Defending Against Eclipse Attacks in Unstructured Overlays

Ido Gonen
Defending Against Eclipse Attacks in Unstructured Overlays

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Ido Gonen

Submitted to the Senate of the Technion — Israel Institute of Technology
Shebat 5775 Haifa February 2015
This research was carried out under the supervision of Prof. Roy Friedman, in the Faculty of Computer Science.

# Contents

**List of Figures**

**List of Algorithms**

**Abstract** 1

**Abbreviations and Notations** 3

1 Introduction 5

2 Related Work 7

3 System Model 11

4 BMON: Byzantine Monitor Service 13
   4.1 Overview 13
   4.2 External Interface and Guarantees 14
   4.3 Implementation 14
      4.3.1 Monitor Selection 14
      4.3.2 Monitor Discovery 15
      4.3.3 Using the Monitoring Service 15
      4.3.4 Byzantine Tolerant Acceptance Mechanism 17
   4.4 Protocol 17
   4.5 Correctness 23
   4.6 Analysis 25
      4.6.1 The Configuration Parameters 25
      4.6.2 Initialization Time 25
   4.7 Adding Churn 27
   4.8 Oral Messages vs. Signed Messages 28

5 Eclipse Attacks Defense Protocol 29
   5.1 Overview 29
   5.2 Correctness 30
      5.2.1 The Prevention of Eclipse Attacks 31
5.3 Overhead ................................................................. 33
   5.3.1 Computational Overhead ....................................... 33
   5.3.2 Network Overhead ................................................ 34

6 Discussion ................................................................. 37

Hebrew Abstract .......................................................... i
List of Figures

4.1 The probability for knowing $L$ out of $K$ monitors, as a function of the
discovery protocol round .................................................. 27
List of Algorithms

4.1 Node \( w \) participates in the discovery of monitoring relationships 18
4.2 Node \( x_1 \) receives a \( \text{Notify}(x_1, y) \) message 18
4.3 Node \( y \) receives a \( \text{Notify}(x_1, y) \) message 18
4.4 Node \( y \) receives a request to get at least \( L_{\text{requested}} \) of its monitors from node \( x_1 \) 19
4.5 Node \( x_1 \) receives a list of \( y \)'s monitors from node \( y \) 19
4.6 Node \( z \) retrieves \( y \)'s monitors 19
4.7 Node \( z \) receives a list of \( y \)'s monitors from node \( y \) 20
4.8 Node \( y \) sends a \( \text{Store}(<y, Key, Value, timestamp_{key} >_y) \) to its monitors 20
4.9 Node \( x_1 \) receives \( \text{Store}(\text{NewTuple}) \) from \( y \) 21
4.10 Node \( x_2 \) receives \( \text{Forward}(\text{NewTuple, ListOfMonitors}) \) from node \( x_1 \) 22
4.11 Node \( z \) sends a \( \text{Get}(y, Key) \) to \( y \)'s monitors 22
4.12 Node \( x_1 \) receives \( \text{Get}(y, Key) \) from node \( z \) 22
4.13 Node \( z \) receives \( \text{Tuple} \) from node \( x_1 \) 23
4.14 Node \( z \) decides whether to accept a tuple for a specific \( \text{ReceivedTuples}(NodeID, Key) \) 23
Abstract

Overlays play a central role in the scalability of many peer-to-peer (P2P) networks and large scale data-center systems. The *Eclipse* attack has been identified as one of the major potential attacks against overlays. In Eclipse attacks, an attacker that controls a small portion of the nodes in the system eclipses a large fraction of the *correct* nodes. By eclipsing correct nodes, attackers isolate correct nodes from the rest of the system, and thereby can completely control what these nodes see and know about the network.

To the best of our knowledge, previous works on defending against Eclipse attacks focused only on structured P2P overlays, where there are structural constraints on the identities of a node’s neighbors. However, structured overlays tend to be much less robust and scalable than unstructured ones. In this work, we present a novel approach to defend against Eclipse attacks in unstructured overlays, where there are no a-priori constraints on a node’s neighbors other than possibly its degree. Our defense bounds the degree of nodes in the overlay and uses a decentralized self-discovered monitoring service called BMON to enforce this bound. In practice, correct nodes disconnect from neighbors whose degree is above a given threshold, thereby bounding the degree of nodes in the overlay. The degree bounding reduces the likelihood of an Eclipse attack to be successfully mounted, as a malicious node is prevented from being the overlay neighbor of too many correct nodes. The work presents the defense protocol, including a detailed description of BMON and its analysis.
Abbreviations and Notations

