2019-09

# BLOCKCHAIN FOR USE IN COLLABORATIVE INTRUSION DETECTION SYSTEMS

Kanth, Vikram K.

Monterey, CA; Naval Postgraduate School

http://hdl.handle.net/10945/63465

NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

# BLOCKCHAIN FOR USE IN COLLABORATIVE INTRUSION DETECTION SYSTEMS

by

Vikram K. Kanth

September 2019

| | |
|---|---|
| Thesis Advisor: | Murali Tummala |
| Co-Advisor: | John C. McEachen |

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

| | | |
|---|---|---|
| **1. AGENCY USE ONLY** ***(Leave blank)*** | **2. REPORT DATE** September 2019 | **3. REPORT TYPE AND DATES COVERED** Master's thesis |

| | |
|---|---|
| **4. TITLE AND SUBTITLE** BLOCKCHAIN FOR USE IN COLLABORATIVE INTRUSION DETECTION SYSTEMS | **5. FUNDING NUMBERS** |
| **6. AUTHOR(S)** Vikram K. Kanth | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Naval Postgraduate School Monterey, CA 93943-5000 | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)** N/A | **10. SPONSORING / MONITORING AGENCY REPORT NUMBER** |

**11. SUPPLEMENTARY NOTES** The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

| | |
|---|---|
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT** Approved for public release. Distribution is unlimited. | **12b. DISTRIBUTION CODE** A |

**13. ABSTRACT (maximum 200 words)**

As the threat of cyber attack grows ever larger, new approaches to security are required. While there are several different types of intrusion detection systems (IDS), collaborative IDS (CIDS) offers particular promise in identifying distributed, coordinated attacks that might otherwise elude detection. Even for this type of IDS, there are unresolved issues associated with trusting participants and aggregating data. Blockchain technology appears capable of addressing those issues. This thesis is focused on presenting a proof-of-concept experiment leveraging an Ethereum-based private blockchain for a CIDS that uses pluggable authentication modules (PAM) to track login activity toward detection of doorknob rattling attacks.

| | |
|---|---|
| **14. SUBJECT TERMS** blockchain, hyperledger, secure communications | **15. NUMBER OF PAGES** 97 |
| | **16. PRICE CODE** |

| | | | |
|---|---|---|---|
| **17. SECURITY CLASSIFICATION OF REPORT** Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified | **20. LIMITATION OF ABSTRACT** UU |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

# BLOCKCHAIN FOR USE IN COLLABORATIVE INTRUSION DETECTION SYSTEMS

Vikram K. Kanth
Lieutenant Junior Grade, United States Navy
BS, U.S. Naval Academy, 2015

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN ELECTRICAL ENGINEERING**

from the

**NAVAL POSTGRADUATE SCHOOL**
**September 2019**

Approved by: Murali Tummala
Advisor

John C. McEachen
Co-Advisor

Douglas J. Fouts
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

As the threat of cyber attack grows ever larger, new approaches to security are required. While there are several different types of intrusion detection systems (IDS), collaborative IDS (CIDS) offers particular promise in identifying distributed, coordinated attacks that might otherwise elude detection. Even for this type of IDS, there are unresolved issues associated with trusting participants and aggregating data. Blockchain technology appears capable of addressing those issues. This thesis is focused on presenting a proof-of-concept experiment leveraging an Ethereum-based private blockchain for a CIDS that uses pluggable authentication modules (PAM) to track login activity toward detection of doorknob rattling attacks.

THIS PAGE INTENTIONALLY LEFT BLANK

# Table of Contents

# List of Figures

# List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

# List of Acronyms and Abbreviations

| | |
|---|---|
| **API** | Application Programming Interface |
| **CIDS** | Collaborative Intrusion Detection System |
| **CLI** | Command Line Interface |
| **CPU** | Central Processing Unit |
| **GUI** | Graphical User Interface |
| **IDS** | Intrusion Detection System |
| **NPS** | Naval Postgraduate School |
| **OS** | Operating System |
| **PAM** | Pluggable Authentication Module |
| **P2P** | Peer-to-Peer |
| **SPoF** | Single Point of Failure |

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 1: Introduction

There is no doubt that cyber attacks are an ever-growing threat. Statistics collected by the U.S. Government Accountability Office (GAO) indicate that the number of reported attacks against federal agencies has steadily increased by an average of 8,000 attacks per year for more than a decade [1], the Department of Defense estimates that 36 million email attacks take place against defense infrastructures every day [2], and an estimated 10% of U.S. residents over the age of 16 were victims of cyber-perpetrated identity theft in 2016 alone [3]. A survey of this threat landscape reveals a clear need for more effective defensive tools.

One of the most commonly used defenses is intrusion detection [4]. The objective of intrusion detection systems (IDS) is to recognize anomalous behavior either within the network as a whole or within individual hosts. In the threat landscape of today, current intrusion detection techniques are not sufficient to address the wide variety of threats [5]. Collaborative intrusion detection is one technique that has potential to address some of these threats [5].

The research conducted for this thesis is a demonstration of a proof-of-concept collaborative intrusion detection system (CIDS) to enable anomaly detection in a networked environment. This research contributes directly to efforts to ensure network security.

Portions of this thesis were used in an upcoming paper submission for HICSS 2020.

## 1.1 Thesis Objective

The objective of this thesis is to develop a mechanism by which a collaborative intrusion detection system can be implemented. This implementation uses blockchain, an emerging technology, to provide the information-sharing framework for a CIDS.

A commercially available blockchain client, Ethereum, is used to provide the trust framework for a CIDS. This thesis develops a workflow that allows for the detection of anomalies using a blockchain-based CIDS. Furthermore, potential classes of attacks are attempted against and detected by this CIDS.

## 1.2 Related Work

The related work for this thesis can be subcategorized into a couple of categories: work related to IDS/CIDS and work related to blockchain and its applications. Seminal work combining the two will be discussed briefly.

Intrusion detection has long been present in literature as a way to counter forms of computer abuse. The concept of IDS is presented in great detail in [4] and [6]. Denning defines abnormal use of a system and presents a number of examples of aberrant usage [6]. Types of IDSs and their various approaches were catalogued in [4] and [7]. Various metrics that are used for network analysis and in IDSs are covered in [8]. The need for CIDS and baseline requirements for such a system are presented in [5]. These requirements and the collection of metrics useful for detecting anomalies form the basis for the architecture presented in Chapter 3.

Blockchain is a popular technology that has been applied to everything from medicine to cryptocurrencies [9]–[11]. Basics and algorithms associated with blockchain are covered in [11]–[13]. Blockchain serves as the trust agent for the CIDS system proposed and implemented in this thesis.

The seminal work that this thesis is based around is the concept proposed by Alexopoulous *et al.* [14]. They propose a framework using blockchain as a mechanism for a CIDS. While their work provided much of the theoretical framework supporting this idea, they did not provide a proof-of-concept. One of their areas for future work was to implement their framework. The focus of this thesis was to modify their proposal and implement it using a commercially available blockchain client.

## 1.3 Organization

There are five chapters and two appendices in this thesis. Chapter 2 covers background information relating to intrusion detection, anomalies, CIDS, and blockchain. The requirements and a proposed approach to a blockchain-based CIDS are presented in Chapter 3. An implementation of the approach detailed in Chapter 3 and the results of testing the resulting proof-of-concept system is presented in Chapter 4. The key findings and considerations for future work are provided in Chapter 5. Appendix A is an installation guide detailing setup instructions for a network of Ethereum nodes. Appendix B contains all of the code used in the creation and testing of the CIDS system. Code was written in the Python, JavaScript, Bash, Go, and C languages.

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 2: From Intrusion Detection Systems to Blockchain: An Overview

Several concepts that are integral to understanding the CIDS system proposed in this thesis are discussed in this chapter. First, an introduction to an IDS and its shortcomings is presented. A system capable of addressing those shortcomings, a CIDS, is described afterwords. Finally, blockchain is discussed as it is a potential solution to overcoming the implementation hurdles of CIDS.

## 2.1 Intrusion Detection Systems

As the name implies, the goal of intrusion detection is to recognize potential intrusions in a host or in a network. While there are several different approaches that are used in intrusion detection, the general process is described in Figure 2.1. This figure provides a framework for the discussion around intrusion detection and systems capable of accomplishing anomaly detection.

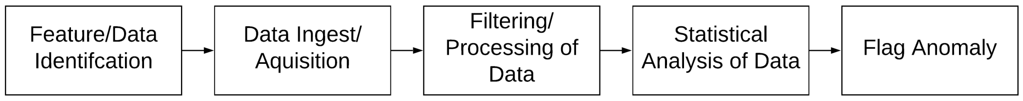

Feature/Data Identifcation
Data Ingest/ Aquisition
Filtering/ Processing of Data
Statistical Analysis of Data
Flag Anomaly

Figure 2.1. General Flow Chart for Intrusion Detection

The intrusion detection model is based on the idea that exploiting system vulnerabilities necessarily requires the introduction or leveraging of some system abnormality [6]. Furthermore, by detecting and analyzing these abnormalities, potential system intruders and violations can be revealed. An abnormality or anomaly is defined as a deviation from normal behavior [6]. Figure 2.2 shows a graphical definition of an anomaly where $X$ and $Y$ could represent hypothetical operational metrics recorded by an IDS. Groups $N_1$ and $N_2$ display normal behavior while groups $O_1$, $O_2$, and $O_3$ represent anomalous behavior. Points in this figure were clustered according to their distances from other points.

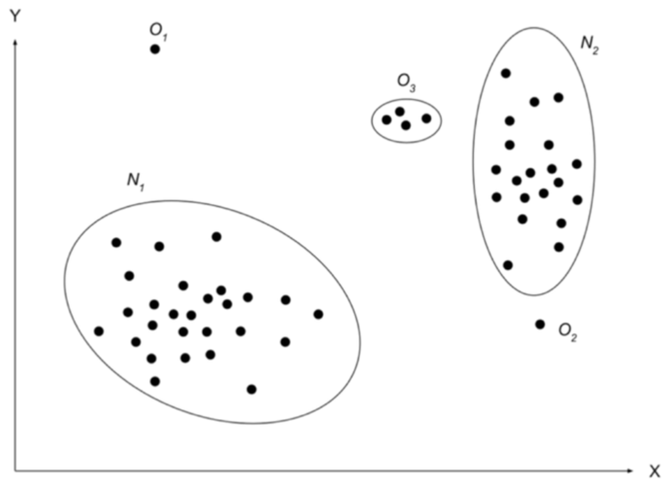


Figure 2.2. Anomalies in a Two-Dimensional Space. Source: [15].

IDSs analyze the produced anomalies and generate alerts based on established criteria. Depending on the type of IDS, that alert can trigger an action or response [6]. For example, if the IDS detects an denial-of-service (DOS) attack because of a high traffic volume from a single IP address, the IDS could change firewall rules to block that IP address. The general characteristics of an IDS are presented in Figure 2.3.

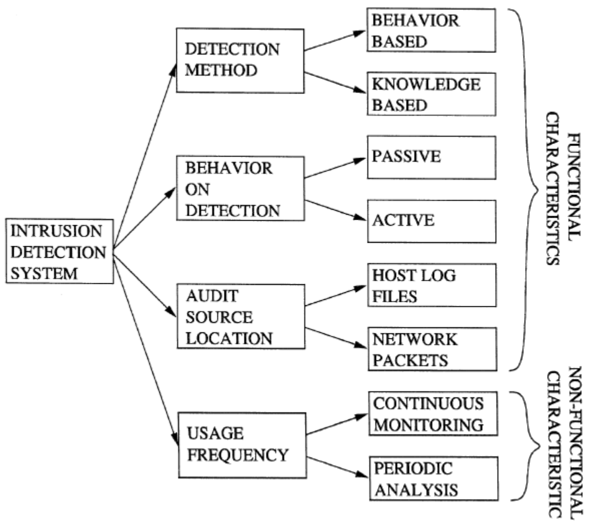


Figure 2.3. Characteristics of Intrusion Detection Systems. Source: [4].

There are several ways to categorize IDSs. One of the criteria that distinguishes IDSs is whether the anomaly detection method is behavior-based or knowledge-based [4]. Knowledge-based systems leverage a priori knowledge about different types of attacks to develop particular signatures to recognize anomalies [4]. Behavior-based systems develop thresholds or profiles of normal activity and compare current behavior patterns to them to detect anomalies. The focus in this thesis is on behavior-based IDSs. Another important criterion for IDS classification focuses on what target the IDS is attempting to protect. An IDS can be considered host-based or network-based depending on the circumstance.

### 2.1.1 Host-Based Intrusion Detection Systems (HIDS)

A HIDS uses data about the host and its computing activities to determine if some malfeasance is taking place. A sampling of these host metrics is presented in Table 2.1.

Table 2.1. Metrics Used in HIDSs. Adapted from [8], [16].

| Metric | Description |
|---|---|
| CPU Usage | High CPU Usage could Indicate an Attacker or a Misperforming Process |
| Memory Access | Incorrectly Timed or Strange Memory Access could Indicate an Attacker |
| File Access | Strange File Access Attempts could Indicate an Attacker |
| Login Attempts/Times | High Numbers of Failed Login Attempts or Login Attempts at Strange Times could Indicate an Attacker |

HIDS can also be either knowledge-based or behavioral-based. The knowledge-based case is akin to commercial virus-checking software [16]. A virus-checking software matches the signature of a virus or malware against characteristics on a host machine. If the signature is matched, then the software throws an alert. In the behavioral-based case, a profile of normal computer activity is developed. If the current measurements of metrics differ from the expected profile, an alert is thrown [16].

For example, if a system expects login attempts to occur between 0800 and 1700 and a login attempts occurs at 0200, an alert should be thrown.

### 2.1.2 Network-Based Intrusion Detection Systems (NIDS)

A NIDS inspects network-related data for anomalies. Examples of this data are network packet traffic and knowledge of the various network protocols [16]. A standard NIDS architecture is presented in Figure 2.4.

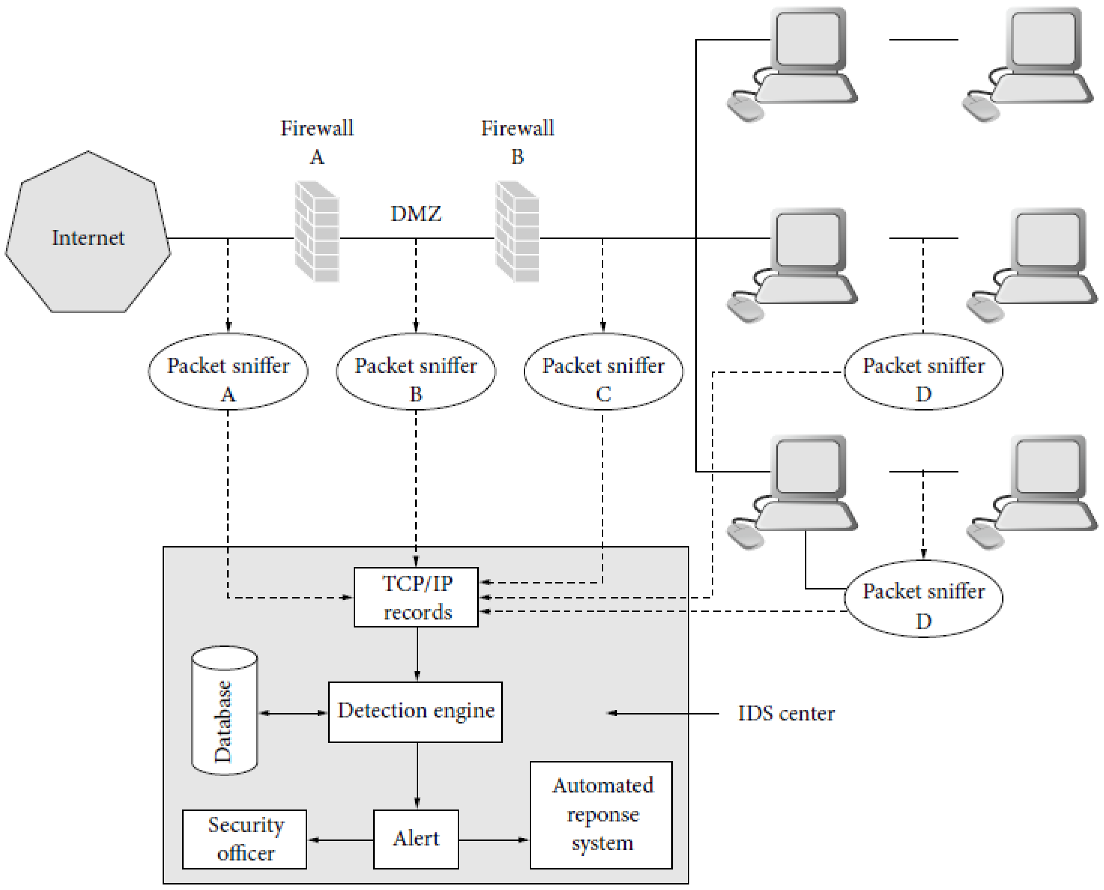


Figure 2.4. Standard NIDS Architecture. Source: [16].

Figure 2.4 highlights various places that packet sniffers can be placed to monitor traffic.

Those sniffed packets are then analyzed using either a knowledge-based or behavioral-based approach. In the knowledge-based case, if the traffic pattern matches a known attack, an alert is triggered [16]. In the behavioral-based approach, a normal profile of network traffic is developed. If the traffic pattern differs from that profile by a specified amount, an alert is triggered [16]. The general approach analyzes the statistical distribution of the attributes of TCP/UDP traffic. These attributes are volume, destination, source, connection time, and protocol [4], [16].

### 2.1.3 IDS: The Scaling Issue

While IDSs are commonly deployed as defensive solutions, they do have drawbacks [5]. Generally, IDSs are stand-alone. They monitor one host or one network for anomolous activity [5]. An issue arises when an attack is distributed across several hosts or several networks. A series of stand-alone IDSs that do not communicate or interact amongst themselves are incapable of fully detecting or responding to such an attack [5]. Specifically, the ability to correlate malicious events occurring across hosts in a network or networks in a system at the same time is not present [5]. In order to address this weakness, the concept of CIDS was introduced.

## 2.2 Collaborative Intrusion Detection Systems

CIDSs are designed to address the weakness of IDSs in thwarting distributed or parallel attacks. CIDSs typically consist of a set of monitor units that are sensors, and a set of analysis units that process the sensor data [5]. These units can be co-located [5]. In this way, a CIDS can aggregate data from multiple hosts or networks in order to make decisions about potential intrusions or anomalies.

### 2.2.1 Types of CIDS

There are three general categories of CIDS. Figure 2.5 shows their architectures by categorization. The blocks labeled ’M’ and ’A’ refer to whether the node is a monitor unit or analysis unit. In some cases, a node can be both.

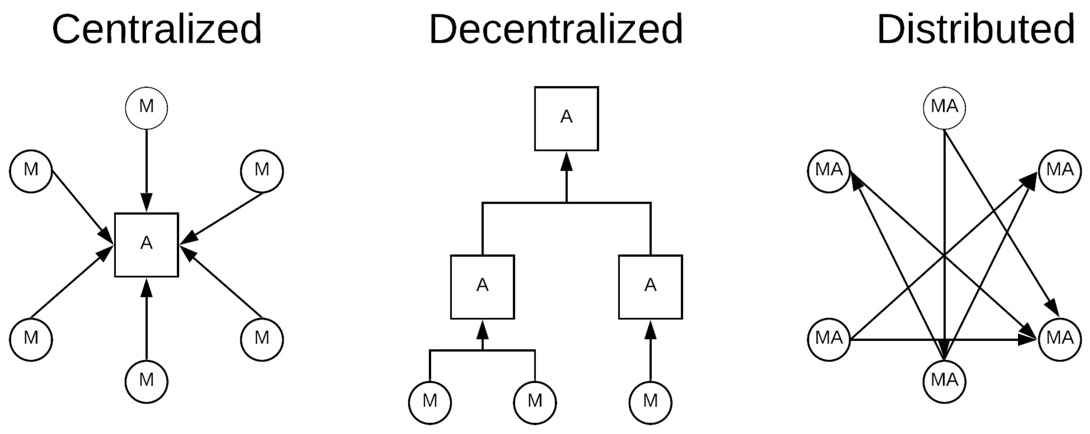


Figure 2.5. Overview of CIDS Architectures. Adapted from [5].

A centralized CIDS is the most straight forward solution to the problem of distributed attacks. This model has multiple monitor units feeding host-based alert data and network-based traffic data to a single central analysis unit [5]. The analysis unit performs either alert correlation algorithms on host-based data or standard detection algorithms on network data [5]. This approach does have drawbacks that can make it undesirable in several circumstances. Centralized CIDS scale very poorly with increasing network size, and the central analysis unit serves as a single point of failure (SPoF) that represents a performance bottleneck [5].

A decentralized CIDS uses a hierarchical structure of analysis and monitoring units [5]. This structure has the advantage of avoiding the SPoF issue from centralized CIDS and is scalable to larger systems [5]. Additionally, performance is improved at the top of the hierarchy because analysis units are processing data at every level of hierarchy. This reduces the burden on the top analysis unit [5]. This level-wise approach does come at a cost. Information is lost at each aggregation step before reaching the top of the hierarchy. This can lead to missing crucial data that prevents detection of attacks [5]. Also, as Vasilomanolakis *et al.* [5] note, modern implementations of decentralized CIDS still have SPoFs and bottlenecks.

A distributed CIDS eschews single analysis units in favor of distributing the analysis tasks amongst all monitors [5]. This requires a peer-to-peer (P2P) architecture to ensure all data is shared, aggregated, and correlated in a distributed manner across the system [5]. This approach is both scalable and avoids SPoFs. However, a distributed CIDS does incur more network cost due to increased signaling overhead [5]. Finally, as Alexopoulous *et al.* [14] note, such a system must have mechanisms to ensure trust amongst its nodes.

### 2.2.2 CIDS: The Trust and Consensus Issue

While a distributed CIDS approach seems effective at addressing the weaknesses of an IDS, certain facets of its implementation are difficult to overcome. The idea of trust is crucial in a CIDS. Alexopoulous *et al.* [14] posit a scenario in which a monitor begins to disseminate false information. The system must be able to determine whether the data produced by a monitor should be accepted and whether that monitor should be trusted. In other words, the system must reach consensus on all alert data and on the trustworthiness of all nodes [14]. Blockchain is proposed as a solution to this trust challenge.

## 2.3 Blockchain

Many industries are exploring how blockchain technology might improve their processes; cyber security is no different. The properties that blockchain exhibits are useful in the context of CIDS. Specifically, the critical component of blockchain technology for CIDS applications is its mechanism of validating and storing data with no need for a central, trusted authority. The following sections provide an overview of useful features of blockchain.

### 2.3.1 Categories of Blockchain

There are three types of blockchain ledgers currently in use: public, consortium, and private [17]. Public blockchain systems allow anyone with internet access and a desire to participate to do so. Consortium blockchain systems are maintained by an established body that grants access to others. Private blockchain systems are maintained by one entity that provides permissions to others. More detailed information can be found in [17]. Depending on the desired use case, different ledgers might apply.

### 2.3.2 Block Structure

At its most basic form, a blockchain is a chain of blocks, with each block connected to the one before it and after it by means of a mathematical relationship. A block is simply a container for pieces of data. The major idea behind blockchain is that each block has a unique self-identifying hash in order to ensure integrity throughout the blockchain. This self-identifying hash is composed of the hash of the block index, data, timestamp, and of course, the hash of the previous block hash [9]. A truncated example of a blockchain is shown in Figure 2.6.

