THESIS

ON THE DESIGN OF A MOVING TARGET DEFENSE FRAMEWORK FOR THE RESILIENCY OF CRITICAL SERVICES IN LARGE DISTRIBUTED NETWORKS

Submitted by

Athith Amarnath

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2018

Master's Committee:

Advisor: Indrajit Ray

Indrakshi Ray
Stephen Hayne

ABSTRACT

ON THE DESIGN OF A MOVING TARGET DEFENSE FRAMEWORK FOR THE RESILIENCY OF CRITICAL SERVICES IN LARGE DISTRIBUTED NETWORKS

Security is a very serious concern in this era of digital world. Protecting and controlling access to secured data and services has given more emphasis to access control enforcement and management. Where, access control enforcement with strong policies ensures the data confidentiality, availability and integrity, protecting the access control service itself is equally important. When these services are hosted on a single server for a lengthy period of time, the attackers have potentially unlimited time to periodically explore and enumerate the vulnerabilities with respect to the configuration of the server and launch targeted attacks on the service. Constant proliferation of cloud usage and distributed systems over the last decade have materialized the possibilities of distributing data or hosting services over a group of servers located in different geographical locations. Existing election algorithms used to provide service continuity hosted in the distributed setup work well in a benign environment. However, these algorithms are not secure against a skillful attackers who intends to manipulate or bring down the data or service. In this thesis, we design and implement the protection of critical services, such as access-control reference monitors, using the concept of moving target defense. This concept increases the level of difficulty faced by the attacker to compromise the point of service by periodically moving the critical service among a group of heterogeneous servers, thereby changing the attacker surface and increasing uncertainty and randomness in the point of service chosen. We describe an efficient Byzantine fault-tolerant leader election protocol for small networks that achieves the security and performance goals described in the problem statement. We then extend this solution to large enterprise networks by introducing random walk protocol that randomly chooses a subset of servers taking part in the election protocol.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr.Indrajit Ray for giving me an opportunity to experience academic research and showcase my abilities. Dr. Ray always encouraged me to explore my passion and inspired me with his dedication towards security research and life skills. Thank you very much sir.

I would like to thank my mother Mrs. Anitha Rao, father Mr. H Amarnath Rao, brother Mr. Anvith Rao and my best friend Mr. Gopalakrishnan Rajagopalan for their consistent support and encouragement. I wouldn't be able to come this far without the support of Gopal's parents, Mrs. Usha Rajagopalan and Dr. Rajagopalan R. Thank you all for believing in me.

I would specially like to thank Dr. Bruhadeshwar Bezawada and Mr. Dieudo Mulamba for their support towards this research and thesis.

DEDICATION

I would like to dedicate this thesis to my mother's late twin brother Mr. Ajit Shenoy. His short, yet beautiful life in this world has always inspired and motivated me to aim higher, even when I didn't get a chance to meet him. You will always be in my heart Ajit mamma.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

There is an increase in the demand for servers hosting wide varieties of web and applications services over the Internet. Critical services like identity and access management, intrusion detection systems, anti-virus software, network and web security are hosted using cloud infrastructures. Such applications are being outsourced to the companies in the form of Security as a Service (SECaaS) products [1]. The servers hosting these critical services are central targets to the attackers who intend to cause harm to the companies that use them. Any compromise of a critical service not only allows the attacker to tamper with the integrity and availability of the service but also opens possibilities for future attacks.

For example, consider a server that hosts a role-based access control system by enforcing file permissions on resources after receiving access request from a user. An attacker will focus on enumerating the vulnerabilities of the services based on current server configuration with an intention to compromise these services. The single point of contact server connected to the Internet provide attackers sufficient time to exploit the vulnerabilities and tamper with the integrity and availability of services. This can lead to short term manipulation of the service, disrupting the service using Denial of Service Attacks (DoS) or long term damage like loss of data. Therefore, protection of these critical services from targeted attacks is an important and challenging problem.

1.1 Use Case

Effective access control depends both on the existence of strong policies and ensuring that the access control enforcement system is adequately protected [2]. Protecting these systems are not well addressed in literature. In general, access control is assumed to be implemented as reference monitor [3]. One way to design a reference monitor is to implement a Trusted Computing Base (TCB). TCBs are computer systems with hardware, firmware and software designed to be secure against hardware/software bugs, vulnerabilities and attacks [4]. They are designed to be small,

thoroughly tested and analyzed to make it tamper-proof. However, according to [2], it is impossible to design a single TCB in distributed computing environments to provide access control.

Hence, we design a network of heterogeneous servers where each server implements all the components of the access control reference monitor. The service that provides the access control to clients is moved around the reference monitor network based on the node chosen by the distributed consensus to serve for a given period of time.

1.2 Moving Target Defense

1.2.1 Definition

It is the concept of moving the target of the attack, *i.e.*, the critical service from one location to another to increase the uncertainty and the apparent complexity of the attacker. Changing the point of contact after a certain period of time will change the attack surface thereby reducing the window of opportunity for an attacker to learn and exploit the vulnerabilities of a system. This would also help increase the cost of an attack and disrupt any ongoing attack.

According to the Trustworthy Cyberspace Strategic plan published by the Executive Office of the President, National Science and Technology Council, December 2011 [5], "Moving Target Defense enables us to create, analyze, evaluate, and deploy mechanisms and strategies that are diverse and that continually shift and change over time to increase complexity and cost for attackers, limit the exposure of vulnerabilities and opportunities for attack, and increase system resiliency." As a result, the defenders are able to create systems that are more dynamic and the attackers have to deal with higher amount of uncertainty. The ultimate goal of Moving Target Defense is to increase the cost of the attacks by dynamically and randomly changing the attack surface. This approach has been heralded as a "game changer" in the field of research [6].

1.2.2 Examples

The concept of moving target defense can be illustrated using the "Shell Game", also known as thimblerig, three shells and a pea, the old army game [7]. Here, the target is hidden under one of

the three shells or cups facing down on the surface. The objective of the game is to find the target after the shells have been randomly shuffled, which is hard. This is analogous to the intent of moving the target of the attack, *i.e.*, the critical service randomly across the nodes in a distributed network.

Another example could be Address Space Layout Randomization (ASLR), which is a famous approach used in the operating systems to fortify against memory corruption vulnerabilities [8]. The idea here is to introduce artificial diversity by randomizing the memory location of certain system components. ASLR is able to randomize the address space of a particular application, making it hard for the attacker to predict them. The advantage of uncertainty in the address space increases the cost of the attack. This approach proves that the concept of moving target defense is used in the real computing world.

1.3 The Problem

When a service is hosted on a centralized static server, an attacker is provided with potentially unbounded time to explore the vulnerabilities of the server with respect to the configuration. This allows the attacker to manipulate or break the service. To avoid this, we consider a network of heterogeneous servers where any server can seamlessly provide the service at any time. The heterogeneity of the servers broadens the attacker's surface and cannot easily exploit the same vulnerability enumerated on one server across all others in the network. This approach is robust against attacks that are launched on a single static server.

Given a set of n servers: $\{S_1, S_2, \cdots, S_n\}$, the objective is to design a protocol that selects one server uniformly at random to host the critical service. This protocol should be repeated periodically so that a server hosts the critical service for a short period of time, known as the *"Election term"*. The protocol should also work in a compromised network environment where the attacker is assumed to control some servers. The server or a node selection should be a decentralized distributed consensus process, where all the nodes participate in the protocol to reach an agreement. Moreover, the protocol should be configured such that the election term of a server is shorter than

the average time taken by an attacker to enumerate the service vulnerabilities on the current server. With this configuration, and the network with heterogeneous servers, the probability of a successful attack can be reduced.

1.3.1 The Solution

Our approach consists of using several networked heterogeneous servers and choosing one server at random, at regular time intervals, as the point of service. The time interval, also known as the Election term is chosen such that the attackers has least probability of success in exploiting the vulnerabilities of the current point of service and launching an attack. The point of service is moved from server to server in a non-deterministic fashion and this makes it difficult to plan and execute an attack. With this design, the system conforms to the concept of *moving target defense*. As the election term of one node completes, we design a leader election protocol that is capable of being resilient to nodes exhibiting byzantine failures [9]. All the nodes taking part in the election protocol are loosely synchronized and the nodes initiate the leader election protocol at the end of the chosen node's election term. During the protocol, each node chooses a random value and broadcasts it to the other servers. The server that broadcasts the smallest random number is chosen as the leader and is confirmed by at least $n - k$ nodes. Here, n is the total number of nodes and $k = \lfloor \frac{n-1}{3} \rfloor$ number of faulty nodes the protocol can withstand [10]. Finally, the consistency of the messages exchanged between the nodes are checked and time-outs are implemented by each node to detect Byzantine-fault behavior. Chapter 4 describes this approach in detail.

1.3.2 Shortcomings of this Solution

The leader election protocol overview described in section 1.3.1 requires at least $n - k$ honest nodes (where $k = \lfloor \frac{n-1}{3} \rfloor$) to participate and choose a leader that hosts the critical service. The resiliency of this system is given by the value of k. As the number of nodes n taking part in the algorithm increases, so does the resiliency of the system k. However, as the number of nodes taking part in the protocol increases, the number of messages exchanged between the nodes and time required by each node to verify the messages increases. This in turn slows down the protocol,

increases the turn-around time to achieve distributed consensus. In short, as the number of nodes taking part in the algorithm increases, the communication complexity of the protocol and time to achieve distributed consensus increases. Moreover, the network of heterogeneous servers that take part in the protocol do not change over time. This fixed group of servers are vulnerable to an attacker who patiently unveils the vulnerabilities of each system and then enumerates the whole network. To address this problem, we consider choosing a random subset of nodes within a larger network, for each election term. The random subset of nodes are then responsible to choose a node that hosts the critical service within the subset by executing the leader election protocol.

1.3.3 Large Networks

Given a set of n servers: $\{S_1, S_2, \cdots, S_n\}$, the objective is to design another protocol that chooses a subset of m servers: $\{T_1, T_2, \cdots, T_M\} \subset N$ uniformly at random that would take part in the election protocol. This protocol is executed before the leader election protocol and repeated periodically to choose a subset that executes the next leader election protocol. It should be designed to work in a compromised network environment, where the attacker is assumed to control some servers. The subset selection is a distributed process and requires that the presence of every node in the subset can be verified by every other node in the network. The protocol is required to complete its process before the next election protocol begins. Furthermore, the election term should be longer than the completion time of the protocol, but shorter than the average time taken by an attacker to enumerate the service vulnerabilities on the current server.

1.3.4 Solution for All Networks

This approach is designed to work closely with the leader election protocol on a network with several heterogeneous servers. A subset of the network is generated at regular intervals, but before the execution of the next leader election protocol. The subset cannot be chosen by a central authority server because it is vulnerable to manipulation attacks. Hence, the subsets are required to be generated in a sequential fashion, where the previous node added to subset is responsible to choose the next node in the subset. A node chooses the next node by generating a random number

and using the modulo arithmetic. This node then *delegates* the process to the next node to continue the sequential process of subset generation. This decentralized sequential algorithm is executed within the distributed network until the predefined number of nodes are added to the subset. All the nodes added in the subset are required to be non-faulty, verifiable and honest. A subset should to be verifiable by every node in the network. The nodes in a subset are responsible for choosing the next node that hosts the critical service by executing the leader election protocol. The sequential node selection algorithm is executed again for the next subset to continue the leader election protocol. Time-outs and message verification algorithms are implemented in each node to detect nodes exhibiting byzantine behavior. Chapter 6 describes this approach in detail.

1.4 System Design Overview

As discussed in section 1.1, our solution requires a network of heterogeneous servers and a critical service as a target of the attack. As a case study, we chose a role based access control as the critical service and this access control engine requires an implementation of a reference monitor [3]. This mediates all access to objects or resources based on the roles provided to the subject or user. The user-role assignments and their respective access privileges are described in a policy file. This access control service is moved around the distributed network of heterogeneous servers, to allow limited opportunity to the attacker to explore and enumerate vulnerabilities, thereby reducing the attack surface of the system.

When a new node is selected and the service is moved, the clients requests are redirected to the new server using service discovery protocol [11]. In this thesis, we used an open source implementation of service discover protocol [12]. However, the working of service discovery protocol are out of scope for our discussions in this thesis.

Chapter 3 describes the system architecture and the modules that are designed to test whether our solution is viable.

1.5 Thesis contribution and Organization

The rest of the thesis is organized as follows. Chapter 2 presents a comprehensive survey of existing leader election protocol and random walk approaches applied to various problems. In chapter 3, we provide an overview of the system architecture built to test our approach. In chapter 4, we describe the leader election protocol and its constructs. Also, we explain how this approach is useful in small enterprise networks. In chapter 5, we discuss the experimental setup and performance analysis of the leader election protocol. In chapter 6 we discuss the problems of the leader election protocol in large networks, propose a better approach to overcome these problems and describe a general solution for all networks. We evaluate our general solution by conducting different experiments and practical analysis in chapter 7. Finally, chapter 8 concludes the thesis and provides directions for future work.

Chapter 2

Related Work

In the world of distributed systems, there are several election algorithms designed and developed to host services hosted in these systems. The ring based algorithm [13] provide safety and aliveness features by implementing failure detectors using alive messages and timeouts. The Bully algorithm [14] proposed by Garcia-Molina is also a famous distributed election algorithm designed to elect a leader with consensus of all the nodes in a distributed environment. These algorithms assume a network with synchronous systems and reliable network communications. The ring based algorithm elects leader in a ring based network. These algorithms choose a node with the highest node identifier among all the nodes in a distributed network. This algorithm conducts election only when the leader goes down. Hence, these algorithms do not hold good in a compromised environment which is considered to be the environment of our problem statement.

Moving target defense (MTD) [7] is the concept of introducing controlled change across multiple components of the network in order to reduce the window of opportunity of attackers, and increase the cost of their attack efforts. Zhuang *et al*. [15] describes the key concepts of MTD and their properties. Evans *et al*. [16] explains different low level techniques of MTD, dynamic diversity defenses, analyze the security properties of a few example defenses and attacks, and identify scenarios where moving target defenses are not effective. Han *et al*. [17] defines and investigates two complementary measures that are applicable when the defender aims to deploy MTD to achieve a certain security goal. At the network level, the work in [18] presents some network-based MTD approaches while the work in [19] introduces a MTD approach for the cloud system.

Int this thesis, we are interested in solving the consensus problem using MTD in an environment with Byzantine failures. Byzantine failures [20] are a type of failures that can affect a computer system and cause it to behave in a arbitrary way. There are many byzantine agreement protocols proposed to conduct election in a compromised environment. Castro and Liskovs *et al*. [21] proposed the earlier protocols that included Practical Byzantine Fault-tolerance protocol. There are

byzantine agreement protocols [20, 22, 23] that work in a different environment as opposed to the one assumed in the problem statement. Some protocols [21, 24, 25] perform the election algorithm with required randomness but do not consider the attackers time-window to explore vulnerabilities to break the system. Among the work proposing the MTD approach in face of the Byzantine failures, [26] proposes a moving target defense approach to switch among Byzantine fault tolerant protocol according to the existing system and network vulnerability. In [27], the authors introduce Turtle Consensus (MPTC), an asynchronous consensus protocol for Byzantine-tolerant distributed systems that uses MTD strategies to tolerate certain attacks. In [28], Tangaroa protocol is proposed, which is an extension of the Raft protocol [25] and the protocol from [21]. In [29, 30], the authors present Byzantine consensus protocols for synchronous and authenticated setting that tolerate less than $\frac{n}{2}$ faults. For mission-critical applications, in [31] the authors describe a practical asynchronous Byzantine fault tolerant protocol, which guarantees liveness without making any network timing assumptions.

In [2], the authors proposed an initial solution for this problem using access control reference monitors, based on the "moving target defense" [15,16,32] where a server is elected for a particular election period using a Byzantine fault-tolerant leader election protocol. Here, the node is chosen as a leader in a deterministic manner [10]. The number of election terms is divided by the total number of nodes and the resultant remainder is the node identifier. The server corresponding to that node identifier value is chosen as the leader for that election term. Moreover, the number of communication rounds designed to elect a leader with the consensus of all the nodes in the network is high. A compromised server has a good chance of being elected as a leader of the protocol. If an attacker compromises a few select nodes, then the attacker has a very high probability of succeeding in tampering with the result of the protocol.

In the current problem statement, we assume the network is reliable and there no churn in the network. If churn is considered, this would provide opportunities for developing an optimized protocol within this setting. State-of-the-art enterprise implementations like [25] are designed as voluntary "voting" mechanisms for replicated log management applications and are inefficient for

protecting critical services. Furthermore, adapting other practical Byzantine-fault tolerant protocols [21, 24] for this problem is not a feasible option as these protocols will require non-trivial modifications to suit the constraints of the problem statement.

Network resilience is defined as the ability of the network to function in face of failures due to natural disaster or malicious attacks, which affect the proper operation of some of its components [33, 34]. This is an important parameter that measures the security level of the system and how protected the system is from byzantine failure. In this thesis, we design a system that increases the network resiliency of the distributed network that provides security-critical service such as access control. Techniques to increase a network resilience include segmentation [35], dynamic composition [36], diversity [32, 37], deception [38], and dynamic reconstruction [39]. Segmentation [35] aims to limit the attack surface of a potential attack by logically or physically separating the network critical components. Dynamic composition [36] is the ability to dynamically provide new capabilities to the network. Diversity [32,37] is the action of using heterogeneous logical separation of physical components in a network. The goal is to limit the attacks exploiting common vulnerabilities. Deception [38] is the action of misleading or confusing attackers in order to hide the critical assets of a network. Dynamic reconstitution [39] is the ability to reconfigure a network in order to render it resilient to ongoing and future attacks or faults while maintaining continuity of operations. One approach to implement a dynamic reconstitution of a network is through Moving Target Defense.

The concept of simple random random walk was first introduced in [40]. The random walk concept is used in many network applications and the main purpose is to perform node sampling. Random walk-based sampling is simple, local, and robust. At present, there are many algorithms that use random walk as an integrated subroutine. Some of the network applications that use random walks are token management [41,42], load balancing [43], small-world routing [44], search [45–47], information propagation and gathering [48] and network topology construction [49, 50]. Random walks have also been used in distributed systems to provide dynamic control in a uniform and efficient way [51]. The paper of [52] describes a broad range of network applications that

can benefit from random walks in dynamic and decentralized settings. A fast and efficient way to perform random sampling in an arbitrary network is shown in [53]. In this thesis, we are using random walk to sample or choose a subset of nodes from a reliable distributed fully connected network that take part in the election protocol in a fast and efficient way.

In this work, we have designed a secure one-way hash commitment based moving target defense protocol for security-critical services that can tolerate byzantine faults, which is new contribution in this problem space. The chapter 4 and its results 5 are part of the paper [54] that is accepted in MTD 2018 ACM conference. We further design and implement the random walk protocol that brings in more uncertainty and randomness in the system that chooses a leader for hosting the service in this thesis.

Chapter 3

System Architecture

In this chapter, we describe system architecture of each server hosted in the fully connected distributed network. Each server implements several modules to test our solution.

3.1 Architecture Overview

Figure 3.1 shows a network of four access control reference monitors. Each server is implemented with four modules,

- ACM - Access Control Module
- EM - Election Module
- FD - Fault Detector Module
- DSA - Digital Signature Algorithm Module

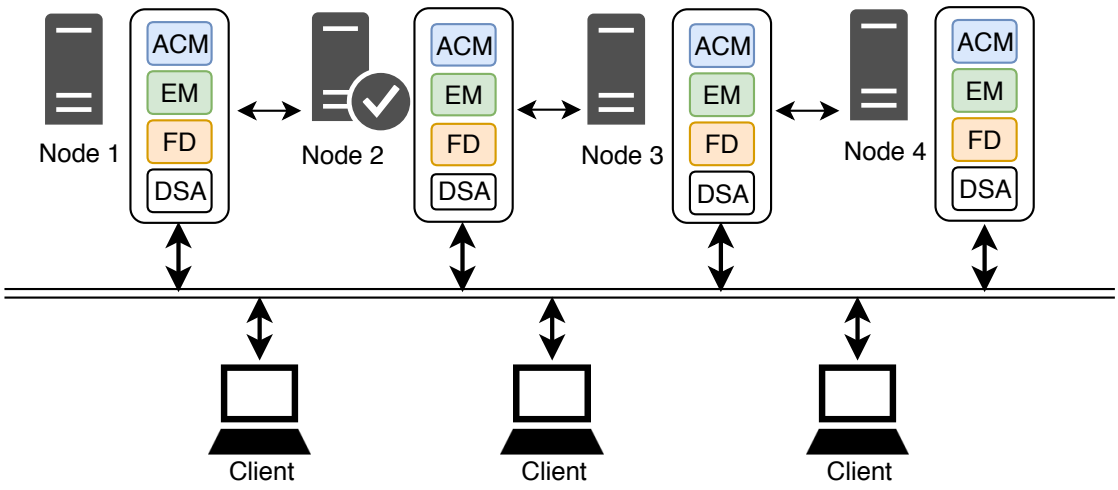


Figure 3.1: System Implementation Architecture

These modules are implemented using Java programming language. Every node in the network is connected to all other nodes to form a fully connected distributed network using Java socket programming. The LAN network used to build this system and is assumed to be reliable.

3.2 Client

The clients are the endpoint entity whose main objective is to send requests to the access control service. Using the help of service discovery protocol, these requests are redirected to the node that hosts the access control service for that particular election term. The node replies back to the each client with its response.