\( N \) : the number of nodes in the overlay  
\( F \) : the number of Byzantine nodes in the overlay  
\( K \) : the number of monitors of each node with high probability  
\( f \) : the upper bound with high probability on the number  
\( \) \ of Byzantine nodes out of the \( K \) monitors of each node  
\( PS(x) \) : the set of nodes that monitor node \( x \)  
\( TS(x) \) : the set of nodes that node \( x \) monitors  
\( NS(x) \) : the neighbor set of node \( x \)  
\( RNS(x) \) : the reverse neighbor set of node \( x \): \( \forall y, y \in RNS(x) \iff x \in NS(y) \)
Chapter 1

Introduction

In large scale distributed systems, such as P2P networks and large data centers, it is often considered impractical for all nodes to constantly be aware and maintain communication links with all other nodes. Instead, such systems often employ overlays, which are logical knowledge (or membership) and communication graphs, such that two nodes “know” each other or may communicate with each other only if they are neighbors in the respective overlay [LGL08, ZH12, Pas12]. That is, each node $p$ is only aware of a node $q$ if there is an edge from $p$ to $q$ in the knowledge (or membership) overlay. Similarly, $p$ may send messages to $q$ only if there is an edge from $p$ to $q$ in the communication overlay. In these cases, we say that $q$ is a neighbor of $p$ in the corresponding overlay.

The structure and maintenance protocol of the overlays are one of the main aspects by which these systems differ from each other. In particular, the overlay may be structured or unstructured. Structured overlays, like Pastry [RD01] and Chord [SMK+01], impose structural constraints on a node’s neighbors’ identities. On the other hand, in unstructured overlays, like Gnutella, Scamp [GKM01] and Araneola [MK04], there are no specific restrictions on the identity of a node’s neighbors, but rather only on their number. Nodes in unstructured overlays typically use flooding or random walks in order to find overlay neighbors, while in structured overlays each node has a unique identifier that is used to choose neighbors satisfying certain constraints.\(^1\)

With the growth of their popularity, overlay security issues have also become a significant concern [Pas12, GCM11, UPS11, Wal02]. In this work we are interested in Eclipse attacks, in which an attacker that controls a portion of the nodes in the system eclipses a large fraction of the correct nodes by dropping or re-routing any messages meant for those nodes [GCM11]. Typically, such an attack is launched on the overlay protocol so that all the neighbors of a given node are chosen to be nodes under the control of the attacker. By eclipsing correct nodes, attackers isolate correct nodes from the rest of the system, and thereby can completely control what these nodes see and know about the network.

\(^1\) Kademlia is a special case in which the overlay has a well-defined structure, but it is much less rigid than Pastry or Chord.
The Eclipse attack may be considered as a close relative of the known Sybil attack [Dou02]. In a Sybil attack, a single attacker can obtain and present multiple identities. By and large, Sybil attacks are directed against reputation based systems, where an attacker uses a large number of identities in order to gain a disproportionately large influence in the system. A successful Sybil attack can be used to mount an effective Eclipse attack on the overlay protocol, so that all the neighbors of a given node are chosen to be nodes under the control of the attacker. A possible defense against Sybil attacks is to use certified identities [Dou02, CDG+02]. However, as mentioned in [SNDW06], an effective Eclipse attack can still be launched in the presence of a defense against the Sybil attack. For example, in unstructured overlays, nodes often use random walks in order to find overlay neighbors. Malicious nodes can exploit this neighbor selection mechanism, and advertise only nodes that are under the control of the attacker. Hence, even a modest number of malicious nodes in an overlay can mount an effective Eclipse attack.

Eclipse attacks have been addressed in the past in the context of structured overlays [SCDR04, SNDW06]. However, these types of solutions are not applicable in unstructured overlays. Further, specific unstructured overlays in which a node’s neighbor set is chosen to be a random sample of nodes in the overlay can be resistant to Eclipse attacks with high probability (w.h.p.) by design. Brahms [BGK+09] is an example of a sampling service that can provide these random samples and overcome Byzantine attacks w.h.p.. Yet, such unstructured overlays have a very specific design, which might not be optimal for all cases. In particular, these overlays are unaware of proximity neighbor selection (PNS) [CDHR03, GGG+03] which is an important and common technique to improve the efficiency of overlays.