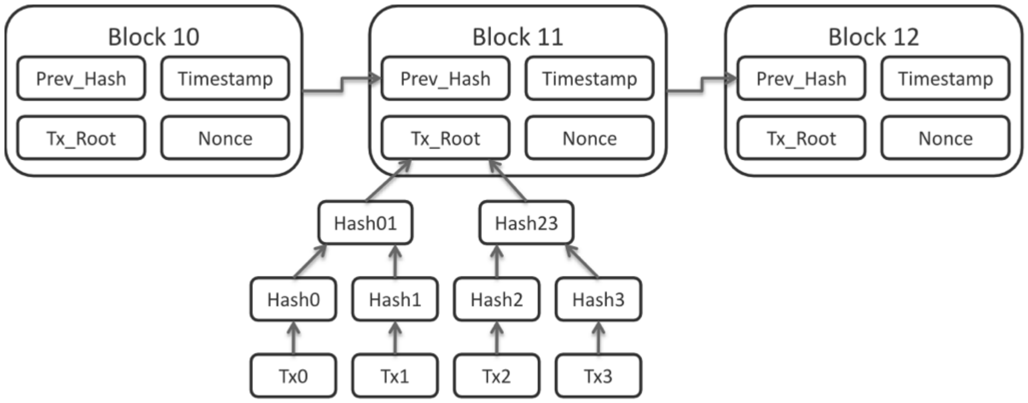


Figure 2.6. Blockchain Example. Source: [18]

Each block also contains a record of all transactions, called a ledger, that occurred during the duration of block production. In this way, each transaction is codified in this block structure [9]. Figure 2.6 illustrates this point. As each block refers to the block before it, there is a record of all of the transactions that have occurred before the production of the current block. Section 2.3.3 will discuss why it is mathematically impractical to modify a block [9].

### 2.3.3 Consensus

Consensus algorithms allow participants in a blockchain network to reach agreement about the state of the network without a central trusted authority [9]. Any system that is designed to use blockchain is only as effective as its consensus model [13]. More information about the origins of the consensus problem and its properties can be found in [13]. In the blockchain world, the most prevalent example of a consensus algorithm is the *proof-of-work* algorithm, which Bitcoin implements [9].

*Proof-of-work* is based around the idea that a participant validates its identity by providing some proof that it performed work. In the case of Bitcoin, the goal of every participant is to find a hash value that is less than a number that is specified as the difficulty level by the network [13]. This is an example of a computational puzzle where the most efficient approach to solving the puzzle is a brute force *guess-and-check* method [19]. This process, called mining, prevents any single participant from having an advantage in producing the next block [9]. As such, no authentication or a priori knowledge is required of participants. This algorithm also makes it mathematically impractical to modify a block or set of transactions as a node would have to reproduce the entire chain before a new block was produced. Nakamoto [9] showed that the probability of a successful modification decreases exponentially with the size of the blockchain. However, *proof-of-work* is vulnerable to the 51% attack where if more than half of the potential mining power is consolidated into one coalition, that coalition can write blocks into the blockchain [13]. In order to combat this, Ethereum (another cryptocurrency) implemented another consensus algorithm, *proof-of-stake*.

*Proof-of-stake* relies on a pool of validators with an economic stake in the network taking turns proposing and voting on the next block [20]. The algorithm selects validators pseudo-randomly for block creation thereby preventing advance knowledge of when a particular participant would create a block. The odds of being selected as a validator is determined by the amount of cyrptocurrency, or stake, that the participant possesses [13] [20]. While there are potential problems with this type of implementation, (see the *Nothing-at-Stake* problem [20]), this algorithm does address the 51% attack and is currently in development by Ethereum [20].

Consensus algorithms form the backbone of a network environment where participants can trust network state information. These attributes are crucial in the context of a CIDS environment. They are an active area of important research but outside the scope of this thesis.

### 2.3.4 Transactions

A transaction is simply a transfer of data from one party to another. This data can be anything. In the context of cryptocurrency, this data would be money or contract data [9] [11]. In the medical field, this data could be the medical records of patients or the transfer of medical equipment [10]. The flexible nature of data allows blockchain to be used in a variety of fields. In the context of CIDS, a transaction could contain alert data [14]. The blockchain ledger contains a record of all transactions that have taken place. The general approach to a cyrptocurrency transaction is presented in Figure 2.7.

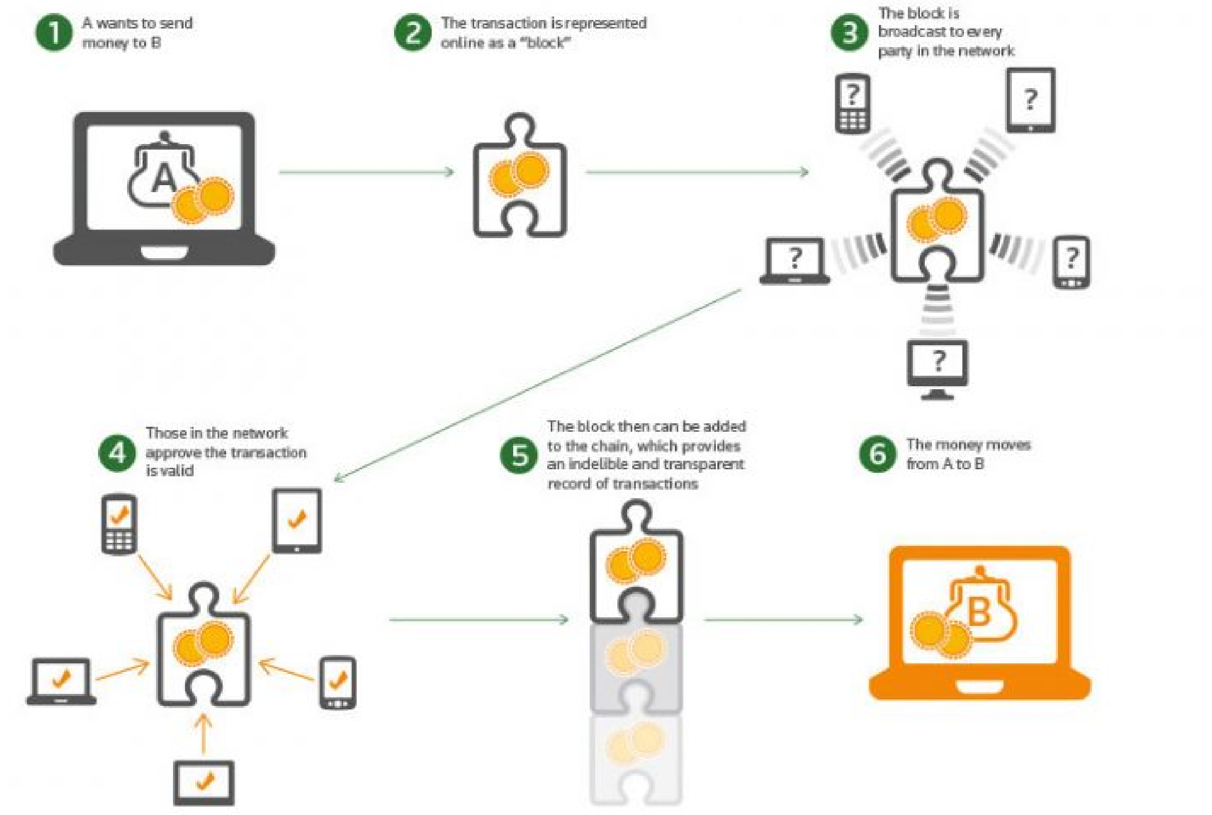

1 A wants to send money to B
2 The transaction is represented online as a "block"
3 The block is broadcast to every party in the network
4 Those in the network approve the transaction is valid
5 The block then can be added to the chain, which provides an indelible and transparent record of transactions
6 The money moves from A to B
A
B

Figure 2.7. Blockchain Transaction Process. Source: [21].

The process modeled in Figure 2.7 illustrates a couple of important points. Every participant has a copy of the ledger of transactions and each of these transactions is permanent and transparent [9]. Additionally, each block is agreed upon using some consensus algorithm.

This provides a trust framework for all participants in the network. The process workflow for Bitcoin is presented as Algorithm 1.

```
Result: All Transactions Codified
1) New transactions are broadcast to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid and not already
  spent.
6) Nodes express their acceptance of the block by working on creating the next
  block in the chain, using the hash of the accepted block as the previous hash.
```

**Algorithm 1:** Bitcoin Transaction Codification Workflow. Source: [9].

This process flow facilitates the trust framework for all participants in the network. The ability to ensure that all participants have knowledge of all transactions and have a stake in approving those transactions is critical to using blockchain for a CIDS.

## 2.4 Summary

Chapter 2 discussed background concepts required for understanding the techniques developed in the remainder of this thesis. IDSs, their categorization, their properties, and their shortcomings were presented. CIDSs were discussed as a solution to address the shortcomings of IDSs. CIDS categorization and the trust issue are also covered. Finally, blockchain was presented as a potential solution to the trust issue in a CIDS. Blockchain structure, consensus, and transactions were defined.

Chapter 3 will present an attack scenario as a vector to discuss CIDS requirements. Those requirements will then be used to propose a blockchain-based CIDS solution.

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 3:
# A Blockchain-based Collaborative Intrusion Detection System Solution

While the theoretical advantages of CIDS have been touted for several years, actual implementation has been difficult to achieve. The primary reason for this difficulty comes from how participants are able to establish trust in one another and how the veracity of the shared information is ensured. [5]. The purpose of this chapter is to model an attack that is countered well by a CIDS and by doing so, present the requirements for a successful CIDS system. Using those requirements, we briefly explore some of the seminal work on the topic of a CIDS and propose our own model for a CIDS using blockchain as an information-sharing tool.

## 3.1 Distributed Attack Model: Doorknob Rattling Attack

Our main series of questions and explorations dealt with improving the data ingest and acquisition process. Envision a scenario in which a network is comprised of several nodes where each monitors and reports various events. The confidence in the anomaly detection capabilities of the system as a whole begins with confidence in the accuracy and completeness of the record of events of interest. A system administrator must ask the question, how can they trust the data that is flagged by the IDS?

The enormity of this question is even further exacerbated when a coordinated attack occurs [5]. When coordinated attacks are choreographed to maintain a low volume or rate at any one host, respectful of common intrusion detection activity thresholds, the network must be able to aggregate events to be able to respond. Consider the example of the doorknob rattling attack on a supervisory control and data acquisition (SCADA) system wherein an adversary attempts common login/password combinations on several machines in the network, maintaining a low number of overall login attempts per device [22]. For the attack to succeed, the adversary needs only gain remote access to one of these devices [22]. Another example of the doorknob rattling attack can be seen in [23] where attackers attempted to login into WordPress servers worldwide. Instead of exhaustively using every username/password combination in their dictionary, they settled on a small subset of hundreds of combinations before moving on to the next target WordPress server [23].

Table 3.1. Username/Password Combinations for WordPress Doorknob Rattling Attack. Adapted from [23].

| Username | Password |
|---|---|
| admin | admin |
| Admin | 123456 |
| user | 12345 |
| support | 12321 |
| qwerty | qwerty |
| manager | 12345678 |
| administrator | 1234 |
| admin1 | 123123 |
| adm | password |
| root | 666666 |
| aaa | 121212 |
| sysadmin | pass |

Table 3.1 provides a sample of the 13 most commonly used username/password combinations that were used in the WordPress attack. Similar types of dictionaries can be used in the context of smaller networks like the SCADA networks discussed in [22]. The only way to detect this type of attack is to aggregate the total number of login attempts. We can express this mathematically as follows.

Let $x_0, x_1, ..., x_{i-1}$, where $i$ is the number of number nodes, correspond to the number of login attempts on each node. Each node has a detection threshold $y$ corresponding to a certain number of incorrect login attempts. If there are $z$ login attempts on each node where $z < y$, this detection threshold will never be exceeded and an alert will not be thrown. A CIDS approach would rely on setting an additional threshold $v$ defined as

$$v = \sum_{n=0}^{i-1} x_n. \tag{3.1}$$

This threshold can be tuned to the network in question. For example, an organization with 100 devices may allow three incorrect login attempts on each machine but might trigger an alert if there are over 50 login attempts in a given time interval. This would correspond to two login attempts on 25 of the 100 devices, thwarting an attempted doorknob rattling attack.

Consider a scenario where ten stand-alone IDS nodes are tracking login rate, with a CIDS watching the system as a whole. For the ten stand-alone IDS nodes expecting to record between zero and three logins per reporting period, a detection threshold of unusual activity might be set at four. These login attempts can be treated as a discrete random variable with a PMF as shown in Figure 3.

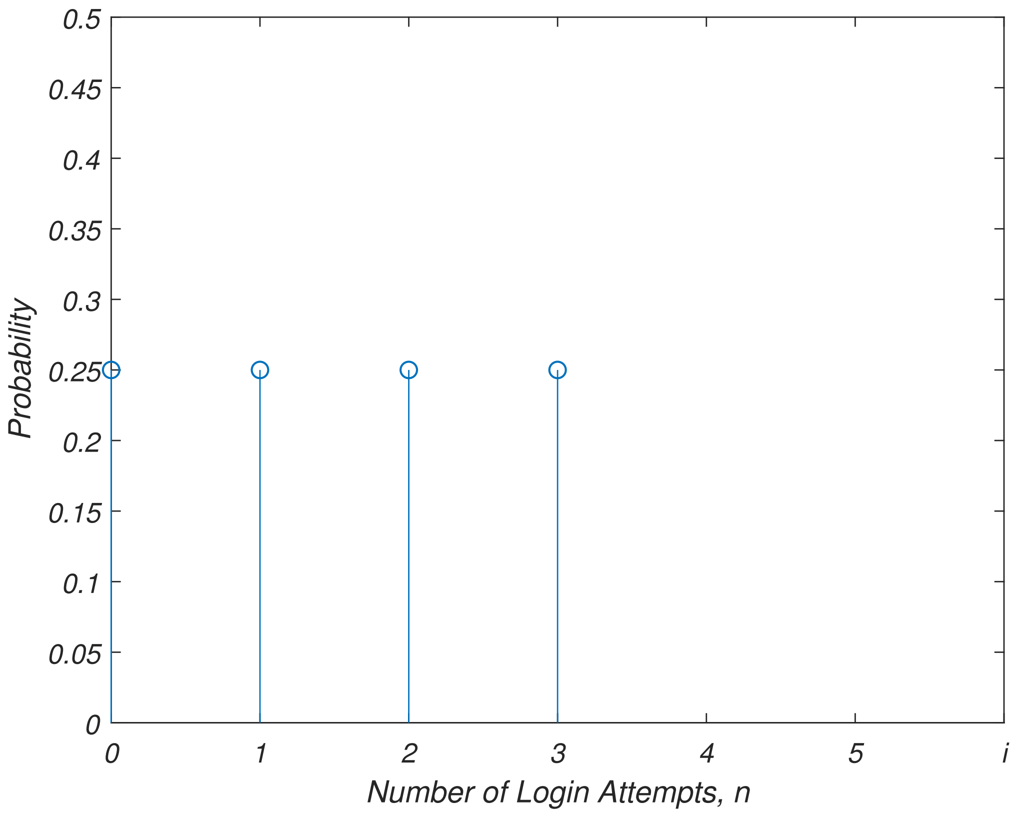


Figure 3.1. PMF for Nominal CIDS Scenario

Let $X$ be uniformly distributed on the sample space of login attempts $\{0, 1, 2, i - 1\}$. Its expected value is given by

$$E(X) = \frac{i - 1}{2} \tag{3.2}$$

In the case of $i = 4$, the expected value would be $\frac{4-1}{2} = 1.5$ login attempts. Assume that an attacker tries a doorknob rattling attack with two login attempts. With the attacker attempting 2 additional login attempts on each of the nodes, we would expect a total of 3.5 login attempts, which is below the detection threshold. The log traces for both the IDS and the CIDS in this scenario are illustrated in Figure 3.2.

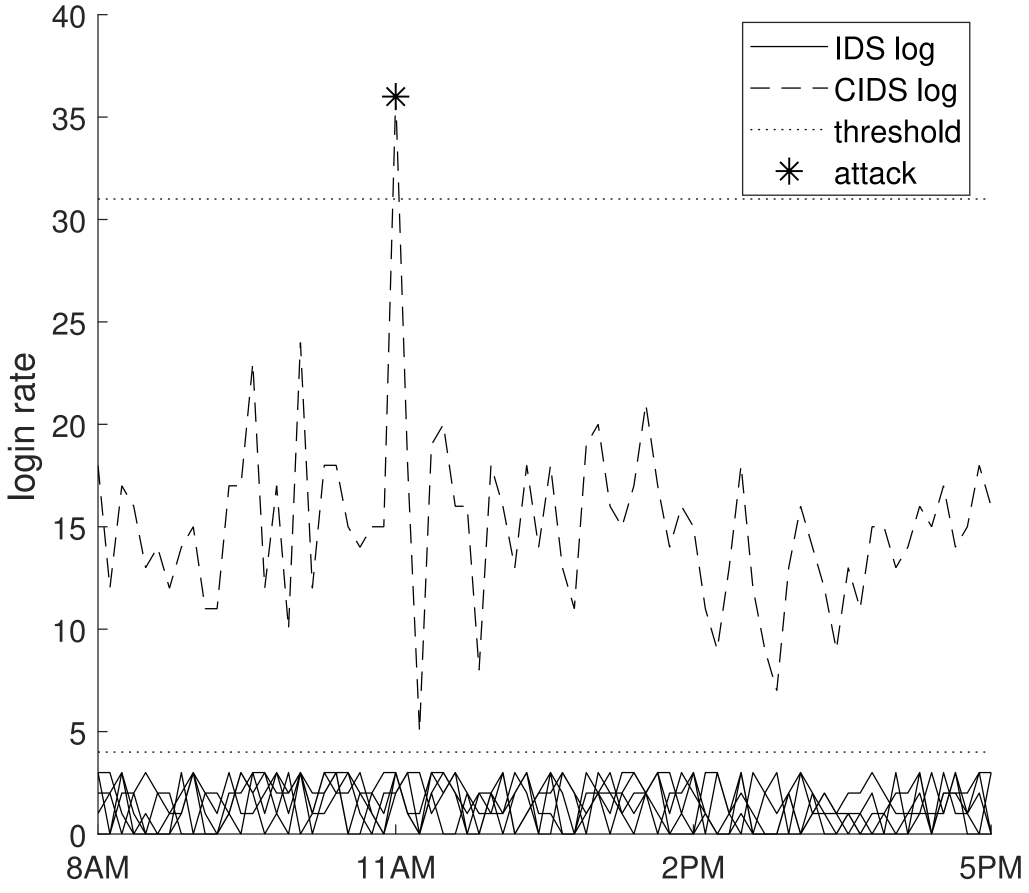


Figure 3.2. Nominal Doorknob Rattling Attack Detection via CIDS

Only nodes that recorded 2 or 3 legitimate login attempts would be able to detect the doorknob rattling attack. Thus, on average, half of the nodes would ignore the 11:00-am doorknob rattling attack as the two extra login attempts per system would not break the detection threshold. This low level of activity disappears in the normal network activity on any one node. In contrast, the attack clearly stands out in the CIDS log trace in Figure 3.2 (as the number of login attempts was greater than the specified CIDS threshold), easing the complexity required in the follow-on data processing and analysis phases of anomaly detection.

While this example supports CIDS as a technique to overcome coordinated attacks, the system administrator is still left with the question of whether to trust the data that is being reported by the individual nodes in the CIDS. After all, if login attempts are not reported, then they cannot be aggregated for analysis. Furthermore, if an adversary is successful in accessing a CIDS, they could alter the access log to remove evidence of the attack, as hackers often tamper with evidence of their presence on a system to thwart future forensic analysis [24]. We need a technique that provides some guarantee that every login attempt will be logged and that those logs are tamper-resistant. This leads to a conversation about requirements for a CIDS system.

## 3.2 Requirements for a CIDS System

The doorknob rattling scenario from Section 3.1 demonstrated the role of CIDS in detecting certain attacks. Vasilomanolakis *et al.* [5] and Alexopolous *et al.* [14] lay out the principal requirements of such a system, which are compiled in Table 3.2.

Table 3.2. CIDS Requirements. Source: [5], [14].

| | |
|---|---|
| Accountability | Actions taken by a participating node must be tied to that node. |
| Integrity | Data cannot be modified once entered into the system. |
| Resilience | The system should not depend on small numbers of node and must avoid single points of failure. |
| Consensus | Nodes in the system must reach consensus about the quality and trustworthiness of data. |
| Scalability | The system must be able to scale to larger numbers of agents. |
| Overhead | Overhead must be minimized to allow flexibility of the system. |
| Privacy | Depending on the type of system, some level of privacy must be considered for participants. |

In the context of the scenario modeled in Section 3.1, some of these requirements are critical for a system administrator to be able to detect and respond to the doorknob rattling attack. Specifically, accountability, integrity, and consensus are integral to the detection of the attack. The system has to record all login attempts on every node (accountability) while ensuring that the network has trust in the quality and accuracy of the data (consensus). Finally, there must be confidence that the login data has not been altered by an attacker once it has been entered in the system (integrity). While our sample scenario had only 10 nodes, in the context of an industrial network where there can be hundreds or thousands of different devices, the system must be able to scale.

There are clear trade-offs between the attributes specified in the table. For example, there are often conflicts between privacy and accountability [14]. Every action that is taken to tie a set of actions to a participant invariably reduces the privacy of that participant. Furthermore, depending on where such a system would be utilized, the requirements shift. For example, database entries with private customer account details require different privacy considerations than logs of privilege escalation on a controlled system. Framing the design of CIDS in the context of these requirements is useful toward capturing the key characteristics of the system while minimizing overhead and cost. For example, overlaying data encryption using a public key infrastructure necessarily increases overhead and thus should be implemented only when needed [14].

As discussed in Section 2.3.2, the central issue with CIDS is how to maintain trust amongst the collaborating nodes in the system. Blockchain provides a potential solution to those trust issues.

## 3.3 Meeting CIDS Requirements with Blockchain

A blockchain-based system fulfills the requirements in Table 3.2 as follows: Because each transaction contains a sender and a receiver that cannot be modified once added to the chain, the requirement of accountability is met. Furthermore, once approved transactions are added to the chain, they cannot be modified without overcoming significant cryptographic barriers [12], fulfilling the requirement of integrity. Resilience and consensus are both met via blockchain's embedded distributed consensus mechanisms [13], which can be implemented to ensure single points of failure are avoided and graceful degradation is possible. As evidenced by wide use of cryptocurrencies like Bitcoin and Ethereum, blockchain is scalable with appropriate implementation considerations and adequate resourcing. The cryptocurrency use case also demonstrates that privacy can be achieved in blockchain-based systems via the assignment of non-attributable identities. As Alexopoulous *et al.* [14] note, overhead can be a problem for blockchain systems, but it is dependent on which consensus mechanism is used and other design decisions and can be mitigated via techniques like alert hashes and bloom filters.

