they had done and/or minimize the security
issues. The developers would say that the report came from an outside group who did not
really understand how software development was done, and the security pen-testers would
say that the developers wrote defective code in the first place and did not appreciate se-
curity, otherwise they would have done something to fix all those problems. Having these
past experiences, in contrast with what I experienced in this research where I worked inside
the development team as a software pen-tester, helped the research team to understand the
impact the outsider/insider dynamic had on effectuating changes in software development
processes.
From our fieldwork experience, we clearly see how this outsider/insider dynamic can play out. When we first found the vulnerabilities about XSS, or code injections, the researcher’s initial thought was that the developers did not know about these security issues. However, after researcher explained to developers and developers had clearly understood the technical details, still some vulnerabilities were not fixed. It was only after further communication with the developers, reflecting on other relevant observations made by the researcher, and brainstorming among the larger research team, that we better recognized why some vulnerabilities were not prioritized to be fixed. Most importantly, it is when we had this understanding, and produced an easy-to-apply solution that fit into the company’s development workflow, that our intervention was the most successful.

The point of view that if developers know better and work harder, they should be able to write software without any security flaws, can be characterized as the so-called “deficit model,” where the problem of software insecurity is attributed to the developers’ lack of knowledge or efforts. The solution driven by this deficit model would mainly involve experts explaining to developers the various software security issues and how to prevent them, and hoping this would drive the needed changes. Research in fields such as education, anthropology, and science communication have examined how using such a deficit model does not prove as useful as imagined because it localizes the problem inside the person and assumes that simply fixing that internal lack will also successfully address larger concerns such as successful learning, cross-cultural understanding, and the application of science to at times controversial topics [102, 103, 104]. One analytic concept of note that emerged through the research was our own use of a “deficit model” to initially interpret why people in the company did not respond to security concerns. We assumed that they might not have the knowledge or awareness to understand security risks and recognize how and why particular aspects of the software might increase those risks. Our research found that this deficit model-driven approach was not working well. Simply communicating security issues found and presenting solutions for fixing them did not lead to the anticipated fixes.
Overcoming this “deficit model” in our own thinking helped us to better interpret why participants responded or not to security issues and to recognize how security concerns existed alongside other factors that shaped their work. We then developed a co-creation model, where developers and security experts collaborate together. Co-creation is a form of collaboration in which ideas and processes are shared and improved together rather than kept to only one-party side. By having a co-creation model, security auditors have the chance to jump into the development process and provide the knowledge and tools that developers can readily apply to prevent vulnerabilities. Part of this co-creation model meant that the researcher did not work exclusively on security but dealt with different tickets. This showed the developers that the researcher knew how to program, and could do so as part of a team, while also having expertise in security that he could draw on if needed.

It appeared to us that developers prefer to trust a person inside their team rather than an outsider. Moreover, our field notes showed that a security person inside the developer team can provide more in-depth knowledge than outside resources such as pen-testing reports, internet, and so on. For example, after the XSS got fixed on the application under study, when developers faced the HTML injection they said:

“Is this HTML Injection going to be easy to fix? It should be very easy to fix...”

and without asking the researcher to provide the solution for them, they fixed the issue.

11.3 Thinking as an Attacker, Thinking as a Developer

It has almost become a platitude in the security field that one must “think as an attacker.” This idea can be traced back to the well-known statement attributed to the ancient Chinese military expert Sun Tzu:

“If you know the enemy and know yourself, you need not fear the result of a hundred battles.”
Applying this to software development, the developers can put themselves in the attackers’ shoes and understand how software may be misused. It is an interesting question as to how much developers need to think as an attacker. These days, understanding the mechanisms of all types of cyber attacks can be overwhelming even to a security expert. Our data implied that the developers and the company were aware of some of those threats that they may face, but just knowing them was not enough. The problem is not necessarily about the lack of understanding the attackers; it is more about not being able to implement security features correctly into software, which unfortunately requires some non-trivial amount of security knowledge. Is it realistic to expect all software developers to become security experts? How much time should developers spend on thinking about how their code may be attacked, among all the other competing demands they face? Security professionals can help bridge this gap by starting to think like developers, just like how we ask developers to think like attackers. Security professionals need to better understand how developers have to negotiate many competing interests, not just a sole focus on security. This could help in providing security knowledge and information at the right level of abstraction that can be easily integrated into the software development process. The co-creation model we used as part of this research allowed the security researchers to think like a developer, and to create some positive impacts in the software development process.
Chapter 12: Conclusion and Future Work