Hence, in this work we focus on preventing Eclipse attacks in unstructured overlays without imposing a specific overlay maintenance protocol. To that end, we present a novel mechanism that prevents w.h.p. correct nodes from neighboring with a single malicious node, and further enables correct nodes to detect attempts to eclipse them by having a malicious node lying about its neighbors. Our solution is based on our novel concept of a Byzantine Monitor Service (or BMON for short). BMON is a decentralized service, in which every node is monitored by a few random monitoring nodes. By keeping track of the claimed set of neighbors of each node, we can detect and thereby prevent nodes from acquiring too many neighbors, which in turn prevents Eclipse attacks [SCDR04]. Let us note that due to scalability considerations we have opted for a decentralized solution rather than relying on a single trusted server to perform all monitoring tasks.

The remainder of this thesis is organized as follows: An overview of related works appears in Chapter 2. The system model is presented in Chapter 3. The details of BMON’s design and implementation are provided in Chapter 4, along with an analysis of the protocol. Chapter 5 describes the defense against Eclipse attacks, which is based on our novel concept of BMON. We conclude with a discussion in Chapter 6.
Chapter 2

Related Work

Distributed Hash Tables (DHTs) are at the heart of many P2P systems as well as several data centers [RD01, SMK+01]. Several works have explored the main threats and solutions facing DHT based systems [CDG+02, DdVP+02]. The interested reader is referred to surveys summarizing P2P security issues and solutions [UPS11, Wal02].

Reputation systems and trust management are important aspects of overlay based P2P systems [MGM06, DA06, SL03]. Some of them can be used in order to implement, for example, decentralized recommendation systems, and some can be used to implement global fairness mechanisms that help protect the system against free-riders. However, by and large, they may be able to overcome certain attempts to manipulate ranking of nodes, but not direct Byzantine attacks on the protocol itself. Further, they do not solve the problem of Eclipse attacks.

As for Eclipse attacks, Castro et al. [CDG+02] identify this attack as a major problem for overlays. They suggest preventing Eclipse attacks by using constraint routing tables (CRT) in structured overlays like CAN [RFH+01], Chord [SMK+01], Pastry [RD01], and Tapestry [ZHS+04]. Each node in the system is assigned with a random, certified ID, and the CRT imposes strong structural constraints on the nodes’ neighbor sets. This approach is designed to work with structured overlays, as in unstructured ones by definition there are no constraints on a node’s neighbor set.

Singh et al. also present a possible solution to defend against Eclipse attacks in [SCDR04] based on distinguishing between the indegree and outdegree of a node. The former is the number of nodes who view a given node $x$ as their neighbor and the latter is the number of nodes that $x$ views as its neighbors. First, they notice that an attacker’s indegree during an Eclipse attack is significantly high. Therefore, if nodes would choose their neighbors out of a subset of nodes whose indegree is lower than a threshold, they can practically bound the indegree of malicious nodes. They continue and argue that this defense is not enough since malicious nodes might also consume the indegree of correct nodes to prevent other correct nodes from neighboring with them. Hence, bounding the outdegree of nodes in the overlay is also needed. Consequently, they present a way to bound both the indegree and outdegree of nodes in the system,
thereby preventing Eclipse attacks. According to their defense, each node in the overlay challenges its neighbors at unpredictable times and in an anonymous way, for their neighbor sets. The challenged node does not know which of its neighbors challenged it, and therefore cannot lie regarding its real neighbor set without risking being exposed. The way they create this anonymity is by forwarding the challenge message to a set of intermediate nodes who will then forward this message to the challenged node. Alas, the solution of [SCDR04] has been found to be inapplicable to unstructured overlays in [BGK+09]. The main problem with this approach, which restricts it to structured overlays, is the way the intermediate nodes that forward the challenges are chosen. As stated in [SCDR04], the intermediate nodes used to audit a node \( x \) are the nodes whose identifiers are closest to \( x \)'s identifier.

This problem is addressed in a followup work [SNDW06], in which unstructured overlays are handled by maintaining a structured overlay alongside the unstructured one. However, this latter approach suffers from the same drawbacks of structured overlays, including scalability, maintenance under churn, etc.

Another kind of defense against Eclipse attacks makes use of proximity constraints on the neighbor set rather than strong structural constraints [HK03]. In this approach, a node \( x \) selects as its neighbors the nodes whose network delay to \( x \) is minimal among all of the nodes that satisfy the structural constraints for \( x \). This solution relies on the assumption that a small subset of malicious nodes cannot be within a small network delay of all correct nodes, making it difficult for them to execute an Eclipse attack. This solution reveals a few pitfalls as stated in [SNDW06]. First, this mechanism depends on accurate high-resolution delay measurements that may not be effective in large overlays. Second, it assumes that malicious nodes cannot attack the delay measurements.