3.3 Digital Signature Algorithm

Elliptic Curve Digital Signature Algorithm (ECDSA) is used by every node to digitally sign and verify the messages exchanged within the distributed system. The strength per key bit is known to be substantially higher in elliptic curve DSA [55]. The public keys of each node is shared with all other nodes taking part in the protocol and this is done during the system configuration. Every node that broadcasts messages during the election algorithm are digitally signed with its own private key. When another node receives the message, it first verifies the message with the sender node's public key. This way the communication between the nodes are secured.

3.4 Fault Detection Module

The Fault Detector Module is designed to closely monitor the behavior of all the nodes that provide the access control service. This module implements phase timeouts, message format verification algorithm and task validation schemes. This module is responsible for verification and validation of messages received by other nodes. All data and observations are shared with the election module for further processing and decision making.

3.5 Election Module

The Election Module implements the leader election protocol described in chapter 4. The module is responsible for maintaining the communications between other nodes and coordinate the election seamlessly. All the messages received as a part of the election process are carefully verified by the fault detector module and then received by the election module. All the outgoing messages are constructed according the protocol format and sent to DSA module for further processing.

3.6 Access Control Module

This module implements all the components of the access control reference monitor. It is responsible to process the clients' access requests, decide based on the policies described and prepare the response that is sent back to the client.

An access control service is also implemented on each server in this module. This service is activated when the node is chosen by the distributed consensus to serve the election term. It is responsible for receiving requests from clients and send the responses back to the clients using java socket programming. The service is deactivated when the node completes its term of service.

Chapter 4

Solution for Small Networks

The first goal of our thesis is to propose a consensus algorithm that works in a compromised environment and resilient to byzantine faults. In this chapter, we aim to design an efficient moving target defense protocol that would move the target of the attack within a distributed network without any service interruption. Our approach begins with designing a *Distributed Leader Election Protocol* where all the nodes in the network communicate and coordinate to choose a leader with quick turn-around time, is able to detect faults and exclude malicious nodes from interrupting the critical service. The protocol follows four stages of complex computations, message verification and time-outs to maintain honest participation of each node taking part in the algorithm. The proposed *Commit and Reveal* approach is used in the protocol to bring in trust within all the nodes and fault detectors are implemented to monitor the behavior of each node in the network. After all the message communication, coordination and verification, the nodes agree to choose one leader that hosts the critical service for a given time period or an *Election term*. Once the term completes, all the nodes wake up to re-start the protocol to choose the next leader.

At first, the thread model is discussed to lay out the capabilities of the adversaries and the limitations of the protocol. We will describe all the components of the protocol in this chapter, later discuss our observations and results of this protocol in the next chapter [1].

4.1 Technical Challenges and Solutions

The first challenge is the choice of random numbers by each node taking part in the leader election protocol. The protocol is designed to choose a smallest of all the random number choices of the nodes. As explained in the problem statement, this algorithm is working in a compromised

[1] A version of this chapter will be published in 5th ACM Workshop on Moving Target Defense (MTD 2018) October 15, 2018, ACM, 2018, A Secure Hash Commitment Approach for Moving Target Defense of Security-critical Services, Dieudonne Mulamba, Athith Amarnath, Bruhadeshwar Bezawada and Indrajit Ray.

environment, where some of the nodes might be controlled by the attacker and might not conform to the protocol honestly. Therefore, such malicious nodes might wait until the other nodes reveal their random values and then choose a smaller value than these nodes to win the election. To avoid this problem, a commitment based protocol is designed in such a way that each node in the protocol requires to first commit to a particular random value in one round and then reveal the value in the next round. This way we can avoid "wait-and-see" attacks by the misbehaving nodes controlled by the attacker.

The second challenge is to design an efficient commitment scheme [56] that the node operates within the time limits of the chosen critical service. To mitigate this challenge, a one-way hash based commitment scheme is designed that uses secure one-way hash functions and public parameters. Here, every node taking part in the algorithm chooses a random value and publishes its publicly-verifiable one-way hash, *i.e.*, the *"commit"* value in the first communication round. During the second communication round, the node publishes, *i.e.*, *"reveals"*, the random value and the other public parameters used to generate the "commit" value, which can be used to verify the commit value shared in the first communication round. Finally, the algorithm chooses the node that published the smallest hash value as the winner. This form of distributed consensus is used to choose the winner without any dependency on the central authority.

The third and final challenge is that, this protocol should be designed to work in a compromised environment. The protocol should be resilient to the nodes exhibiting byzantine behavior and try to compromise or disrupt the service by breaking the consensus protocol. To achieve this, we have devised a fault detector mechanism in our protocol that monitors all the nodes and detect malicious or byzantine behavior of other nodes. All the observations are shared with the rest of the nodes as one round within an election to avoid nodes making malicious decisions. Once the majority of nodes vote against a node being malicious, it will be placed in a faulty node list and would not be allow to participate or become the leader in the future.

4.2 Threat Model

Byzantine failures [20] are a type of failures that can affect a computer system and cause it to behave in a arbitrary way. These failures can be caused either by wear and tear of physical components leading to failure of the system or by a generic adaptive adversary who intends to compromise the system. In this thesis, we are protecting the critical service on hosted on a server and move the service at regular intervals to reduce the window of opportunity for attacks.

We assume that the adversary has the ability to enumerate the vulnerabilities of the service with respect to the server configuration and can launch the described attack. To launch an attack, we assume that the attacker requires certain time-window to learn about the vulnerabilities of server and then enumerate them to launch the attack. Considering that the servers are heterogeneous, the attacker's effort to enumerate vulnerabilities of one server cannot be the same as to another server.

Finally, we assume that that network model consists of a minimum of 4 or more nodes. This is to tackle the issues with the Byzantine Generals Problem [20]. The consensus is achievable in a distributed system if and only if the number of faulty nodes in the system is: (1) less than one-third of the total number of nodes; and (2) less than one-half of the connectivity of the system's network [57]. Hence, we assume that there are at most $\lfloor \frac{n-1}{3} \rfloor$ nodes (n - total nodes) that are faulty or malicious in a distributed network that run the leader election protocol. When the faulty or malicious nodes are greater than $k = \lfloor \frac{n-1}{3} \rfloor$ nodes, the network fails to obtain consensus and this must be avoided. Moreover, higher the value of n, the network is more resilient to byzantine failures. In other words, a distributed network is k-resilient, where $k = \lfloor \frac{n-1}{3} \rfloor$.

Furthermore, through periodic service audits or anomaly detection approaches, the network administrator detects a compromised server and fortifies or patches the vulnerabilities of the server.

4.3 Public Parameters

All the nodes are initialized with two public-parameters for the *Hash commitment* phase of our protocol. The first public parameter is a large integer Q, which is used for modulo the result of one-way hash function at each node to reduce the size and is a fixed value throughout. The second

public parameter is P, a critical component of the protocol and is one of the inputs to a hash function $\mathbb{H}$ along with the random value chosen by a node. P plays the role of public randomness in our protocol, which is a difficult property in distributed systems [58].

Lets assume $\{S_1, S_2, \cdots, S_n\}$ are the set of n heterogeneous servers that are taking part in the leader election algorithm. Let Q denote a public parameter that is configured and common to all the nodes. Let P^i denote another public-value that is used in a particular election round i. For each subsequent round, a node locally updates the P^i value used in the previous election round as follows: $P^{i+1} = P^i + 1$. Nodes are allowed to use older values of P^i and the P^i value used in the first phase is shared with all nodes in the second phase for verification. However, if a node is using an older P^i value that has not been incremented beyond a certain system defined threshold, then the other nodes will consider the inputs of this node as faulty and ignore them. We use the notation P^i_j represents the local value of P in S_j node for ith round, where S_j, $\forall 1 \leq j \leq n$. The key property of this parameter is that it is incremented after each election session to provide the necessary randomness for the one-way hash computations.

4.4 Secure One-Way Hash Commitment

A pseudo-random one-way hash function $\mathbb{H}$ is a function that takes in a message x and return a fixed-size alphanumeric string which is called a Hash value $X = \mathbb{H}(x)$. This function is computationally easy to calculate a hash for any given data. However, it is extremely difficult to reverse the hash function $\mathbb{H}(x)$ to get x. Moreover, two messages that are slightly different generate unique hash values. This function is the core concept behind the security of our protocol.

In the existing consensus protocols [2, 21, 29], the general approach is for the nodes to declare a random value and then agree on the lowest value among those published. This approach can be broken by a malicious node controlled by an attacker, waiting for inputs from other nodes before declaring its own value. If the attacker controls a sufficient number of nodes, the probability of a malicious node winning the election is high. Moreover, considering our network model, where the nodes being connected by a reliable broadcast channel, the attacker has instant access to inputs of

all the nodes. To address this problem, we have designed a commit and reveal approach, where the nodes first commit a value $\mathcal{C}$ to other nodes in the network and then reveal a value $\mathcal{R}$ that generated $\mathcal{C}$. We are using a secure one-way hash function $\mathbb{H}$ such as SHA-256, to calculate the commit value $\mathcal{C}$. using the random value $\mathcal{R}$.

Let assume $\{S_1, S_2, \cdots, S_n\}$ are the set of n heterogeneous servers that are taking part in the leader election protocol. Let $\mathbb{H} : \{0,1\}^* \rightarrow \{0,1\}^t$ denote a strong pseudo-random one-way hash function that takes any length input and generates a fixed t-bit output. Each node S_j, $\forall 1 \leq j \leq n$ calculates the **One-way Hash Commit Value** $\mathcal{C}_j$ such that,

$$\mathcal{C}_j = \mathbb{H}(\mathcal{R}_j, P_j^i) \mod Q$$

Each node S_j, broadcasts its commit value, $\mathcal{C}_j$, and the public value local to the node S_j for i-th election term, P_j^i, used in the hash computation above to the rest of the network.

Once the commit phase is complete, each node S_j broadcasts the **Reveal value** $\mathcal{R}_j$ and P_j^i used in the hash computation to the network. Once the Reveal Value is received by all the other nodes, each node S_j, compares all the received commit values $\{\mathcal{C}_j\}$, and verifies the integrity of these values by recomputing the hash values based on the respective random value, $\mathcal{R}_j$, and the public value, P_j^i, pairs. Among the correctly verified hash values, the node selects the smallest hash value amongst all the values and confirms the corresponding publishing node as the winner.

A possible attack on the hash commitment approach explained above is that a node fixes the value of P and calculates the best values that result in the lowest hash values over multiple election terms using brute-force. To prevent this, we define a certain *stale* threshold, which is the maximum difference of a node's copy of P from any other node's copy of P. This threshold reduces the attack surface of an attacker and, the brute-force attack will no longer be a feasible option. In our security analysis, we show that this parameter helps in achieving strong security guarantees in our protocol.

4.5 Leader Election Protocol

4.5.1 Overview

The goal of our protocol is to keep *"moving"* the service, which is the attack target, from one server node to another and change the attackers surface to increase the cost of an attack and disrupt any ongoing ones. We start with the assumption that one or more nodes, but no more than $\lfloor \frac{n-1}{3} \rfloor$ nodes, might be partially compromised by an attacker. In this section, we are going to talk about different stages of the leader election protocol and its operation in detail. There are 4 main phases in our protocol, *COMMIT, ESTIMATE, CONFIRM* and *FAULT DETECTION* phase. Algorithm 1 describes all the four phases of the leader election protocol in the form of a state machine.

4.5.2 Commit Phase

This phase is the beginning of every election term where each node selects a random value independently and computes a one-way hash on the random value using the semantics of the *One-way hash commit* step of the commitment technique explained above, which is called as commit value $\mathcal{C}$. This value is digitally signed to create a $COMMIT$ message and broadcasted to all the remaining node servers. The figure 4.1 shows the timing diagram of the $COMMIT$ phase.

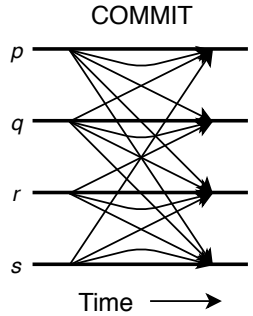


Figure 4.1: COMMIT phase Timing Diagram

After the $COMMIT$ message broadcast, each node server then starts a timer G and uses this value to wait for the next protocol phase to start. During this wait period, it is expected that each

Algorithm 1: Leader Election Protocol at S_c

Input: Election term Number $\mathcal{E} \leftarrow 0$, $L_{suspect} \leftarrow \{\phi\}$, $L_{faulty} \leftarrow \{\phi\}$
$S = \{S_1, \cdots, S_n\}$ where $S_m = (S_m, \sigma_{S_m}, P_m^{\mathcal{E}}[\cdots])$, $P_m^{\mathcal{E}}[\cdots]$ is the list of past P values
$I = [\cdots]$ where $I_j = (\mathcal{C}_j, \mathcal{R}_j, P_j^{\mathcal{E}}, S_j)$, reset for each election term $\mathcal{E}$
$register_{flag} \leftarrow false$, $state \leftarrow COMMIT$
$n \leftarrow$ Total number of nodes in the network, $k = \lfloor \frac{n-1}{3} \rfloor$, $\Omega \leftarrow$ Election Term
Output: S_c protocol output variable $register_{flag}$

loop
switch $(state)$
case COMMIT**:**
$\mathcal{E} \leftarrow \mathcal{E} + 1$, For random number $\mathcal{R}_c$, Calculate $\mathcal{C}_c = \mathbb{H}(\mathcal{R}_c, P_c^{\mathcal{E}}) \mod Q$
Broadcast $\{Sc \quad COMMIT \quad \mathcal{E} \quad \{\mathcal{C}\}\}_{\sigma_{Sc}}$ to all nodes, Start Timer G
if Timeout(G) **and** $n - k$ $COMMIT$ messages received **then**
$state = ESTIMATE$
end if
case ESTIMATE**:**
Broadcast $\{Sc \quad ESTIMATE \quad \{\mathcal{R}_c, P_c^{\mathcal{E}}\}\}_{\sigma_{Sc}}$ to all nodes, Start Timer G
if Timeout(G) **and** $n - k$ $ESTIMATE$ messages received **then**
$state = CONFIRM$
end if
case CONFIRM**:**
S_{min} = node with minimum $\mathcal{C}_{min}$, given by Algorithm 3
Broadcast $\{Sc\, CONFIRM\ (S_{min})\ \ LIST[I]\}_{\sigma_{Sc}}$ to all nodes, Start Timer G
if Timeout(G) **and** $n - k$ $CONFIRM$ messages received **then**
$validate(I)$ using Algorithm 2
$state = SUSPECT$
end if
case SUSPECT**:**
Broadcast $\{Sc \quad SUSPECT \quad LIST[(1,\ S_1), \cdots, (k,\ S_k)]\}_{\sigma_{Sc}}$ to all nodes
Start Timer G
if Timeout(G) **and** $n - k$ $SUSPECT$ messages received **then**
if $S_{min} == S_c$ **then**
$state = LEADER$
else
$state = IDLE$
end if
end if
case IDLE**:**
Sleep (Ω)
$register_{flag} \leftarrow false$, $state = COMMIT$
case LEADER**:**
$register_{flag} \leftarrow true$, $state = IDLE$
end switch
end loop

node server receives all $COMMIT$ values from other nodes taking part in this protocol. This round completes when the timer expires and moves to the next round.

4.5.3 Estimate Phase

At the end of the timeout of the $COMMIT$ phase, each node sends out an $ESTIMATE$ message containing the random value $\mathcal{R}$ and public value P used for generating the commit value $\mathcal{C}$ in the $COMMIT$ phase and starts a new timer G for this phase. We call this phase as the $ESTIMATE$ phase and it is an implementation of the ***Reveal*** step of the commitment technique explained above. The figure 4.2 shows the timing diagram of the $ESTIMATE$ phase. Also, the

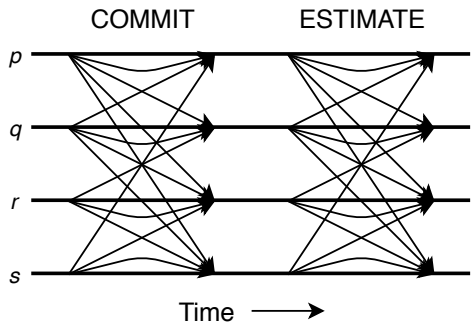


Figure 4.2: ESTIMATE Phase Timing Diagram

node stores the random values $\mathcal{R}$ received from other nodes along with their respective $\mathcal{C}$ values.

After the $ESTIMATE$ message broadcast, each node server then starts a timer G and uses this value to wait for the next protocol phase to start. During this wait period, it is expected that each node server receives all $ESTIMATE$ values from other nodes taking part in this protocol. This round completes when the timer expires and moves to the next round.

4.5.4 Confirm Phase

After the completion of the $ESTIMATE$ phase, the nodes validate the respective $\{\mathcal{C}, \mathcal{R}, P\}$ pairs from different nodes using the algorithm 2. Figure 4.3 shows the timing diagram for the $CONFIRM$ phase.

Algorithm 2: Validate the List I at S_c

Input: $I = \{I_1, \cdots, I_n\}$ where $I_m = (\mathcal{C}_m, \mathcal{R}_m, P_m^{\mathcal{E}}, S_m)$
$L_{suspect} \leftarrow \{\phi\}$

Let $Valid = \{\phi\}$
for $I_m \in \{1, ..., n\}$ **do**
  if $[\mathcal{C}_m == \mathbb{H}(\mathcal{R}_m, P_m^{\mathcal{E}}) \mod Q]$ **then**
    $Valid = Valid \bigcup I_m$
  else
    $L_{suspect} = L_{suspect} \bigcup S_m$
  end if
end for

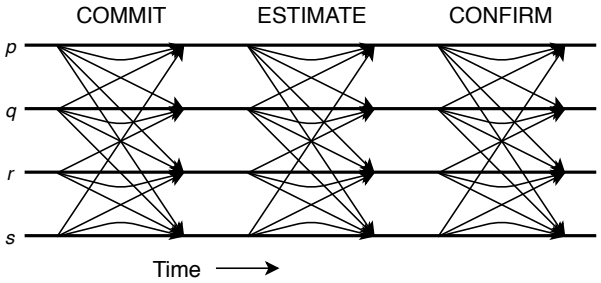


Figure 4.3: CONFIRM Phase Timing Diagram

Algorithm 3: Minimum Commit Value at S_c

Input: $I = \{I_1, \cdots, I_n\}$ where $I_m = (\mathcal{C}_m, \mathcal{R}_m, P_m^{\mathcal{E}}, S_m)$
Output: S_i publishing minimum $\mathcal{C}_i$

Let $I_{min} = I_1$,
$\Rightarrow\ S_{min} = S_1\ and\ \mathcal{C}_{min} = \mathcal{C}_1$

for $I_t \in \{1, ..., |Valid|\}$ **do**
  if $\mathcal{C}_{min} > \mathcal{C}_t$ **then**
    $I_{min} = I_t$
    $\mathcal{C}_{min} = \mathcal{C}_t$
  end if
end for
Return S_{min};

Using Algorithm 3, each node selects the minimum hash value from among the valid values and returns the S_{min} of the corresponding $\mathcal{R}$. At the end of the computation, each node broadcasts, the winning node identifier, the list of hash values and the corresponding sender identifiers, to the rest of the network.

After receiving the $CONFIRM$ message with the minimum hash value and list of hash values, each node verifies the digital signatures on the received messages and then, performs the validation step by checking if: $\{\mathcal{C}_j == H(\mathcal{R}_j, P_j^i) \mod Q\}$, for the node S_j in election session i. The leader of the election is the node that advertised the smallest value in the *Commit* phase and marked by at least $\frac{n}{2}$ other nodes.

4.5.5 Fault Detection Phase

During the fault detection phase, every node broadcasts the node identifiers added in their respective *suspect* list. The nodes added into this list are nodes that advertised inconsistent values during the first 3 phases or did not follow the protocol or failed to verify the source during signature verification. The inconsistent values are considered to be sent by Byzantine nodes that are compromised by the attacker. Figure 4.4 shows the timing diagram for the $SUSPECT$ phase.

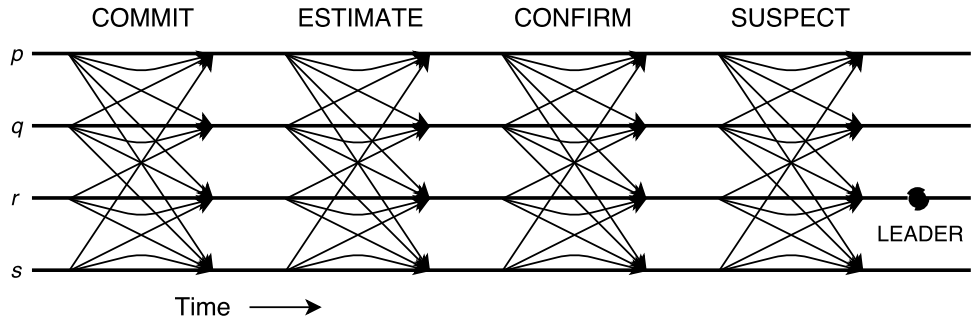


Figure 4.4: Leader Election Protocol Timing Diagram

Using the list of received hash values in the $CONFIRM$ phase, each node determines inconsistent messages by misbehaving nodes and creates a $suspect$ list, which is broadcasted to the rest of the network in a $SUSPECT$ message. Next, each node compares all the received suspect

Algorithm 4: Fault Detection Algorithm

Input: $L_1, \cdots, L_n$ where $L_m = \{S_a, \cdots, S_m\}$
Output: $\{L_{faulty}$ where $\forall S \in L_{faulty}$ each S is in at least $|n-k|$ L_is

$L_{faulty} = \{\phi\}$
$k = \lfloor \frac{n-1}{3} \rfloor$
$THRESHOLD = \lfloor n - k \rfloor$

for $L_i \in L_1, ..., n\}$ **do**
  for $S_j \in L_i$ **do**
    $COUNT = 1$
    if [$(S_j \in L_p)$ AND $(p \neq i)$ AND $(COUNT \neq THRESHOLD)$] **then**
      $COUNT = COUNT + 1$
      if [$COUNT == THRESHOLD$ AND $S_j \notin L_{faulty}$] **then**
        $L_{faulty} = L_{faulty} \bigcup S_j$
        Break;
      end if
    end if
  end for
end for
Return L_{faulty}

nodes' lists, including its own, and identifies the nodes that are identified as suspect by a threshold majority of the nodes. The threshold majority is given by $\lfloor n - k \rfloor$ where $k = \lfloor \frac{n-1}{3} \rfloor$. All suspect nodes satisfying the threshold majority are placed in the *fault* list, as shown in Algorithm 4, and their inputs are ignored for the life time of the algorithm. The administrator is able to patch up the server and fortifies them so that server can be included in the protocol manually. At the end of this phase, each node sets the time to the election term and waits until the current leader's term is complete.