This part of this dissertation, shows how security intersects with software development on the ground, based on a embedded researcher with six months’ data. Our study has revealed two factors that play a part in determining how much security advice can be taken: “Security considerations need to come early in the development life cycle”. In the case of code injection, we didn’t get to be there when the features were developed. Because of this, we could not successfully intervene. We believe that if we were there at the time, we might have been able to fix this bug. Just as the other vulnerability I presented in section 10.3, with which developers have successfully fixed the HTML Injection after I introduced it to them without asking me to fix it. Moreover, we highlighted that “Security experts can be more effective when they work within a development team.” As I mentioned before, as a penetration tester, I have worked in multiple companies and from my experience, the security people are not incorporated into the development process. However, in this study when I stepped into the role of a security worker in the team, we realized that we could be a lot more helpful if we intervened as team members. This conclusion is confirmed by the success of interventions with XSS injections vulnerability we had in our research. There remains a considerable gap between security and developers. Our research shows that security professionals can better bridge the gap by understanding how (in)security emerges from the interacting technological and human factors in the development process. Our ethnographic study provided a way to understand this complicated phenomenon, both by better understanding the competing demands under which developers work, and by demonstrating how security can successfully be integrated into software development through a co-creation model.
References


Appendix A: Copyright Permissions

Copyright permission for content used in Chapters 2, 3, 4, 5 and 6 can be seen below. The publishers state that authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.
Copyright permission for content used in Chapters 8, 9, 10, 11 and 12 can be seen below. The publishers state that the authors own the copyright of the content published in the paper.

An Ethnographic Understanding of Software (In)Security and a Co-Creation Model to Improve Secure Software Development

Hernan Palombo*  Armin Ziaie Tabari*  Daniel Lende  Jay Ligatti  Xinming Ou
University of South Florida, Tampa, FL, USA
Email: [hpalombo, aziaietabari, dlende, ligatti, xou] @usf.edu

Abstract
We present an ethnographic study of secure software development processes in a software company using the anthropological research method of participant observation. Two PhD students in computer science trained in qualitative methods were embedded in a software company for 1.5 years of total research time. The researchers participated in everyday work activities such as coding and meetings, and observed software (in)security phenomena both through investigating historical data (code repositories and ticketing system records), and through pen-testing the developed software and observing developers’ and management’s reactions to the discovered vulnerabilities. Our study found that 1) security vulnerabilities are sometimes intentionally introduced and/or overlooked due to the difficulty in managing the various stakeholders’ responsibilities in an economic ecosystem, and cannot be simply blamed on developers’ lack of knowledge or skills; 2) accidental vulnerabilities discovered in the pen-testing process produce different reactions in the development team, often times contrary to what a security researcher would predict. These findings highlight the nuanced nature of the root causes of software vulnerabilities and indicate the need to take into account a significant amount of contextual information to understand how and why software vulnerabilities emerge during software development. Rather than simply addressing deficits in developer knowledge or practice, this research sheds light on at times forgotten human factors that significantly impact the security of software developed by actual companies. Our analysis also shows that improving software security in the development process can benefit from a co-creation model, where security experts work side by side with software developers to better identify security concerns and provide tools that are readily applicable within the specific context of the software development workflow.

1 Introduction
It has long been recognized that human factors play a dominant role in ever-present software vulnerabilities, with substantial research devoted to this area [1–10]. These past efforts have used a variety of research methods including surveys, interviews, controlled experiments, studying code artifacts, and analyzing data collected from secure-coding competitions. It is also understood that there is a fundamental economic problem underlying software insecurity [11], and in general there often appears to be an unwillingness in industry to give code security equal importance as other business considerations, such as time to market and richness of features. It is therefore important to recognize that the (in)security of software produced by software companies is impacted not only by individual developers’ knowledge and skills and the types of programming languages/environment they use, but also by the various incentives at play both in the market and at the organizational level. Thus, to produce real impact in secure software development, it is indispensable to study this problem in the context of where the process happens, i.e., in the software companies.