In this work, we also cap the indgree and outdegree of nodes. However, we do so for unstructured overlays by auditing these values rather than restricting the structure of the overlay. As mentioned in the Introduction, our auditing mechanism uses a novel service called BMON. The initial inspiration for BMON comes from the AVMON work [MG09], which presents an availability monitoring overlay in non-Byzantine environments. As stated in [MG09], the problem of monitor overlay can be formally described as follows:

For each node \( x \), select and discover a small subset of nodes to monitor \( x \).

Let us briefly review the basic ideas behind AVMON with respect to the select and discover parts. In AVMON, each node \( x \) in the system is assigned a fixed-sized set of \( K \) random monitor nodes, consistent over time. The role of these monitors is to monitor \( x \)'s availability. Moreover, the monitoring relationships are locally verifiable, i.e., given two nodes \( x, y \), any third node is able to correctly verify if \( x \) monitors \( y \) (or vice versa) without any communication. The way AVMON satisfies these goals (randomness, consistency and verifiability) is by relying on a hash-based implementation of the monitoring relationship. I.e., selecting who monitors whom is done by using a one-way hash-based function \( H \). Specifically, node \( x \) monitors node \( y \) iff \( H(x, y) \leq \frac{K}{N} \) where \( N \) is the number of nodes in the system. This condition is called the hash-consistent
condition. While the hash-consistent condition works regardless of Byzantine failures, its inputs can be manipulated by a Byzantine node. I.e., a Byzantine node can choose its identifier in a wise manner, effecting the randomness of the process.

In order to discover any consistent and verifiable monitoring relationships, AVMON maintains and uses a coarse overlay. Each node maintains a fixed-size neighbor list, called the coarse view, that is, a random subset of the remaining nodes in the system. This coarse overlay is used by nodes to discover monitoring relationships between other pairs of nodes and inform the relevant nodes of such a discovery.

In particular, this is how the coarse overlay works: a node that joins the system is given a random starting node \( y \), sends \( y \) a Join message, inherits \( y \)'s view, and starts participating in the discovery protocol. A node that receives a Join message, adds the new joined node to its view and forwards this message to a pair of other random members of its view. The forwarding of Join messages is stopped when this message has been forwarded to enough nodes in the overlay. The way that this condition is checked in AVMON is by using a counter at each Join message that is divided by 2 at each forwarding. The discovery protocol includes choosing one random member out of the local node’s view and fetching its view. Once the view is fetched, the hash-consistent condition is checked for every pair \((x, y)\) of nodes that can be created out of these two views. For every pair \((x, y)\) that the hash-consistent condition is satisfied for, Notify\((x, y)\) messages are being sent to \( x \) and \( y \). Finally, the local node’s view is updated to be a fixed number of random entries of its own old view and the fetched view.

The discovery protocol of AVMON suffers from several pitfalls when facing Byzantine behavior. First, a newly joined node starts the process of constructing its view by contacting a single random node. If this node is Byzantine, it can quickly and efficiently poison the new node’s view with other Byzantine collaborating nodes. Second, the described discovery protocol is very similar to traditional gossip based membership protocols, and as such, it is exposed to the Byzantine attacks identified in [BGK+09].

Finally, Morales et al. state that AVMON is not designed to work in an adversarial environment where nodes might be Byzantine. Moreover, AVMON is designed to solve the availability discovery of nodes, while we are interested in defending against Eclipse attacks. In this work, we augment AVMON’s basic ideas to create a monitor service that is resilient to Byzantine behavior w.h.p. and is helpful in preventing Eclipse attacks in unstructured overlays.
Chapter 3

System Model

We consider a collection $U$ of nodes, each identified by a unique ID. IDs are assigned to nodes in a uniformly random manner, such that the nodes have no influence on the IDs that they are given. Moreover, each node has only a single ID and a node cannot use multiple IDs, and in particular cannot impersonate other nodes or present itself using unassigned IDs, which rules out Sybil attacks [Dou02]. Such an assignment of identifiers can be implemented, e.g., by a trusted certification authority and a PKI infrastructure.