4.5.6 Message Formats

There are four main messages in our protocol: $COMMIT, ESTIMATE, CONFIRM$ and $SUSPECT$. We use S to denote a node-identifier field, $\mathcal{C}$ to denote the hash commitment value field, P to denote the public-parameter field, and $\mathcal{R}$ to denote the random value field. We use σ_{Sc} to denote the private-key signature of a message by the sending node Sc where c means "current sending node". Let $\mathcal{E}$ be the current election term number.

The $COMMIT$ phase parameter has only one field, $\{\mathcal{C}\}$. The message format is as follows

$$\{Sc \quad COMMIT \quad \mathcal{E} \quad \{\mathcal{C}_c^i\}\}_{\sigma_{Sc}}$$

Here, $\mathcal{C}_c^i$ represents the commit value $\mathcal{C}$ of node S_c for ith election term.

The current election term number $\mathcal{E}$ is shared by all nodes to verify whether the node is participating in the current election term.

The $ESTIMATE$ phase parameter contains only two fields: $\{R, P\}$. The message format is as follows

$$\{Sc \quad ESTIMATE \quad \{\mathcal{R}_c^i, P_c^i\}\}_{\sigma_{Sc}}$$

The $CONFIRM$ message is composed of two fields: the node identifier of the node that published the lowest $\mathcal{C}$ and, the list of all the $\mathcal{C}$ values and corresponding Ss received by this node.The format is as follows:

$$\{Sc \quad CONFIRM \quad (S_{min}) \quad LIST[(S_1, \mathcal{C}_1, \mathcal{R}_1, P_1^i), \cdots, (S_{n-1}, \mathcal{C}_{n-1}, \mathcal{R}_{n-1}, P_{n-1}^i)]\}_{\sigma_{Sc}}$$

where n is the maximum number of nodes in the system.

Finally, the $SUSPECT$ message contains the Ss of suspected faulty nodes:

$$\{Sc \quad SUSPECT \quad LIST[(1,\ S_1), \cdots, (k,\ S_k)]\}_{\sigma_{Sc}}$$

where n is the maximum number of nodes in the system.

4.5.7 Time-outs

In our protocol, every phase of the protocol is terminated by a properly chosen time-out value: G. This time-out value is common to all the 4 phases in the algorithm. This time-out value is chosen by measuring the time required for the nodes to perform local computations, prepare the

messages with results and broadcast them to the rest of the network. The main requirement of this time-out for each phase is to provide just enough time to perform necessary computations. This inhibits any malicious node from manipulating the protocol in an unfair manner. If a malicious node is waiting for inputs from all the nodes in the network, the time-out proves beneficial in detecting the malicious node. Consider a malicious node that tries to brute-force the best random value to win the election, if the node is unable to do so within the time-out value, then it is put in the *fault* list. In our protocol, we choose the time-outs within a Δ-fraction of the estimated round-trip network delay, $RTDelay$, such that

$$G_p = (RTDelay + \Delta * RTDelay + f(n)) where 0 < \Delta \leq 1$$

, to ensure that such "wait-and-see" behavior of nodes is detected.

Also, f is a network dependent time function that accounts for the size of the network and the type of broadcast capability of the network. Finally, we do not assume that the nodes are tightly synchronized and allow staggering of timeouts across the network.

4.6 Security Analysis

In this section, we consider theoretical and practical attack scenarios to discuss the security of our protocol. For theoretical analysis, we consider an *Adaptive Byzantine Adversary* who has the ability to modify the inputs to the protocol by (partially) controlling a threshold number of servers, not necessarily the same, in any given session. For practical situations, we consider two kinds of attacks: brute-force and clone attacks.

4.6.1 Adaptive Byzantine Adversary

The definition of an adaptive adversary $\mathcal{A}$ in the context of our protocol is as follows:

Definition 1: An adaptive Byzantine adversary $\mathcal{A}$ is a Byzantine adversary who has access to all the inputs and results of the protocol for a polynomial q number of election sessions, *i.e.*, the attacker knows all the hash commit values, the random values and the winning values. The

adversary is also in control of some of the inputs, not more than $k = \lfloor \frac{n-1}{3} \rfloor$, which is required for a safe quorum to be achieved. We denote the adversary storage by: $A = \{A_1,\ A_2, \cdots,\ A_q\}$ where $A_i = \{(\mathcal{C}_1^i,\ R_1^i,\ P_1^i,\ S_1^i), \cdots,\ (\mathcal{C}_n^i, R_n^i,\ P_n^i,\ S_n^i)\}$ denotes the i^{th} protocol session's input-output pairs.

ϵ-advantage. The ϵ-advantage is the advantage probability of an adversary to win the election in any election session while being in control of k servers. Ideally, ϵ-advantage should be no more than $\frac{1}{k}$.

Theorem 1 (ϵ-security)**.** *Assuming the presence of a strong pseudo-random function, our protocol is ϵ-secure against Byzantine adversaries where $\epsilon \leq \frac{1}{k}$ and $k = \lfloor \frac{n-1}{3} \rfloor$, and the $stale$ threshold of the public parameter P is $\leq q$.*

Proof. We base our proof on two important assertions and show that the adversary advantage does not increase based on the history of the interactions and the strategy used.

Assertion 1. The protocol is history independent as long as the public parameter P is incremented by the honest majority.

This assertion is made based on the properties of the pseudo-random one-way hash function $\mathbb{H}$, *i.e.*, given a set of $(\mathcal{R}_m, P_m^i)$ input pairs for a node S_m for ith election round, the output $\mathcal{C}_m$ is independent of all past rounds. For each round, the value of P is incremented and $\mathcal{R}$ is randomly chosen with bigger seed. If the adversary controls or predicts the $\mathcal{R}$ space, then the adversary does not gain any additional advantage from knowing the output history of the hash function $\mathbb{H}$ because the honest majority increment P after every q^{th} round. Moreover, this assertion states that the ϵ-advantage is not cumulative and forces the adversarial nodes to increment P after the q^{th} round to ensure that the adversarial inputs are acceptable.

Assertion 2. Within a given session, the adversarial advantage is not increased due to attack strategies used.

To increase advantage, there are two simple attack strategies for an adversary: (a) Use all k compromised servers to select an input that is better than a particular target server that the attacker ϵ-estimates to be the winner, and (b) Use each of k servers to target all the honest servers with a

divide-and-conquer approach, *i.e.*, each compromised server targets a distinct subset of the honest servers.

For the first attack strategy, the advantage of the adversary is ϵ for selecting an input better than the target server. However, there are still $n-k-1$ honest servers that will pick their inputs independently at random and hence, the probability of a winner from this coalition is $\frac{1}{n-k-1}$. Therefore, the final advantage of the adversary in winning the election is: $\epsilon \times (1 - \frac{1}{n-k-1})$, which is smaller than ϵ.

For the second attack strategy, assume that the attacker divides the network into $B = \{B_1, \cdots, B_k\}$ where $|B| = \lceil \frac{n-k}{k} \rceil$, coalitions and assigns one server to each coalition B_i. But now, each compromised server only has (ϵ/k) advantage of winning the election against a given coalition, B_i. To compute the cumulative adversary advantage, we observe that, the adversary advantage within each coalition is independent of the results in the other coalitions. Therefore, the adversarial advantage in a single coalition B_i is no better than $((\epsilon/k)/|B_i|) = (\epsilon/k)/(\frac{n-k}{k}) = \frac{\epsilon}{n-k}$, which is again less than ϵ.

These attacks are a good baseline for modeling real-world adversaries as other strategies can be modeled similarly. Therefore, based on these two assertions, our protocol remains ϵ-secure under the conditions of monotonically increasing P, regardless of the adversarial strategies.

4.6.2 Brute Force Attack

An adversary might try to win the election by brute-forcing the random number space and keeping the P fixed up to the $stale$ threshold. There are two reasons why this attack is likely to fail. First, though the attacker controlled servers do not increment P, the honest majority of nodes will increment P. As a result, the attacker's brute-force on the random numbers will result in $\mathbb{H}$ outputs that are different from the honest majority's $\mathbb{H}$ outputs, even for the same random numbers. Therefore, the likelihood of the attacker's random numbers being the winning choices is as good as the honest majority's choice of random numbers. Second, some nodes in the honest majority may not update their P value due to various reasons like down time or lack of synchrony. Even in this

scenario, brute-forcing is unlikely to be successful because it is more than likely that these honest nodes will still have a P that is different from the attacker's servers. Furthermore, the honest nodes might be out of synchrony with respect to P for at most one or two election sessions. Once the election starts and when these nodes start receiving $COMMIT$ messages, these nodes will update their P values according to the higher P values seen in the $COMMIT$ messages. Therefore, the attacker will have no additional advantage even if some of the honest nodes are not in synchrony with the remaining honest majority. Hence, based on the above two reasons, brute-force attack is not a practical option for an adversary against our protocol.

4.6.3 Clone Attacks

A more feasible attack on our protocol is a message "clone" attack, in other words, an attacker server "waits-and-sees" for the honest servers' inputs-outputs and "clones" these messages. This attack is as follows: an adversary waits for the commit values, the $\mathcal{C}$s, from other servers and picks the smallest $\mathcal{C}$ value. The adversary announces this value to the rest of the network. Now, for the next step, the attacker has two choices to complete the $Estimate$ phase, (a) either the attacker brute-forces the random number space and finds a suitable random number matching the $\mathcal{C}$ value it has advertised, or (b), the attacker repeats the "wait-and-see" step and announces the winning random number. If successful, in either attempts, there will be two winning nodes, the attacker node and the "cloned" node.

First, we consider the brute-force scenario and show that it can be addressed with properly chosen timeouts. Note that, if the attacker is successful then the attacker has essentially followed the protocol steps correctly and is a valid winner. However, brute-forcing an input based on the output $\mathbb{H}$ is a time-consuming operation as it involves computing a $\mathbb{H}$ output and reducing it modulo Q, till the desired modulo value is found. As a result, there are no guarantees of finding a suitable random value within the timeout of the current protocol phase. Therefore, an attacker is unlikely to choose this option to be successful in the clone attack.

The second scenario can be addressed with an additional tie-breaking round. The remaining nodes can randomly choose one of the winning nodes as the leader and announce it to the rest of the network. The node that gets the majority votes will be the winner of the election. While this mitigation step is likely to allow an attacker server to be a leader, the probability is still bounded by: $\frac{1}{2(n-k)}$, as remaining honest majority of $n-k$ nodes will choose either nodes with equal likelihood. Timeouts can prevent this attack as the attacker has to wait for almost all the nodes in the network to send their values. In practice, such waiting will timeout the attacker and put the attacker in a faulty node as desired.

A modified version of this attack, that may not timeout, is that the attacker clones the input-output messages of a selected honest server across all sessions. But this attack has a far less likelihood of winning as the attacker's winning chances are only as good as the honest server selected. However, clone attack still remains one of the limitations of our protocol at present and there is a possible solution for this through log analysis. As the attacker is assumed to be in (partial) control of k nodes and, if the same nodes keep winning the election, a statistical analysis of the past logs will reveal this pattern and all the suspected nodes will be decommissioned and fortified. Also, any attempts by the attacker to remain under the radar of the statistical analysis is likely to reduce the impact of the attacker in the long run.

Chapter 5

Small Networks - Evaluations and Discussion

5.1 Experimental Setup

To measure and validate the protocol, we implemented a network with fifty HP-Z440-XeonE5-1650v4 servers, running Fedora 26, each with 8 cores, 3.6 GHz clock, and 16 GB RAM, communicating over a LAN network with 1 Gbps capacity[2]. The protocol was implemented in Java with Apache Maven Build Environment. Apache Maven build environment was used to maintain the project dependencies as well as to deploy the final jar file for simulation. The software was configured to read config.properties file to setup the simulation environment. The config.properties file contained all the network address information of each node, timeout values, public parameters and digital signature keys as an input to all nodes in the network. The protocol communication used blocking server-client TCP communication design with a fully connected distributed network was implemented to replicated the broadcast capabilities of the LAN. We used Apache Log4j2 logging service (https://logging.apache.org/log4j/2.x/), which uses asynchronous loggers, to log the node activities. We used OpenSLP 2.0.0 [12] Java implementation to register the leader that provides the access control request. The service location protocol (SLP) is used to publish/broadcast the leader service to all the clients. The simple access control service was designed using the Balana XACML implementation. The Policy Enforcement Point (PEP) was built as a wrapper to communicate to the client as well as maintain access to resources. All nodes were equipped with self signed public-key private-key pairs and used the Elliptic-Curve DSA signature algorithm with 256-bit keys. The secure one-way hash function chosen was SHA-256, the public-parameter P was a 512-bit integer and the modulo reducer Q was a 256-bit integer. The configurations of the number of nodes per

[2]A version of this chapter will be published in 5th ACM Workshop on Moving Target Defense (MTD 2018) October 15, 2018, ACM, 2018, A Secure Hash Commitment Approach for Moving Target Defense of Security-critical Services, Dieudonne Mulamba, Athith Amarnath, Bruhadeshwar Bezawada and Indrajit Ray.

election term were: $5,\ 10, \ldots,\ 50$, and each experiment was averaged over 50 to 200 trials. The election term of a winning server was arbitrarily chosen to be 120 seconds.

5.2 Performance Analysis

5.2.1 Timing Analysis

The minimum time required to choose a new leader is the time required by the system to converge onto a leader. The goal is to keep this value as low as possible while maintaining the security requirement and consensus. By achieving this, we can assure service continuity and aliveness with very minimum down time during migration. Figure 5.1, 5.2 and 5.3 shows the summary of the performance of the protocol and its phases. In Figure 5.1(a), we show the minimum average time for leader election, which shows a minimum time of 25 ms for 5 nodes and about 250 ms for 50 nodes. In Figure 5.1(b), we show the average election time for each network configuration, which varies from 10s of milliseconds, for 5 nodes, to about 400 ms, for 50 nodes. In Figure 5.2(a), the average time taken for detecting Byzantine faults was in the range of 5 to 70 ms as the number of nodes increased from 5 to 50. In Figure 5.2(b),5.3(a) and 5.3(c) the execution times of the key phases of our protocol is shown over 50 iterations, which include the timeout values, the message transmission times and the computation required for each phase. The $Commit$ and $Estimate$ phases averaged from 10 ms up to 90 ms for 5 up to 50 server configurations, respectively.

The execution times amount to $0.008 - 0.3\%$, of the election term which is manually calibrated to 120 seconds. This shows that our protocol is fast and efficient. This also provides more leverage for the system administrator to adjust the election term according the minimum time required to explore vulnerabilities and break the system. Out of the four phases, the $Confirm$ phase required more time as it includes the validation of the committed $\mathcal{C}$ values from various nodes and selecting the minimum of the valid values.

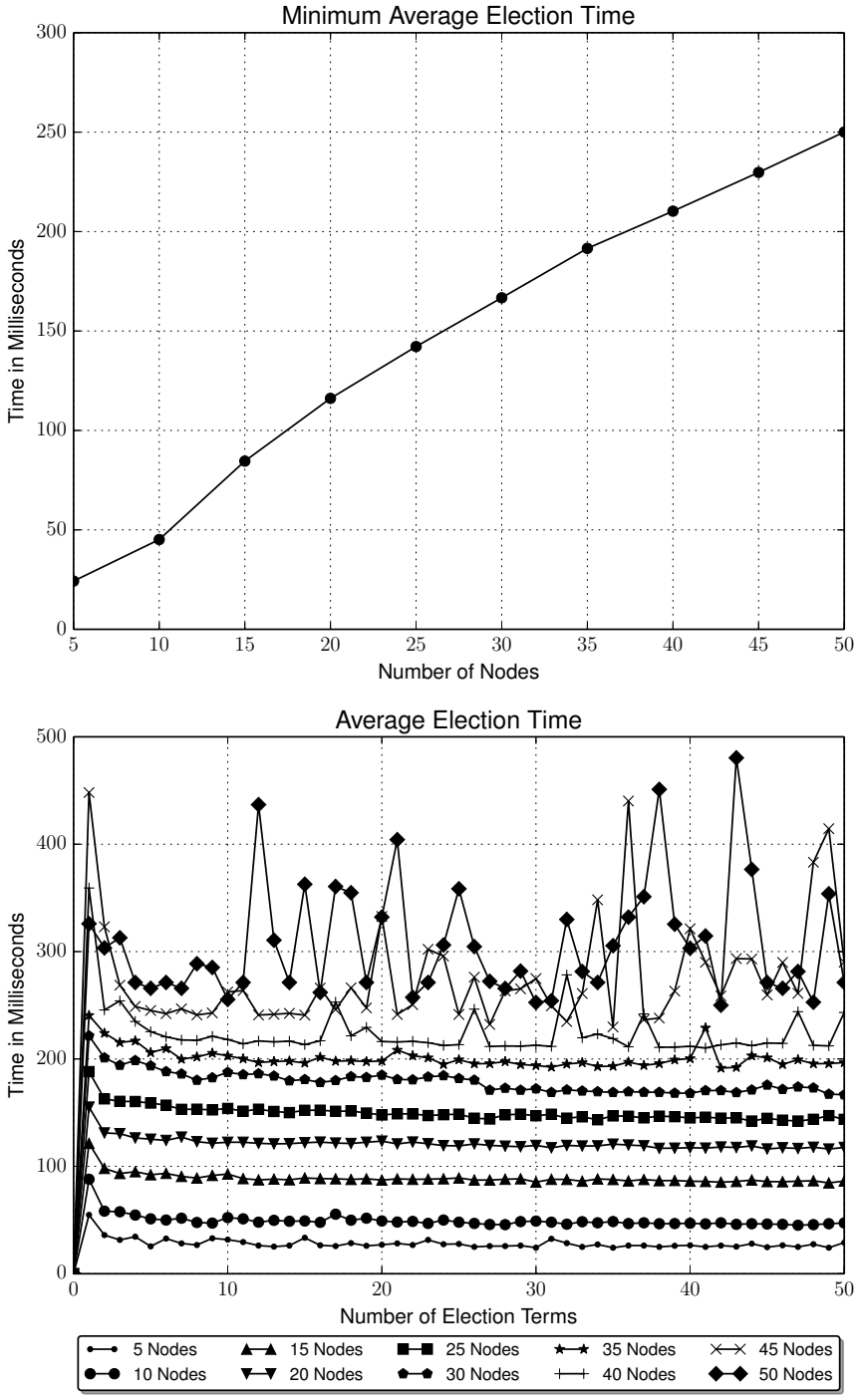


Figure 5.1: (a) Minimum Average Election Time
(b) Average Election Time over 50 iterations

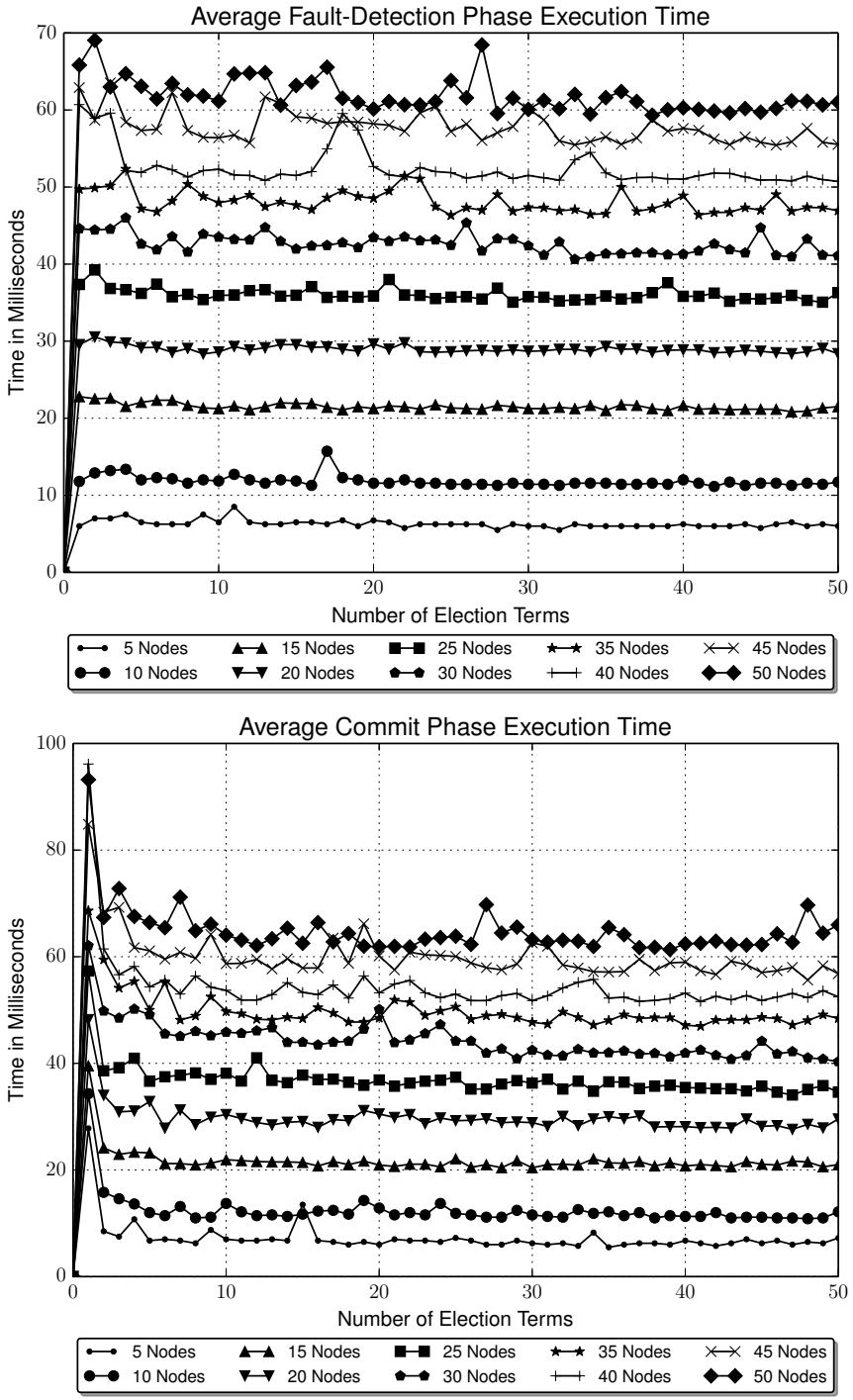


Figure 5.2: (a) Fault Detection Phase Execution Time (b) Commit Phase Execution Time