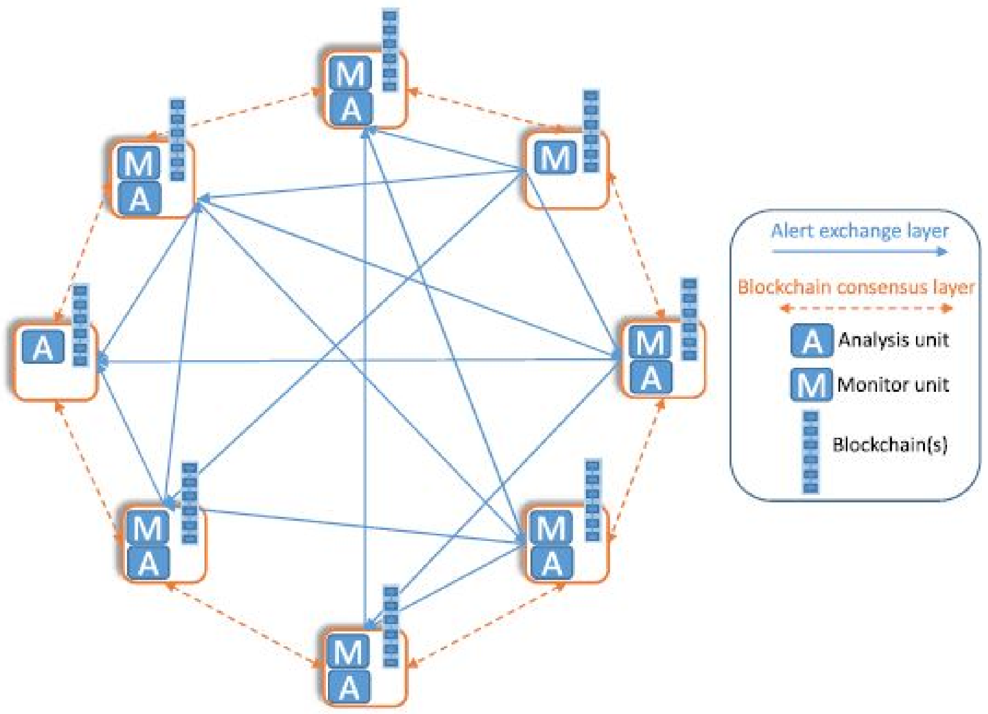


Figure 3.3. Generic Architecture of a Blockchain-Based CIDS. Source: [14].

To satisfy the requirements from Table 3.2, Alexopoulous *et al.* [14] propose the CIDS framework shown in Figure 3.3 and utilize a secure distributed ledger implemented by blockchain technology to exchange alerts between collaborating nodes. These alerts become the transactions of the blockchain system. Depending on the types of attacks or behaviors that an organization is concerned about, different metrics must be utilized. A more detailed discussion on different metrics can be found in [8]. As a few key examples, finding an adversary running unwanted programs might require logging CPU utilization statistics. Source and destination ports for data access requests could be another potential metric of interest. To thwart the doorknob rattling attack from Section 3.1, the transactions would record login attempts.

## 3.4 Proposed Blockchain Solution

As a proof-of-concept, we develop a CIDS to thwart the doorknob rattling attack by leveraging blockchain technology to facilitate a distributed ledger of login and authentication attempts across a network. The basic architecture of the system is displayed in Figure 3.4.

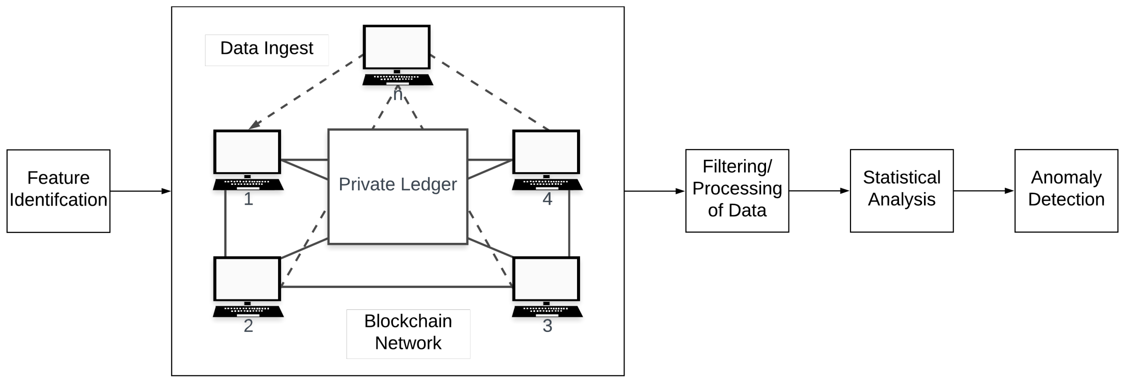


Figure 3.4. Blockchain Solution for CIDS

We make a few assumptions in this general approach. The first is that for most organizations, a private ledger is the most logical mechanism for this log, with the system administrator validating participant identity upon initialization of the system, at a minimum, which leads to our second assumption that we have a priori knowledge of participants on the network. For example, most organizations and companies register and sign out equipment to their employees. Similarly, in the context of our general approach, the system administrator would register devices to the user and ensure proper registration into the blockchain network. This is a crucial distinguishing factor between this approach and current blockchain implementations like Bitcoin where anonymity of the users is a feature. In the Bitcoin environment, it would not be possible to have a priori knowledge of the participants in transactions [9]. More to the point, Bitcoin does not support tying an actual identity to a Bitcoin address. In fact, Bitcoin.org makes recommendations on how to protect individual privacy using techniques like having multiple wallets and using unique addresses for each transaction [25].

Although the ledger in Figure 3.4 is pictured in the center of the nodes, this is only to emphasize that consensus is maintained on the ledger's contents; it is physically stored at every node to maintain the distributed attributes important to the system's benefits. Data from the private ledger can be used for aggregate analysis much like in any other IDS, with the specific anomaly detection mechanism outside the scope of our proposed solution.

In the solution proposed in [14], nodes are categorized as either monitoring units or analysis units. Any node in the blockchain network can be either although the technical requirements for each will be different. An analysis node will need significantly more computing power based on the algorithm chosen to process the data. Some systems may also require two layers of communication: an alert layer and a consensus layer to allow flexibility in scenarios where permissioned viewing lists are necessary to achieve required privacy levels [14]. The range of scenarios that the system must handle will heavily influence design decisions, but as a proof-of-concept we have stuck to the case illustrated in Figure 3.4, where all nodes participate fully as both monitoring and analysis units.

## 3.5 Summary

An example of a distributed attack, the doorknob rattling attack, was presented as a way to discuss CIDS requirements. Some use cases and tradeoffs of the various requirements were discussed. An explanation of why blockchain is well-suited to address the trust issues inherent to CIDS was presented. Finally, a brief discussion about what types of metrics might be used as alerts in the blockchain system was held.

Chapter 4 will present an implementation of our general blockchain solution from Section 3.4 and present results from a series of experiments conducted to test the effectiveness of our solution. Specific tools, languages, techniques, and algorithms will be presented and explained.

# CHAPTER 4: Implementation and Results

This chapter discusses the variety of technology and approaches that were required to create a proof-of-concept CIDS and presents results of some experiments that were conducted to show the potential capabilities of the system. Specifically, in the first two sections of this chapter, the approach is presented by which the general blockchain solution discussed in Chapter 3 section 4 and Figure 3.4 was implemented in small scale as a proof-of-concept. Specific platform, programming language, software packages, and hardware choices are discussed at length. Chapter 4 sections 3 and 4 cover a series of experiments designed to examine how such a test bed can be used to thwart attacks like the doorknob rattling attack from Chapter 3.

## 4.1 Recording Metrics

In order to properly implement and test our CIDS, we needed to control and capture our metrics of interest. We implemented our CIDS system to aggregate two different metrics, login data and CPU utilization. Both of these metrics are good indicators of different types of attacks [8]. Login data is crucial to identifying the doorknob rattling attack discussed in Chapter 2. CPU utilization information can help to identify when malicious processes are running on a particular node [8]. The following sections describe how our system interacts with each host machine to extract the data necessary for CIDS operation.

### 4.1.1 CPU Utilization

The basic idea for recording this metric is that unexpected spikes in CPU utilization are potential indications of a possible anomaly. Our goal was to sample one of our nodes every minute and report those statistics as a transaction in our blockchain system. In order to properly perform this test, we created a program that would spike CPU utilization on our node to an expected amount. This was implemented this using the OpenMP library in C [26]. The OpenMP library provides an API specification for parallel programming. Our test computer had 4 cores and 2 threads per core for a total of 8 virtual cores. In a typical non-parallel environment, we can only fully load one of these 8 virtual cores leading to approximately 12.5% CPU utilization (maxing out 1 of 8 cores). Using the OpenMP library we are able to load any number of our cores giving us more granularity in setting utilization values.

While OpenMP provides the ability to use multiple threads, careful attention must be paid to how it is used. It was necessary to select a problem where using multiple threads would speed up execution. Furthermore, the problem had to take long enough to solve that it was measurable by our system. We used factorization of a large number. The code is shown in Listing 4.1 and also in Appendix B.1.3. The *omp_set_num_threads()* command allows for the selection of a specific number of threads. While the code snippet in Listing 4.1 uses 8 threads, any number from 1 to 8 is selectable.

Listing 4.1: Factorization of a Prime Number

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
/*
    The purpose of this program is to force CPU utilization
       by factoring a large number
*/
void main(int argc, char *argv[])
{
    if(argc != 2)
    {
        printf("Usage:␣./para␣numToFactor\n");
        return;
    }
    long long int n = (long long int) strtol(argv[1], (char
       **)NULL, 10);
    long long int range = floor(sqrt(n));
    omp_set_num_threads(8);
    #pragma omp parallel for
    for(long long int i = 1; i < n; i++)
```

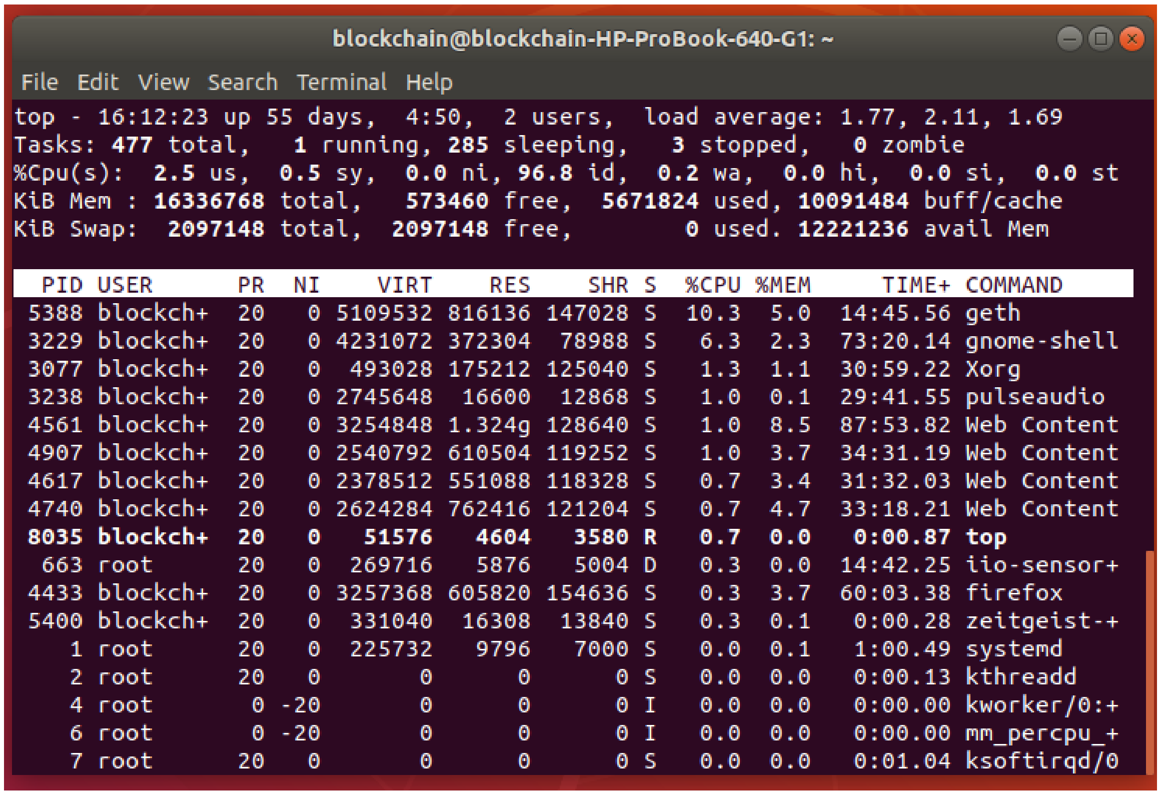
```
        {
            if(n % i == 0)
            {
                printf("%lld␣is␣divisble␣by␣%lld\n",n,i);
                printf("%lld␣is␣divisble␣by␣%lld\n",n,n/i);
            }
        }
    }
```

System monitoring was accomplished using the *top* command from the Linux operating system. From the Linux manual pages, "The *top* program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of tasks currently being managed by the Linux kernel. The types of system summary information shown and the types, order and size of information displayed for tasks are all user configurable and that configuration can be made persistent across restarts [27]." The command *top -b -n2 -d 1* takes two samples of our system in a one second duration in batch mode. This allows the output to be sent to a variety of different programs. In this case, the output was redirected to a text file, which was then read by the CIDS and recorded in the blockchain ledger. The *top* code snippet is shown in Listing 4.2. The full Python script that read the text file and submitted a transaction to the Ethereum nodes is in Appendix B.1.3.

Listing 4.2: Using *top* and other Linux utilities to record CPU Utilization

```
 #!/bin/bash
top -b -n2 -d 1 | awk "/^top/{i++}i==2" | grep -Ei "cpu\(s\)
   \s*:" > out.txt
```

Figure 4.1 presents the basic output of the top command. The typical steady state utilization of our *geth* node is about 10% of one thread. The total CPU utilization on one thread was approximately 20%. The utilization over all 8 threads was 2.5%, which can be seen by the header *%Cpu(s)*. This brings up an important point: there is a cost to running a *geth* node. Future research will have to answer questions about how expensive (from both a computational and network perspective) the overhead of a blockchain system is.

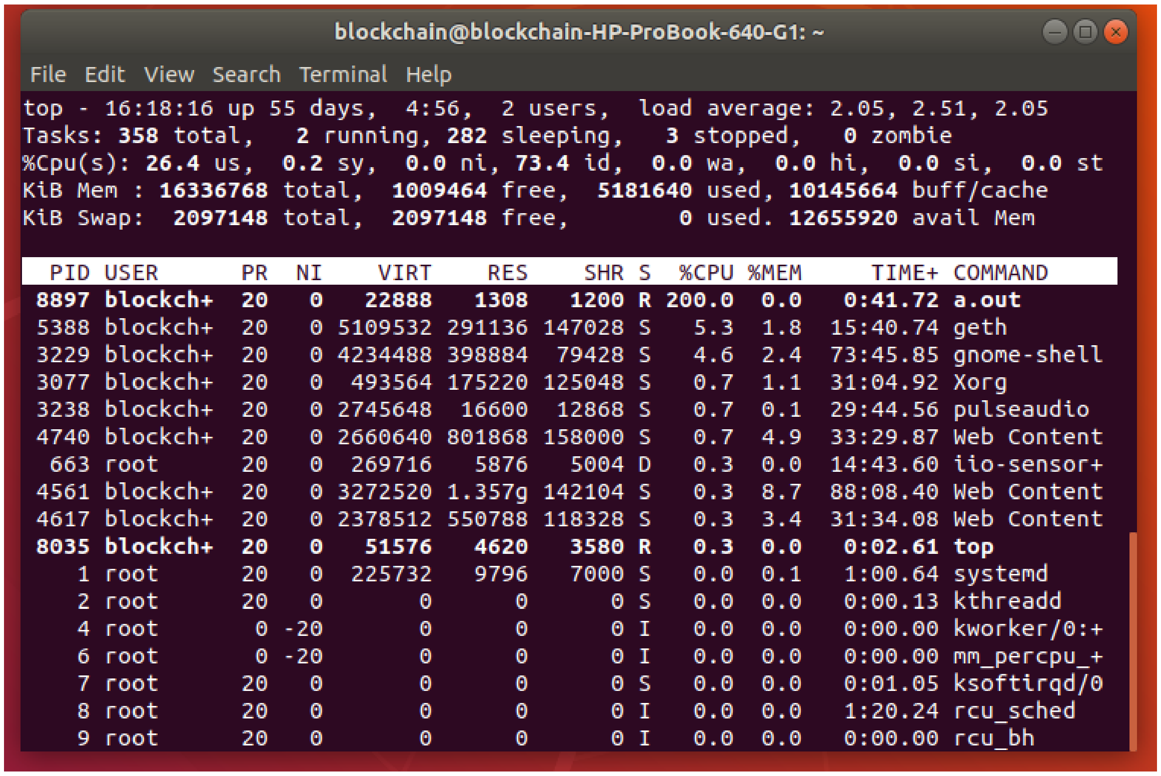

blockchain@blockchain-HP-ProBook-640-G1: ~
File Edit View Search Terminal Help

```
top - 16:12:23 up 55 days,  4:50,  2 users,  load average: 1.77, 2.11, 1.69
Tasks: 477 total,   1 running, 285 sleeping,   3 stopped,   0 zombie
%Cpu(s):  2.5 us,  0.5 sy,  0.0 ni, 96.8 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16336768 total,   573460 free,  5671824 used, 10091484 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used. 12221236 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 5388 blockch+  20   0 5109532 816136 147028 S  10.3  5.0  14:45.56 geth
 3229 blockch+  20   0 4231072 372304  78988 S   6.3  2.3  73:20.14 gnome-shell
 3077 blockch+  20   0  493028 175212 125040 S   1.3  1.1  30:59.22 Xorg
 3238 blockch+  20   0 2745648  16600  12868 S   1.0  0.1  29:41.55 pulseaudio
 4561 blockch+  20   0 3254848 1.324g 128640 S   1.0  8.5  87:53.82 Web Content
 4907 blockch+  20   0 2540792 610504 119252 S   1.0  3.7  34:31.19 Web Content
 4617 blockch+  20   0 2378512 551088 118328 S   0.7  3.4  31:32.03 Web Content
 4740 blockch+  20   0 2624284 762416 121204 S   0.7  4.7  33:18.21 Web Content
 8035 blockch+  20   0   51576   4604   3580 R   0.7  0.0   0:00.87 top
  663 root      20   0  269716   5876   5004 D   0.3  0.0  14:42.25 iio-sensor+
 4433 blockch+  20   0 3257368 605820 154636 S   0.3  3.7  60:03.38 firefox
 5400 blockch+  20   0  331040  16308  13840 S   0.3  0.1   0:00.28 zeitgeist-+
    1 root      20   0  225732   9796   7000 S   0.0  0.1   1:00.49 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.13 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:+
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_+
    7 root      20   0       0      0      0 S   0.0  0.0   0:01.04 ksoftirqd/0
```

Figure 4.1. Baseline Output of Top Command

Figure 4.2 shows what happens when the factorization program *a.out* is run maxing out two threads (one core). The overall CPU utilization went up to 26.4%, which again can be seen by the header *%Cpu(s)*. This is slightly more than a quarter of the total available resources. We have maxed out two threads and have some additional utilization taking place based on the other processes running at the same time.

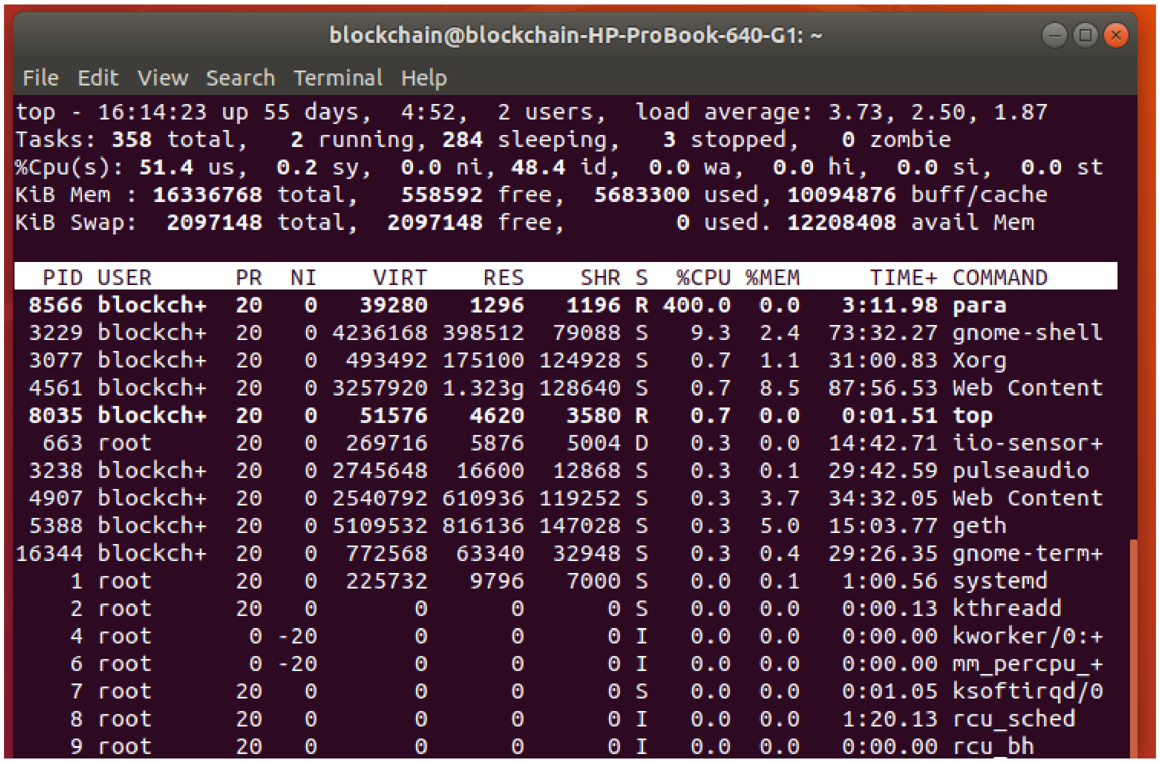

blockchain@blockchain-HP-ProBook-640-G1: ~
File Edit View Search Terminal Help