Each node has a local persistent storage that can be accessed only by this node. Nodes are supposed to execute a prescribed protocol for them, which includes the computation steps and messages they need to send in reaction to messages they receive and to the passage of time. A node can be correct, in which case if follows its protocol. Other nodes, called Byzantine, faulty or malicious interchangeably, do not have to follow any protocol and can try to exploit the protocol in order to attack other nodes.

Any pair of nodes can communicate with each other directly through bidirectional FIFO reliable links, provided that they know each other’s IDs. A node can determine the source of every incoming message, and cannot intercept messages addressed to other nodes (this is the standard Byzantine model with oral messages [AW04]). Yet, some of the messages in our protocols are signed by their creators at the protocol level, i.e., they are signed messages in the terminology of [AW04]. We denote a signed message $M$ by a node $x$ as $<M>_x$. Signing on messages by nodes in the system can be achieved by a PKI infrastructure.

Individual nodes only know a small subset of $U$; these nodes are called the neighbors of $p$ or the neighbor set of $p$. The entire set of nodes in the system $U$ and the links between them (derived from the neighbor sets) is referred to as the overlay of the system. In this work we deal with unstructured overlays, that is, overlays in which there is no a-priori mapping between the node IDs and their neighbor sets.

Similarly to [SCDR04], in addition to the neighbor set, each node $x$ maintains a reverse neighbor set, which consists of all of the nodes that include $x$ in their neighbor sets. At the application level, correct nodes answer only to nodes that are included in their reverse neighbor set. We denote the neighbor set of node $x$ by $NS(x)$ and the
reverse neighbor set of $x$ by $RNS(x)$. We also refer to nodes in the $RNS(x)$ as the reverse neighbors of $x$. We assume a synchronous model with a global discrete clock and zero processing times. The execution of the protocol proceeds in rounds, each of which lasts a single clock tick and all messages sent have a latency of a single round. Specifically, nodes receive and process messages at the beginning of a round and then generate and send messages, which are received at the beginning of the next round etc. Relaxing these assumptions remains for future work.

For the simplicity of explanation and analysis, we first assume a stable system that is not subject to churn. I.e., there is no change in the subset of the correct nodes in $U$. Relaxing this assumption is discussed in Section 4.7, where churn is added to the system.

Let us note that Byzantine nodes can send any message at any time, regardless of their state or the messages they receive. A restricted form of Byzantine nodes is called selfish. These nodes obey the protocol only when they benefit from this in some way. I.e., their behavior is not driven by malice, but rather by personal gain. Li et al. describe in their paper [LCW+06] the rational node, which is similar to our selfish node.
Chapter 4

BMON: Byzantine Monitor Service

4.1 Overview

In this chapter we present BMON, a Byzantine-resilient monitor service that is a building block for overcoming Eclipse attacks in unstructured overlays, as detailed in Chapter 5.

In BMON, each node $x$ in the overlay is assigned a randomly chosen fixed-size set of monitoring nodes whose role is to monitor $x$. The monitoring relationships are also verifiable. That is, given any three nodes $x, y,$ and $z$, node $z$ can verify locally (with no communication) whether $x$ is assigned to monitor $y$ (or vice-versa). In case indeed $x$ is assigned to monitor $y$ (or vice versa), this is true during the entire lifetime of the system. I.e., the monitoring relationships are also consistent over time. Additionally, node $x$ gradually learns who its monitoring nodes are, and whom it needs to monitor.

In the rest of this chapter, we refer to the set of nodes that monitor $x$ as $PS(x)$ and the set of nodes that $x$ monitors as $TS(x)$. We also use the terms $PS(x)$ and monitor set of $x$ interchangeably.

In general, the monitoring works as follows: Every $PS(x)$ node stores tuples of the form of $<x, Key, Value, timestamp_key>_x$, created by $x$. Each monitor of $x$ stores only a single tuple per each key created by $x$, the one that corresponds to the largest timestamp. Then, every node in the overlay $y$ can contact any node in $PS(x)$ and ask it for a specific key, matched to a specific tuple, that it stores. After receiving the replies, $y$ can either accept these replies or not based on the protocol (i.e., treat these replies as correct).

The remainder of this Chapter is organized as follows: First, we introduce BMON’s external interface and guarantees in Section 4.2. Next, we present the design blocks of BMON in Section 4.3 and the full monitoring protocol divided into small, yet well explained, pseudo-code parts in Section 4.4. Section 4.5 presents theorems to support BMON’s correctness with respect to the guarantees it provides, and is followed by an analysis in Section 4.6. Lastly, we discuss the possibility of adding churn to BMON in
4.2 External Interface and Guarantees

First, let us present the external interface that BMON provides. BMON accepts two types of messages:

1. **Store** \(< x, Key, Value, timestamp >_x\)
2. **Get** \((x, Key)\)

The **Store** message is used for nodes to send new tuples to their monitors for storage. The **Get** message is used by nodes to query their neighbors’ monitors for tuples that they store.