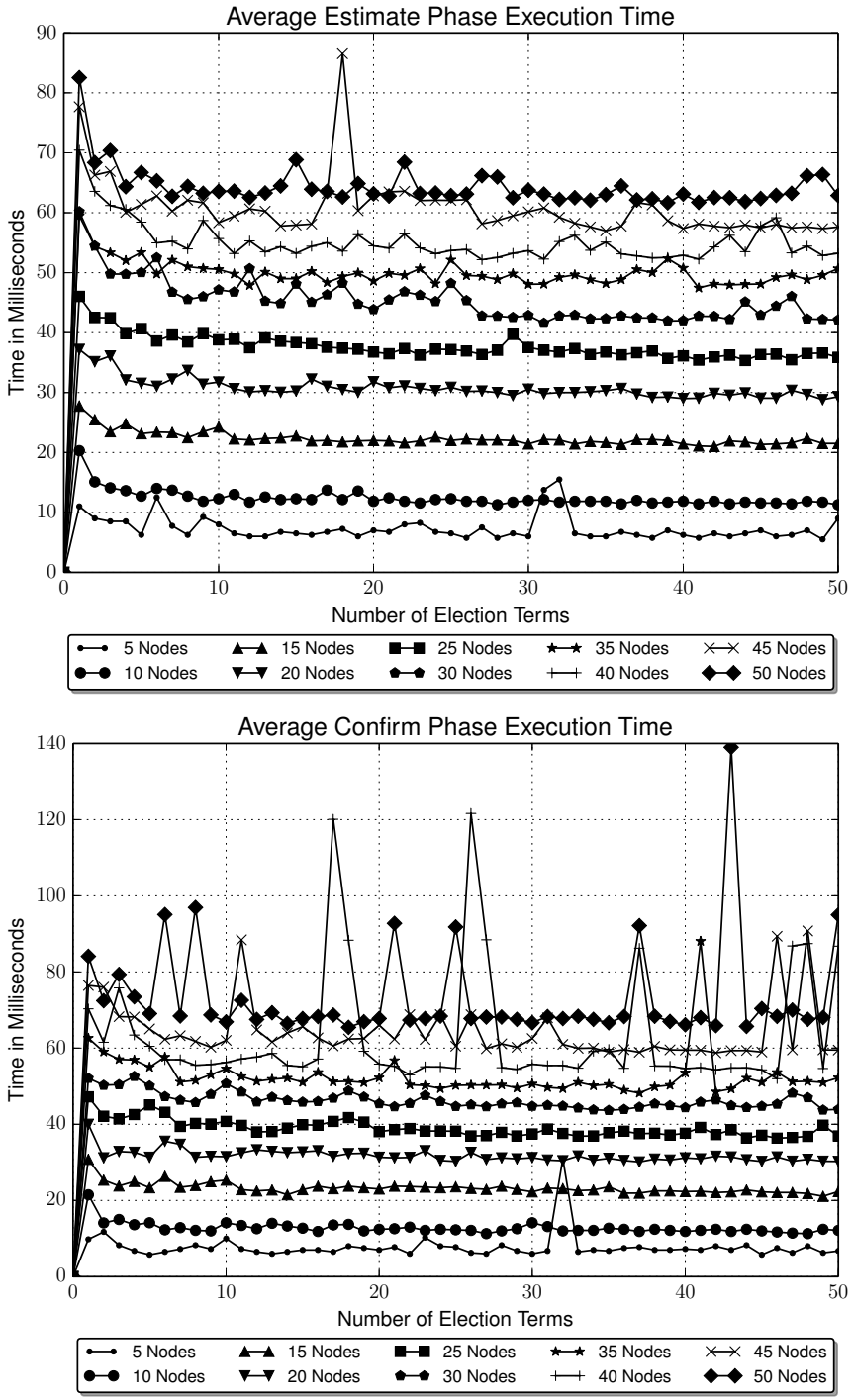


Figure 5.3: (a) Estimate Phase Execution Time (b) Confirm Phase Execution Time

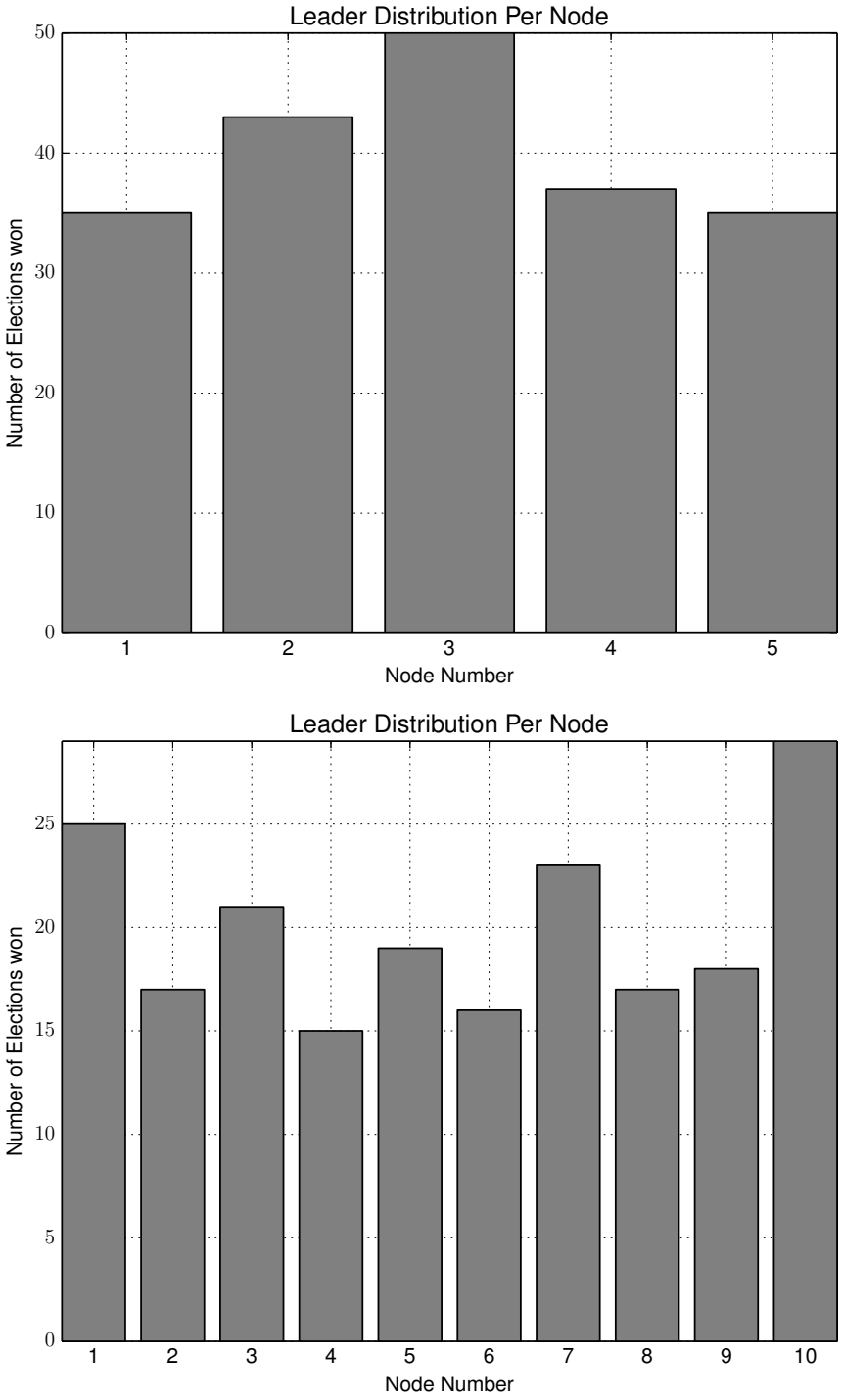


Figure 5.4: Leader Distribution Plots for (a) 5 Nodes (b) 10 Nodes network

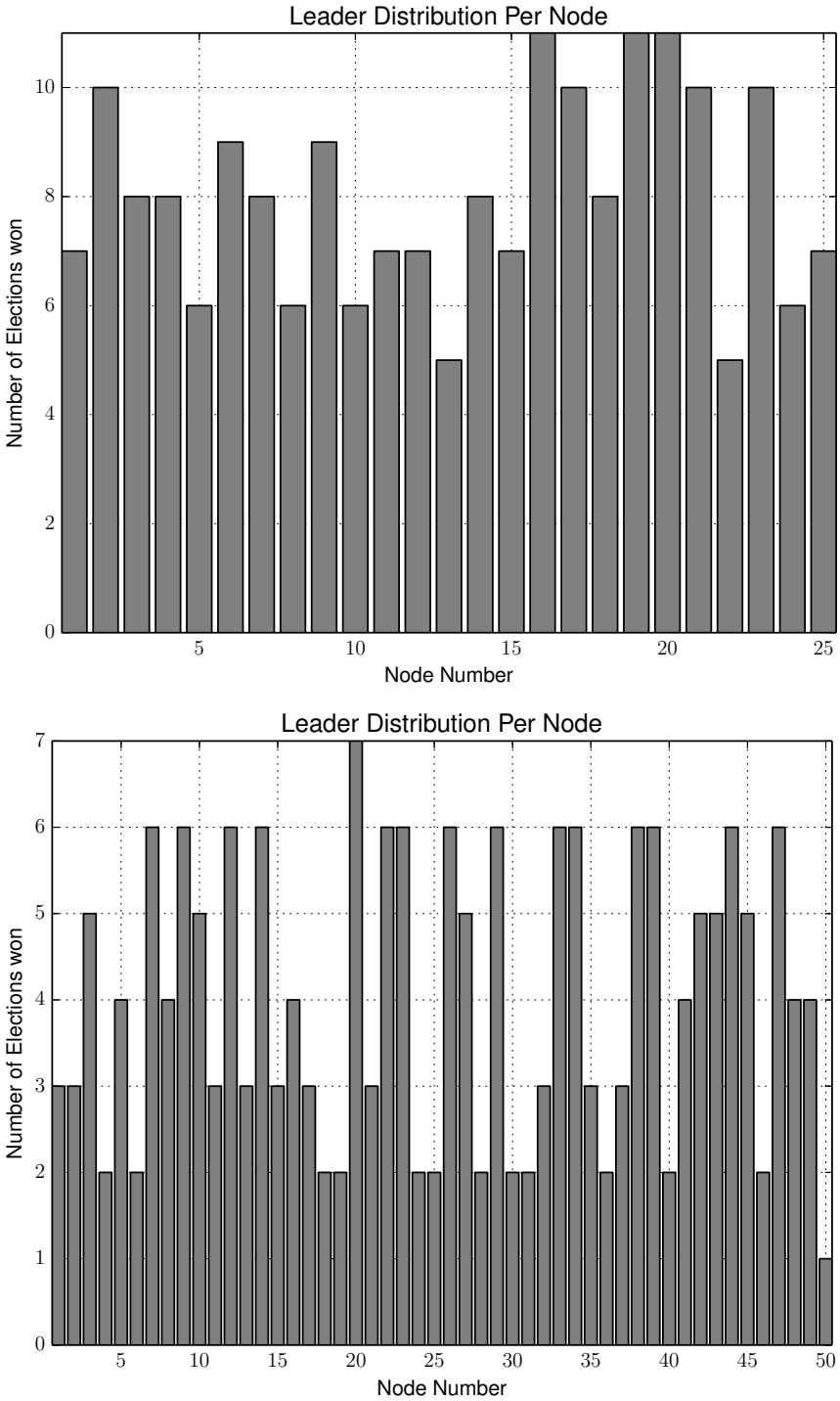


Figure 5.5: Leader Distribution Plots for (a) 25 Nodes (b) 50 Nodes Network

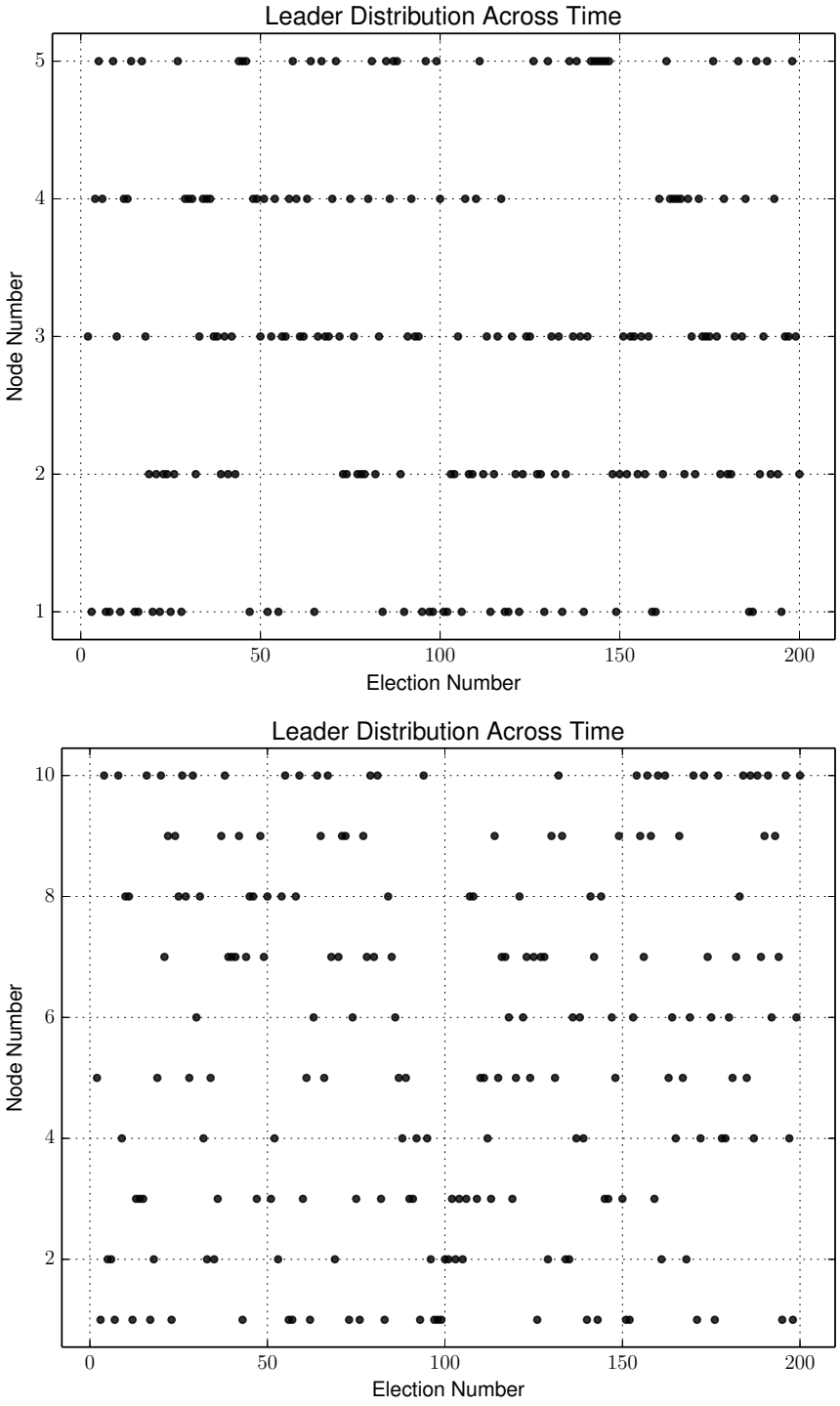


Figure 5.6: Scatter Plots for (a) 5 Nodes (b) 10 Nodes Network

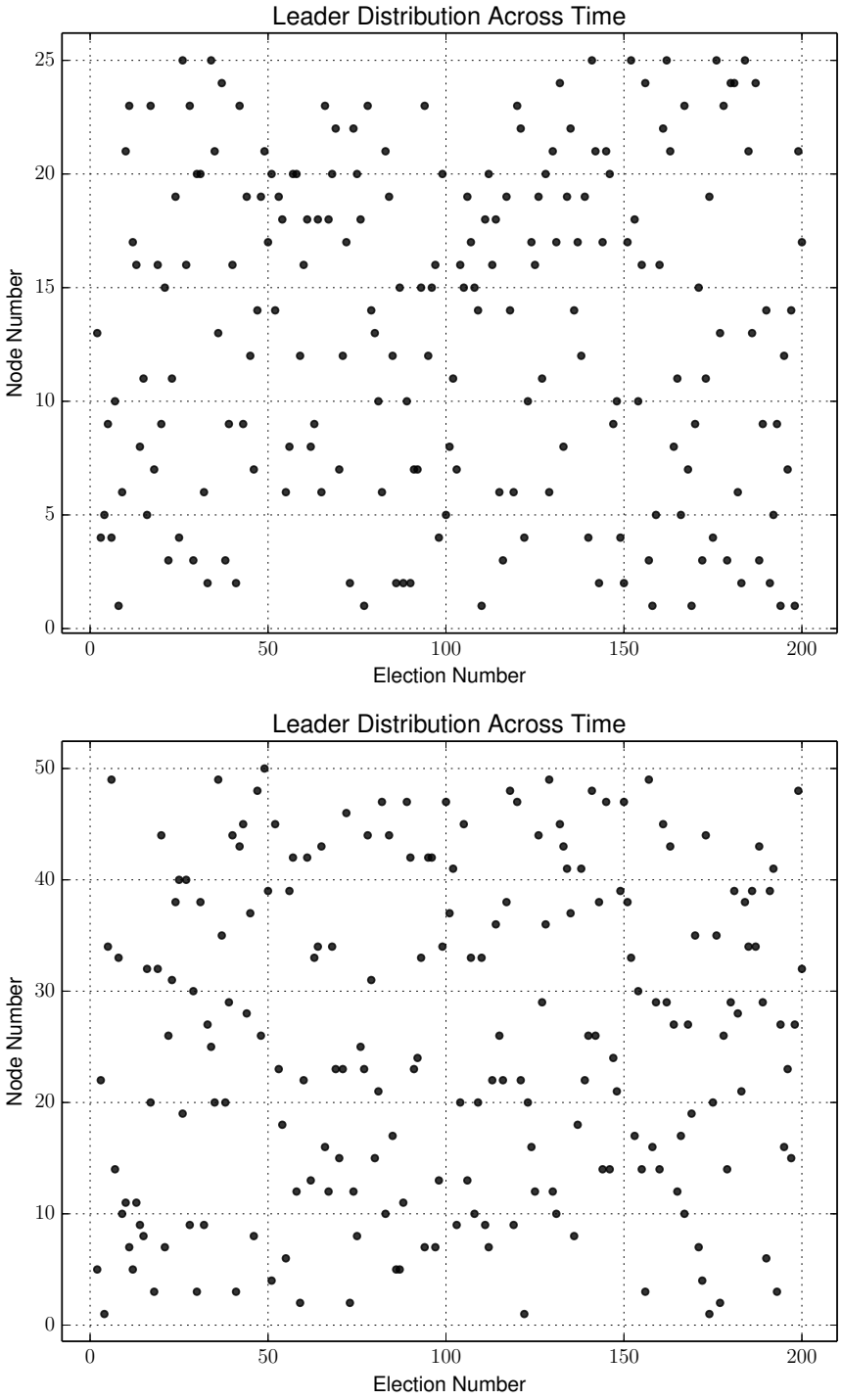


Figure 5.7: Scatter Plots for (a) 25 Nodes (b) 50 Nodes Network

5.2.2 Leader Distribution

In Figure 5.4 and 5.5 , we show the number of times a node is elected as leader across 200 election terms for server configuration of $5,\ 10,\ 25$, and 50 nodes. From these plots, we can see that the leader is distributed across the spectrum of nodes taking part in the algorithm. This means that any given node is equally likely to be the leader and our protocol maintains uniform distribution of leader elected for a given amount of election terms.

The leader distribution on scatter plots in Figure 5.6 and 5.7 shows that there are many instances of the same node elected as a leader more than once. However, the plots show a distribution where there is not regular pattern over 200 iterations. This clearly demonstrates that our approach is successful in choosing leader uniformly at random.