```
top - 16:18:16 up 55 days,  4:56,  2 users,  load average: 2.05, 2.51, 2.05
Tasks: 358 total,   2 running, 282 sleeping,   3 stopped,   0 zombie
%Cpu(s): 26.4 us,  0.2 sy,  0.0 ni, 73.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16336768 total,  1009464 free,  5181640 used, 10145664 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used. 12655920 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 8897 blockch+  20   0   22888   1308   1200 R 200.0  0.0   0:41.72 a.out
 5388 blockch+  20   0 5109532 291136 147028 S   5.3  1.8  15:40.74 geth
 3229 blockch+  20   0 4234488 398884  79428 S   4.6  2.4  73:45.85 gnome-shell
 3077 blockch+  20   0  493564 175220 125048 S   0.7  1.1  31:04.92 Xorg
 3238 blockch+  20   0 2745648  16600  12868 S   0.7  0.1  29:44.56 pulseaudio
 4740 blockch+  20   0 2660640 801868 158000 S   0.7  4.9  33:29.87 Web Content
  663 root      20   0  269716   5876   5004 D   0.3  0.0  14:43.60 iio-sensor+
 4561 blockch+  20   0 3272520 1.357g 142104 S   0.3  8.7  88:08.40 Web Content
 4617 blockch+  20   0 2378512 550788 118328 S   0.3  3.4  31:34.08 Web Content
 8035 blockch+  20   0   51576   4620   3580 R   0.3  0.0   0:02.61 top
    1 root      20   0  225732   9796   7000 S   0.0  0.1   1:00.64 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.13 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:+
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_+
    7 root      20   0       0      0      0 S   0.0  0.0   0:01.05 ksoftirqd/0
    8 root      20   0       0      0      0 I   0.0  0.0   1:20.24 rcu_sched
    9 root      20   0       0      0      0 I   0.0  0.0   0:00.00 rcu_bh
```

Figure 4.2. Two Thread Output of Top Command

Figure 4.3 shows what happens when the program *para* is run maxing out four threads (two cores). The overall CPU utilization went up to 51.4%. This is slightly more than half of the total available resources.

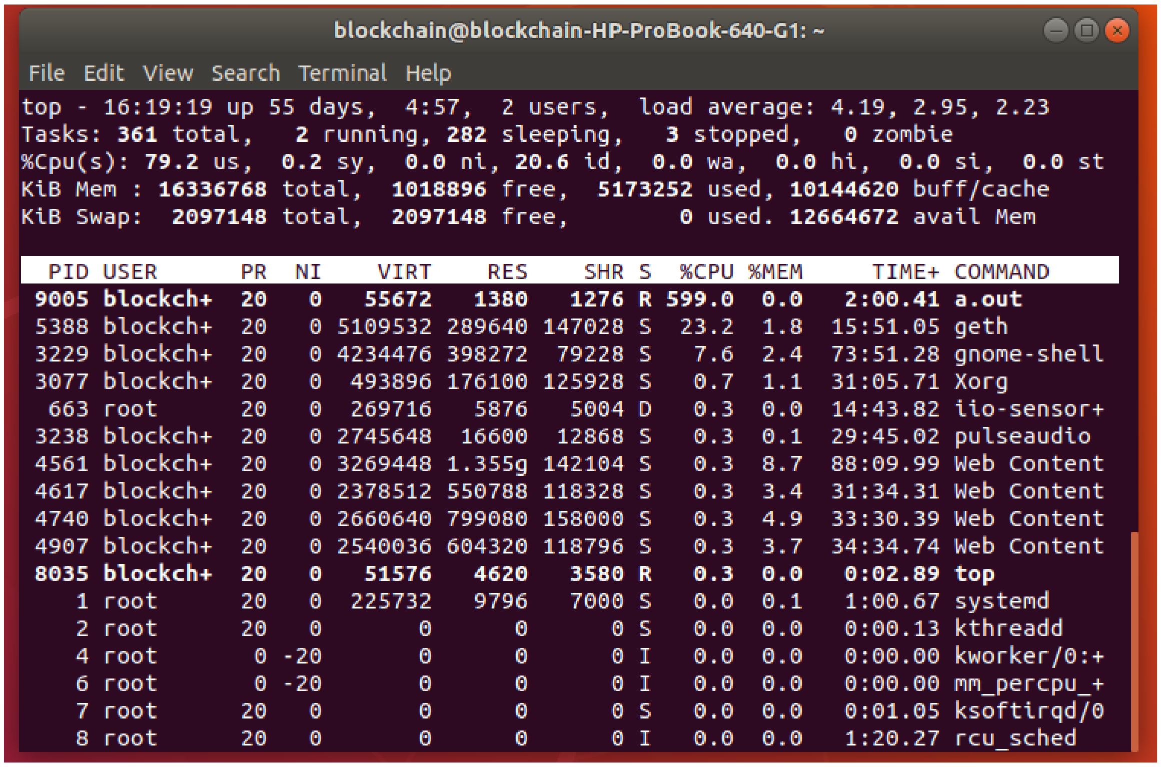

blockchain@blockchain-HP-ProBook-640-G1: ~
File Edit View Search Terminal Help

```
top - 16:14:23 up 55 days,  4:52,  2 users,  load average: 3.73, 2.50, 1.87
Tasks: 358 total,   2 running, 284 sleeping,   3 stopped,   0 zombie
%Cpu(s): 51.4 us,  0.2 sy,  0.0 ni, 48.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16336768 total,   558592 free,  5683300 used, 10094876 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used. 12208408 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 8566 blockch+  20   0   39280   1296   1196 R 400.0  0.0   3:11.98 para
 3229 blockch+  20   0 4236168 398512  79088 S   9.3  2.4  73:32.27 gnome-shell
 3077 blockch+  20   0  493492 175100 124928 S   0.7  1.1  31:00.83 Xorg
 4561 blockch+  20   0 3257920 1.323g 128640 S   0.7  8.5  87:56.53 Web Content
 8035 blockch+  20   0   51576   4620   3580 R   0.7  0.0   0:01.51 top
  663 root      20   0  269716   5876   5004 D   0.3  0.0  14:42.71 iio-sensor+
 3238 blockch+  20   0 2745648  16600  12868 S   0.3  0.1  29:42.59 pulseaudio
 4907 blockch+  20   0 2540792 610936 119252 S   0.3  3.7  34:32.05 Web Content
 5388 blockch+  20   0 5109532 816136 147028 S   0.3  5.0  15:03.77 geth
16344 blockch+  20   0  772568  63340  32948 S   0.3  0.4  29:26.35 gnome-term+
    1 root      20   0  225732   9796   7000 S   0.0  0.1   1:00.56 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.13 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:+
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_+
    7 root      20   0       0      0      0 S   0.0  0.0   0:01.05 ksoftirqd/0
    8 root      20   0       0      0      0 I   0.0  0.0   1:20.13 rcu_sched
    9 root      20   0       0      0      0 I   0.0  0.0   0:00.00 rcu_bh
```

Figure 4.3. Four Thread Output of Top Command

Figure 4.4 shows what happens when the program *a.out* is run maxing out six threads (three cores). The overall CPU utilization went up to 79.2%. This is slightly more than three quarters of the total available resources. In this case, the *geth* node was performing an operation in the background that pushed the utilization for that particular process to 23.2% of one core. This accounted for $\frac{23.2}{8} = 2.9\%$ to be added to the 75% expected directly from our program *para*. The remaining 1.2% came from the remainder of processes running.

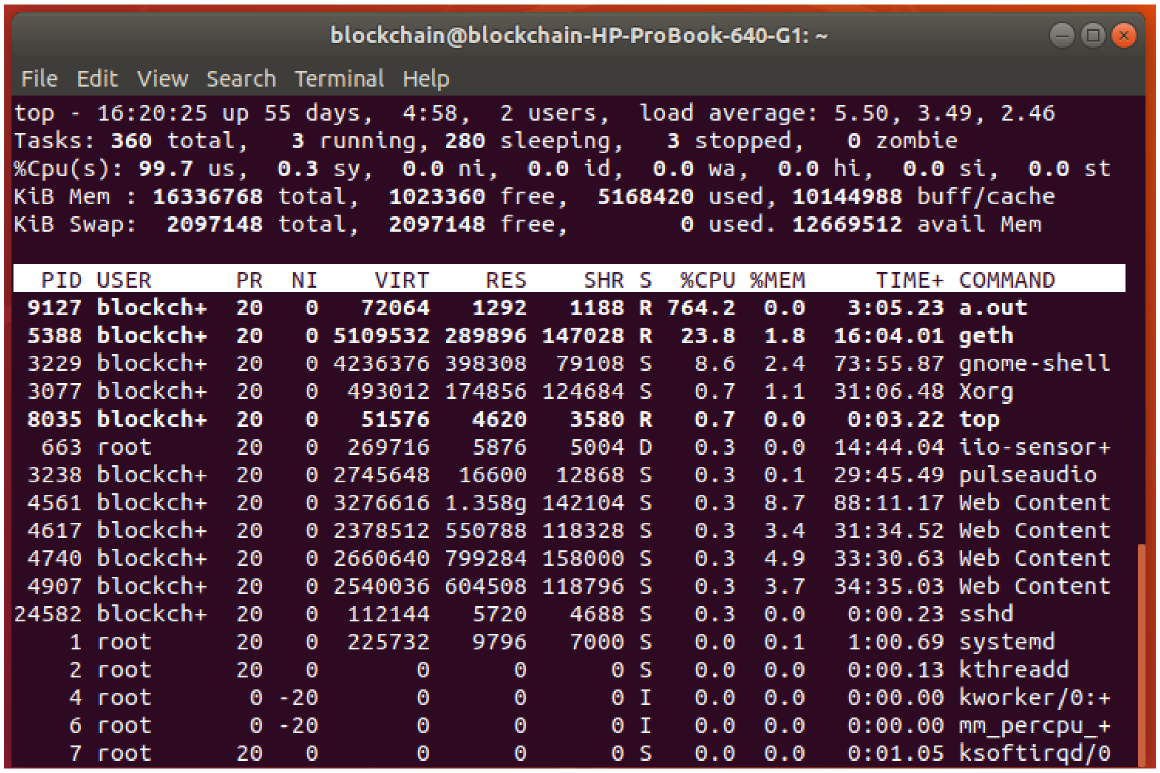

blockchain@blockchain-HP-ProBook-640-G1: ~
File Edit View Search Terminal Help

```
top - 16:19:19 up 55 days,  4:57,  2 users,  load average: 4.19, 2.95, 2.23
Tasks: 361 total,   2 running, 282 sleeping,   3 stopped,   0 zombie
%Cpu(s): 79.2 us,  0.2 sy,  0.0 ni, 20.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16336768 total,  1018896 free,  5173252 used, 10144620 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used. 12664672 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 9005 blockch+  20   0   55672   1380   1276 R 599.0  0.0   2:00.41 a.out
 5388 blockch+  20   0 5109532 289640 147028 S  23.2  1.8  15:51.05 geth
 3229 blockch+  20   0 4234476 398272  79228 S   7.6  2.4  73:51.28 gnome-shell
 3077 blockch+  20   0  493896 176100 125928 S   0.7  1.1  31:05.71 Xorg
  663 root      20   0  269716   5876   5004 D   0.3  0.0  14:43.82 iio-sensor+
 3238 blockch+  20   0 2745648  16600  12868 S   0.3  0.1  29:45.02 pulseaudio
 4561 blockch+  20   0 3269448 1.355g 142104 S   0.3  8.7  88:09.99 Web Content
 4617 blockch+  20   0 2378512 550788 118328 S   0.3  3.4  31:34.31 Web Content
 4740 blockch+  20   0 2660640 799080 158000 S   0.3  4.9  33:30.39 Web Content
 4907 blockch+  20   0 2540036 604320 118796 S   0.3  3.7  34:34.74 Web Content
 8035 blockch+  20   0   51576   4620   3580 R   0.3  0.0   0:02.89 top
    1 root      20   0  225732   9796   7000 S   0.0  0.1   1:00.67 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.13 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:+
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_+
    7 root      20   0       0      0      0 S   0.0  0.0   0:01.05 ksoftirqd/0
    8 root      20   0       0      0      0 I   0.0  0.0   1:20.27 rcu_sched
```

Figure 4.4. Six Thread Output of Top Command

Figure 4.5 shows what happens when the program *a.out* is run maxing out eight threads (four cores). The overall CPU utilization went up to 99.7%. This is slightly less than 100% of our total available resources. As stated earlier, the system only had 8 virtual cores giving us a total of 800% of CPU utilization (based on one thread). Even forcing the program to attempt to consume all of the available resources, the OS is smart enough to realize that some amount of resources must be saved for the other processes running at the same time as the program *a.out*. This leads to the 764.2% figure for *a.out* vice the 800% figure we might have expected.


blockchain@blockchain-HP-ProBook-640-G1: ~
File Edit View Search Terminal Help

```
top - 16:20:25 up 55 days,  4:58,  2 users,  load average: 5.50, 3.49, 2.46
Tasks: 360 total,   3 running, 280 sleeping,   3 stopped,   0 zombie
%Cpu(s): 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16336768 total,  1023360 free,  5168420 used, 10144988 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used. 12669512 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 9127 blockch+  20   0   72064   1292   1188 R 764.2  0.0   3:05.23 a.out
 5388 blockch+  20   0 5109532 289896 147028 R  23.8  1.8  16:04.01 geth
 3229 blockch+  20   0 4236376 398308  79108 S   8.6  2.4  73:55.87 gnome-shell
 3077 blockch+  20   0  493012 174856 124684 S   0.7  1.1  31:06.48 Xorg
 8035 blockch+  20   0   51576   4620   3580 R   0.7  0.0   0:03.22 top
  663 root      20   0  269716   5876   5004 D   0.3  0.0  14:44.04 iio-sensor+
 3238 blockch+  20   0 2745648  16600  12868 S   0.3  0.1  29:45.49 pulseaudio
 4561 blockch+  20   0 3276616 1.358g 142104 S   0.3  8.7  88:11.17 Web Content
 4617 blockch+  20   0 2378512 550788 118328 S   0.3  3.4  31:34.52 Web Content
 4740 blockch+  20   0 2660640 799284 158000 S   0.3  4.9  33:30.63 Web Content
 4907 blockch+  20   0 2540036 604508 118796 S   0.3  3.7  34:35.03 Web Content
24582 blockch+  20   0  112144   5720   4688 S   0.3  0.0   0:00.23 sshd
    1 root      20   0  225732   9796   7000 S   0.0  0.1   1:00.69 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.13 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:+
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_+
    7 root      20   0       0      0      0 S   0.0  0.0   0:01.05 ksoftirqd/0
```

Figure 4.5. Eight Thread Output of Top Command

With our implementation, CPU utilization as a metric has a weakness. If a malicious process is not running exactly when we sample our node, we will not see the anomaly. The examples above work because we were able to somewhat control the execution time by specifying the number to be factored. As a brute force method was used, the larger the number, the longer the execution time. The CIDS that was put into place has a lag time associated with passing data throughout the network and codifying transactions. This can create a scenario where short duration malicious programs can be executed without being detected. Further work is needed to improve this basic system. This provided the impetus to find other metrics that would enable anomaly detection in a more robust fashion.

### 4.1.2 Login Attempts

Thwarting the doorknob rattling attack described in Chapter 2 requires an accurate recording of all login attempts throughout a system. This provided the original inspiration to use login attempts as a metric for anomaly detection. Our difficulty was in finding a way to securely record login data for propagation throughout the network. By default, several Linux distributions already log successful and failed login attempts to a variety of log locations [28]. However, there are two issues to trying to read these logs. The first can be summed up by asking the question: How often would the system administrator check the logs? A doorknob rattling attack can take place quickly [22], and having to read information from a log before sending it could make it difficult to see the attack. Secondly, as noted in Chapter 3, hackers want to erase evidence of their actions [24]. A hacker would want to erase evidence of their login attempts from the logs. Therefore, we concluded that the best way to securely record login information was to make the recording and sending of the information a part of the Linux login process. We decided to use Pluggable Authentication Modules (PAM) to accomplish this task. The flowchart for how login attempts are recorded is shown in Figure 4.6. Each of the steps will be discussed in more detail in the following sections.

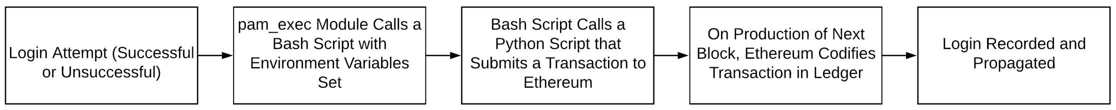

Login Attempt (Successful or Unsuccessful)
pam_exec Module Calls a Bash Script with Environment Variables Set
Bash Script Calls a Python Script that Submits a Transaction to Ethereum
On Production of Next Block, Ethereum Codifies Transaction in Ledger
Login Recorded and Propagated

Figure 4.6. Using PAM to Record and Propagate Login Attempts

## PAM

The Linux-PAM System Administrator Guide summarizes PAM as "a suite of shared libraries that enable the local system administrator to choose how applications authenticate users [29]". PAM is leveraged by applications to ensure that a user is properly authenticated using a set of shared libraries called modules. A complete list of modules can be found in [29]. This functionality allows us to create additional tasks that must be completed in order for the login process to take place. More precisely, we leveraged the *common-auth* configuration file. This configuration file is called in most situations where authentication is required. Listing 4.3 shows the edited version of the file.

Listing 4.3: The *common-auth* Configuration File

```
# /etc/pam.d/common-auth - authentication settings common to
    all services
#
```

```
# This file is included from other service-specific PAM
   config files ,
# and should contain a list of the authentication modules
   that define
# the central authentication scheme for use on the system
# (e.g., /etc/shadow, LDAP, Kerberos, etc.).  The default is
    to use the
# traditional Unix authentication mechanisms.

# here are the per-package modules (the "Primary" block)
auth    [success=2 default=ignore]      pam_unix.so
   nullok_secure

# here's the fallback if no module succeeds
#auth    requisite                       auth_fail.so
auth requisite pam_exec.so /home/blockchain/login_success.sh
    2
auth    requisite                       pam_deny.so

# prime the stack with a positive return value if there isn'
   t one already;
# this avoids us returning an error just because nothing
   sets a success code
# since the modules above will each just jump around
auth    required                        pam_permit.so

# and here are more per-package modules (the "Additional"
   block)
#auth required  pam_example.so
auth    optional                        pam_cap.so
auth optional pam_exec.so /home/blockchain/login_success.sh
   1
# end of pam-auth-update config
```

When a user attempts to login, our system goes through the typical Linux authentication process (*pam_unix*) but makes an additional call to *pam_exec*, which permits the system to run an external command. Note that the call to the *pam_exec* module must be put into the correct PAM service file (in Ubuntu the service file location is */etc/pam.d*). We used the *common-auth* service to call a Bourne Again SHell (Bash) script that interacted with our blockchain client. The *pam_exec* module sets several environmental variables that record several important pieces of information for logging purposes. A subset of these variables is shown in Table 4.1.

Table 4.1. PAM Fields Exported as Environment Variables by *pam_exec*. Source: [29].

| | |
|---|---|
| PAM_RHOST | Remote user attempting to authenticate |
| PAM_RUSER | Remote host that is being authenticated to |
| PAM_SERVICE | Service module that made request |
| PAM_USER | User that made request |
| PAM_TYPE | Type of module (account, auth, password, open-session, close-session) |

By altering the *common-auth* file, we ensured that every time an authentication request was made, it would be logged. Listing 4.4 shows the lines that were added. These two lines covered the two different cases of a login failure and a login success. Based on whether the login was successful, they called the Bash script *login_success.sh* with a different input variable (2 for failure, 1 for success).

Listing 4.4: Lines added to *common-auth*

```
auth requisite pam_exec.so /home/blockchain/login_success.sh
    2
auth optional pam_exec.so /home/blockchain/login_success.sh
    1
```

Additionally, this method not only includes initial login attempts but any other service that requires user authentication, such as the use of *sudo* to escalate privilege. The *pam_exec* module allowed us to run an external Bash script that had access to all of these environment variables.

**Bash Script *login_success.sh***

A cursory explanation of the Bash script *login_success.sh* is required to understand the output in Section 4.2. Listing 4.5 contains the text of *login_success.sh*.

Listing 4.5: *login_success.sh*

```
#!/bin/bash
d=$(date)
py="/usr/bin/python3"
loc="/home/blockchain/paperNet/login.py"

case $1 in
1)
var="Authentication Successful"
eval $py $loc $PAM_USER $PAM_TYPE $PAM_SERVICE $PAM_RUSER
   $PAM_RHOST $d $var;; #>> /home/blockchain/out.txt;;
2)
var="Authentication Failure"
eval $py $loc $PAM_USER $PAM_TYPE $PAM_SERVICE $PAM_RUSER
   $PAM_RHOST $d $var;;
esac
```

The *pam_exec* module passed either a 1 or a 2 based success or failure. Those cases are represented in the Bash script by the *case* statement. This Bash script calls a Python script *login.py* (Appendix B.1.4) using the Bash command *eval*. The script *login_success.sh* acts as a bridge between *pam_exec* and the Python script by passing all of the correct variables as well as the timestamp from the *date* command.

## 4.2 Testbed Implementation and Software Selection

Our goal was to create a small network that would be able to share information using a blockchain client. Our network architecture is shown in Figure 4.7. Our test bed consisted of two Linux Ubuntu 18.04.1 systems with the Ethereum client running on both. The version of Ubuntu is important as it determines the default version of Python available. While this can be changed either via installation or via a virtual environment, we found it easier when a suitable version of Python was natively installed with the OS. In this case, one of the Python libraries that we leveraged, *web3.py*, only works with Python 3.5 and later [30]. We ran into issues using the *web3.py* library using the Ubuntu 16.04 distribution with Python 3.5.1 as the default Python 3 distribution. These issues resolved when we migrated to Ubuntu 18.04.1, which by default uses Python 3.6.5.

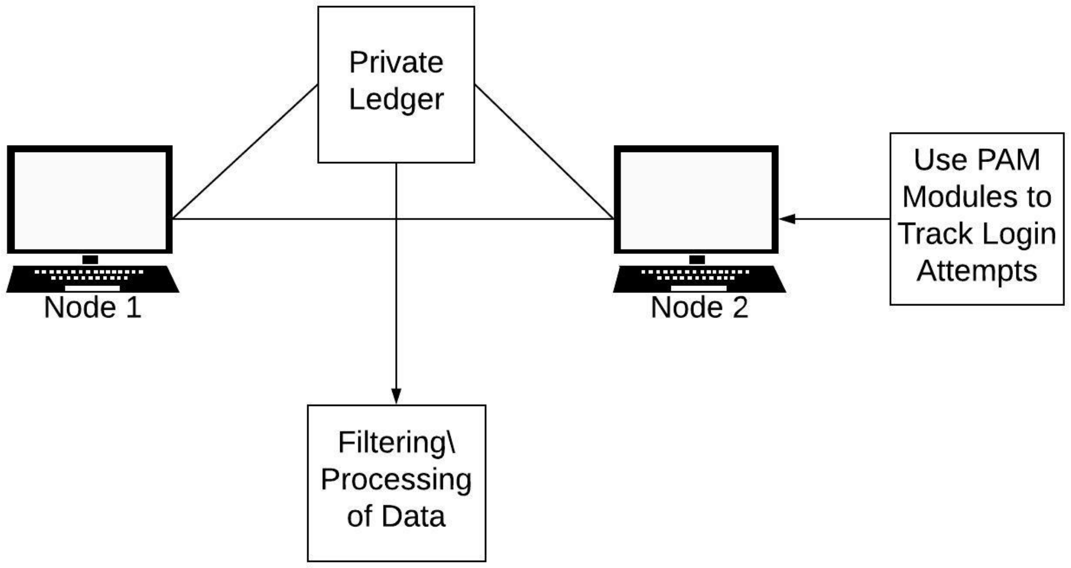

Private Ledger
Node 1
Node 2
Use PAM Modules to Track Login Attempts
Filtering\ Processing of Data

Figure 4.7. CIDS Set Up Used in Experiments

### 4.2.1 Blockchain Client: Ethereum

We leveraged a number of commercially available and open-source products to implement our proof-of-concept. We considered two different products for our blockchain client, Ethereum [31] and Hyperledger [32]. We chose Ethereum as it supported our blockchain-based distributed ledger via the *testnet* functionality.

Ethereum supports a variety of clients in different programming languages. These options are shown in Table 4.2.

Table 4.2. Ethereum Clients. Source: [33].

| Client | Language | Developers | Latest release |
|---|---|---|---|
| go-ethereum | Go | Ethereum Foundation | go-ethereum-v1.4.18 |
| Parity | Rust | Ethcore | Parity-v1.4.0 |
| cpp-ethereum | C++ | Ethereum Foundation | cpp-ethereum-v1.3.0 |
| pyethapp | Python | Ethereum Foundation | pyethapp-v1.5.0 |
| ethereumjs-lib | Javascript | Ethereum Foundation | ethereumjs-lib-v3.0.0 |
| Ethereum(J) | Java | <ether.camp> | ethereumJ-v1.3.1 |
| ruby-ethereum | Ruby | Jan Xie | ruby-ethereum-v0.9.6 |
| ethereumH | Haskell | BlockApps | no Homestead release yet |

The Ethereum documentation states that the Go and Rust (Parity) implementations are the most popular [33]. We leveraged the standard *go-ethereum* client and the private test network functionality for all experiments. The *go-ethereum* client is referred to as *geth*. A detailed guide on installation of Go, an Ethereum node, *geth*, and specific implementation directions is provided in Apprendix A.

### Interacting with the Ethereum Client

We utilized two different methods to interact with an existing Ethereum node, the JavaScript command line interface (CLI) and a Python interface using the *web3.py* library.

**JavaScript Command Line Interface:** *Geth* is the command-line interface (CLI) for interacting with an Ethereum node. Command line instructions can be given in the JavaScript language via the *Web3.js* library. The full documentation for this library can be found at [34]. The *Web3.js* library covers many core functions of Ethereum including mining, sending and receiving transactions, querying the state of the *testnet*, and node management. The code snippet in Listing 4.6 illustrates some of these core functions. This snippet unlocks an already created account with a plaintext password and initializes a transaction to be sent to another account. A mining operation is then started for a duration of three seconds.

Listing 4.6: JavaScript Example: Interacting with Ethereum Node