Next, we define a new term in order to specify formally the guarantees provided by BMON. We distinguish between a correct and cooperating monitor of \(x\) and a correct but non-cooperating monitor of \(x\). A correct monitor \(m\) of \(x\) is called a cooperating correct monitor of \(x\) iff \(m\) answers all queries regarding \(x\) to other nodes in the overlay and has stored at least one tuple for \(x\). In the rest of our work, for convenience, the abbreviation a cooperating monitor of \(x\) refers to a cooperating correct monitor of \(x\).

Now, BMON’s guarantees can formally be described as follows:

1. Let \(y\) be a cooperating monitor of \(x\) that receives a new tuple for a specific key created by \(x\) at time \(t\) and the timestamp included in that tuple is the largest received by any cooperating monitor of \(x\) for this key by time \(t + 2\). Then, w.h.p., by time \(t + 2\) all cooperating monitors of \(x\) have received this tuple as well.

2. Provided that an arbitrary correct node \(y\) in the overlay has already obtained the identities of \(x\’s\) monitors, then whenever \(y\) contacts \(x\’s\) monitors, \(x\) does not get any prior indication about this.

We prove these guarantees in Section 4.5, and use them in Chapter 5, where we present a defense mechanism against Eclipse attack that uses BMON.

4.3 Implementation

4.3.1 Monitor Selection

Each node \(x\) is assigned a subset of up to \(K\) nodes w.h.p. that are responsible for storing tuples created by \(x\). This assignment is based on a consistent one-way hash function \(H\) that maps ordered pairs \((x, y)\) of nodes’ IDs onto the real interval \([0, 1]\). Further, for every given ID \(x\), it is assumed that the possible IDs \(y\) are mapped uniformly onto \([0, 1]\) by \(H\). Thus, node \(x\) monitors node \(y\) iff \(H(x, y) \leq \frac{K}{N}\), where \(N\) is the number of nodes in the system. As mentioned in [PG07]), \(H\) can be a **SHA-1** or a **MD5** hash function, but
with the result normalized to the range \([0, 1]\). The consistent hash function \(H\) enables each node \(z\) to verify locally whether a node \(x\) is supposed to monitor node \(y\) and vice versa. Hence, verifying who should monitor whom is resilient to Byzantine failures. Furthermore, as we assumed in Chapter 3, nodes are given their IDs from a trusted entity and cannot fake them or adopt themselves IDs other than their assigned ID, and these IDs are assigned to the nodes in a uniform random manner. Consequently, w.h.p., the ratio of Byzantine nodes that are assigned to monitor a given node is similar to their ratio in the entire system [FKA10]. We denote \(f\) as the upper bound w.h.p. of the number of Byzantine (possibly collaborating) monitoring nodes out of the \(K\) monitors of each node.

### 4.3.2 Monitor Discovery

While the consistent hashing function enables verifying whether a given node should monitor another node, there is still an issue of how to detect these actual nodes. For this, we assume the existence of a Byzantine tolerance node sampling service such as Brahms [BGK+09]. Brahms ensures that each node eventually obtains changing uniformly random samples of nodes out of all nodes in the system. This is despite having a linear number of Byzantine nodes in the system and storing only a sub-linear number of IDs \(\Theta(\sqrt{N})\) at each node. Given the sampling service, periodically (every discovery protocol round), each node \(z\) obtains a new sample and checks for each ordered pair of nodes \((x, y)\) from the sample whether \(H(x, y) \leq \frac{K}{N}\). If this condition holds, then \(z\) sends a \texttt{Notify}(\(x, y\)) message to both \(x\) and \(y\). When \(x\) and \(y\) receive this message, they first verify that indeed \(H(x, y) \leq \frac{K}{N}\) and if so, node \(x\) updates its \(TS(x)\) and \(y\) updates its \(PS(y)\) accordingly. This test is used to prevent Byzantine nodes from sending bogus \texttt{Notify} messages. With this mechanism, each node gradually learns the identity of other nodes that monitor it and the identity of nodes it should monitor.