5.2.3 Failure Scenarios

The test-bed was designed considering various failure scenarios. When the leader fails after being elected, the protocol follows a policy of no-service for that election term. This could be avoided by leader periodically sending "heart-beat" or "alive" messages to all other nodes in the network. If nodes do not receive the messages for certain amount of time during the election term, they trigger a new election. However, this is additional complexity that is not considered in our protocol.

We have considered different failure scenarios: *node failures during protocol phases,* and *leader failures*. When the node fails in between the protocol phases, other nodes would immediately detect the faulty node and vote to exclude from the protocol. When a node fails before, after the election or within the election term, the other honest nodes would detect the faulty node in the next election term and place it in the faulty list. The system administrator manually fixes the nodes and adds it back to the network. For leader failure scenarios, when the leader fails during the election term, the server node is put in faulty node in the next election and other nodes continue to choose the next leader. If the leader node fails right after the $CONFIRM$ phase, the service is down for one election term and the faulty node is voted out in the next election term.

Moreover, any deviations in the values shared by nodes during the phases of the protocol is also immediately detected by other nodes and excluded from the protocol by adding it to the faulty list. Message constructs and signature verification is tightly coupled with the fault detector and any deviations would lead the node being added to the faulty list. This is how we have designed and tested different failure scenarios in our protocol.

Chapter 6

General Solution for Large Enterprise Networks

In the previous chapter, we explained the leader election protocol and its constructs. The protocol solves the problem of selecting a leader node after reaching a consensus by all the nodes taking part in the protocol. However, as the number of nodes taking part in the election protocol increases, the communication complexity increases as well as the protocol takes longer time to elect a leader (refer to section 6.5.3). Moreover, a node is chosen within a fixed set of node servers communicating and coordinating over a network that is exposed to the outside world. An attacker could patiently unveil all the nodes taking part in the protocol and plan the attack by enumerating the nodes one by one. This can lead to a total compromise of the system.

To resolve this issue, we describe another protocol that chooses a subset of nodes that are allowed to take part in the leader election protocol. Using the concept of *Random Walk* [59], each selected node in the network is delegated a task of choosing another node uniformly at random to include in the subset. This process goes on until the predefined number of node servers are added in a subset. All the nodes in the subset wait until the current leader's election term is completed to start the next leader election protocol. We name this protocol as *Random Walk Protocol*.

In this chapter, we describe the random walk protocol and its constructs in detail. In the next chapter we discuss the results and the performance analysis of random walk protocol.

6.1 Technical Challenges and Solutions

The first challenge is to choose a subset of nodes M, such that the nodes are selected uniformly at random in the distributed network. A central authority cannot be used to choose the subset because of its vulnerability to manipulation and attacks. To resolve this, the nodes are delegated the act of choosing the next node in the subset and this serialized process needs to start from a verifiable source to avoid attackers manipulating the subset to win the election. Using the public parameters, one-way hash used in the leader election protocol, and modulo arithmetic to choose the

next node, we are able to resolve this challenge. Once the winner or *"Leader"* node is confirmed by the election protocol, it would start the process of random walk by choosing the first node in the subset. The leader would then delegate the random walk process to the chosen node T_1 to continue this process. The node T_1 chooses the next node T_2 in the subset and delegates the random walk process. This act of delegation continues until the total number of m nodes are present in the subset. Here, the leader is a verifiable source to start the random walk process because the leader is known to all other nodes during the end of the election protocol. This way we could solve the challenge of choosing the subset of nodes uniformly at random in a distributed fashion.

The second challenge is to make sure that subset does not include invalid nodes, *i.e.*, without being chosen, the nodes do not include themselves to be part of the election process. To deal with is, we have designed a verifiable delegated message signature scheme that helps traced back to the origin of random walk process. Every node that takes part in this process would publish all the public parameters used to choose the next node in the subset, include the delegating message and its signature from the previous node that delegated this process, sign the encapsulated message and send it to next chosen node as the part of delegation to continue the random walk. The next node receiving this message would first, verify the signature, public parameters and then continue with the random walk process until the total number of m nodes are present in the subset. This way the source of the delegation can be verified and assures that an invalid node cannot be included in the subset.

The third challenge is to make sure that the protocol converges within the election term of the leader. If the subset is not created within the election term, this would break in the chain leading to denial of service. Also if the node is faulty or down at the time of random walk delegation, the delegating node needs to take some countermeasures to continue the random walk process. To make sure the algorithm converges with the election term, we introduced timeouts to monitor the random walk delegation process. Once the node, delegates the random walk process to the next chosen node, the delegating node would wait for a predefined time for an acknowledgement message from the delegated node. If the delegating node does not receive the acknowledgement

message within the predefined time, the node would re-run the random walk process to choose another node as the next delegated node to continue the random walk process. The fourth and the final challenge is not allow faulty and malicious nodes to be part of the subset. This is achieved by the fault detector mechanism used in the leader election protocol. During the random walk process, a node needs to make sure that the next chosen nodes is not included in the fault list. Moreover, all the observations made during the random walk process is shared with the other nodes during the leader election protocol to decide if the nodes are faulty and vote to add them to the fault list.

6.2 Threat Model

The threat model is similar to the model defined for the leader election protocol. We assume that the skilled adversary is able to enumerate the server vulnerabilities and launch the attack. An attacker would require a certain amount of time to explore the vulnerabilities in the server. Also the attacker would require time to device the attack and launch them.

We are also assuming that the network consists of a minimum of 4 or nodes according to the byzantine consensus protocol [10]. But the assumption here is slightly different than the one made for the leader election protocol. In the leader election protocol described in chapter 4, we assume that for a network with n nodes, the protocol can handle a maximum of $k = \lfloor \frac{n-1}{3} \rfloor$ faulty nodes. In this case, the total number of nodes n is fixed by the system administrator and resiliency k is evaluated using the formula $k = \lfloor \frac{n-1}{3} \rfloor$. Here, we can say that the leader election protocol is resilient to k faults.

Suppose the system administrator builds a system by fixing the resiliency value k. If a system is designed of be resilient against k faults, then a minimum of $n = 3k+1$ nodes need to participate in the algorithm. For example, if the system is required to withstand $k = 33$ faulty nodes, there must be $n = 3 \cdot 33 + 1 = 100$ nodes participating in the protocol.

Hence, the main assumption for the random walk protocol is as follows: For a network of n nodes, the number of malicious nodes in the network is always less than or equal to k, where k is the resiliency value of the system with n nodes fixed by the system administrator. With this, the

minimum number of nodes required in the subset M is given by $m = 3k + 1$, where $m < n$. Here, we are choosing the minimum number of nodes required in the subset M to withstand up to k faulty or malicious nodes. This makes sure that for all election terms, the leader election protocol can withstand up to k faulty or malicious nodes in the subset M.

For example, let the total number of nodes n in the network be 100. If the system administrator decides the resiliency value k to be 10, the minimum number of nodes in the subset to be chosen is $m = 3 \cdot 10 + 1 = 31$ nodes, which is less than $n = 100$ nodes in the network.

Furthermore, a skilled adversary has prior knowledge on the history of random walk messages and is able to brute-force them by controlling one or more nodes in the network, at the most k nodes in the network. The attackers intent is to somehow include the malicious node in the subset so that the malicious node can get chosen as a leader or to disrupt the system operation by stalling the random walk protocol.

6.3 Public Parameters

The main goal of this protocol is to choose a subset of M servers: $\{T_1, T_2, \cdots, T_m\} \subset N$, where $\{S_1, S_2, \cdots, S_n\}$ are the set of N heterogeneous servers. Like in the previous protocol, every node server taking part in the algorithm is initialized with two public parameters, 1) Let P denote a large public value used in a particular election round i, 2) n be the total number of node servers in the network taking part in the algorithm.

In the random walk protocol, we are going to make use of the large public value P. If a node participates in the election round, P is updated after the round, say P^i. If this node is chosen again to take part in the leader election protocol, this value P^i is used in this protocol as one of the inputs to the one way hash function $\mathbb{H}$ along with the random value chosen by the node. After the hash operation is completed, the value of P^i is incremented as follows: $P^{i+1} = P^i + 1$. The nodes are allowed to use older values of P^i and these values are shared as a part of the delegated message verification scheme. Just like in the leader election protocol, a threshold is maintained beyond

which a node with older P^i value is treated as faulty. We use the notation P_k^i represents the local value of P in T_k node for ith round, where $T_k, \forall 1 \leq k \leq m$ in the subset M.

The resiliency of the system k is fixed by the system administrator. With this value, the minimum number of nodes in the subset M is calculated using the formula $m = 3k + 1$, where $m < n$, the total nodes in the network. Higher the resiliency value k, more the minimum number of nodes required in the subset to take part in the leader election algorithm.

6.4 Walk Function

This function is responsible to choose the next node added to the subset of m servers: $\{T_1, T_2, \cdots, T_m\} \subset N$. The function is only called by the delegated node T_k, $\forall 1 \leq k \leq m$ once the delegation message is received by the previous node T_{k-1}. This function uses a pseudo-random one-way hash function $\mathbb{H}$ similar to that of the hash commitment scheme in the previous chapter. The property of being extremely difficult to reverse the hash function, lesser collusion with higher bit-order hash function and less computation time helps the protocol converge faster. This is an important security feature against attackers who are trying to break the protocol. We are using a secure one-way hash function $\mathbb{H}$ such as SHA- 256, to calculate the next chosen node T_{k+1}, $\forall 1 \leq k \leq m$ using the random value $\mathcal{V}$. The random value $\mathcal{V}$ is different than the random value $\mathcal{R}$ used in the leader election protocol.

Let assume $\{S_1, S_2, \cdots, S_n\}$ are the set of n heterogeneous servers that are taking part in the leader election algorithm. The random walk protocol has to choose a subset of m servers: $\{T_1, T_2, \cdots, T_m\} \subset n$. Let $\mathbb{H} : \{0,1\}^* \rightarrow \{0,1\}^t$ denote a strong pseudo-random one-way hash function that takes any length input and generates a fixed t-bit output. Each delegated node in the subset T_k, $\forall 1 \leq k \leq m$ calculates the **One-way Hash Walk Function** to choose the next node T_{k+1} such that,

$$T_{k+1} = \mathbb{H}(\mathcal{V}_k, P_k^i) \mod n + 1$$

The T_{k+1} result from the walk function also need to conform to all the following conditions.

- T_{k+1} result is not same as the current node identifier T_k.
- T_{k+1} result is not same as the current Leader L.
- T_{k+1} result is not previously added to the subset M to avoid duplication, *i.e.*, $\{T_{k+1}\} \notin M$
- T_{k+1} result is not in the faulty list of the Leader Election protocol.

If any of the above conditions fail, the node updates the public value P, generates another random value $\mathcal{V}$ and calculates a new T_{k+1} using the walk function $\mathbb{H}$. In the second condition, the protocol does not allow the leader to be re-elected in the next consecutive election term. This is because the leader of the current term does not allow itself or other chosen node to add the current leader L to the subset of the next election term. This avoids one node winning two consecutive election terms.

Each delegated node that chooses the next node T_{k+1} would transmit the delegation message with random value $\mathcal{V}_k$ and public value P_k^i used in the walk function for verification. The chosen node would then verify the origin of the message using the delegated message signature scheme and continue with random walk protocol until m required number of nodes are added to the subset.

6.5 Random Walk Protocol

6.5.1 Overview

The goal of this protocol is to choose a verifiable subset of nodes in a distributed network using random walk. This subset of nodes then take part in the leader election protocol to choose a leader that hosts the service for that election term. The extra step of choosing a subset of nodes is performed in order to increase the unpredictability of where the service is hosted and reduce the scaling problems and server enumeration.

During the start of the service hosting, the first leader is provided by the system configuration. The leader hosts the service and then starts the random walk process. Lets consider a node with

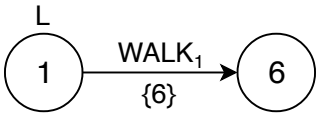


Figure 6.1: Random Walk First Message

identification number 1 to be the leader configured that starts the random walk protocol. Lets assume that the total number of nodes in the network n is 10. Lets say that the resiliency k is chosen as 1 for this example. Hence the number of nodes in the subset M is calculated as $m = 3 \cdot 1 + 1 = 4$. Node 1 computes the next chosen node T_1 using the walk function explained in section 6.4 and sends the $WALK_1$ message to node 6 as shown in the figure 6.1.

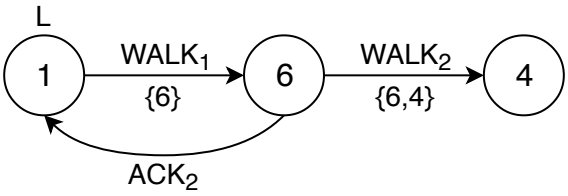


Figure 6.2: Random Walk Second Message and Acknowledgment

Once the node 6 receives the $WALK_1$ message, the node would first verify the source of the message and authenticity of the $WALK_1$ delegation message. Once the verification process is completed, the node would choose the next node T_2 using the walk function. For example, the next node chosen is 4. The subset with chosen identification numbers and required information is packed in the second $WALK_2$ message and sent to node 4 to continue the process. As a part of acknowledgement to the delegating node 1, the node creates an ACK_2 message with the chosen node along with other information and send it to node 1 as show in the figure 6.2.

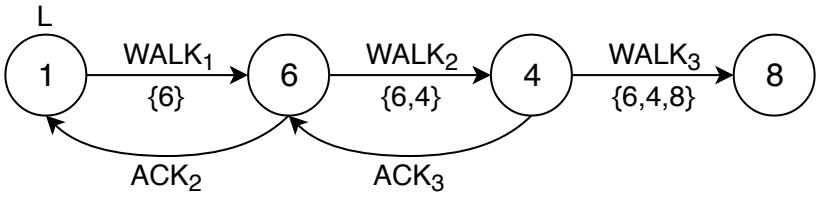


Figure 6.3: Random Walk Third Message and Acknowledgment

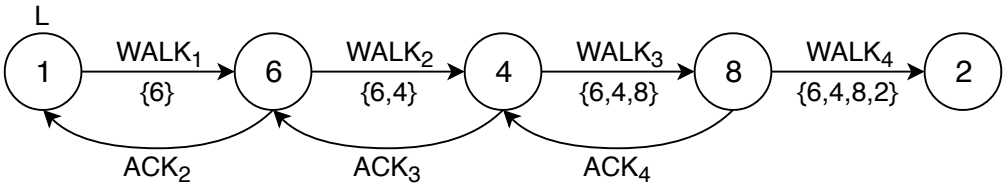


Figure 6.4: Random Walk Fourth Message and Acknowledgment

Node 4 after receiving the $WALK_2$ message would repeat the process of verification, chooses the next node 8 and sends the $WALK_3$ delegation message. It would also send the ACK_3 message to node 6 as shown in the figure 6.3. Node 8 performs the same operation and chooses the next node 2 as shown in the figure 6.4.

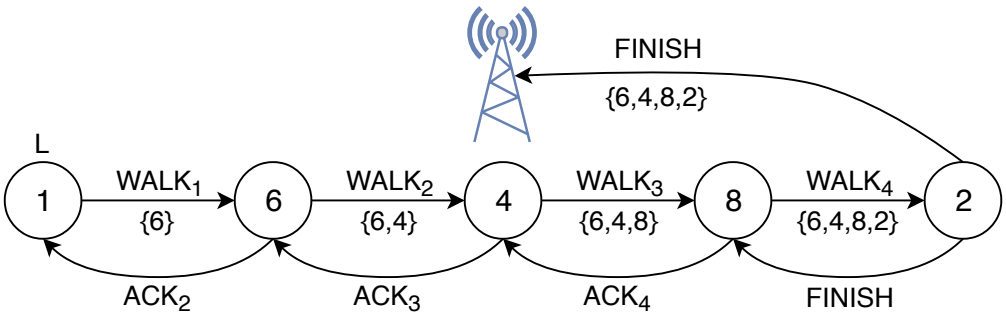


Figure 6.5: Finalization and Broadcast

When node 2 receives the $WALK_4$ message, it validates that the random walk process completed the process of choosing $m = 4$ nodes $\{6, 4, 8, 2\}$ in the subset M. This node would broadcast a $FINISH$ message containing the subset to all other nodes in the network as shown in the figure 6.5. All the other nodes in the network would verify the message and the subset.

Now that the subset is chosen and broadcasted to the rest of the network, every node in the subset prepares to start the election once the leader node completes its term. The leader election protocol on the subset would choose the next leader for election term. That leader would then start the random walk protocol to choose the next subset.

Now that the leader election protocol is executed only on the subset, all the protocol message exchanges occur within the nodes of the subset. During the Fault Detection phase of the protocol

explained in chapter 4 section 4.5.5, the suspected nodes are voted out and added to the *fault* list. This $fault$ list is broadcasted to all the nodes in the network. This is to maintain the uniformity of the $fault$ list across all nodes in the network and help nodes exclude faulty nodes during the random walk protocol.

6.5.2 Delegated Message Signature Scheme

The scheme defines a message format that the delegating node uses to create the $WALK$ and ACK message. This message format encapsulates the delegation message from the previous delegating node as proof of delegation. This scheme is used by every delegated node to verify the source, validate the walk function result shared by the delegating node as well as other nodes chosen in the subset. In this section, we are going to talk about this scheme in detail.

There are 3 main messages in this protocol: $WALK, ACK$ and $FINISH$. We use T to denote the node-identifier field, P to denote the public-parameter field, and $\mathcal{V}$ to denote the random value field. We use σ_{T_k} to denote the private-key signature of a message by the sending node T_k where k means "current sending node".

The $WALK$ message parameter has 3 fields, $\{T, \mathcal{V}, P\}$. The message from the previous delegating node is encapsulated with the message sent to the next delegated node. Lets consider the overview to explain the scheme, figure 6.1 shows that the leader node 1 selects node 6 and adds to subset M. It then sends the $WALK1$ message to node 6. The message format is as follows.

$$WALK_1 = \{T_L \quad WALK \quad \{T_1, \mathcal{V}_{T_L}, P^1_{T_L}\}\}_{\sigma_{T_L}}$$

where in this case, T_L is the leader node 1, T_1 is node 6 which is the result of the walk function $T_1 = \mathbb{H}(\mathcal{V}_{T_L}, P^1_{T_L}) \mod n + 1$. Here, n is the total number of nodes in the network, $\mathcal{V}_{T_L}$ is the random value chosen by leader node 1, and $P^1_{T_L}$ is the public value used by the leader node 1 for the 1st election term. After the hash operation is completed, the value of $P^2_{T_L}$ is incremented as follows: $P^2_{T_L} = P^1_{T_L} + 1$.

Once the node 6 receives the $WALK_1$ message, it first verifies the signature and the walk function result. Once verified, it would then choose a random value $\mathcal{V}_{T_1}$ and calculate the next chosen node $T_2 = \mathbb{H}(\mathcal{V}_{T_1}, P^1_{T_1}) \mod n+1$. It would then create the following $WALK_2$ message and send the message to node 4 as shown in the figure 6.2.

$$WALK_2 = \{T_1 \quad WALK \quad \{T_1, T_2, \mathcal{V}_{T_1}, P^1_{T_1}\}[WALK_1]\}_{\sigma_{T_1}}$$

Here, we can see that the delegating $WALK_1$ message received from leader node 1 is encapsulated within [].