```
web3.personal.unlockAccount(web3.personal.listAccounts[0],"
    test1",150);
```

```
toAddr = "0x842686d96bbdfd540293622d17fa8eb1d1604b0a";

transData = web3.fromAscii("Text for a Transaction");

web3.eth.sendTransaction({to: toAddr, from: web3.eth.
   coinbase, value: web3.toWei(1, "ether"), data: transData
   });

web3.miner.start(1);

setTimeout(function() {
   web3.miner.stop();
}, (3 * 1000));
```

**Python Interface:** While the JavaScript CLI is very useful, it has some shortcomings. We needed our operating system to be able to interact with Ethereum. The JavaScript CLI was unable to support that requirement. This led us to use the *Web3.py* library written in the Python programming language, which allowed various system components to talk to Ethereum. The full documentation can be found at [35]. The code snippet in Listing 4.7 provides the exact functionality as the snippet in Listing 4.6.

Listing 4.7: Python Example: Interacting with Ethereum Node

```
from web3.auto import w3
import time

w3.personal.unlockAccount(w3.eth.accounts[0],"test1", 150)

toAddr = w3.toChecksumAddress('0
   x842686d96bbdfd540293622d17fa8eb1d1604b0a')

transData = 'Text␣for␣a␣Transaction'
transData = "0x" + "".join(hex(ord(c))[2:] for c in
    transData.strip('\n'))

a = w3.eth.sendTransaction({'to': toAddr, 'from': w3.eth.
    coinbase, 'value': w3.toWei(1, "ether"), 'data':
    transData})

w3.miner.start(1)
time.sleep(3)
w3.miner.stop()
```

### 4.2.2 Transaction Structure in Ethereum

In Section 2.3, we discuss the idea of a transaction in a blockchain environment. In essence, a transaction can be thought of as a data package being passed between two participants. In Ethereum, the primary purpose of a transaction is to transfer money from one party to another. The structure of an Ethereum transaction is presented in Table 4.3.

In our system, we repurposed this structure to instead contain the alert information that was to be propagated throughout the network. For example, using the login information passed by *pam_exec* and our Bash script from Listing 4.5, a node in our CIDS submits a transaction wherein the *data* field contains a hex string encoding the user, service, time, and type of authentication. In another use case, we use the data field to send information about CPU utilization. This hex string contains the timestamp and CPU data. Once a mining operation takes place to produce a new block, either type of transaction is codified into the ledger and can be viewed by any node in the CIDS. For our specific implementation, while all other fields in the transaction were included, only the data field was important for the operation of the CIDS.

Table 4.3. Fields in an Ethereum Transaction. Adapted from [36].

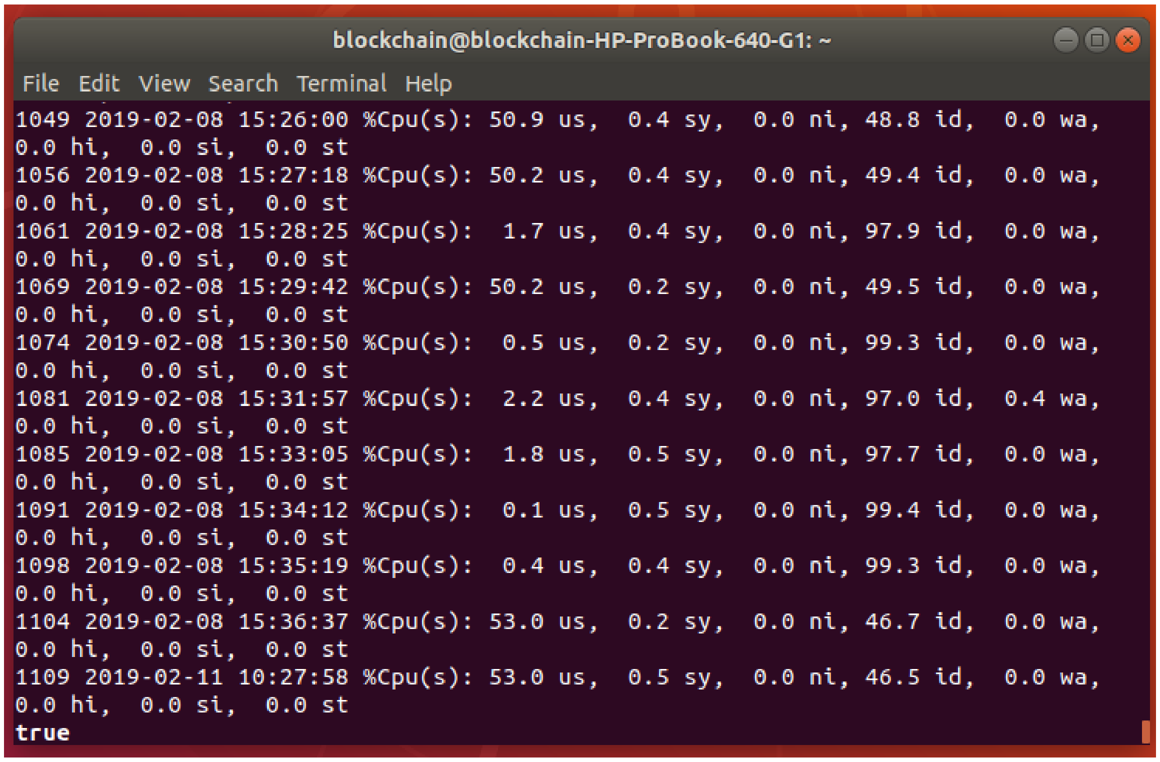
| Field | Data Type | Description |
|---|---|---|
| from | String\|Number | The address for the sending account. [Can be an address or a local] wallet. |
| to (optional) | String | The destination address of the message, left undefined for a contract-creation transaction |
| value (optional) | Number\|String\|BN\|BigNumber | The value transferred for the transaction in wei, also the endowment if it's a contract-creation transaction |
| gas | Number | The amount of gas to use for the transaction (unused gas is refunded) |
| gasPrice | Number\|String\|BN\|BigNumber | The price of gas for this transaction in wei |
| data (optional) | String | Either a ABI byte string containing the data of the [function call] on a contract, or in the case of a contract-creation transaction, the [initialization] code |
| nonce (optional) | Number | Integer of a nonce. This allows the overwriting of pending transactions that use the same nonce |

## 4.3 Results

This section contains the results from various experiments crafted using both of the metrics discussed in Section 4.1. Additionally, a scenario was modeled and executed on the test bed. Data from that scenario was collected and shared with a collaborator who applied a statistical analysis algorithm, resulting in the identification of an anomaly [37].

### 4.3.1 CPU Utilization

We designed a simple experiment to confirm that our CIDS setup would be able to accurately record CPU utilization information. Our node would sample its CPU utilization every minute and record the result. Separately, another program (*para*) was running in the background. The purpose of *para* was to run the factorization process with four threads at random intervals causing CPU spikes. The results are shown in Figure 4.8. Our system was able to record what was occurring on the machine at a specific point in time. We could have used CPU utilization over 50% as a threshold as a trigger for a specific action, such as shutting down the target machine or killing the specific process that was responsible for the high CPU utilization. This threshold, much like in the case of login attempts, is flexible based on system needs.


blockchain@blockchain-HP-ProBook-640-G1: ~
File Edit View Search Terminal Help

```
1049 2019-02-08 15:26:00 %Cpu(s): 50.9 us,  0.4 sy,  0.0 ni, 48.8 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
1056 2019-02-08 15:27:18 %Cpu(s): 50.2 us,  0.4 sy,  0.0 ni, 49.4 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
1061 2019-02-08 15:28:25 %Cpu(s):  1.7 us,  0.4 sy,  0.0 ni, 97.9 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
1069 2019-02-08 15:29:42 %Cpu(s): 50.2 us,  0.2 sy,  0.0 ni, 49.5 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
1074 2019-02-08 15:30:50 %Cpu(s):  0.5 us,  0.2 sy,  0.0 ni, 99.3 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
1081 2019-02-08 15:31:57 %Cpu(s):  2.2 us,  0.4 sy,  0.0 ni, 97.0 id,  0.4 wa,
0.0 hi,  0.0 si,  0.0 st
1085 2019-02-08 15:33:05 %Cpu(s):  1.8 us,  0.5 sy,  0.0 ni, 97.7 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
1091 2019-02-08 15:34:12 %Cpu(s):  0.1 us,  0.5 sy,  0.0 ni, 99.4 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
1098 2019-02-08 15:35:19 %Cpu(s):  0.4 us,  0.4 sy,  0.0 ni, 99.3 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
1104 2019-02-08 15:36:37 %Cpu(s): 53.0 us,  0.2 sy,  0.0 ni, 46.7 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
1109 2019-02-11 10:27:58 %Cpu(s): 53.0 us,  0.5 sy,  0.0 ni, 46.5 id,  0.0 wa,
0.0 hi,  0.0 si,  0.0 st
true
```

Figure 4.8. CPU Utilization Output in Blockchain Ledger

### 4.3.2 Login Attempts

With the CIDS in place, we were able to capture a variety of different authentication requests including login attempts. Table 4.4 shows output from the blockchain ledger for *gdm-password*, which occurs when one attempts to log in from the GUI.

Table 4.4. Login from GUI Example

| $PAM_USER | $PAM_TYPE | $PAM_SERVICE | Date | Success/Failure |
|---|---|---|---|---|
| blockchain | auth | gdm-password | Sun Jul 21 9:20:34 | Success |
| blockchain | auth | gdm-password | Sun Jul 21 9:26:53 | Success |

Table 4.5 shows two other services, *sudo* and *su* being logged. There is an additional field in use with these commands. This is the *$PAM_RUSER* field because both *sudo* and *su* can change the context of the user. PAM modules need to determine both the identity of the user who requests a service, and also the identity of the service granter [29]. Oftentimes, these two fields will not be the same. Envision a scenario where one user uses the *su* command to switch to another user. Listing 4.8 is output for the user *vikram* using the *su* to switch to user *blockchain*. This is an example of where the *$PAM_RUSER* field changes.

Listing 4.8: Example Using *su* to Switch Between Users

```
vikram auth su blockchain Sun Jul 21 09:27:29 PDT 2019
    Authentication Successful
```

Table 4.6 demonstrates the case of an external host using the *ssh* command to log into our CIDS node remotely. This example uses the *$PAM_RHOST* field, which contains information about the requesting host. In this case, an external agent (172.20.157.112) successfully logged into our CIDS node as user *blockchain* via the *ssh blockchain@172.20.157.111*

Table 4.5. Authentication Requests by *sudo* and *su*

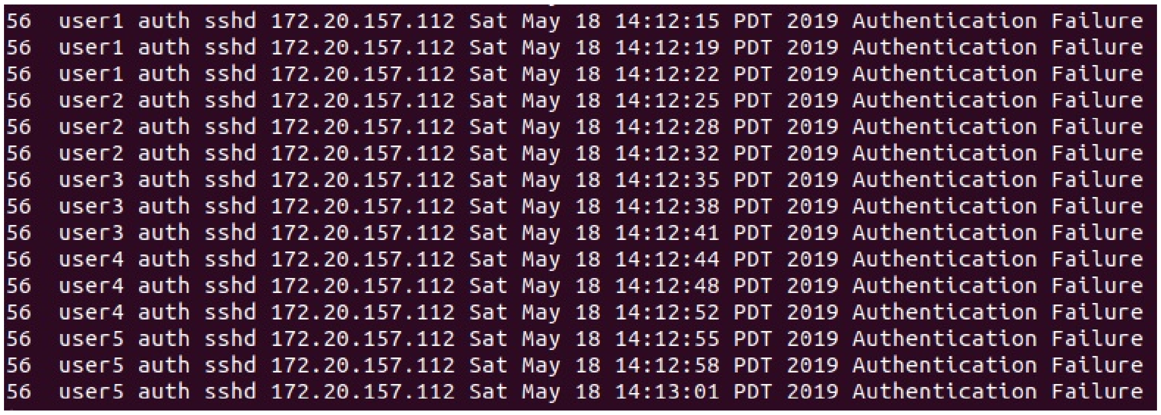
| $PAM_USER | $PAM_TYPE | $PAM_ SERVICE | $PAM_RUSER | Date | Success /Failure |
|---|---|---|---|---|---|
| blockchain | auth | sudo | blockchain | Sun Jul 21 9:28:00 | Success |
| blockchain | auth | su | blockchain | Sun Jul 21 9:29:45 | Success |

command. This is an incredibly important use case as the doorknob rattling attack typically involves remote users attempting to penetrate a target network [22].

Table 4.6. Login Attempt by a Remote User

| $PAM_USER | $PAM_TYPE | $PAM_ SERVICE | $PAM_RHOST | Date | Success /Failure |
|---|---|---|---|---|---|
| blockchain | auth | sshd | 172.20.157.112 | Sun Jul 21 9:29:04 | Success |

Using the technique developed to record external login attempts evidenced by Table 4.6, we simulated a doorknob rattling attack against one of our machines. In this test, the victims were different user accounts on a single machine. Output from our CIDS ledger is shown in Figure 4.9. The attacking machine (172.20.157.112) attempted to login to each victim user account three times via secure shell (SSH). With our modified *common-auth* file in place, the CIDS recorded each login attempt as a separate transaction in the blockchain as depicted in Figure 4.9. The logged information is summarized in Table 4.7.



```
56  user1 auth sshd 172.20.157.112 Sat May 18 14:12:15 PDT 2019 Authentication Failure
56  user1 auth sshd 172.20.157.112 Sat May 18 14:12:19 PDT 2019 Authentication Failure
56  user1 auth sshd 172.20.157.112 Sat May 18 14:12:22 PDT 2019 Authentication Failure
56  user2 auth sshd 172.20.157.112 Sat May 18 14:12:25 PDT 2019 Authentication Failure
56  user2 auth sshd 172.20.157.112 Sat May 18 14:12:28 PDT 2019 Authentication Failure
56  user2 auth sshd 172.20.157.112 Sat May 18 14:12:32 PDT 2019 Authentication Failure
56  user3 auth sshd 172.20.157.112 Sat May 18 14:12:35 PDT 2019 Authentication Failure
56  user3 auth sshd 172.20.157.112 Sat May 18 14:12:38 PDT 2019 Authentication Failure
56  user3 auth sshd 172.20.157.112 Sat May 18 14:12:41 PDT 2019 Authentication Failure
56  user4 auth sshd 172.20.157.112 Sat May 18 14:12:44 PDT 2019 Authentication Failure
56  user4 auth sshd 172.20.157.112 Sat May 18 14:12:48 PDT 2019 Authentication Failure
56  user4 auth sshd 172.20.157.112 Sat May 18 14:12:52 PDT 2019 Authentication Failure
56  user5 auth sshd 172.20.157.112 Sat May 18 14:12:55 PDT 2019 Authentication Failure
56  user5 auth sshd 172.20.157.112 Sat May 18 14:12:58 PDT 2019 Authentication Failure
56  user5 auth sshd 172.20.157.112 Sat May 18 14:13:01 PDT 2019 Authentication Failure
```

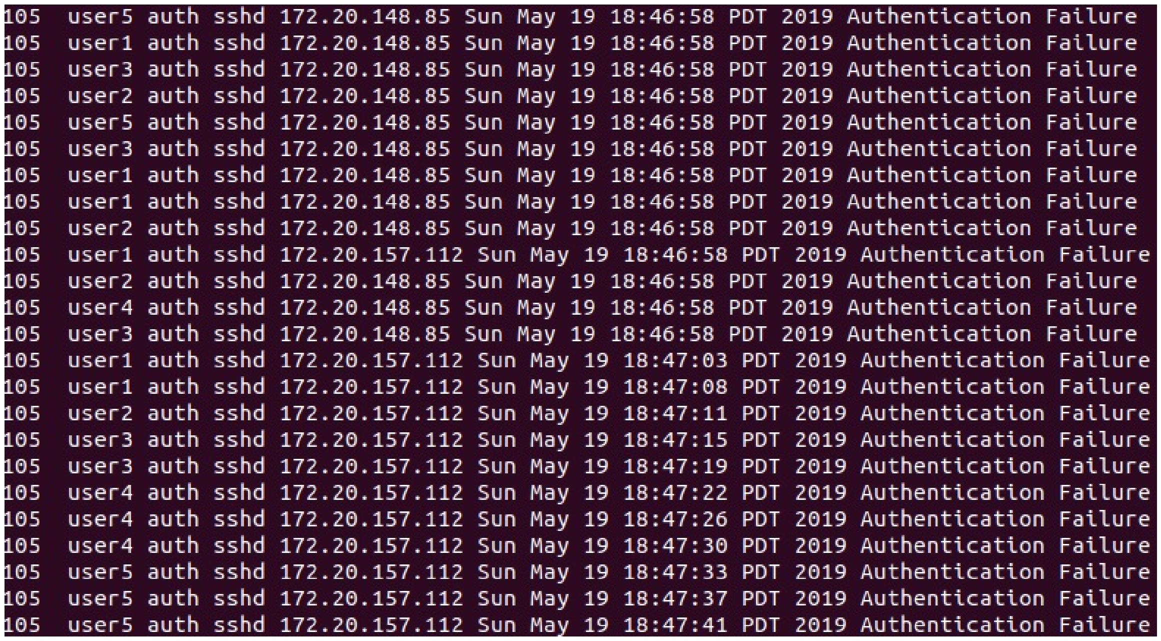
Figure 4.9. Doorknob Rattling Attack in Ledger

Table 4.7. Summary of Doorknob Rattling Attack Events, Single Attacker

| User | IP Address of Request | Number of Login Attempts | Duration of Attack (seconds) |
|---|---|---|---|
| user1 | 172.20.157.112 | 3 | 7 |
| user2 | 172.20.157.112 | 3 | 10 |
| user3 | 172.20.157.112 | 3 | 9 |
| user4 | 172.20.157.112 | 3 | 11 |
| user5 | 172.20.157.112 | 3 | 9 |

This attack took place over 46 seconds, with 15 total login attempts across five different accounts. As this ledger is visible from any CIDS node, extraction of these login attempts for follow on analysis toward anomaly detection becomes possible from any of the distributed participants. Although beyond the scope of the current experiment, our system might have then responded by blocking any subsequent traffic from the attack-related internet protocol (IP) address.

In a second experiment, multiple attackers acted against our victim machine. Output from the ledger in this case is shown in Figure 4.10. This scenario contained two attackers that acted in differing ways as revealed by the detail the ledger contains. The first attacker (172.20.148.85) made all of its SSH requests simultaneously whereas the second attacker (172.20.157.112) made its SSH requests in sequence. This simple difference can reveal some information about the attacking machine. The first attacker was a Windows machine using the puTTY program for its SSH requests. The second attacker was another Ubuntu machine using OpenSSH to conduct its login attempts. The differing nature of these programs can be seen from the log entries in Figure 4.10. A summary of the attack is shown in Table 4.8. There were a total of 24 login attempts by two attackers across a time interval of 43 seconds.