Recent work by Sundaramurthy et al. [12, 13] showed that by employing the anthropological research method of participant observation [14, 15], researchers successfully obtained deep insights into the challenges faced by security analysts in security operations centers (SOCs). Moreover, embeddings in the SOCs allowed researchers to produce both technical and non-technical interventions that improved SOC operations by uncovering and addressing the pain points in the overall work process and environment. Encouraged by the success in that work, we conducted an extensive ethnographic study in a software company, using the same method of participant observa-

Copyright is held by the author/owner. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee.
August 9-11, 2020, Virtual Conference.
Appendix B: Institutional Review Boards (IRB) Approval

This research involves study of human subjects and the researchers have obtained approval from IRB to conduct this research. The documentation for the approval is shown below.

![IRB Approval](image)

On 6/30/2021, the IRB reviewed and approved the following protocol:

<table>
<thead>
<tr>
<th>Application Type:</th>
<th>Continuing Review</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRB ID:</td>
<td>Pro0003617 CR000002</td>
</tr>
<tr>
<td>Review Type:</td>
<td>Expedited</td>
</tr>
<tr>
<td>Funding:</td>
<td>National Science Foundation</td>
</tr>
<tr>
<td>IND, IDE, or IDE:</td>
<td>None</td>
</tr>
<tr>
<td>Approved Protocol and Consent(s)/Assent(s):</td>
<td>• IRB Protocol Version 1 June 2018.docx; • Verbal Consent Form Version 2.pdf;</td>
</tr>
</tbody>
</table>

The IRB approved the protocol from 7/24/2021 to 7/24/2022. Within 45 days of 7/24/2022, submit a continuing review/study closure request in BullsIRB by clicking Create Modification/CR.

If continuing review approval is not granted before the expiration date of 7/24/2022, approval of this protocol expires on that date.

In conducting this protocol you are required to follow the requirements listed in the INVESTIGATOR MANUAL (IRP-103).

Sincerely,

Oria Larsen  
IRB Manager

Figure B.1: IRB Approval
Appendix C: Clusters Identified from Honeypot Logs

We will take a look at the commands found inside each cluster in this appendix.
Appendix D: Coding

Coding – including the development of specific codes and of a codebook – proceeded in an iterative fashion. Weekly meetings facilitated the discussion of emerging research themes and specific examples. Codes emerged from these discussions, where the researchers built consensus on specific analyses. The researchers relied on a general inductive approach [93], as well as techniques derived from grounded theory and related approaches for doing qualitative analysis [94]. Overall, the development of codes initially focused on the Silently Allow example (The researcher 1), then on emerging results from penetration testing (the author of this dissertation), the development of a specific coding system for each researcher for their field notes, and a final collaborative phase to find commonalities in codes for both the specific examples and overall corpus of field notes.

D.1 Full Set of Codes

Subsequent research focused on developing a full set of codes by each of the researchers. Because the two embedded researchers often worked on different projects and at different times, each wrote their own field notes and then subsequently engaged in coding of their own notes. This process permitted inductive analysis from their own data, which could then be shared in research meetings to produce consensus. Listed below are the sets of codes developed by the author of this dissertation:

- Caring-about-subject
- Changing-attitude
- Development-process
• Documents-not-updating-frequently
• Fix-first-update-later
• Joking-about-intern-work
• Knowing-bug-do-nothing
• Lack-of-knowledge
• Learning-process-with-company
• Looking-for-new-idea
• New-idea-vs-tasks
• Not-caring-about-subject
• Not-trusting-other-developer-or-intern
• Performance-reaction
• Protective-about-subject
• Say-something-do-something-else
• Security-vs-performance
• Security-vulnerabilities-blocked
• Security-vulnerabilities-concern
• Security-vulnerabilities-denying
• Security-vulnerabilities-execution
• Security-vulnerabilities-fixing
• Security-vulnerabilities-interested
• Security-vulnerabilities-process
• Security-vulnerabilities-reaction
• Security-vulnerabilities-thinking
• Security-vulnerabilities-upgrading

D.2 Final Collaborative Phase

Researchers often worked on a whiteboard during the final collaborative phase of a project to find the overlap between different types of data, specific examples, and inductive conclusions. For example, when working through the data to find common themes and observations in both field notes and in the researchers’ experience during participant observation, the idea of “co-creation” emerged as an overarching conclusion. Upon reviewing the similarities, via the coding and then the notes, we were also able to make data-driven conclusions about what was successful and what appeared to be bottlenecks or limitations throughout the months of embedded research.