The ACK message is similar to that of walk message except the encapsulation of delegating $WALK$ message received from leader node 1.

$$ACK_2 = \{T_1 \quad ACK \quad \{T_1, T_2, \mathcal{V}_{T_1}, P^1_{T_1}\}\}_{\sigma_{T_1}}$$

Node 4 receives the $WALK_2$ messages, verifies the source and walk function result, creates the following $WALK_3$ message as shown below and sends it to the next chosen node 8. Also, it creates the ACK_3 message as shown below and sends it as an acknowledgement to its delegating node 6 as shown in the figure 6.3.

$$WALK_3 = \{T_2 \quad WALK \quad \{T_1, T_2, T_3, \mathcal{V}_{T_2}, P^1_{T_2}\}[WALK_2]\}_{\sigma_{T_2}}$$
$$ACK_3 = \{T_2 \quad ACK \quad \{T_1, T_2, T_3, \mathcal{V}_{T_2}, P^1_{T_2}\}\}_{\sigma_{T_2}}$$

Node 8 performs the same operation, creates the following messages and sends $WALK_4$ message to node 2 and ACK_4 message to node 4 as shown in the figure 6.4. The message formats are as follows.

$$WALK_4 = \{T_3 \quad WALK \quad \{T_1, T_2, T_3, T_4, \mathcal{V}_{T_3}, P^1_{T_3}\}[WALK_3]\}_{\sigma_{T_3}}$$
$$ACK_4 = \{T_3 \quad ACK \quad \{T_1, T_2, T_3, T_4, \mathcal{V}_{T_3}, P^1_{T_3}\}\}_{\sigma_{T_3}}$$

As the random walk process completes, the last chosen node 2, sends the $FINISH$ message to all the nodes in the network. With the encapsulation strategy mentioned above, the $FINISH$ message would contain all the $WALK$ messages for verification. This delegated message signature scheme is able to verify all the nodes chosen in the subset M and will lead the verification to the starting point, *i.e.*, the leader. Following is the message format for $FINISH$ message.

$$FINISH = \{T_4 \quad FINISH \quad [\{T_3 \quad WALK \quad \{T_1, T_2, T_3, T_4, \mathcal{V}_{T_3}, P^1_{T_3}\}$$
$$[\{T_2 \quad WALK \quad \{T_1, T_2, T_3, \mathcal{V}_{T_2}, P^1_{T_2}\}[WALK_2]\}_{\sigma_{T_2}}]\}_{\sigma_{T_3}}]\}_{\sigma_{T_4}}$$

or

$$FINISH = \{T_4 \quad FINISH \quad [WALK_4]\}_{\sigma_{T_4}}$$

In general, the message formats for $WALK, ACK$ and $FINISH$ are as follows.

$$WALK_k = \{T_{k-1} \quad WALK \quad \{T_1, T_2 \cdots T_k, \mathcal{V}_{T_{k-1}}, P^i_{T_{k-1}}\}[WALK_{k-1}]\}_{\sigma_{T_{k-1}}}$$
$$WALK_{k-1} = \{T_{k-2} \quad WALK \quad \{T_1, T_2 \cdots T_{k-1}, \mathcal{V}_{T_{k-2}}, P^i_{T_{k-2}}\}[WALK_{k-2}]\}_{\sigma_{T_{k-2}}}$$
$$WALK_{k-2} = \{T_{k-3} \quad WALK \quad \{T_1, T_2 \cdots T_{k-2}, \mathcal{V}_{T_{k-3}}, P^i_{T_{k-3}}\}[WALK_{k-3}]\}_{\sigma_{T_{k-3}}}$$
$$WALK_2 = \{T_1 \quad WALK \quad \{T_1, T_2, \mathcal{V}_{T_1}, P^1_{T_1}\}[WALK_1]\}_{\sigma_{T_1}}$$
$$WALK_1 = \{T_L \quad WALK \quad \{T_1, \mathcal{V}_{T_L}, P^i_{T_L}\}\}_{\sigma_{T_L}}$$

$$ACK_k = \{T_{k-1} \quad ACK \quad \{T_1, T_2 \cdots T_k, \mathcal{V}_{T_{k-1}}, P^i_{T_{k-1}}\}\}_{\sigma_{T_{k-1}}}$$
$$ACK_{k-1} = \{T_{k-2} \quad ACK \quad \{T_1, T_2 \cdots T_{k-1}, \mathcal{V}_{T_{k-2}}, P^i_{T_{k-2}}\}\}_{\sigma_{T_{k-2}}}$$
$$ACK_{k-2} = \{T_{k-3} \quad ACK \quad \{T_1, T_2 \cdots T_{k-2}, \mathcal{V}_{T_{k-3}}, P^i_{T_{k-3}}\}\}_{\sigma_{T_{k-3}}}$$
$$ACK_2 = \{T_1 \quad ACK \quad \{T_1, T_2, \mathcal{V}_{T_1}, P^i_{T_1}\}\}_{\sigma_{T_1}}$$

$$FINISH = \{T_k \quad FINISH \quad [WALK_k]\}_{\sigma_{T_k}}$$

ACK_1 message does not exist because the Leader node is the one that starts the random walk protocol and delegates the $WALK_1$ message to T_1 node. Algorithm 5 is the pseudo code for verifying each chosen node in the $WALK$ message.

Algorithm 5: Delegated Message Signature Verification Algorithm

Input: $W = \{W_1, \cdots, W_k\}$ where $W_j = \{T_j, \mathcal{V}_{T_{j-1}}, P^i_{T_{j-1}}\}$ after processing the encapsulated message

Output: $WALK$ message $VALID$ or $NOT \ \ VALID$

```
Let Valid = {φ}
for W_j ∈ {1, ..., k} do
    if [T_j == H(V_{T_{j-1}}, P^i_{T_{j-1}})  mod n + 1] then
        Valid = Valid ⋃ T_j
    end if
end for
```

6.5.3 Communication Complexity Analysis

From the leader election protocol described in chapter 4, the communication complexity is given by the number of messages exchanged in the network to elect a leader for each election term. Every node sends and receives n messages in each round of the protocol. Hence, there are n^2 message exchanges in each round. There are 4 rounds described in the algorithm. Therefore, The value is calculated as $4n^2$, where n is the total number of nodes in the network. As the number of nodes participating in the algorithm increases, the number of messages exchanged also increases exponentially. From the results of the leader election protocol describes in chapter 5, the time required to elect a leader also increases linearly with respect to the number of nodes participating in the protocol in section 5.2.1.

Considering the random walk protocol choosing a subset of m nodes from a network of n nodes, we show that the communication complexity is minimized as compared to all n nodes participating in the leader election protocol.

We know that number of nodes in the subset m is given by

$$m = 3k + 1, m < n, \text{where } k \text{ is fixed}$$

Number of message exchanges by running the leader election protocol on the subset M with m nodes is given by

$$\text{Number of Message Exchanges} = 4m^2$$

Random Walk protocol includes sequential messages exchanges between the chosen and the final broadcast. There are $m\,WALK$ messages, $(m-1)\,ACK$ messages and $n\,FINISH$ messages exchanged during the random walk protocol.

Hence, the total message exchange is given by

$$\text{Total number of Message Exchanges} = 4m^2 + (2m - 1) + n$$

The total message exchanges by choosing a subset and executing the leader election protocol within that subset is always less than all the nodes in the network participating in the election protocol. Lets illustrate this with an example.

Lets say $n = 100$, if all nodes participate in the election protocol, the number of messages exchanges is given by $4 \cdot 100^2 = 40000$

If $k = 10$, m can be calculated as $3 \cdot 10 + 1 = 31$. The total number of messages exchanges by choosing the subset is $4 \cdot 31^2 + 2 \cdot 31 - 1 + 100 = 4005$, which is $\approx 10\%$ of the number of messages exchanges when all the nodes in the network participate in the election protocol.

6.5.4 Time-Outs

In this protocol, time-out value: Z is used to make sure that the delegated node continues the random walk protocol by waiting for the ACK message. This time-out value is common to all the nodes in the network and provided using the system configuration. It is chosen by measuring the time required for the nodes to perform local computations, prepare the messages with results and broadcast them to the rest of the network. In our protocol, we choose the time-outs within a Δ-fraction of the estimated round-trip network delay, $RTDelay$, such that

$$Z_p = (RTDelay + \Delta * RTDelay + f(n)) \;\; where \;\; 0 < \Delta \leq 1$$

Also, f is a network dependent time function that accounts for the size of the network and the type of broadcast capability of the network.

6.5.5 Fault Detection

The fault detection in this algorithm works in conjunction with the fault detection phase of the leader election algorithm. When a delegating node does not receive an ACK message from the delegated node, the node would re-run the walk function and update the public value. A new $WALK$ message is created and sent the new delegated node to continue the random walk process. The faulty node is added to the suspected list and the information is shared with the network during the fault detection phase. The node that is added to suspected node list from the random walk protocol is not erased after each election term. When majority of the nodes in the network vote to exclude the faulty node, the node is then put to a faulty node list and excluded from both the protocols.

6.6 Security Analysis

In this section, we are going to discuss the attack scenarios described the threat model and talk about the security of our protocol. For this analysis, we consider an *Adaptive Byzantine*

Adversary who has the ability to modify the inputs to the protocol by (partially) controlling a threshold number of servers, not necessarily the same, in any given session.

6.6.1 Adaptive Byzantine Adversary

The definition of an adaptive adversary $\mathcal{A}$ in the context of our protocol is as follows:

Definition 1: An adaptive Byzantine adversary $\mathcal{A}$ is a Byzantine adversary who has access to all the inputs and results of the protocol for a polynomial q number of random walk subset sessions, *i.e.*, the attacker knows all the public values, random values and corresponding node values in the subset. The adversary is also in control of some of the inputs, not more than $j = \lfloor \frac{m-1}{3} \rfloor$, which is required for a safe quorum to be achieved. We denote the adversary storage by: $A = \{A_1,\ A_2, \cdots,\ A_q\}$ where $A_i = \{(T_1^i, \mathcal{V}_{T_L^i}, P_{T_L}^i), \cdots,\ (T_1^i. \cdots, T_k^i, \mathcal{V}_{T_{k-1}}^i, P_{T_{k-1}}^i)\}$ denotes the i^{th} protocol session's input-output pairs.

With this, we analyze the security threats in random walk.

Lemma 1 (Node chosen are always random)**.** *Assuming the presence of a strong pseudo-random function, our protocol ensures that the next node chosen T_k, $\forall 1 \leq k \leq m \subset N$ and always chosen uniformly at random.*

Proof. We show that our protocol results in random nodes chosen nodes, history independent and adversary advantage does not increase based on the history of the interactions and the strategy used.

Out protocol uses a pseudo-random hash function $\mathbb{H}$ takes in 2 inputs $(\mathcal{V}, P)$, where $\mathcal{V}$ is a random number chosen by the delegating node and P is the large public value, *i.e.*, updated after every hash function execution. The property of the hash function *i.e.*, given a set of $(\mathcal{V}_k, P_k^i)$ input pairs for a node S_k for i^{th} election round, the output T_k is independent of all past rounds. After each Hash function execution, the value of P is incremented and $\mathcal{V}$ is randomly chosen with bigger seed. If the adversary controls or predicts the $\mathcal{V}$ space, then the adversary does not gain any additional advantage from knowing the output history of the hash function $\mathbb{H}$ because the honest

majority increment P after every hash function execution and $\mathcal{V}$ is randomly chosen before the hash execution.

Lemma 2 (Unauthorized node do not participate in the election or become part of the random walk subset)**.** *All the nodes included in subset of M node: $\{T_1, T_2, \cdots, T_m\} \subset N$ are publicly verifiable and unauthorized node cannot participate or take part in the election protocol.*

Proof. This is ensured by the delegated message signature scheme. Every delegating node publishes the $(\mathcal{V}, P)$ used in the hash function $\mathbb{H}$ that chooses the next node T in the subset. The delegating node would also append the $WALK$ message received from the previous delegating node as a proof of verification. The last chosen node in the subset would broadcast a $FINISH$ message to all the nodes in the network with all the messages encapsulated so that all nodes can publicly verify each node that is added to the subset. If a malicious node replays this message, the updated P values would rule out the chance of malicious node being verified as a chosen node. Moreover, the random subset is started by the leader of the previous election. At any given moment, the random walk messages can be traced back to the leader of the previous election term and the leader is known by all the nodes in the network. Hence, there is no way that an unauthorized node can participate in the election or become a part of random walk subset.

Lemma 3 (Random walk protocol continuity)**.** *The random walk protocol ensures that the protocol does not stall due to node failures and makes sure that a subset is chosen for the next election round.*

Proof. Every delegating node starts maintains a timer with a time-out value: Z, *i.e.*, used to make sure that the delegated node continues the random walk protocol by waiting for the ACK message. Once the delegating node T_{k-1} sends the $WALK$ message to the delegated node T_k, the delegated node verifies the source of the $WALK$ message, chooses another node T_{k+1} using the walk function and sends the $WALK$ message to T_{k+1}. Also, the node T_k sends an ACK with the chosen node and related public parameters back to the delegating node T_{k-1}. This needs to occur within the time-out value Z. If the delegated node T_k fails to send the ACK message within the time-out value Z, T_{k-1} rolls back and chooses another node by updating the public value P

and choosing another random value $\mathcal{V}$. This ensures that the random walk protocol continues and finalizes the subset.

Lemma 4 (Random walk protocol not affected by byzantine faults)**.** *The protocol is not affected by the byzantine faults detected by the leader election protocol.*

Proof. The random walk protocol used the fault detectors list used in the leader election protocol. When the delegating node chooses the next node in the subset, it would check if the the node is present in the faulty node list. If yes, it would update the P value and re-run the walk function until a valid node is chosen. If a chosen node is down when the $WALK$ message is sent, the node is added to the suspected list and in the information is shared during the $SUSPECT$ phase of the leader election protocol.

Chapter 7

Large Enterprise Networks - Evaluations and Discussion

7.1 Experimental Setup

The network implemented to test, measure and validate the random walk protocol is similar to the one used for testing the leader election protocol. A network of fifty HP-Z440-XeonE5-1650v4 servers, running Fedora 26, each with 8 cores, 3.6 GHz clock, and 16 GB RAM, communicating over a LAN network with 1 Gbps capacity was used to test the random walk protocol. All nodes were equipped with self signed public-key private-key pairs and used the Elliptic-Curve DSA signature algorithm with 256-bit keys. The protocol communication used blocking server-client TCP communication design as the broadcast capabilities of the LAN are not utilized due to departmental restrictions. We used 20 nodes network with $3, 4, 5$ and 6 node subset selection configuration and 50 node network with $5, 10, 15$ and 20 node subset selection configuration over 50 to 200 trials. Every subset selected for each election were recorded along with the time required to choose the subset as well as the computation time of the walk function.

7.2 Performance Analysis

7.2.1 Walk Function Computation Time

The minimum computation times to choose the random nodes are shown in the figure 7.1 and 7.2. These plots measure the walk function explained in the previous chapter. The minimum computation time to choose a random node in each delegated node is collected and added as shown in the figure 7.1 for 20 and 50 node network. This plot shows that as the number of nodes in the subset increases, the time required for the walk function increases linearly. Moreover, the

walk function calculates the next node in microseconds. This proves the walk function has little overhead on the total time required to choose a random subset.

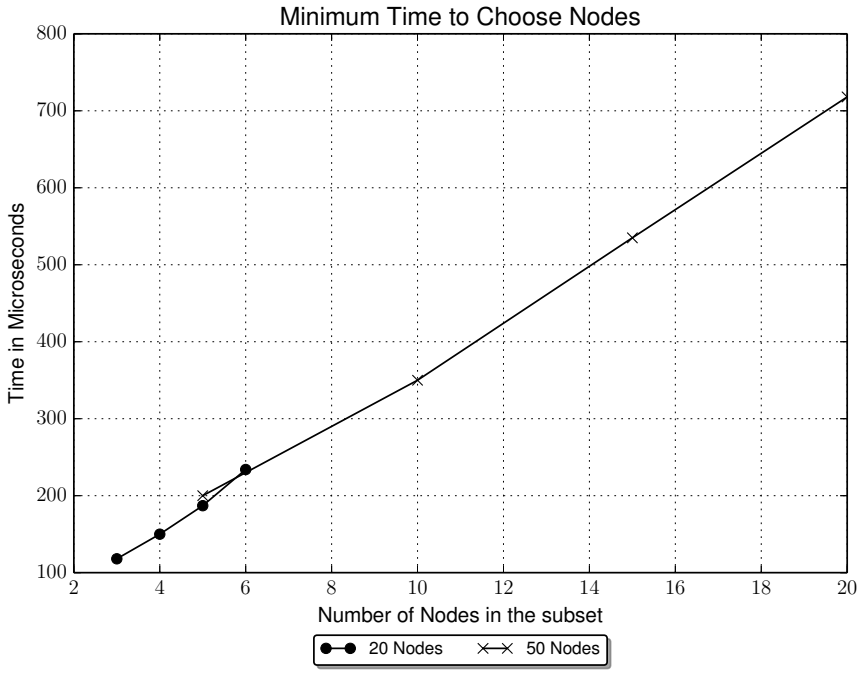


Figure 7.1: Minimum computation time to choose random nodes

Figure 7.2 shows the average time the network nodes require to choose one random node per election term for 50 and 100 iterations. This is captured by averaging all the walk function computation time from the nodes in the subset. From these plots we can conclude that average computation time to choose a random node is within the range of 40 microseconds and 80 microseconds, irrespective of the number of nodes in the network and percentage of total nodes in the subset.

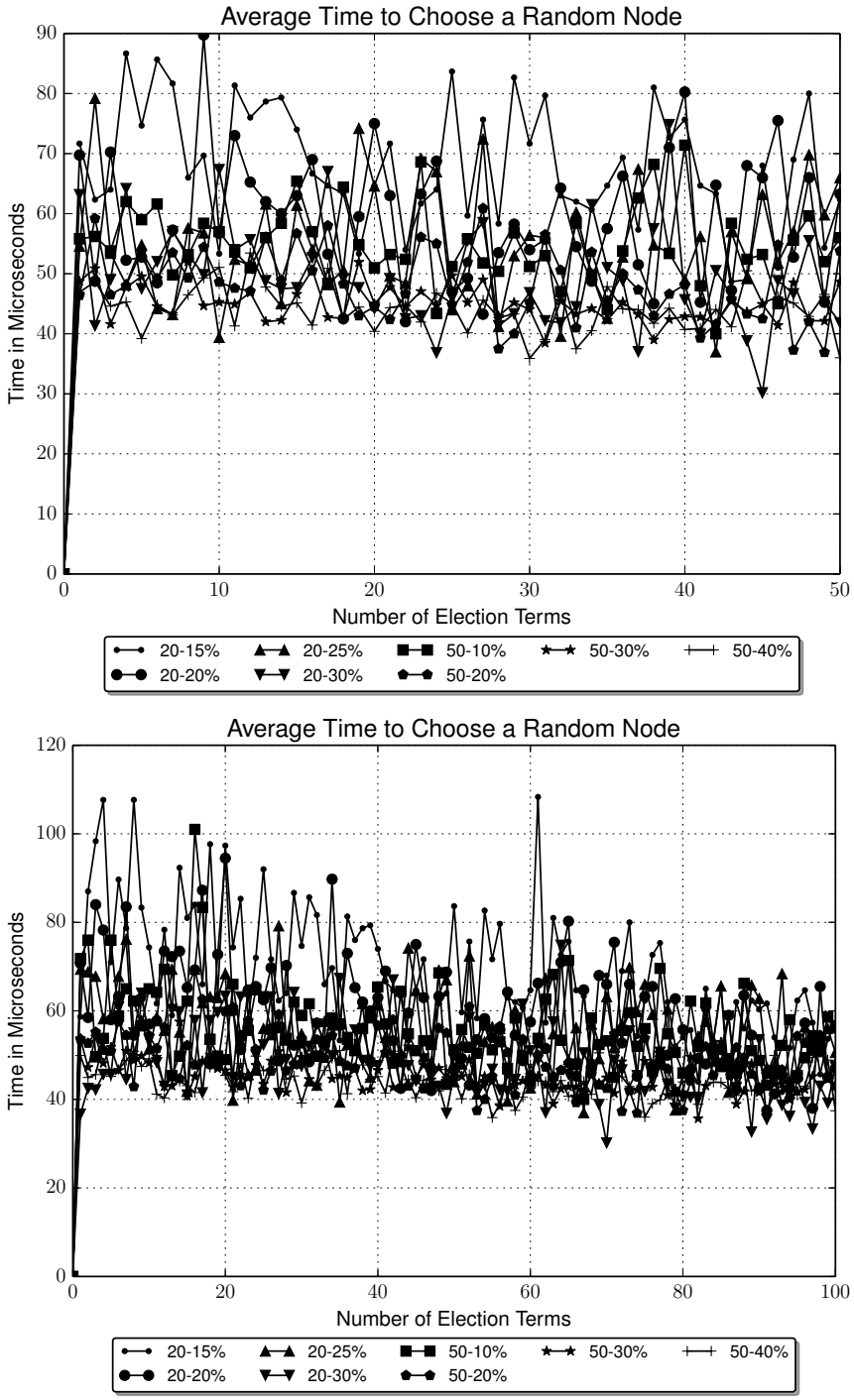


Figure 7.2: Average Time to choose a random node over (a) 50 iterations (b) 100 iterations

7.2.2 Total Subset Finalization Time

The total subset finalization time are shown in the figure 7.3 and 7.4 . These plots show the total time required by the nodes in the network to finalize the subset that take part in the next election term. This value is critical with respect to the election term period because the subset needs to be finalized before the election term ends. The time required to finalize a subset is calculated by adding the time required to execute the walk function, random walk message creation with signature and verification of the acknowledge message from all the delegated nodes in each term. Figure 7.3 shows the minimum time required to choose a subset for 20 and 50 node network. There is a slight exponential relationship between the number of nodes in the subset to the time required to finalize the subset.