```
105  user5 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user1 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user3 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user2 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user5 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user3 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user1 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user1 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user2 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user1 auth sshd 172.20.157.112 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user2 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user4 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user3 auth sshd 172.20.148.85 Sun May 19 18:46:58 PDT 2019 Authentication Failure
105  user1 auth sshd 172.20.157.112 Sun May 19 18:47:03 PDT 2019 Authentication Failure
105  user1 auth sshd 172.20.157.112 Sun May 19 18:47:08 PDT 2019 Authentication Failure
105  user2 auth sshd 172.20.157.112 Sun May 19 18:47:11 PDT 2019 Authentication Failure
105  user3 auth sshd 172.20.157.112 Sun May 19 18:47:15 PDT 2019 Authentication Failure
105  user3 auth sshd 172.20.157.112 Sun May 19 18:47:19 PDT 2019 Authentication Failure
105  user4 auth sshd 172.20.157.112 Sun May 19 18:47:22 PDT 2019 Authentication Failure
105  user4 auth sshd 172.20.157.112 Sun May 19 18:47:26 PDT 2019 Authentication Failure
105  user4 auth sshd 172.20.157.112 Sun May 19 18:47:30 PDT 2019 Authentication Failure
105  user5 auth sshd 172.20.157.112 Sun May 19 18:47:33 PDT 2019 Authentication Failure
105  user5 auth sshd 172.20.157.112 Sun May 19 18:47:37 PDT 2019 Authentication Failure
105  user5 auth sshd 172.20.157.112 Sun May 19 18:47:41 PDT 2019 Authentication Failure
```

Figure 4.10. Multi-attacker Doorknob Rattling Attack in Ledger

Table 4.8. Summary of Doorknob Rattling Attack Events, Multi-attacker

| User | IP Address of Request | Number of Login Attempts |
|---|---|---|
| user1 | 172.20.157.112 | 3 |
| user2 | 172.20.157.112 | 1 |
| user3 | 172.20.157.112 | 2 |
| user4 | 172.20.157.112 | 3 |
| user5 | 172.20.157.112 | 3 |
| user1 | 172.20.148.85 | 3 |
| user2 | 172.20.148.85 | 3 |
| user3 | 172.20.148.85 | 3 |
| user4 | 172.20.148.85 | 1 |
| user5 | 172.20.148.85 | 2 |

In both experiments, the timestamps of the transactions in the blockchain indicate the attack was permanently recorded in the CIDS distributed ledger within seconds of attack initiation. This rapid indication of the event and protection of related data could facilitate action in time to protect other network nodes in near real time and also ensures that a trustworthy forensic record of the attack events is preserved for follow on analysis. Although a specialized system might improve functionality and efficiency, we have concluded that Ethereum's unaltered *testnet* blockchain client integrates smoothly with existing Linux system architecture to accomplish data ingest toward intrusion detection with minimal adjustment or overhead and fidelity sufficient for an initial investigation.

## 4.4 Detecting an Anomaly: Thwarting a Doorknob Rattling Attack

The purpose of our proposed CIDS architecture was to record data that could be used to detect anomalies. Ultimately, collecting data is not enough. That data has to be processed and analyzed in order to find potential threats. Working in collaboration with LCDR Stephanie Pendino [37], we designed an experiment in which we were able to demonstrate detecting a doorknob rattling attack using statistical analysis and machine learning techniques.

We used the *cron* software utility to schedule a job every five minutes. This job represented steady-state login traffic over a defined interval. The time interval could have been made every hour or every day by adjusting the *cron* parameters. The *cron* job executed the script shown in Listing 4.9

Listing 4.9: Bash Script for Anomaly Detection Scenario

```
#!/bin/bash

FLOOR=20;
CEILING=30;
RANGE=$(($CEILING-$FLOOR+1));
NUMLOG=$RANDOM;
let "NUMLOG %= $RANGE";
NUMLOG=$(($NUMLOG+$FLOOR));
echo $NUMLOG
```

```
FLOOR=1;
CEILING=5;
USER=$RANDOM;
let "USER %= $RANGE";
USER=$(($USER+$FLOOR));
echo "user"$USER

for j in `seq 1 $NUMLOG`;
do
sshpass -p "wrong pass" ssh -o StrictHostKeyChecking=no "
   user"$USER@172.20.157.111 &
done
```

We ensured that between 20 and 30 logins were attempted of a random user on our victim machine in five-minute intervals. We let this traffic pattern continue for several iterations before introducing a spike in login attempts (done by adjusting the *NUMLOG* parameter in Listing 4.9 to a value larger than 30). Additionally, on the CIDS node, we adjusted our login Python script to also vary the transaction cost of each record of a login attempt (adjusting the *gas* parameter) to provide another variable to analyze for potential anomalies. The idea is that the more transactions take place, the more transaction costs will be paid during that time duration. This code is provided in Appendix B.1.4. Table 4.9 contains the data from the experiment. Some of the transactions that occurred at the borderline between intervals were categorized into the wrong intervals. This explains the transaction values of 19, which were below the *NUMLOG* parameter specified. The processing algorithm was unaware that 30 was the threshold value.

Table 4.9. Experimental Transaction Data with Gas Costs

| Time | Number of Transactions | Average Gas Cost | Total Gas Cost |
|---|---|---|---|
| 10:43:34 | 25 | 1252000 | 31300000 |
| 10:48:34 | 20 | 1295000 | 25900000 |
| 10:53:34 | 22 | 1250000 | 27500000 |
| 10:58:34 | 25 | 1288000 | 32200000 |
| 11:03:34 | 19 | 1215789.47 | 23099999.93 |
| 11:08:34 | 21 | 1285714.29 | 27000000.09 |
| 11:13:34 | 24 | 1166666.67 | 28000000.08 |
| 11:18:34 | 28 | 1310714.29 | 36700000.12 |
| 11:23:34 | 19 | 1268421.05 | 24099999.95 |
| 11:28:34 | 22 | 1268181.82 | 27900000.04 |
| 11:33:34 | 36 | 1225581.395 | 44200000 |
| 11:38:34 | 41 | 1255882.353 | 51200000 |
| 11:43:34 | 21 | 1266666.67 | 26600000.07 |
| 11:48:34 | 21 | 1319047.62 | 27700000.02 |
| 11:53:34 | 33 | 1275757.576 | 42100000 |
| 11:58:34 | 27 | 1251851.85 | 33799999.95 |

By inspection, there were three time intervals (11:33:34, 11:38:34, 11:53:34) when the number of transactions eclipsed the threshold (30). Figures 4.11 and 4.12 show histograms and boxplots for both the number of transactions and transaction (gas) costs.

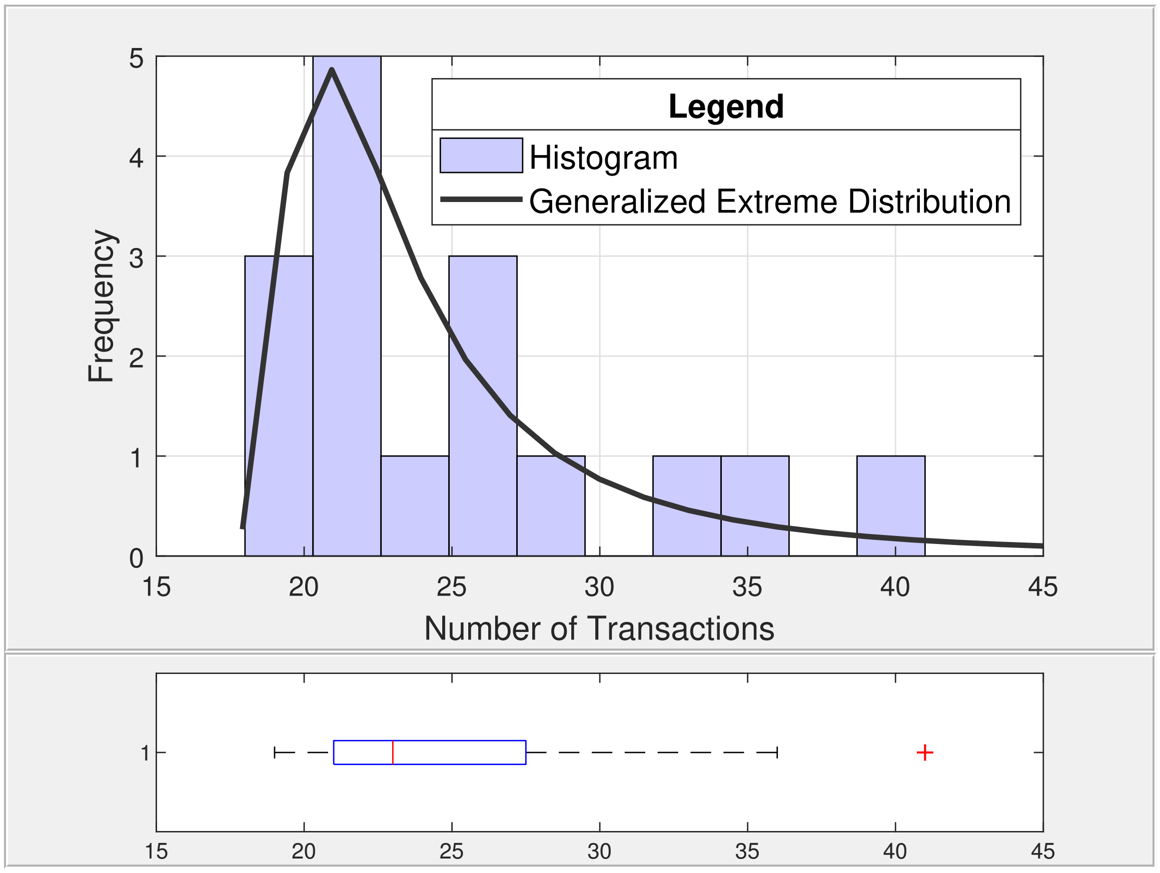


Figure 4.11. Transaction Histogram for Doorknob Rattling Scenario. Source: [37].

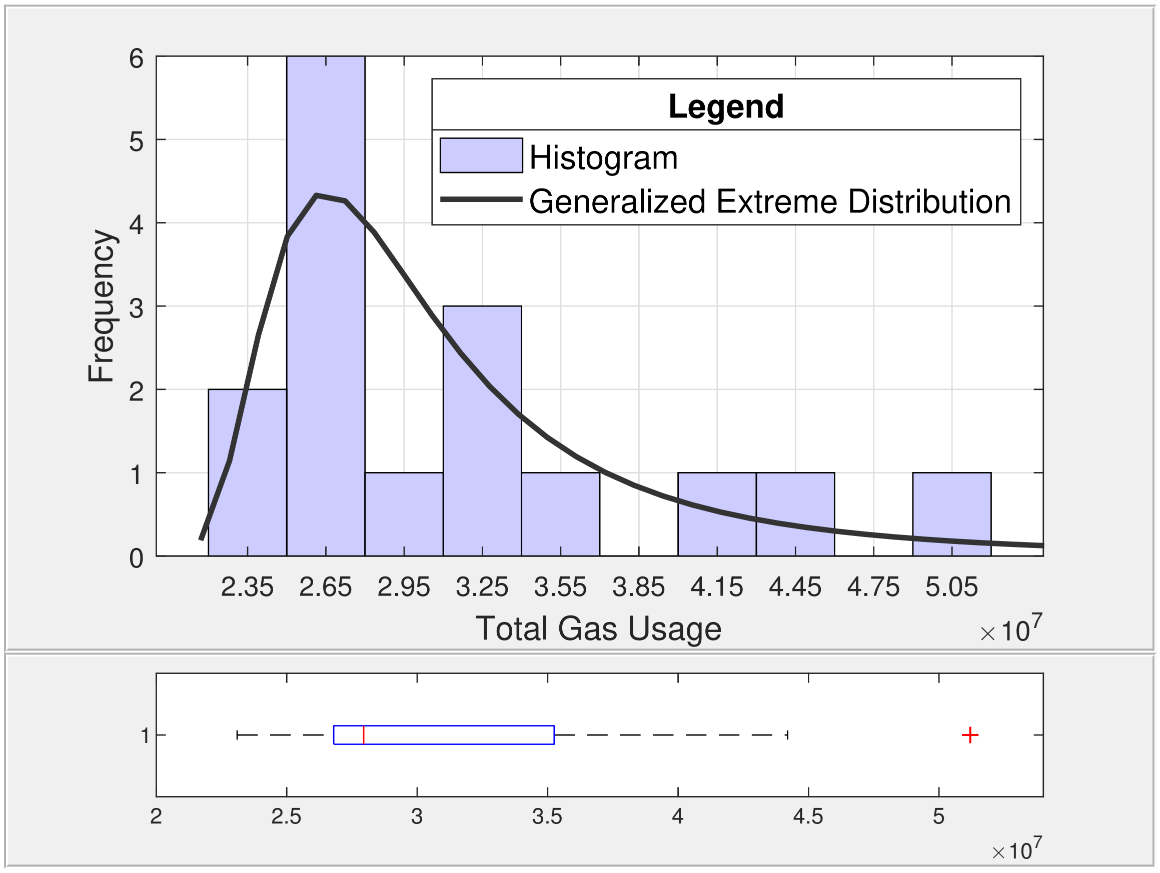


Figure 4.12. Gas Histogram for Doorknob Rattling Scenario. Source [37].

The attack intervals are clearly highlighted in the histograms and skew the distribution. From a statistical perspective, only the interval where 41 transactions occurred was flagged as an anomaly. We used the standard definition of an outlier: a point that is more than 1.5 times the interquartile range below the first quartile or above the third quartile [38]. Although the inability to detect all three attacks was problematic, this statistical method did correctly identify that there was an anomaly, which would lead a system administrator to further investigate.

We then used an unsupervised machine learning technique to identify clusters in our data. The *k*-means method was used to identify numbers of clusters in the data. This technique takes a set of data and divides it into *k* different partitions on the basis of similarities. More specific information on *k*-means can be found in [39] and [40]. Based on prior analyses of Ethereum, we determined that clustering the data into two sets was a good starting point [37]. Those results are shown in Figure 4.13. This clustering was insufficient to answer which transaction intervals were suspicious.

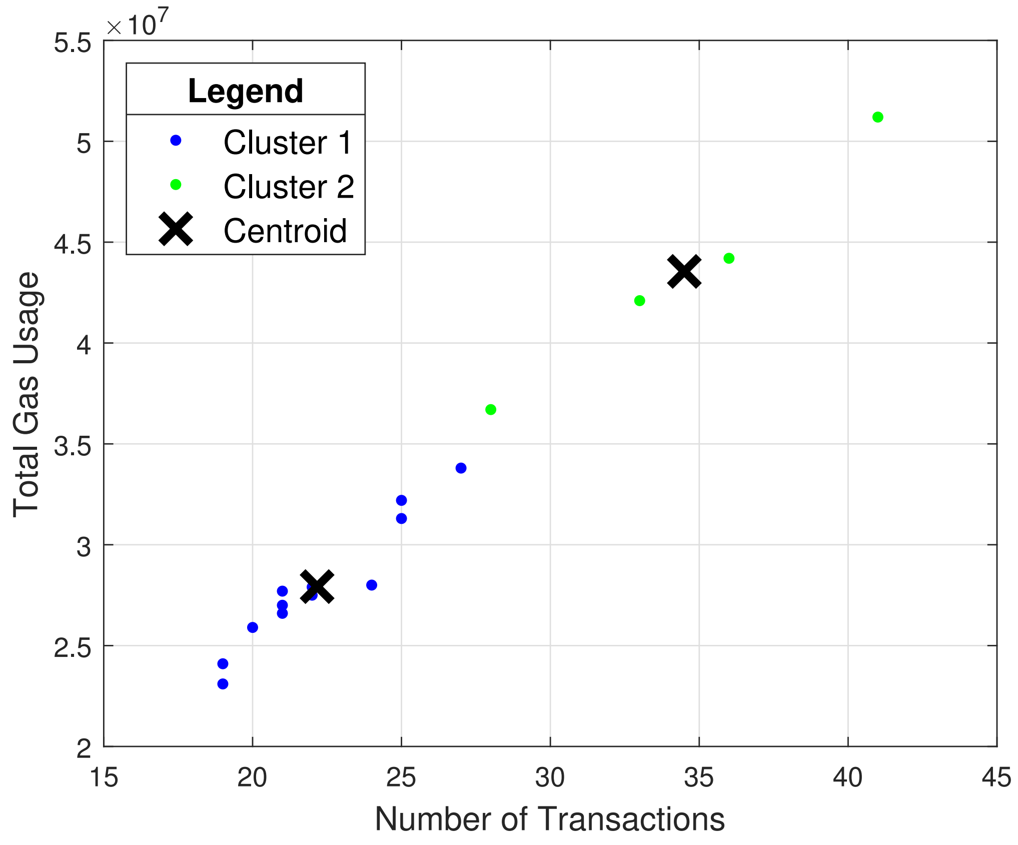


Figure 4.13. Initial Clustering for Doorknob Ratting Scenario. Source: [37].

We then used a dendrogram to further refine our clusters. A dendrogram is a tree diagram that represents the hierarchy of similarities between groups of data points [41]. The similarities are determined by the Euclidean distance between data points. More information of dendrograms, their applications, and methods to create them are covered in [41] and [42]. Figure 4.14 is a display of the dendrogram for our data set. The dendrogram shows three clusters of interest and correctly identifies a similarity between the three attack intervals. This provides us with a model of what behavior in our network might be considered normal (blue) vice suspicious (green and red). It also further informs the *k*-means algorithm.

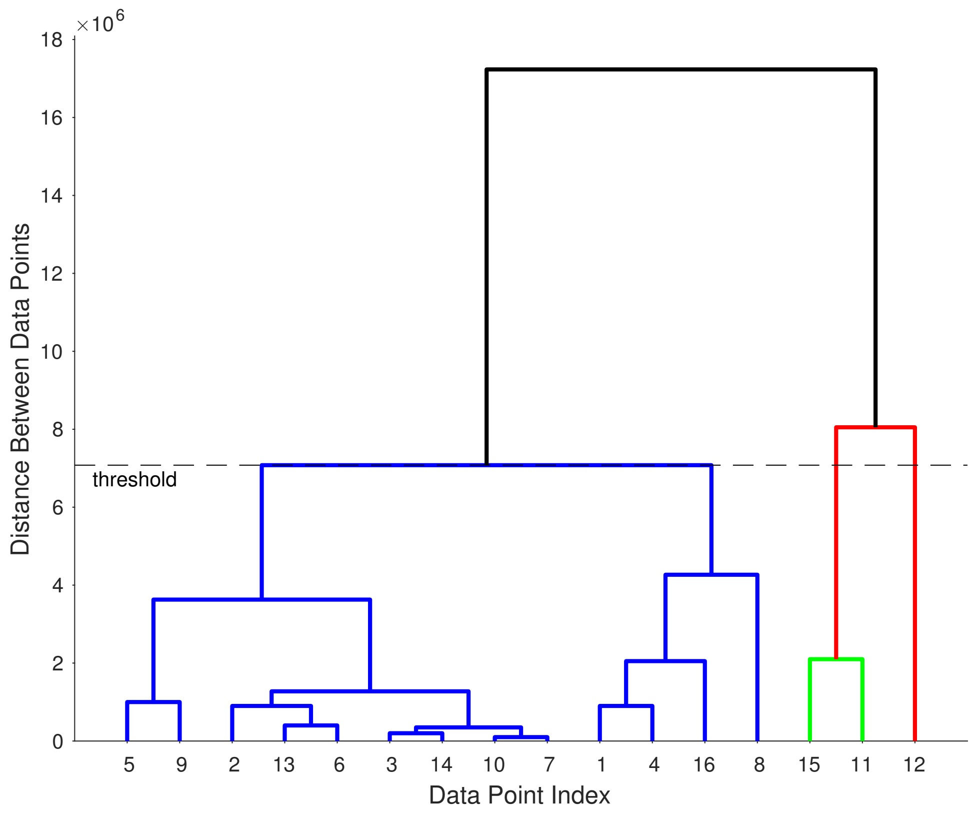


Figure 4.14. Dendrogram for Doorknob Rattling Scenario. Source: [37].

Figure 4.15 shows the results of the *k*-means algorithm with the information gleaned from the dendrogram. Once our data had been refined using the hierarchical clustering approach, it became clear that three clusters was the optimal number of clusters for the *k*-means algorithms. These clusters corresponded to normal behavior, suspicious behavior, and our sole statistical outlier. Our second clustering in Figure 4.15 correctly identifies the attack intervals that would be of interest to a system administrator. In addition, it is visually clear that there is a demarcation at 30 transactions. A number of transactions above 30 is suspicious or an outlier and a number of transacions under 30 is normal. This is significant because both the statistical approach and the unsupervised machine learning approach did not have a priori information about the threshold that we set during the design phase of the experiment. These techniques were able to identify that threshold and could be used in a network to establish baseline behavior that would form the basis for anomaly detection.

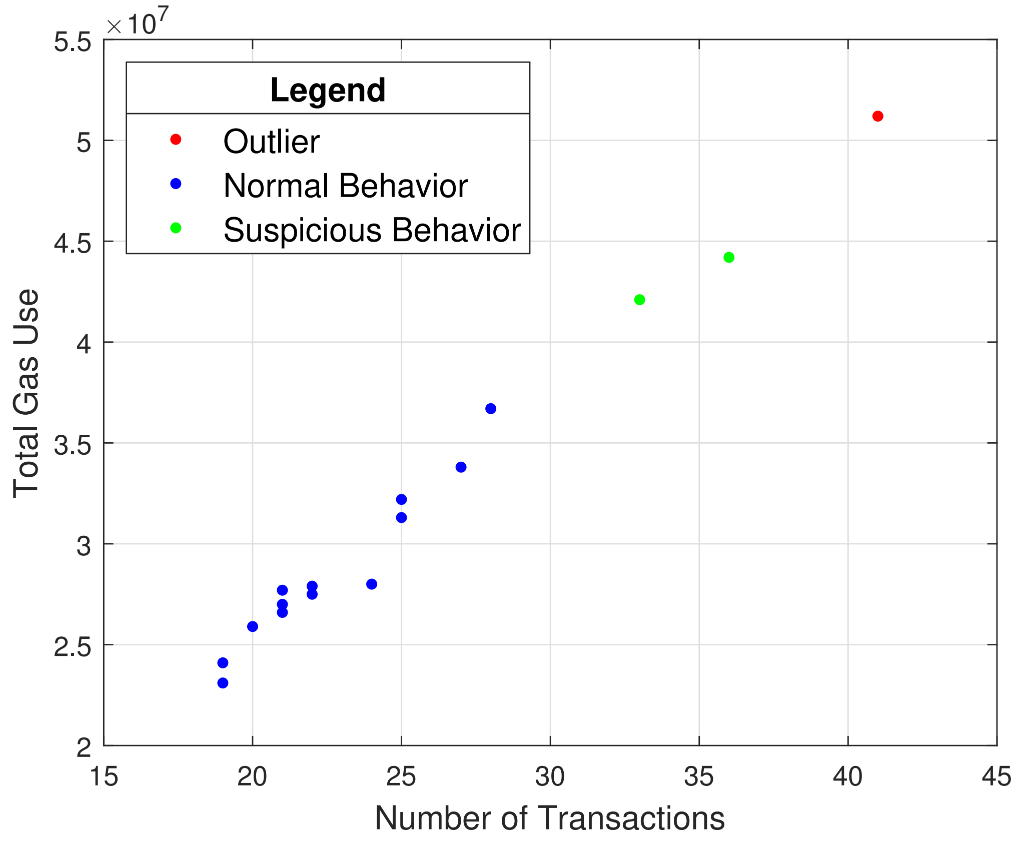


Figure 4.15. Second Clustering for Doorknob Ratting Scenario. Source: [37].

## 4.5 Summary

The general blockchain solution proposed in Chapter 3 was implemented in this chapter in small scale with specific implementation details provided. The choices of hardware, blockchain client, metrics, and software interfaces were explained and analyzed. A set of experiments was conducted to present the methodology of logging potentially anomalous data. Both CPU utilization and login attempts were discussed.

Finally, a scenario was devised involving a fictitious network and attackers conducting a doorknob rattling attack. This data was then analyzed using statistical techniques and machine learning leading to the detection of anomalies.

# CHAPTER 5: Conclusions

The sharing of information between cyber defenders has become crucial toward preventing attacks against the system as a whole. This information sharing is even more crucial in an environment where distributed attacks are becoming increasingly common. CIDS are one way in which to address this need, and blockchain technology appears to be well-suited for the task of ingesting data in a distributed and secure fashion. This research has demonstrated the ability to use commercial and open source blockchain solutions to implement an information sharing system to record both the doorknob rattling attack using PAM and CPU utilization information as transactions in the blockchain. This proof-of-concept also showed that a blockchain system is capable of acting as a logging mechanism for multiple attacking machines and can be used to aggregate data for further processing toward intrusion detection. While it is true that blockchain is not a solution to every type of problem, it shows promise in this area. These positive indications point to the value of future investment in understanding how blockchain technology could improve CIDS.

## 5.1 Significant Contributions

The most significant contribution in this thesis is the presenting of the proof-of-concept system for CIDS. This proof-of-concept while considered in literature has not yet been implemented in practice. This is a significant contribution to this field as it provides a physical structure for analysis and improvement. Furthermore, this thesis reveals the potential feasibility of using open-source blockchain clients, like Ethereum, to conduct further research in this relatively new topic area.

Another contribution is the use of PAM to record information for use as a transaction in a blockchain environment. This is an innovative use of PAM that has heretofore not been used in literature, or used practically to the extent of our knowledge. This method provides a more secure alternative to writing information to a log file, which could potentially be modified by an adversary. This proof-of-concept also demonstrates an example of the interface between a blockchain client and existing systems and software. Being able to trust recorded alerts is the first step in detecting and preventing cyber attacks.

Finally, this research demonstrated the ability to develop thresholds of anomalous activity using statistical and machine learning techniques based on data acquired from the blockchain-based system developed in this thesis. The scenario described in Chapter 4 illustrated how a network administrator might analyze potentially anomalous data in order to prevent intrusions in their networks.

## 5.2 Recommendations for Future Work

This thesis provides several avenues for additional work. The proof-of-concept system engineered in this thesis is small in scale. Creating a larger testbed would be the first step in further exploration of this system approach. Additional research must also be conducted to determine how the system would work at scale with several devices. One issue that would need to be addressed as the system increases in scale is the network overhead to support the blockchain client. Developing an understanding of the overhead of a blockchain client must be a priority for this research to move forward.

The system engineered in this thesis utilized the Ethereum client. By its very nature, this client is designed around supporting a cryptocurrency. A CIDS-specific blockchain apparatus might differ significantly from existing clients in terms of consensus algorithm, transaction formatting, and other details. Work is needed to explore what that customization should entail. Furthermore, if designing a CIDS-specific blockchain system is too high of a hurdle at the current time, additional research on which general purpose blockchain client best meets CIDS requirements should be conducted.

Finally, more work is also necessary to explore how this blockchain-based data ingest module will interact with follow on filtering and processing of data and the other remaining steps in the intrusion detection process outlined in Figure 2.1. There is potential for the blockchain system itself to provide enhanced detection of attacks. For example, an increased activity level in the blockchain could indicate some attack. Methods to use blockchain-activity data to detect and prevent attacks are a rich area for further exploration.

# APPENDIX A:
# Installation Guide and Implementation Notes

This appendix contains an installation guide that should prove useful to a user new to Ethereum. Steps to create a networked blockchain using Ethereum Client:

**1. Update machine**

NOTE: Location of go-ethereum is assumed to be in the home directory.

Listing A.1: Basic Update and Setup Instructions

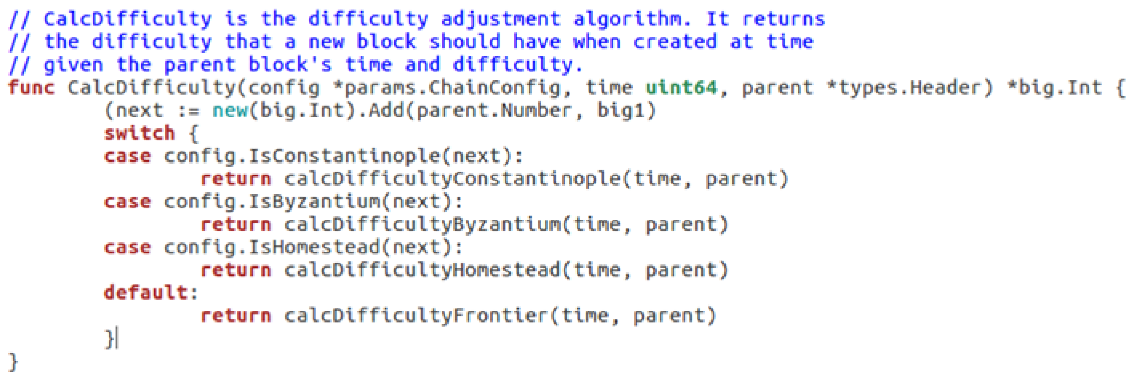
```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install -y build-essential
sudo apt-get install git
cd ~