### 4.3.3 Using the Monitoring Service

Next, after defining who monitors whom and the way nodes in the system discover these monitoring relationships, we describe how nodes in the overlay use the monitors. I.e., we describe how nodes in the overlay store tuples at their monitors, and in turn get, and later accept, tuples from other nodes’ monitors.

Whenever a node \(x\) in the overlay wishes, it sends each of its monitoring nodes it is aware of a \texttt{Store}(<\(x, Key, Value, timestamp\)\_\text{key} >\(x\)) message, where \(x\) is the ID of the node that created and signed this tuple, \(Key\) is the key of this tuple, \(Value\) is the value that is matched to this specific \(Key\) at this specific \(timestamp\), and \(timestamp\) is a monotonically increasing timestamp for that specific \(Key\). Each monitor of \(x\) stores only a single tuple per each key created by \(x\). A node \(y\) that has received this tuple, stores it instead of the currently stored tuple with the same key of node \(x\) (if \(y\) stores such a tuple) iff \(y\) is supposed to monitor node \(x\) \((H(y, x) \leq \frac{K}{N})\) and the \(timestamp\)
value of the new tuple is larger than the timestamp value of the currently stored tuple with the same key of $x$ (if $y$ already stores such a tuple) and the tuple is properly signed by node $x$.

When a monitor $y$ of node $x$ receives a properly signed tuple by $x$ that includes the same key and timestamp as the one that it currently stores, but with a different value, it serves as a proof that node $x$ is Byzantine, as node $x$ created 2 tuples with the same key and timestamp, but with different values. If this happens, node $y$ has a proof that $x$ is Byzantine and it can handle it in a number of ways. For example, $y$ may disseminate a revocation message against node $x$ in the overlay, attaching these two tuples as a proof of Byzantine behavior. Notice that this is the only case in which a node that notices a Byzantine behavior also obtains signed proofs to support it. In other cases of Byzantine behaviors, signed proofs are not obtained and only the node that noticed the Byzantine behavior can take local action against the Byzantine node. In the latter case, we say that the correct node punishes the Byzantine node at the application level. The application level refers to the application that uses BMON, and in our case, the defense protocol against Eclipse attacks, as described in Chapter 5.

Whenever a node $z$ wishes to know the value of a specific key that $x$ has stored at its monitoring nodes, it sends a message to $x$ asking for at least $L$ IDs out of the of the $K$ monitoring nodes of $x$. Node $z$ does not ask for all $K$ monitors because the discovery process might fail to discover all of the monitors due to its probabilistic manner and since some of the nodes that should monitor node $x$ might be Byzantine and avoid any communication with any other node, thereby making themselves undiscoverable. Hence, $L$ is less than $K$ — we discuss a bound on its value in Section 4.6.1. The reply of $x$ is considered complete if it includes at least $L$ IDs of its monitoring nodes (the monitoring relationships are checked locally using the $H$ function). Once the system has been operating for an amount of time $t_c$ after which each node is assumed to know at least $L$ of its monitoring nodes w.h.p., if $x$ fails to return a complete answer in a timely manner, it serves as an indication that $x$ is Byzantine w.h.p.. We discuss the amount of time $t_c$ in Section 4.6.2.

After obtaining at least $L$ monitors of $x$, node $z$ asks them for a specific key matched to a specific tuple that they store for $x$ by sending each of them a $\text{Get}(x, \text{Key})$ message. Once the replies have been received, $z$ can either accept them (i.e., treat these replies as correct) or not, based on the acceptance mechanism described in Section 4.3.4.

Recall that in Chapter 3 we assumed a synchronous model with a global discrete clock, zero processing times, and message latencies of a single round. Using these assumptions, and in order to prevent races between the $\text{Store}$ and the $\text{Get}$ messages received at the monitors, we define that $\text{Store}$ messages can be received at the monitors only in even rounds. Respectively, $\text{Get}$ messages can be received at the monitors only in odd rounds. This way, it is guaranteed that nodes in the overlay that wish to get a stored tuple of another correct node in the overlay, will succeed in accepting it according to Section 4.3.4.
4.3.4 Byzantine Tolerant Acceptance Mechanism

Since Byzantine nodes may return false answers, in BMON we require obtaining multiple replies from monitors in order to accept them. Accepting an answer means that this answer has been received from enough nodes and can be treated as the most up-to-date answer (in terms of timestamps). In order to overcome Byzantine attacks, we present the following Byzantine tolerant acceptance mechanism. Before presenting the acceptance mechanism, we remind some assumptions and notations. First, recall that the model is synchronous and the message latencies are of a single round. Also, recall that $f$ is the upper bound w.h.p. of the number of Byzantine (possibly collaborating) monitoring nodes out of the $K$ monitors of each node, $L$ is the number of requested monitors when a node in the overlay wishes to get stored tuples from the monitors of another node, and that a correct monitor $m$ of $x$ is called a cooperating monitor of $x$ iff $m$ answers queries regarding $x$ to other nodes in the overlay and has stored at least one tuple for $x$.