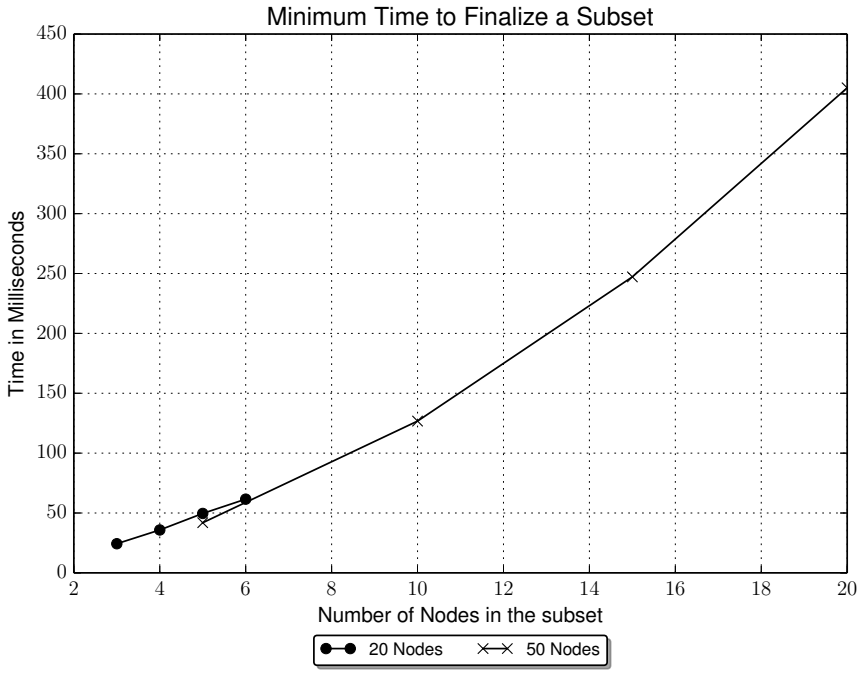


Figure 7.3: Minimum Subset Finalization time

Figure 7.4 shows the total time the network nodes require to finalize the subset per election term for 50 and 100 iterations. From these plots we can conclude that the protocol requires about

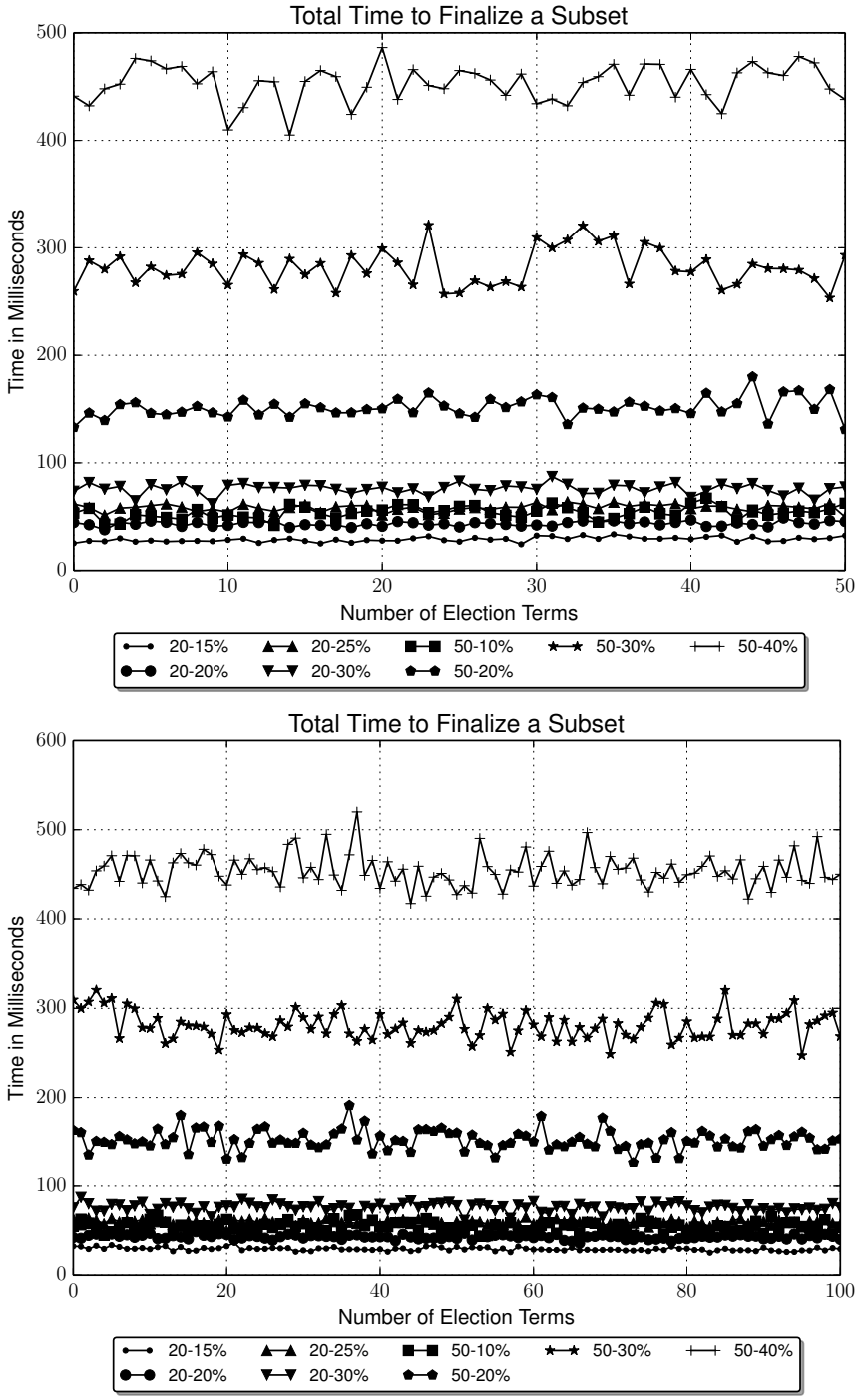


Figure 7.4: Subset finalization time over (a)50 iterations (b)100 iterations

20 to 25 milliseconds per node added to the subset. This performance proves that the protocol can scale up to networks with higher number of nodes and higher number of subset configuration.

7.2.3 Node Distribution in a Subset

The figure 7.5 and 7.6 shows the number of times a node is chosen in to the subset for next election term over 200 iterations. Figure 7.5 shows the number of times a node is added to the subset in 20 node network with 4 node subset capacity over 200 iterations. Figure 7.5(a) and figure 7.6(b) shows the number of times a node is added to the subset in 50 node network with 15 and 20 node subset capacity over 200 iterations respectively.

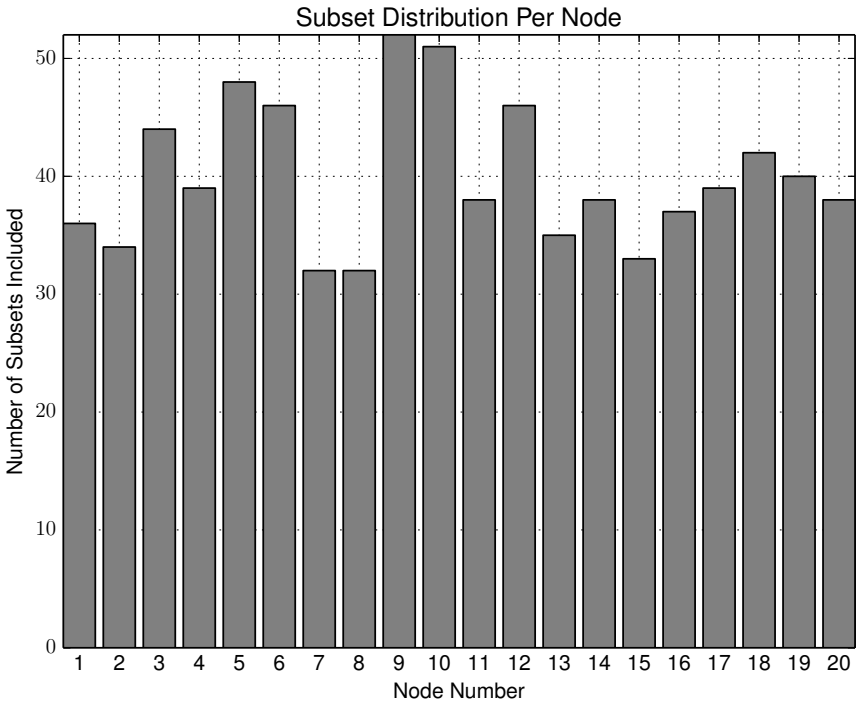


Figure 7.5: Node Distribution in a subset over 200 iterations for 20 total nodes with 4 node subset capacity

For a 20 node network with 4 node subset capacity, the ideal probability of a node chosen in the subset over 200 iterations is $\frac{40}{(20\times0.2)\times200} = 0.05$. In figure 7.5 (a), we can see that the probabilities are $0.045, 0.0425, 0.055, 0.04875, 0.06, 0.0575, 0.04, 0.04, 0.065, 0.06375, 0.0475, 0.0575, 0.04375, 0.0475, 0.04125, 0.04625, 0.04875, 0.0525, 0.05$ and 0.0475. These values are very close to the

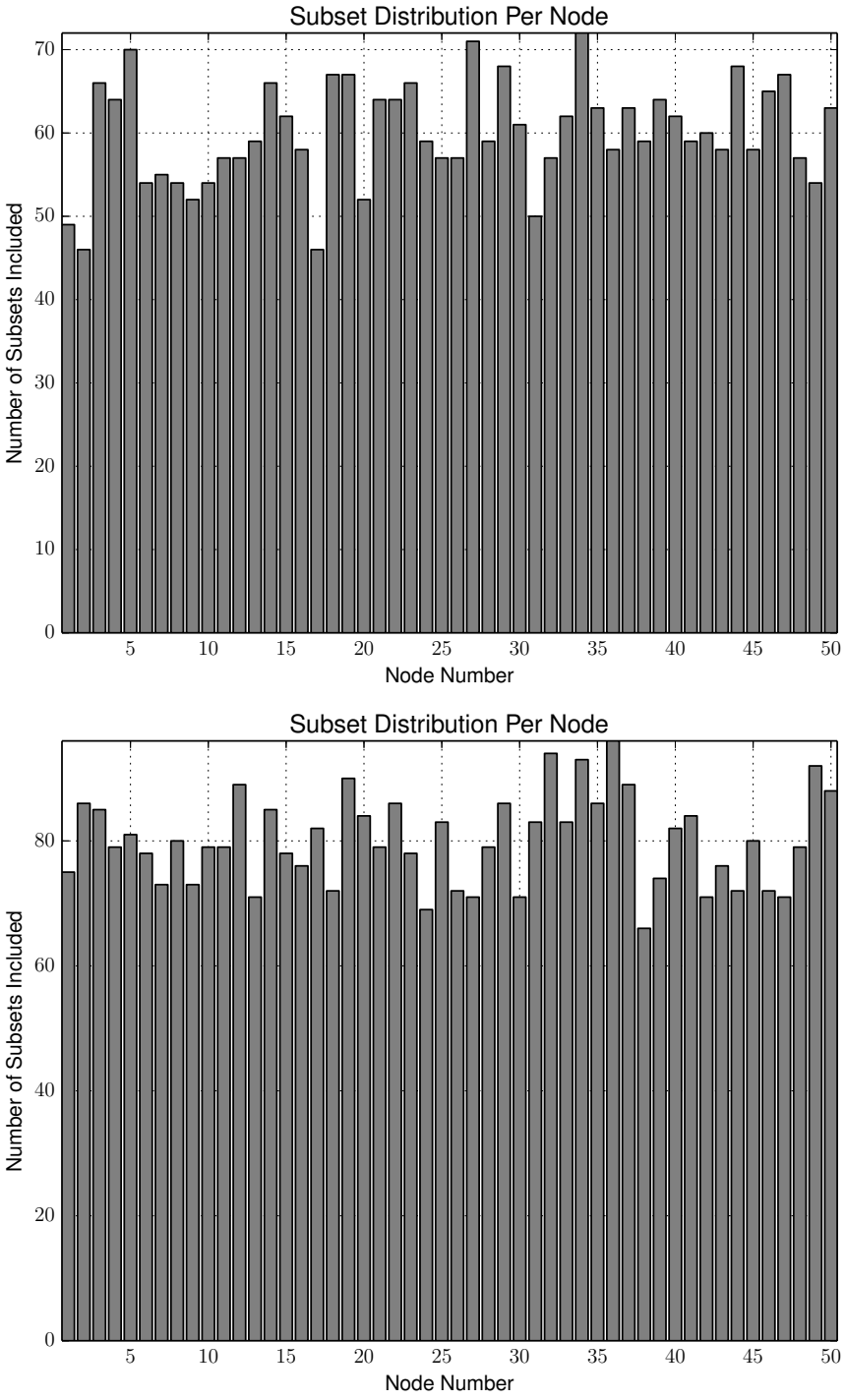


Figure 7.6: Distribution of a Node in a subset over 200 iterations (a) 50 total nodes with 15 node subset capacity (b) 50 total nodes with 20 node subset capacity

ideal probability of 0.05. This proves that all the nodes are equally likely to be chosen into the subset for participating in the next election.

7.2.4 Leader Distribution

The figure 7.7 and 7.8 shows the number of times a node is chosen as leader of over 200 iterations. Figure 7.7 shows the leader distribution for a 20 node network with 4 node subset capacity over 200 iterations. Figure 7.8 (a) and figure 7.8 (b) shows the leader distribution for a 50 node network with 15 and 20 subset capacity over 200 iterations respectively.

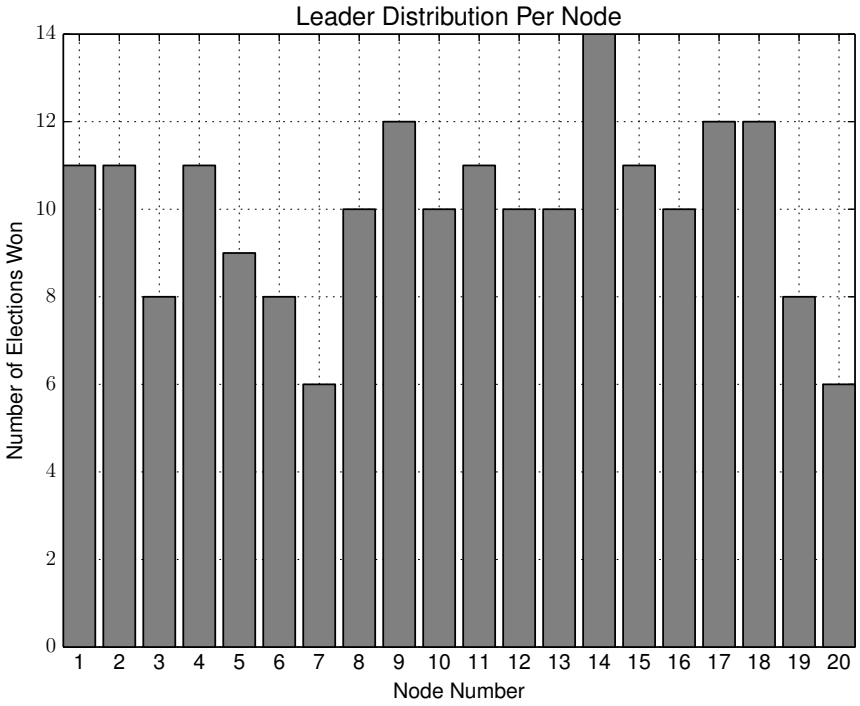


Figure 7.7: Leader Distribution of a Node over 200 iterations over 20 total nodes with 4 node subset capacity

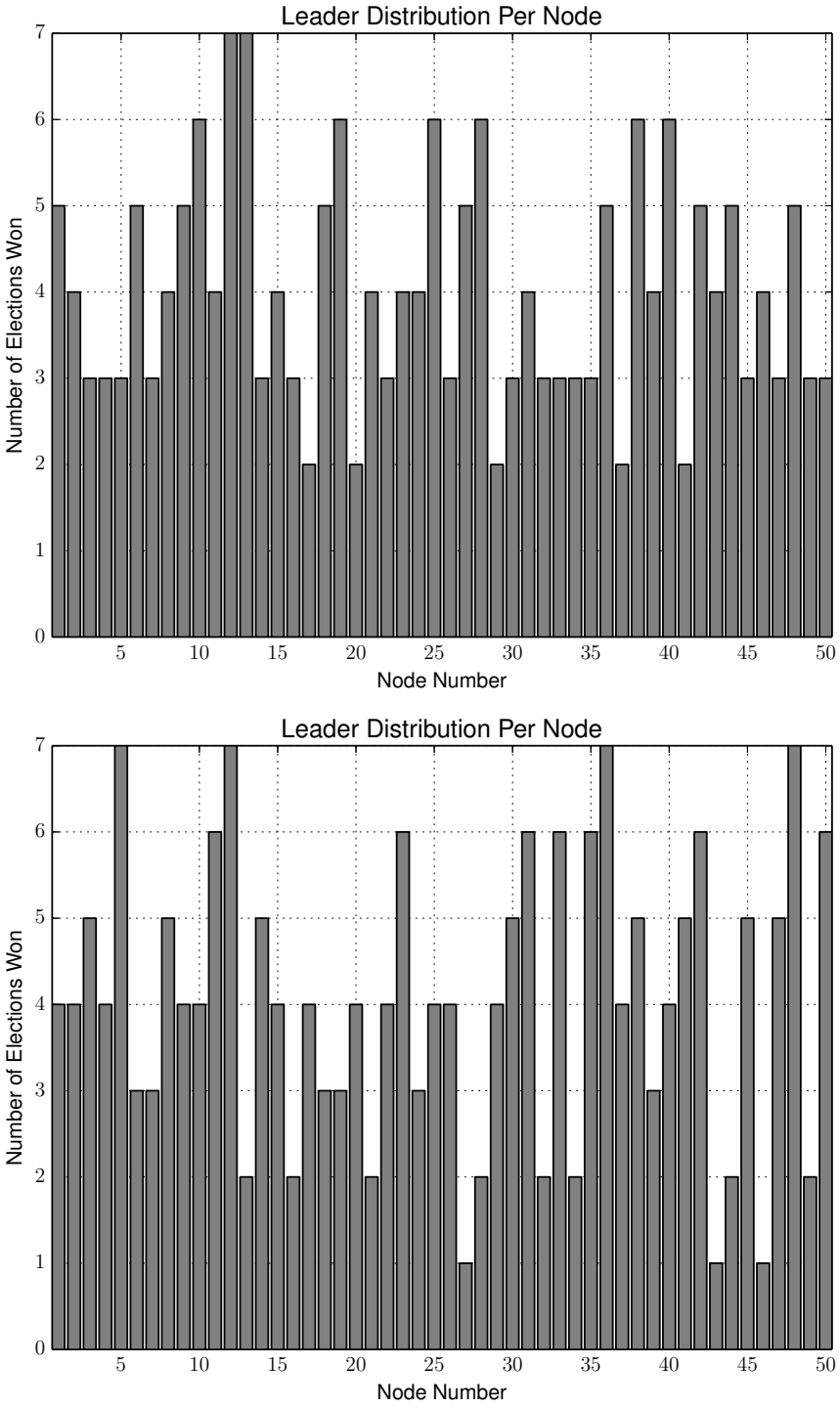
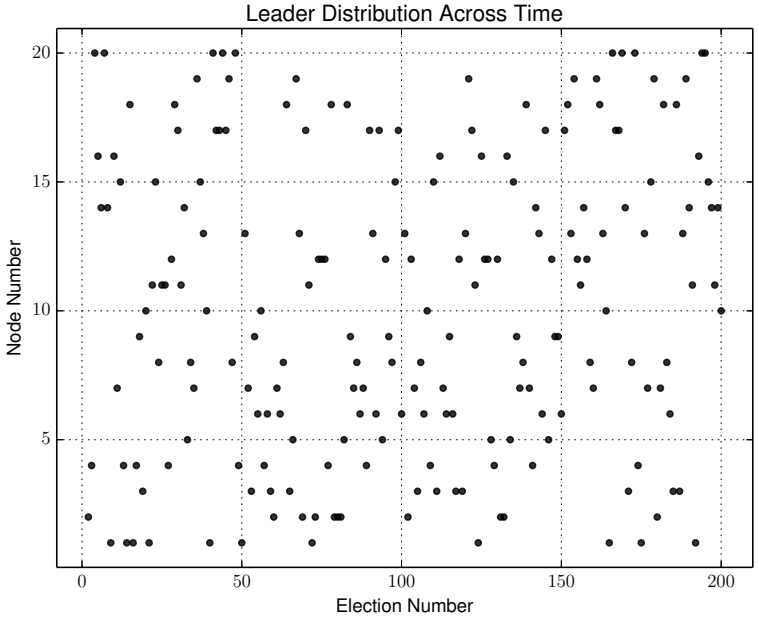
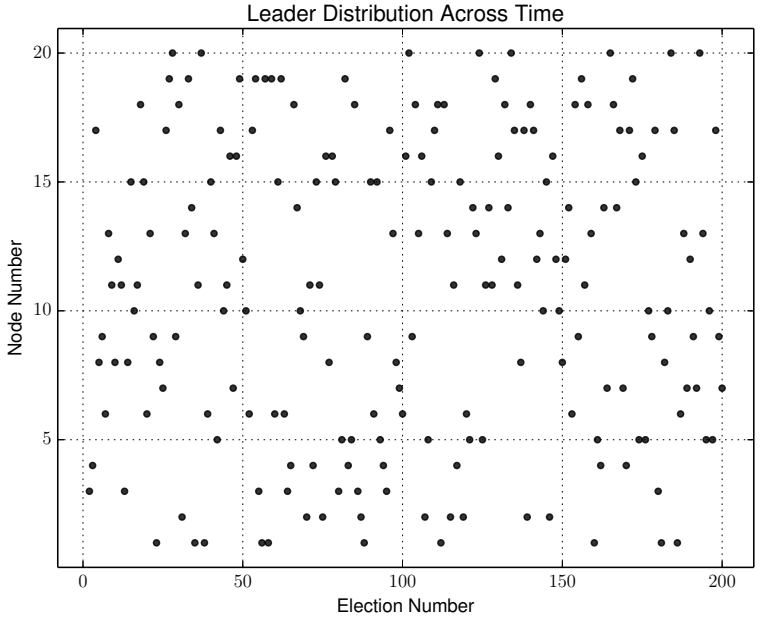


Figure 7.8: Leader Distribution of a Node over 200 iterations (a) 50 total nodes with 15 node subset capacity (b) 50 total nodes with 20 node subset capacity

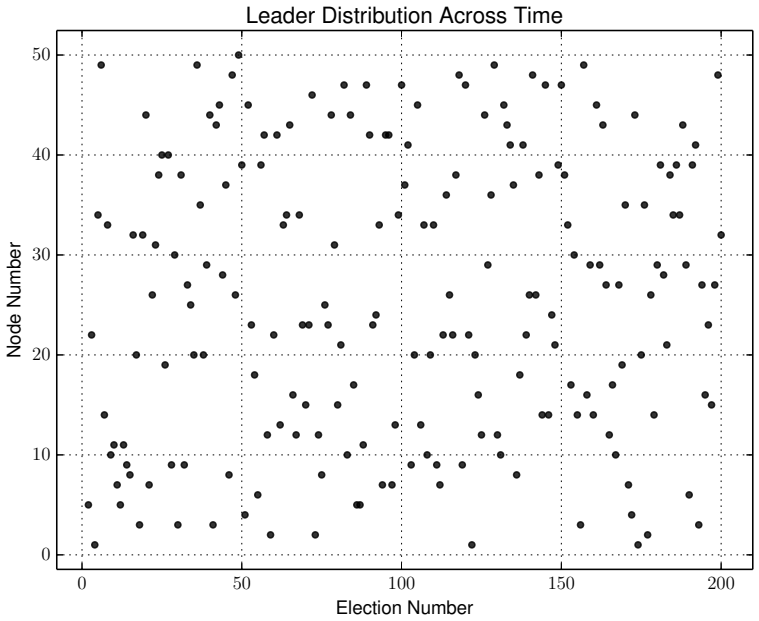


(a) 20 node Network without Random Walk Protocol.