git clone https://github.com/ethereum/go-ethereum.git
```

**2. Install go-lang for geth**

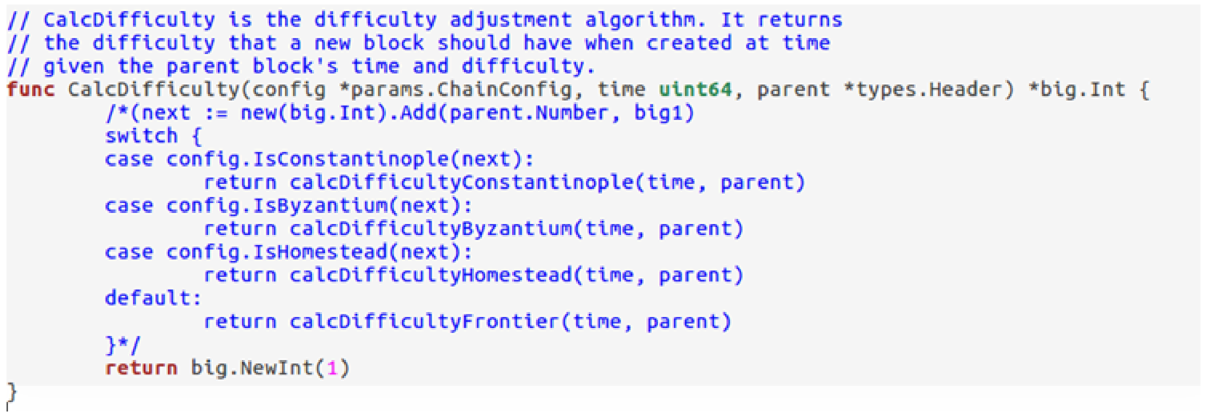
Use update-golang.sh from [43]

Modify consensus algorithm *calcDifficulty* in *consensus.go* ( For ease of testbed use and experimentation)



```
// CalcDifficulty is the difficulty adjustment algorithm. It returns
// the difficulty that a new block should have when created at time
// given the parent block's time and difficulty.
func CalcDifficulty(config *params.ChainConfig, time uint64, parent *types.Header) *big.Int {
        (next := new(big.Int).Add(parent.Number, big1)
        switch {
        case config.IsConstantinople(next):
                return calcDifficultyConstantinople(time, parent)
        case config.IsByzantium(next):
                return calcDifficultyByzantium(time, parent)
        case config.IsHomestead(next):
                return calcDifficultyHomestead(time, parent)
        default:
                return calcDifficultyFrontier(time, parent)
        }
}
```

Figure A.1. Original *calcDifficulty Function*



```
// CalcDifficulty is the difficulty adjustment algorithm. It returns
// the difficulty that a new block should have when created at time
// given the parent block's time and difficulty.
func CalcDifficulty(config *params.ChainConfig, time uint64, parent *types.Header) *big.Int {
        /*(next := new(big.Int).Add(parent.Number, big1)
        switch {
        case config.IsConstantinople(next):
                return calcDifficultyConstantinople(time, parent)
        case config.IsByzantium(next):
                return calcDifficultyByzantium(time, parent)
        case config.IsHomestead(next):
                return calcDifficultyHomestead(time, parent)
        default:
                return calcDifficultyFrontier(time, parent)
        }*/
        return big.NewInt(1)
}
```

Figure A.2. Modified *calcDifficulty Function*

This step is particularly important because it sets our mining difficulty to its lowest possible level. This allows us to acquire a large amount of faux currency for our transactions between accounts. Once we have completed the above step, we must again rebuild *geth*. Thankfully, the developers of this code have provided us with a *makefile* to automate this process. Please note that you must be in the *go-ethereum* folder in order to run the *make* command.

## 3. Rebuild *go-ethereum*

Listing A.2: Updating *geth* with the *make* Command

```
cd ~/go-ethereum
make geth
```

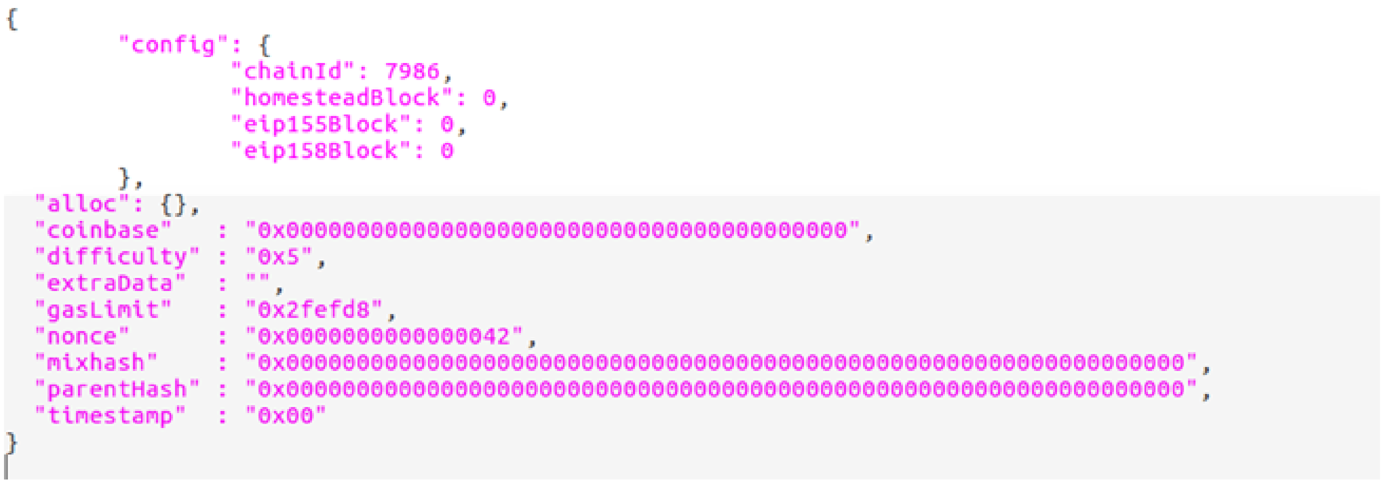
For the sake of simplicity, details have been included about how to setup a two-node network. For a multi-node network, every node must be peered with every other node.

**4. Setting up Node 1**

Listing A.3: Creating a Data Directory

```
mkdir gethDataDir #(Name is completely arbitrary)
```

**create genesis JSON block**



```
{
        "config": {
                "chainId": 7986,
                "homesteadBlock": 0,
                "eip155Block": 0,
                "eip158Block": 0
        },
  "alloc": {},
  "coinbase"   : "0x0000000000000000000000000000000000000000",
  "difficulty" : "0x5",
  "extraData"  : "",
  "gasLimit"   : "0x2fefd8",
  "nonce"      : "0x0000000000000042",
  "mixhash"    : "0x0000000000000000000000000000000000000000000000000000000000000000",
  "parentHash" : "0x0000000000000000000000000000000000000000000000000000000000000000",
  "timestamp"  : "0x00"
}
```

Figure A.3. Sample Genesis Block

For more information on each of these fields, one can consult the Ethereum documentation [20]. Keep the difficulty low in order to mine more currency. Also note that the network ID of the blockchain network is determined by the *chainId* variable above. It is highly advised that the number is large as to prevent conflicts.

**b. Initialize chain with genesis JSON block ensuring chain ID is large**

Listing A.4: Initializing a Genesis Block

```
geth --datadir ~/gethDataDir/ init genesis.json
```

**c. Open IPC endpoint with the same network Id as designated in JSON block**

Listing A.5: Opening an IPC Endpoint

```
geth --datadir ~/gethDataDir --networkid 7986
```

**d. Attach to the IPC endpoint**

Listing A.6: Attaching to the IPC Endpoint

```
cd ~
geth attach ipc:gethDataDir/geth.ipc
```

**5. Node 2 Setup Directions**

**NOTE: FOR EVERY SUBSEQUENT NODE, THE SAME GENESIS BLOCK AS THE FIRST NODE MUST BE USED. IF NOT, NODES WILL NOT BE ABLE TO BE PEERS.**

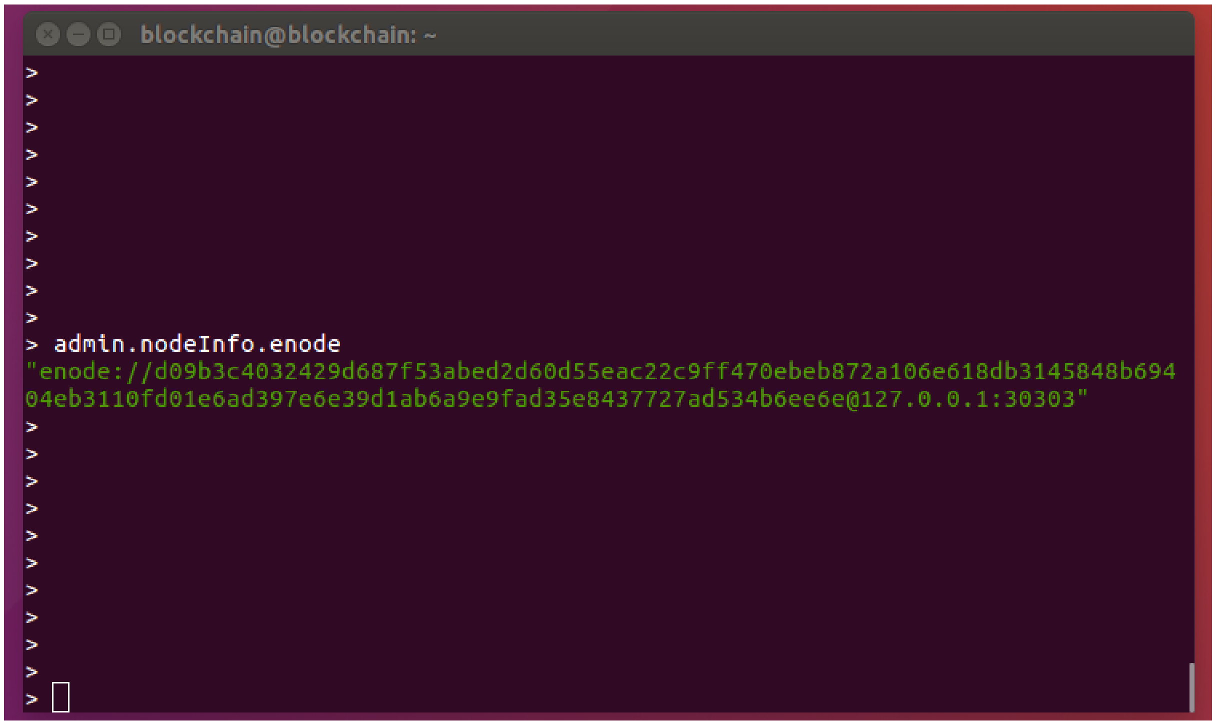
**a. Create data directory as shown in Listing A.3 and use the same genesis JSON block as Node 1**.

**b. Follow initialization instructions from Listing A.4**.

**c. Follow opening IPC endpoint instructions from Listing A.5**.

**c. Follow attaching to the IPC endpoint instructions from Listing A.6**.

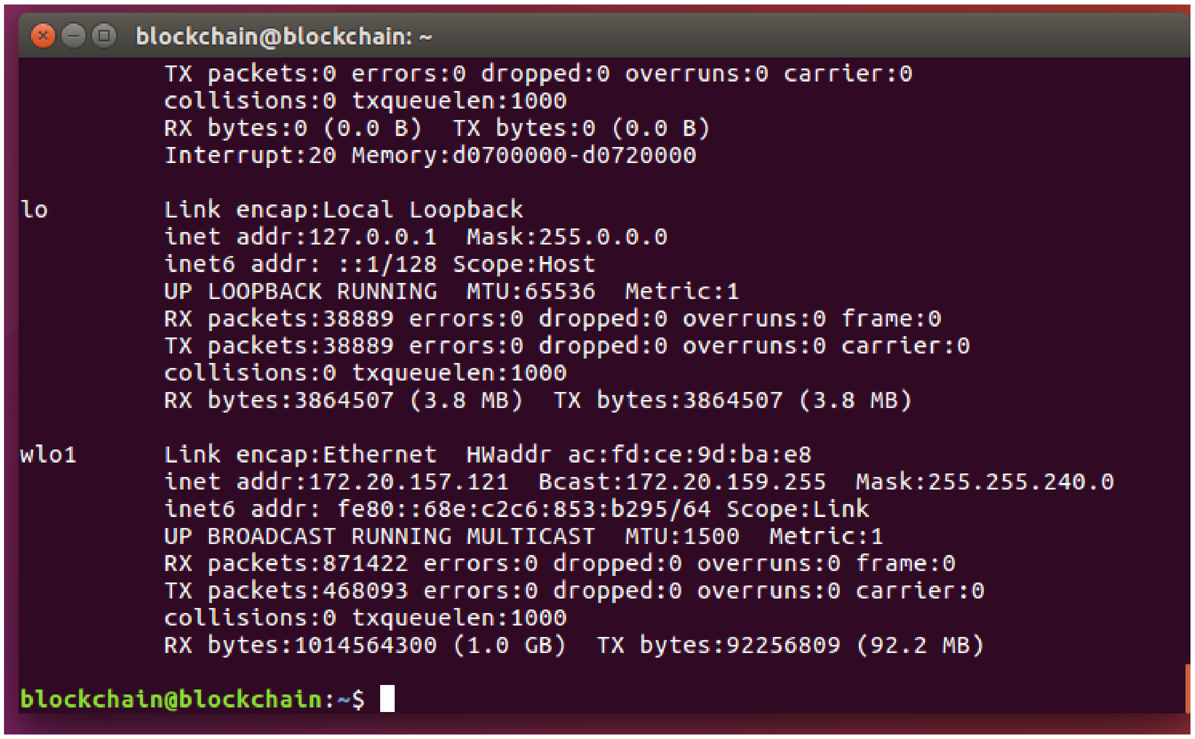
**d. Use *enode* address of Node 1 to add them as a peer:** The *enode* address of Node 1 can be determined by using *admin.nodeInfo.enode*. Details can be seen in A.4.



```
>
>
>
>
>
>
>
>
>
>
> admin.nodeInfo.enode
"enode://d09b3c4032429d687f53abed2d60d55eac22c9ff470ebeb872a106e618db3145848b694
04eb3110fd01e6ad397e6e39d1ab6a9e9fad35e8437727ad534b6ee6e@127.0.0.1:30303"
>
>
>
>
>
>
>
>
>
>
> 
```

Figure A.4. Determining *enode* Information

The IP address shown in the *enode* value above is *localhost* (127.0.0.1). Node 2 must replace that address with the address of the active network interface on Node 1. This is accomplished via running the command *ifconfig* on Node 1. This step is displayed in Figure A.5.



```
blockchain@blockchain: ~
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:20 Memory:d0700000-d0720000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:38889 errors:0 dropped:0 overruns:0 frame:0
          TX packets:38889 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3864507 (3.8 MB)  TX bytes:3864507 (3.8 MB)

wlo1      Link encap:Ethernet  HWaddr ac:fd:ce:9d:ba:e8
          inet addr:172.20.157.121  Bcast:172.20.159.255  Mask:255.255.240.0
          inet6 addr: fe80::68e:c2c6:853:b295/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:871422 errors:0 dropped:0 overruns:0 frame:0
          TX packets:468093 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1014564300 (1.0 GB)  TX bytes:92256809 (92.2 MB)

blockchain@blockchain:~$ 
```

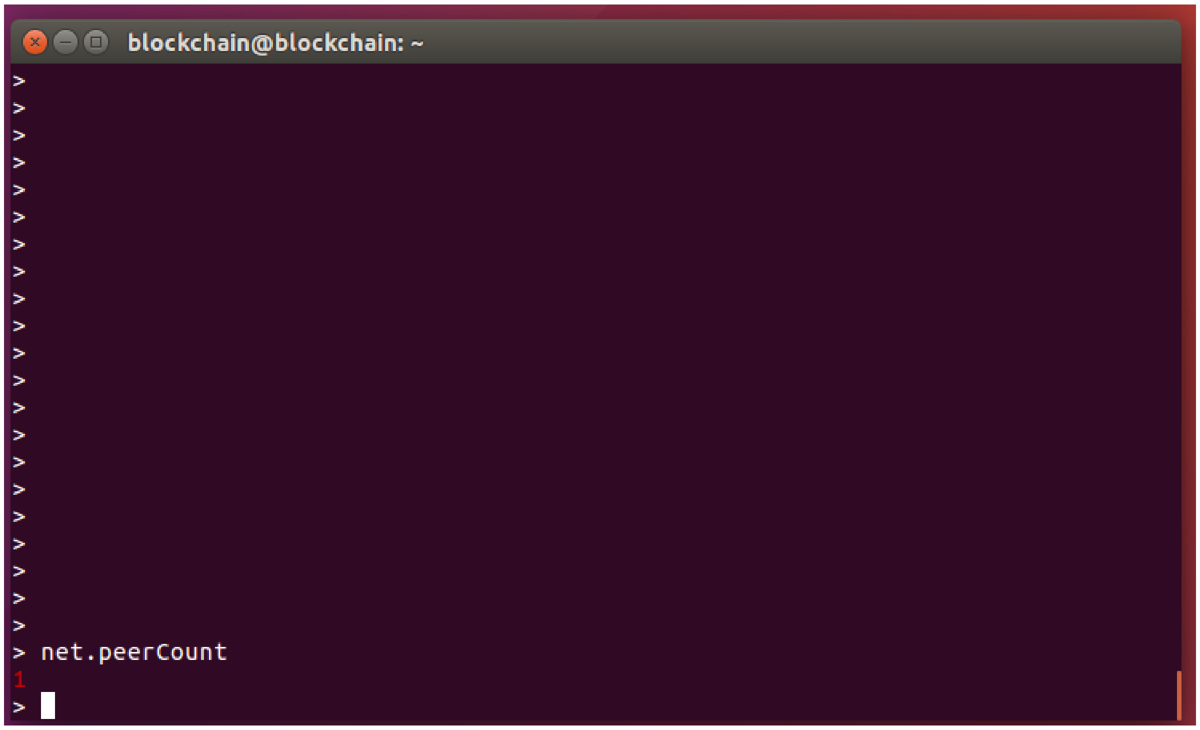
Figure A.5. *ifconfig* Output

Based on the above values, the command that must be run from Node 2 is:

Listing A.7: Adding a Peer

```
admin.addPeer("enode://
d09b3c4032429d687f53abed2d60d55eac22c9ff470ebeb
872a106e618db3145848b69404eb3110fd01e6ad397e6e39d1ab6a9e9fa
d35e8437727ad534b6ee6e@172.20.157.121:30303?discport=0")
```

The output should return true and if the command *net.peerCount* is executed it should return a value of 1 signifying that Node 2 has one peer, Node 1 as can be seen in Figure A.6. The same process could have been reversed with Node 1 adding Node 2 as a peer using its *enode* address. If there are more than two nodes, each node must add each other node as can be seen in Figure 3.4.



```
blockchain@blockchain: ~
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
> net.peerCount
1
> 
```

Figure A.6. *net.peerCount* Output

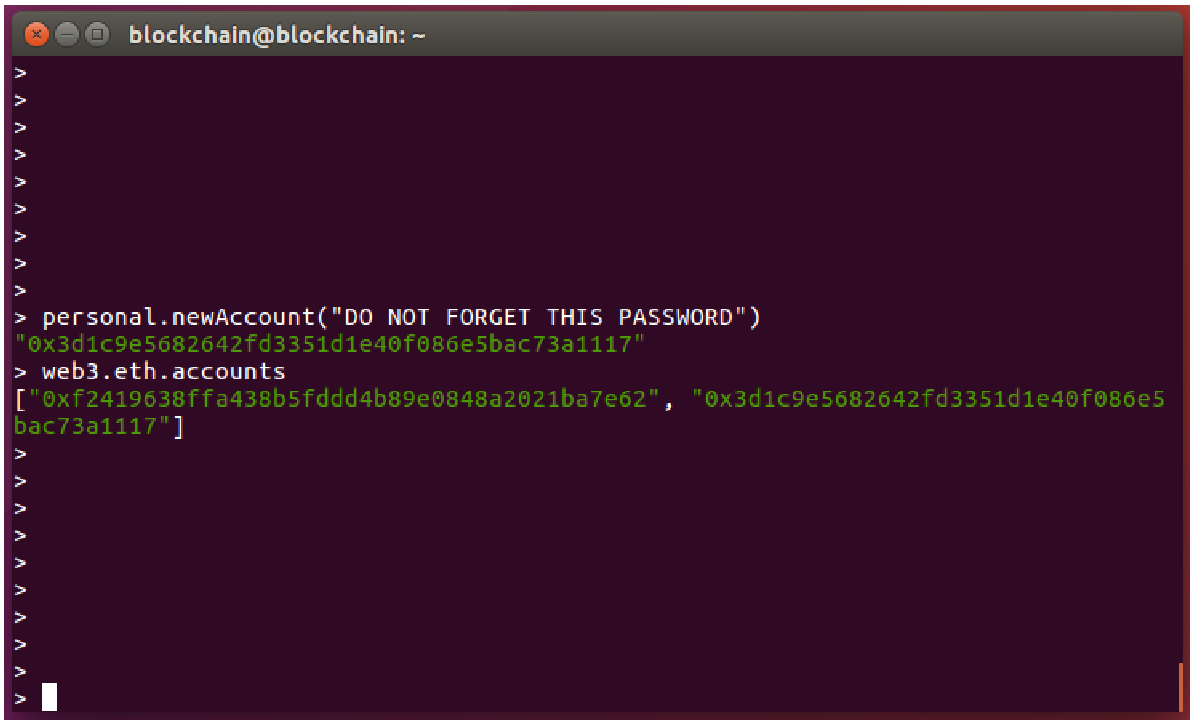
Once the nodes have been connected, a variety of operations can now be conducted. We will cover three basic operations, adding accounts, mining, and sending transactions.

**Adding Accounts (Can be done before peering nodes)**

Listing A.8: Adding Accounts and Verifying

```
personal.newAccount("password")
web3.eth.accounts
```

There is no password recovery option. If the password for the account is forgotten, all money contained in the account is inaccessible.