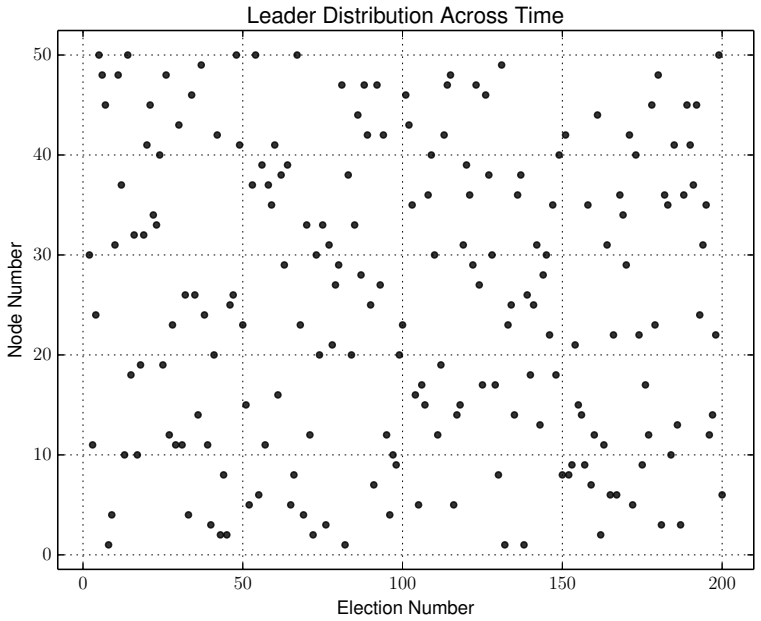


(b) 20 node Network with Random Walk Protocol 30% subset capacity

Figure 7.9: Leader Distribution of a Node over 200 iterations for a 20 node network (a) without Random Walk Protocol (b) with Random Walk Protocol choosing 15 node subset



(a) 50 node Network without Random Walk Protocol.



(b) 50 node Network with Random Walk Protocol 40% subset capacity

Figure 7.10: Leader Distribution of a Node over 200 iterations for a 50 node network (a) without Random Walk Protocol (b) with Random Walk Protocol choosing 20 node subset

For a 20 node network with 4 node subset capacity, the ideal probability of a node chosen as a leader is $\frac{1}{20} = 0.05$. In figure 7.7, we can see that the probabilities are $0.055, 0.055, 0.04, 0.055, 0.045, 0.04, 0.03, 0.05, 0.06, 0.05, 0.055, 0.05, 0.05, 0.07, 0.055, 0.05, 0.06, 0.06, 0.04$ and 0.03. These values are very close to the ideal probability of 0.05. Hence, we can say that the random walk maintains the uniform leader distribution of the Leader Election protocol for a given network. Moreover, the random walk protocol, does not allow the leader to be re-elected in the next consecutive election term because the leader of the current term does not allow itself to be added to the subset of the next election term.

7.2.5 Average Computation Time

The computation time measured is the time required by a delegated node to create the random walk message with signature and verification of acknowledge message for one election term. The minimum computation time per node is shown in the figure 7.11. All the computation times required by the nodes in the subset are averaged over 50 iterations and 100 iterations are shown the figure 7.12 (a) and 7.12 (b).

Figure 7.11 shows the minimum computation time per node for 20 and 50 node network. There is linear relationship between the minimum computation time to the time required to finalize the subset. Figure 7.12 (a) and 7.12 (b) shows the average computation time required per node over 50 and 100 iterations for 20 and 50 node network. From these plots we can conclude that the protocol requires about 1 to 2 milliseconds computation time per node in the subset. This performance proves that the protocol can scale up to higher node networks with lesser computation overhead.

7.2.6 Failure Scenarios

The random walk protocol implementation was designed considering various failure scenarios. The different failure scenarios we considered are: *delegated node failures during the random walk protocol* and *leader failures*. When the delegated node fails in between the random walk protocol, delegating node would immediately detect the faulty node and vote to exclude from the protocol. The delegating node would update the public parameters and choose another node to

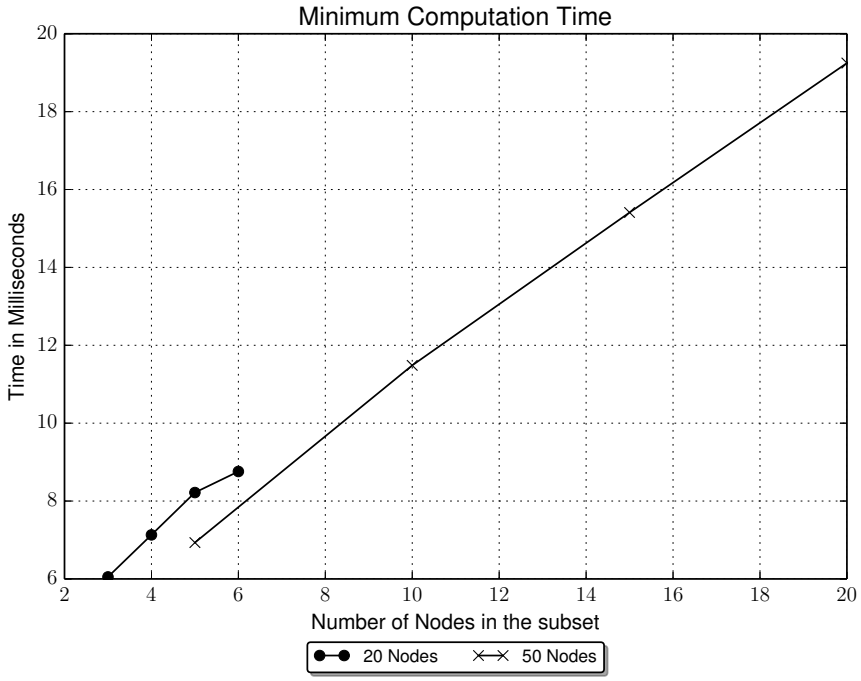


Figure 7.11: (a) Minimum Computation Time per Node

delegate the random walk. If the delegated node fails to send an acknowledgment message before the timeout, the delegating node would update the public parameters and continue the random walk by choosing another node. All the nodes in the system would accept the subset with updated public parameters and continue with the leader election protocol. If the leader node fails to start the random walk process, the previous leader would start the random walk process to continue leader election protocol and maintain the aliveness of the service. Moreover, any deviations in the values shared by nodes during the phases of the protocol is also immediately detected by other nodes and excluded from the protocol by adding it to the faulty list. Message constructs and signature verification is tightly coupled with the fault detector and any deviations would lead the node being added to the faulty list. This is how we have designed and tested different failure scenarios in our protocol.

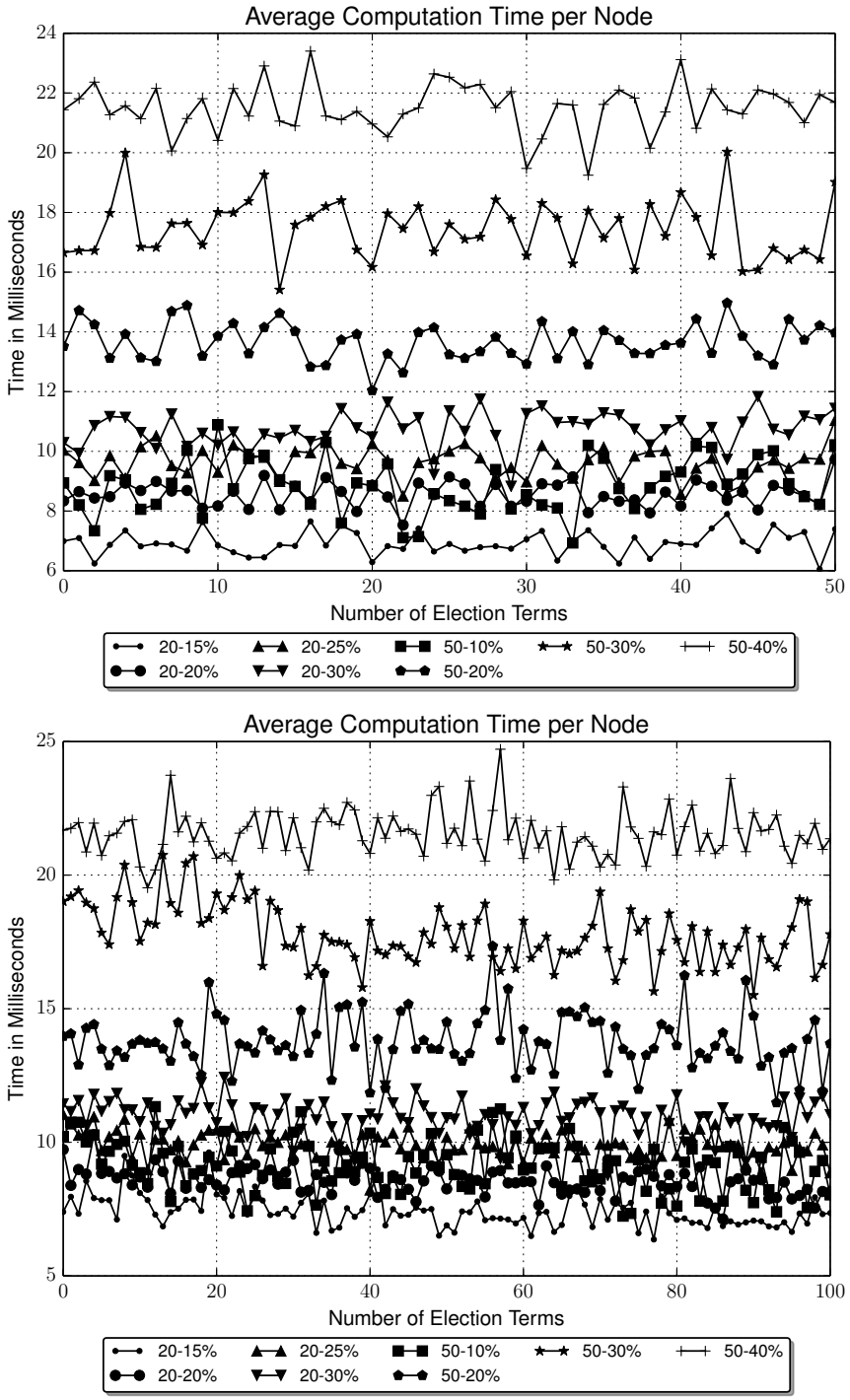


Figure 7.12: Average Computation time per node over (b) 50 iterations (c) 100 iterations

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this work, we addressed the problem of protecting critical services from attackers who use system vulnerabilities using the concept of moving target defense. First we sketched a moving target defense architecture aiming to defend an access control reference monitor. This design allows the access control service to be periodically and randomly moved to the next point of service in the distributed network. Using one-way hash commitment values and choosing a random subset of nodes that participate in the election, we have described an efficient moving target defense algorithm that chooses the next point of service with uniform randomness. Moving the point of service ensures that the attack surface is changed periodically and adversary effort of service vulnerability enumeration is wasted on the current server. The advantages of one-way hash functions provide security advantages to both leader election protocol such that the attacker has no better advantage of winning the election than any honest node in the network. The one-way hash functions used in random walk protocol provides quick turnaround time in honestly building the subset that take part in the election protocol.

We have implemented a prototype test bed of fifty servers for an access control monitor. The leader election protocol shows that the next leader can be chosen within a few hundred milliseconds with the consensus of all the nodes taking part in the algorithm. The random walk protocol is able to choose a subset of m servers within $(20 - 25) \times m$ milliseconds which lets the election term of 120 seconds adjustable according to the security standards set by the system administrator. Our extensive experimental results show that the average probability of a node being the winner of an election is close to the ideal probability.

8.2 Limitations

One of the limitations of the leader election protocol is that, when the leader stops providing the critical service during the election term, this fault is not detected by other nodes in network. Hence, the service is down for at most one election term and the clients are not able to access the service.

Our protocol uses fast hashing algorithms like SHA256. With the advancements in computing power, it is shown that fast hashing algorithms are vulnerable to brute force attacks [60]. The attackers are able to break the SHA256 algorithm faster than before. Breaking the hash function can lead to a compromise in our system. To avoid this, slower hashing functions like bcrypt hashing algorithm [61] can be used in our protocols. Slower hashing algorithm does not promise a secure protocol. It helps in slowing down the brute force attacks as it takes time break the algorithm.

8.3 Future Work

Our future work is to consider the different attack strategies possible and expand the protocol to work for larger networks. Another direction is to consider the churn in the network and adapt the design of both the protocols to handle churn in the distributed network seamlessly. Addition of heartbeat messages or "alive" messages to make sure the leader is up and providing service to the clients is another approach being considered to seamlessly provide service without any down time. Finally, we would like to explore the usage of the protocol for realistic workloads on different kinds of services besides access control and explore the service specific challenges in such deployments.

Bibliography

[1] Mohammed Hussain and Hanady Abdulsalam. Secaas: security as a service for cloud-based applications. In *Proceedings of the Second Kuwait Conference on e-Services and e-Systems*, page 8. ACM, 2011.

[2] Dieudonne Mulamba and Indrajit Ray. Resilient reference monitor for distributed access control via moving target defense. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 20–40. Springer, 2017.

[3] JP Anderson. Computer security technology planning study [r]. air force electronic systems division, hanscom afb, bedford. Technical report, MA: Technical Report ESDTR-73-51, 1972.

[4] John M Rushby. *Design and verification of secure systems*, volume 15, Issue 5. ACM, 1981.

[5] White House. Trustworthy cyberspace: Strategic plan for the federal cyber security research and development program. *Report of the National Science and Technology Council, Executive Office of the President*, 2011.

[6] Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, and X Sean Wang. *Moving target defense: creating asymmetric uncertainty for cyber threats*, volume 54. Springer Science & Business Media, 2011.

[7] dhs.gov. U.S. Homeland Security Cyber Security R&D Center: Moving Target Defense (MTD) program. https://www.dhs.gov/science-and-technology/csd-mtd, 2017.

[8] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[9] Felix C Gärtner. Byzantine failures and security: Arbitrary is not (always) random. Technical report, IC/2003/20, 2003.

[10] Kim Potter Kihlstrom, Louise E Moser, and P Michael Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[11] Hai Dong, Farookh Khadeer Hussain, and Elizabeth Chang. Semantic web service matchmakers: state of the art and challenges. *Concurrency and Computation: Practice and Experience*, 25(7):961–988, 2013.

[12] The SCO Group. Openslp- service location protocol. http://www.openslp.org, January 2011.

[13] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[14] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE transactions on Computers*, (1):48–59, 1982.

[15] Rui Zhuang, Scott A DeLoach, and Xinming Ou. Towards a theory of moving target defense. In *Proc. of the First ACM Workshop on Moving Target Defense*, pages 31–40, 2014.

[16] David Evans, Anh Nguyen-Tuong, and John Knight. Effectiveness of moving target defenses. In *Moving Target Defense*, pages 29–48. Springer, 2011.

[17] Yujuan Han, Wenlian Lu, and Shouhuai Xu. Characterizing the power of moving target defense via cyber epidemic dynamics. In *In Proc. of the ACM Symp. and Bootcamp on the Science of Security*, page 10, 2014.

[18] Spyros Antonatos, Periklis Akritidis, Evangelos P Markatos, and Kostas G Anagnostakis. Defending against hitlist worms using network address space randomization. *Computer Networks*, 51(12):3471–3490, 2007.

[19] Noor O Ahmed and Bharat Bhargava. Mayflies: A moving target defense framework for distributed systems. In *Proc. of ACM Workshop on Moving Target Defense*, pages 59–64, 2016.

[20] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[21] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[22] John Augustine, Gopal Pandurangan, and Peter Robinson. Fast byzantine agreement in dynamic networks. In *Proc. of ACM PODC*, pages 74–83, 2013.

[23] Augustine John, Pandurangan Gopal, and Robinson Peter. Fast byzantine leader election in dynamic networks. In *Proc. of DISC*, pages 276–291, 2015.

[24] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proc. of the 6th USENIX NSDI*, pages 153–168, 2009.

[25] Diego Ongaro and John Ousterhout. Raft consensus algorithm, 2015.

[26] Sisi Duan, Yun Li, and Karl Levitt. Cost sensitive moving target consensus. In *Proc. of the IEEE Int. Symp. on Network Computing and Applications (NCA)*, pages 272–281, 2016.

[27] Stavros Nikolaou and Robbert Van Renesse. Turtle consensus: Moving target defense for consensus. In *Proc. of the 16th Annual Middleware Conference*, pages 185–196. ACM, 2015.

[28] Christopher Copeland and Hongxia Zhong. Tangaroa: a byzantine fault tolerant raft, 2016.

[29] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Efficient synchronous byzantine consensus. *arXiv preprint arXiv:1704.02397*, 2017.

[30] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proc. of ACM CCS*, pages 31–42, 2016.

[32] Michael L Winterrose, Kevin M Carter, Neal Wagner, and William W Streilein. Adaptive attacker strategy development against moving target cyber defenses. *arXiv preprint arXiv:1407.8540*, 2014.

[33] Luke Rodriguez, Darren Curtis, Sutanay Choudhury, Kiri Oler, Peter Nordquist, Pin-Yu Chen, and Indrajit Ray. Action recommendation for cyber resilience. In *Proc. of the 22nd ACM CCS*, pages 1620–1622, 2015.

[34] A. Newell, D. Obenshain, T. Tantillo, C. Nita-Rotaru, and Y. Amir. Increasing network resiliency by optimally assigning diverse variants to routing nodes. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2013.

[35] Neal Wagner, Cem Ş Şahin, Michael Winterrose, James Riordan, Jaime Pena, Diana Hanson, and William W Streilein. Towards automated cyber decision support: A case study on network segmentation for security. In *Proc. IEEE Symp. Series on Computational Intelligence (SSCI)*, pages 1–10, 2016.

[36] Shangguang Wang, Ao Zhou, Mingzhe Yang, Lei Sun, Ching-Hsien Hsu, et al. Service composition in cyber-physical-social systems. *IEEE Trans. on Emerging Topics in Computing*, 2017.

[37] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Softw. Pract. Exper.*, 44(6):735–770, June 2014.

[38] Stefan Achleitner, Thomas La Porta, Patrick McDaniel, Shridatt Sugrim, Srikanth V Krishnamurthy, and Ritu Chadha. Cyber deception: Virtual networks to defend insider reconnaissance. In *Int. Workshop on Managing Insider Security Threats*, pages 57–68. ACM, 2016.

[39] Pradeep Ramuhalli, Mahantesh Halappanavar, Jamie Coble, and Mukul Dixit. Towards a theory of autonomous reconstitution of compromised cyber-systems. In *Proc. of Int. IEEE Conf. on Technologies for Homeland Security (HST)*, pages 577–583, 2013.

[40] Frank Spitzer. *Principles of random walk*, volume 34. Springer Science & Business Media, 2013.

[41] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 119–131. ACM, 1990.

[42] Don Coppersmith, Prasad Tetali, and Peter Winkler. Collisions among random walks on a graph. *SIAM Journal on Discrete Mathematics*, 6(3):363–374, 1993.

[43] David R Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43. ACM, 2004.

[44] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 163–170. ACM, 2000.

[45] Lada A Adamic, Rajan M Lukose, Amit R Puniyani, and Bernardo A Huberman. Search in power-law networks. *Physical review E*, 64(4):046135, 2001.

[46] Brian F Cooper. Quickly routing searches without having to move content. In *International Workshop on Peer-to-Peer Systems*, pages 163–172. Springer, 2005.

[47] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM, 2002.

[48] David Kempe, Jon Kleinberg, and Alan Demers. Spatial gossip and resource location protocols. *Journal of the ACM (JACM)*, 51(6):943–967, 2004.

[49] Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 395–406. ACM, 2003.

[50] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Hybrid search schemes for unstructured peer-to-peer networks. In *IEEE INFOCOM*, volume 3, page 1526. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 2005.

[51] Marc Bui, Thibault Bernard, Devan Sohier, and Alain Bui. Random walks in distributed computing: A survey. In *International Workshop on Innovative Internet Community Systems*, pages 1–14. Springer, 2004.

[52] Ming Zhong and Kai Shen. Random walk based node sampling in self-organizing networks. *ACM SIGOPS Operating Systems Review*, 40(3):49–55, 2006.

[53] Atish Das Sarma, Danupon Nanongkai, and Gopal Pandurangan. Fast distributed random walks. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 161–170. ACM, 2009.

[54] Bruhadeshwar Bezawada Dieudonne Mulamba, Athith Amarnath and Indrajit Ray. A secure hash commitment approach for moving target defense of security-critical services. In *5th ACM Workshop on Moving Target Defense (MTD 2018) October 15, 2018*, MTD 2018. ACM, 2018.

[55] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[56] Yehuda Lindell and Jonathan Katz. *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.

[57] Danny Dolev. The byzantine generals strike again. *Journal of algorithms*, 3(1):14–30, 1982.

[58] Baruch Awerbuch and Christian Scheideler. Robust random number generation for peer-to-peer systems. *Theoretical Computer Science*, 410(6-7):453–466, 2009.

[59] Harry Kesten, Mykyta V Kozlov, and Frank Spitzer. A limit law for random walk in a random environment. *Compositio Mathematica*, 30(2):145–168, 1975.

[60] Kanwalinderjit Gagneja and Luis G Jaimes. Computational security and the economics of password hacking. In *International Conference on Future Network Systems and Security*, pages 30–40. Springer, 2017.

[61] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.