```
blockchain@blockchain: ~
>
>
>
>
>
>
>
>
>
> personal.newAccount("DO NOT FORGET THIS PASSWORD")
"0x3d1c9e5682642fd3351d1e40f086e5bac73a1117"
> web3.eth.accounts
["0xf2419638ffa438b5fddd4b89e0848a2021ba7e62", "0x3d1c9e5682642fd3351d1e40f086e5
bac73a1117"]
>
>
>
>
>
>
>
>
>
> 
```

Figure A.7. Creating a New Account

### Mining

Listing A.9: Mining

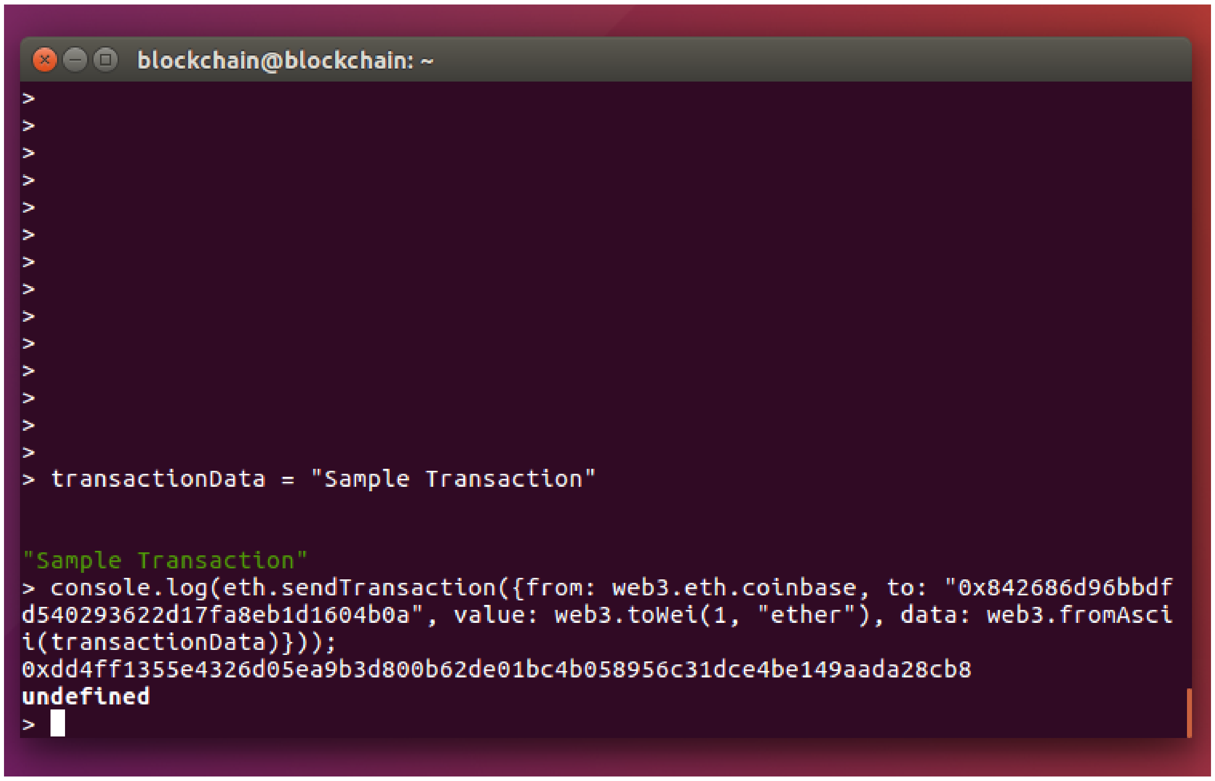
```
miner.start(1)
miner.stop()
```

### Transactions

Listing A.10: Transaction

```
web3.personal.unlockAccount(web3.personal.listAccounts
    [0],"account password", 15000)
console.log(eth.sendTransaction({from: web3.eth.coinbase
    , to: "0x842686d96bbdfd540293622d17fa8eb1d1604b0a",
    value: web3.toWei(1, "ether"), data: transData}));
```

The variable transData is a 32-byte string that is hex encoded. The transaction script in Appendix B shows how to correctly encode those types of values. Also recall that transactions will be processed when blocks are mined. Output of a transaction and of a script designed to crawl through all transactions on a network are shown in Figures A.8 and A.9.

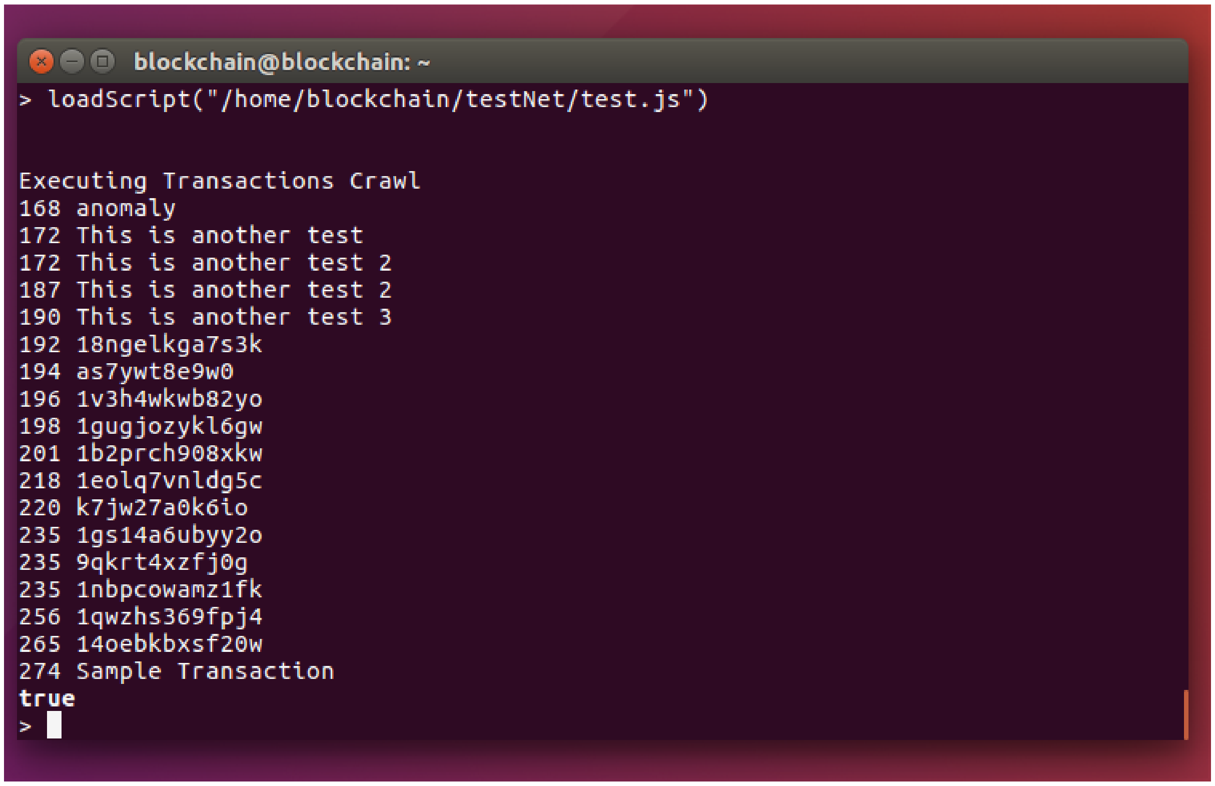


blockchain@blockchain: ~

```
>
>
>
>
>
>
>
>
>
>
>
>
>
>
> transactionData = "Sample Transaction"


"Sample Transaction"
> console.log(eth.sendTransaction({from: web3.eth.coinbase, to: "0x842686d96bbdf
d540293622d17fa8eb1d1604b0a", value: web3.toWei(1, "ether"), data: web3.fromAsci
i(transactionData)}));
0xdd4ff1355e4326d05ea9b3d800b62de01bc4b058956c31dce4be149aada28cb8
undefined
> 
```

Figure A.8. Sample Transaction



```
blockchain@blockchain: ~
> loadScript("/home/blockchain/testNet/test.js")


Executing Transactions Crawl
168 anomaly
172 This is another test
172 This is another test 2
187 This is another test 2
190 This is another test 3
192 18ngelkga7s3k
194 as7ywt8e9w0
196 1v3h4wkwb82yo
198 1gugjozykl6gw
201 1b2prch908xkw
218 1eolq7vnldg5c
220 k7jw27a0k6io
235 1gs14a6ubyy2o
235 9qkrt4xzfj0g
235 1nbpcowamz1fk
256 1qwzhs369fpj4
265 14oebkbxsf20w
274 Sample Transaction
true
> 
```

Figure A.9. Output of Crawl Script

This ends the basic installation guide and Ethereum tutorial.

# APPENDIX B:
Code Snippets

This Appendix contains code snippets that were written for the implementation of a CIDS node that was capable of recording information.

## B.1 Sample Code for Various Applications

### B.1.1 JavaScript Function for Sending a Transaction

Listing B.1: Sending a Transaction

```
web3.personal.unlockAccount(web3.personal.listAccounts[0],"
   paperNet1", 15000)

//Must encode data string as a hex string. The fromAscii
   function accomplishes this.
transData = web3.fromAscii(Math.floor(Math.random()*Math.pow
   (2,63)).toString(36));
console.log(eth.sendTransaction({from: web3.eth.coinbase, to
   : "0x599e76d919dcaa7274cce5e299dce9b46eda989b", value:
   web3.toWei(1, "ether"), data: transData}));

miner.start(1);

setTimeout(function() {
   miner.stop();
}, (5 * 1000));
```

### B.1.2 JavaScript Function for Crawling Through all Transactions

Listing B.2: Crawling Through all Transactions

```
console.log("Executing Transactions Crawl")
var transTest = []
var zero = 0;
var temp;
var data;
//We are able to start at any block by setting i to be
   whatever block we wish. In this example it is set to 1.
for (i = 1; i < eth.getBlock("latest").number; i++)
{
        temp = eth.getBlock(i).transactions;
        if(temp.length != 0)
        {
                for(j = 0; j < temp.length; j++)
                {
                        transTest.push(temp[j]);
                        currentTrans = eth.getTransaction(
                           temp[j])
                        data = currentTrans.input
                        if(data != "0x")
                        {
                                //Records both the data and
                                   gas consuption for
                                   further analysis.
                                //Gas is not needed for CIDS
                                    function
                                console.log(i,web3.toAscii(
                                   data),currentTrans.gas);
                        }
                }
        }
}
```

### B.1.3 Python Script for Recording CPU Utilization

This script utilizes a Bash script shown in Listing B.4 to record utilization information in a file *out.txt* to be read by this Python script. It uses a C program to act as a CPU hog. That code is shown in Listing B.5.

Listing B.3: Python Example: Recording CPU Utilization

```
from web3.auto import w3
import importlib
import time
import subprocess
import datetime
import random

w3.personal.unlockAccount(w3.eth.accounts[0],"paperNet1",
   15000)
toAddr = w3.toChecksumAddress('0
   xf2419638ffa438b5fddd4b89e0848a2021ba7e62')

#Two possible paths, one with the cpu hog running and one
   without
if random.random() < .50:
     subprocess.Popen(['./para','9223372036854775807'])
     time.sleep(10)

transData = ''
subprocess.call("/home/blockchain/atkNet/cpu.sh")
cur_time = datetime.datetime.fromtimestamp(time.time()).
   strftime('%Y-%m-%d␣%H:%M:%S')

with open('out.txt','r') as f:
     transData = f.read()

transData = cur_time + '␣' + transData
```

```
print(transData)

transData = "0x" + "".join(hex(ord(c))[2:] for c in
    transData.strip('\n'))

a = w3.eth.sendTransaction({'to': toAddr, 'from': w3.eth.
    coinbase, 'value': 12345678, 'data': transData})

print(a)
print('\n')

subprocess.run(['pkill','para'])

w3.miner.start(1)
time.sleep(5)
w3.miner.stop()
```

### Bash Script for Recording CPU Utilization

Listing B.4: Bash Script for Recording CPU Utilization

```
#!/bin/bash

top -b -n2 -d 1 | awk "/^top/{i++}i==2" | grep -Ei "cpu\(s\)
    \s*:" > out.txt
```

### C Script for Factorization of a Large Number

Listing B.5: Factorization of a Large Number

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
/*
```

```
    The purpose of this program is to force CPU utilization
        by factoring a large number
*/
void main(int argc, char *argv[])
{
    if(argc != 2)
    {
        printf("Usage:␣./para␣numToFactor\n");
        return;
    }
    long long int n = (long long int) strtol(argv[1], (char
        **)NULL, 10);
    long long int range = floor(sqrt(n));
    omp_set_num_threads(8);
    #pragma omp parallel for
    for(long long int i = 1; i < n; i++)
    {
        if(n % i == 0)
        {
            printf("%lld␣is␣divisble␣by␣%lld\n",n,i);
            printf("%lld␣is␣divisble␣by␣%lld\n",n,n/i);
        }
    }
}
```

### B.1.4 Python Script for Recording Login Attempts

This script was designed to take input from a Bash script shown in Listing B.7 and process it as a transaction.

Listing B.6: Python Example: Recording Login Attempts

```
from web3.auto import w3
import sys
```

```
w3.personal.unlockAccount(w3.eth.accounts[0],"paperNet1",
   15000)

toAddr = w3.toChecksumAddress('0
   xf2419638ffa438b5fddd4b89e0848a2021ba7e62')

transData = ''
for i in range(1,len(sys.argv)):
transData = transData + '␣' + sys.argv[i]

print(transData)

price = random.randint(10,15)*100000

transData = "0x" + "".join(hex(ord(c))[2:] for c in
   transData.strip('\n'))

a = w3.eth.sendTransaction({'to': toAddr, 'from': w3.eth.
   coinbase, 'gas': price, 'gasPrice' : 1000000000, 'value':
    42, 'data': transData})

print(a)
print('\n')

w3.miner.start(1)
time.sleep(5)
w3.miner.stop()
```

**Bash Script for Taking Input from PAM and Outputting it to a Python Script**

Listing B.7: Bash Bridege Between PAM and Python

```
#!/bin/bash
d=$(date)
```

```
py="/usr/bin/python3"
loc="/home/blockchain/paperNet/login.py"

case $1 in
1)
var="Authentication Successful"
eval $py $loc $PAM_USER $PAM_TYPE $PAM_SERVICE $PAM_RUSER
   $PAM_RHOST $d $var;; #>> /home/blockchain/out.txt;;
2)
var="Authentication Failure"
eval $py $loc $PAM_USER $PAM_TYPE $PAM_SERVICE $PAM_RUSER
   $PAM_RHOST $d $var;;
esac
```

THIS PAGE INTENTIONALLY LEFT BLANK

# List of References

[1] U.S. Government Accountability Office, "Information security: Agencies need to improve controls over selected high-impact systems," Washington, DC, USA, GAO Report No. GAO-16-501, 2016.

[2] F. Konkel, "Pentagon thwarts 36 million email breach attempts daily," Jan 2018. Available: https://www.nextgov.com/cybersecurity/2018/01/pentagon-thwarts-36-million-email-breach-attempts-daily/145149/

[3] U.S. Government Accountability Office, "Data breaches: Range of consumer risks highlights limitations of identity theft services," Washington, DC, USA, GAO Report No. GAO-19-230, 2019.

[4] H. Debar, M. Dacier, and A. Wespi, "Towards a taxonomy of intrusion-detection systems," *Comput. Netw.*, vol. 31, no. 9, pp. 805–822, Apr. 1999. Available: http://dl.acm.org/citation.cfm?id=324119.324126

[5] E. Vasilomanolakis, S. Karuppayah, M. Mühlhäuser, and M. Fischer, "Taxonomy and survey of collaborative intrusion detection," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 55:1–55:33, May 2015. Available: http://doi.acm.org/10.1145/2716260

[6] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2, pp. 222–232, Feb 1987.

[7] P. Kabiri and A. A. Ghorbani, "Research on intrusion detection and response: A survey," *International Journal of Network Security*, vol. 1, pp. 84–102, 2005.

[8] T. Proffitt, *How Can You Build and Leverage SNORT IDS Metrics to Reduce Risk?* SANS Institute, Sep 2013. Available: https://www.sans.org/reading-room/whitepapers/tools/paper/34350

[9] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. Available: https://bitcoin.org/bitcoin.pdf

[10] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "Medrec: Using blockchain for medical data access and permission management," in *2016 2nd International Conference on Open and Big Data (OBD)*, Aug 2016, pp. 25–30.

[11] B. Vitalik, "A next-generation smart contract and decentralized application platform," Ethereum, 2014. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[12] S. Singh and N. Singh, "Blockchain: Future of financial and cyber security," in *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, Dec 2016, pp. 463–467.

[13] A. Baliga, "Understanding blockchain consensus models," Persistent Systems, Santa Clara, California, USA, 2017. Available: https://www.persistent.com/whitepaper-understanding-blockchain-consensus-models/

[14] N. Alexopoulos, E. Vasilomanolakis, N. R. Ivánkó, and M. Mühlhäuser, "Towards blockchain-based collaborative intrusion detection systems," in *Critical Information Infrastructures Security*, G. D'Agostino and A. Scala, Eds. Cham: Springer International Publishing, 2018, pp. 107–118.

[15] S. Kibish, "A note about finding anomalies," Apr. 2018. Available: https://towardsdatascience.com/a-note-about-finding-anomalies-f9cedee38f0b

[16] J. Hu, *Host-Based Anomaly Intrusion Detection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 235–255. Available: https://doi.org/10.1007/978-3-642-04117-4_13

[17] H. Okada, S. Yamasaki, and V. Bracamonte, "Proposed classification of blockchains based on authority and incentive dimensions," in *2017 19th International Conference on Advanced Communication Technology (ICACT)*, Feb 2017, pp. 593–597.

[18] "What Is Hashing? Step-by-Step Guide-Under Hood Of Blockchain," Aug. 2017. Available: https://blockgeeks.com/guides/what-is-hashing/

[19] A. Miller and J. LaViola, "Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin," University of Central Florida, 2014. Available: https://socrates1024.s3.amazonaws.com/consensus.pdf

[20] "The Ethereum Wiki. Contribute to ethereum/wiki development by creating an account on GitHub," Aug. 2019, original-date: 2014-02-14T23:05:17Z. Available: https://github.com/ethereum/wiki

[21] "How does blockchain technology work? Is it really the future?" Nov. 2017. Available: https://cryptotechies.com/blockchain-technology-future/

[22] B. Zhu, A. Joseph, and S. Sastry, "A taxonomy of cyber attacks on scada systems," in *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, Oct 2011, pp. 380–388.

[23] P. Ducklin, "Wordpress blogs and more under global attack – check your passwords now!" Apr 2013. Available: https://nakedsecurity.sophos.com/2013/04/13/wordpress-blogs-and-more-under-global-attack-check-your-passwords-now/

[24] J. Erickson, *Hacking the art of exploitation*, 2nd ed. San Francisco, Calif: No Starch Press, 2008.

[25] "Protect your privacy – Bitcoin." Available: https://bitcoin.org/en/protect-your-privacy

[26] Tim Lewis, "Openmp: Enabling hpc since 1997." Available: https://www.openmp.org/

[27] "top(1) – Linux manual page." Available: http://man7.org/linux/man-pages/man1/top.1.html

[28] "12 Critical Linux log files you must be monitoring," Apr. 2018. Available: https://www.eurovps.com/blog/important-linux-log-files-you-must-be-monitoring/

[29] A. Morgan and T. Kukuk, "The Linux-PAM system administrators' guide, version 1.1.2," 2010. Available: http://www.linux-pam.org/Linux-PAM-html/Linux-PAM_SAG.html

[30] "web3: Web3.py." Available: https://github.com/ethereum/web3.py

[31] "Ethereum." Available: https://ethereum.org

[32] "Hyperledger – Open source blockchain technologies." Available: https://www.hyperledger.org/

[33] "Choosing a client – Ethereum Homestead 0.1 documentation." Available: http://ethdocs.org/en/latest/ethereum-clients/choosing-a-client.html

[34] "web3.js Ethereum JavaScript API – web3.js 1.0.0 documentation." Available: https://web3js.readthedocs.io/en/1.0/

[35] "Web3.py – Web3.py 4.9.2 documentation." Available: https://web3py.readthedocs.io/en/stable/

[36] "web3.eth – web3.js 1.0.0 documentation." Available: https://web3js.readthedocs.io/en/1.0/web3-eth.html#eth-sendtransaction-return

[37] S. Pendino, "Blockchain network behavior based anomaly detection," 2019, unpublished.

[38] D. S. Moore and G. P. McCabe, *Introduction to the practice of statistics*, 4th ed. New York: W.H. Freeman and Co., 2003.

[39] J. MacQueen, "Some methods for classification and analysis of multivariate observations." The Regents of the University of California, 1967. Available: https://projecteuclid.org/euclid.bsmsp/1200512992

[40] A. K. Jain, “Data clustering: 50 years beyond K-means,” *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, June 2010. Available: http://www.sciencedirect.com/science/article/pii/S0167865509002323

[41] K. M. Carter, R. P. Lippmann, and S. W. Boyer, “Temporally oblivious anomaly detection on large networks using functional peers,” in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (IMC ’10). New York, NY, USA: ACM, 2010, pp. 465–471. Available: http://doi.acm.org/10.1145/1879141.1879201

[42] L. K. Szeto, A. W.-C. Liew, H. Yan, and S.-s. Tang, “Gene expression data clustering and visualization based on a binary hierarchical clustering framework,” *Journal of Visual Languages & Computing*, vol. 14, no. 4, pp. 341–362, Aug. 2003. Available: http://www.sciencedirect.com/science/article/pii/S1045926X03000338

[43] udhos, “update-golang is a script to easily fetch and install new Golang releases with minimum system intrusion: udhos/update-golang,” July 2019, original-date: 2017-04-10T21:50:01Z. Available: https://github.com/udhos/update-golang

# Initial Distribution List