The acceptance mechanism for overcoming Byzantine attacks requires that the value of the system parameter $L$ is set (e.g., by the system administrator) such that the intersection of any two sets of at least $L$ monitors of the same node includes at least one correct monitor w.h.p.. A lower bound on the value of $L$ with respect to the other parameters is given in Section 4.6.1.

When this requirement holds, the acceptance mechanism of the protocol provides the following properties:

1. An answer from the monitoring nodes of $x$ is accepted by a correct node only if the node has received at least $f + 1$ identical answers, corresponding to the largest timestamp among all the replies it has received.

2. A correct monitor of node $x$ is cooperating in monitoring node $x$ iff it knows about at least $L - 1$ other monitors of node $x$.

3. A cooperating monitor $y$ of node $x$ forwards tuples created by $x$ to all other monitors of $x$ known to $y$ using a Forward message sent immediately after $y$ has stored them.

4. Forward messages include (in addition to the corresponding tuple) the list of all monitors of $x$ known to $y$.

5. Nodes cache the IDs of the monitors they have obtained from other nodes for an unlimited amount of time. When they wish to query these monitors, they address them directly using the cached identities.

4.4 Protocol

For the sake of the algorithms in this section, and without loss of generality, we define a few conventions regarding nodes in the overlay. Node $y$ and node $w$ are arbitrary nodes.
in the overlay. Nodes $x_1$ and $x_2$ monitor node $y$. Node $z$ is a node in the overlay that wishes to get stored tuples created by node $y$ from $y$’s monitors.

As an arbitrary node in the overlay, node $w$ participates in the discovery of monitoring relationships (Algorithm 4.1). Periodically, $w$ obtains new samples from Brahms (Line 2), and for each ordered pair of nodes that can be created using these samples (Line 3), the hash condition is checked (Line 4). If the condition holds, a Notify message is sent to both nodes of the corresponding pair (Line 5). Notice that the monitoring relationships of node $w$ are mainly found and reported to $w$ by other nodes in the overlay as part of the discovery protocol and not necessarily found by node $w$ itself.

**Algorithm 4.1** Node $w$ participates in the discovery of monitoring relationships

1: Periodically do,
2: \( \text{NodesSample} \leftarrow \text{Obtain a new uniformly random sample of nodes using Brahms} \)
3: for all \((v_1, v_2)\) such that \(v_1, v_2 \in \text{NodesSample} \) and \(v_1 \neq v_2\) do
4: \( \text{if} \ H(v_1, v_2) \leq \frac{K}{N} \text{ then} \)
5: \( \text{Send Notify}(v_1, v_2) \) to both \(v_1\) and \(v_2\)
6: \( \text{end if} \)
7: \( \text{end for} \)

Without loss of generality, we assume that node $w$ is the node that found out about the monitoring relationship between node $x_1$ and node $y$, and thus sent them a Notify message. The handling of $x_1$ and $y$ when receiving a Notify message from node $w$ is presented in Algorithm 4.2 and Algorithm 4.3 respectively. First, in case a previous similar Notify message has already been received, and the monitoring relationship is already known to the local node, it ignores the Notify message (Line 1). Otherwise, the hash condition is checked, and only if it holds, the other node is added to the corresponding set (Line 3). Recall that if $x_1$ needs to monitor node $y$, it also needs to know about $L - 1$ additional monitors of $y$ (for the sake of forwarding the tuples), so it addresses $y$ with such a request (Line 5 in Algorithm 4.2).

**Algorithm 4.2** Node $x_1$ receives a Notify($x_1, y$) message

1: if \( y \in TS(x_1) \) then
2: Return
3: else if \( H(x_1, y) \leq \frac{K}{N} \text{ then} \)
4: Add $y$ to \(TS(x_1)\)
5: Send $y$ a request to get at least $L - 1$ additional monitors of $y$
6: end if

**Algorithm 4.3** Node $y$ receives a Notify($x_1, y$) message

1: if \( x_1 \in PS(y) \) then
2: Return
3: else if \( H(x_1, y) \leq \frac{K}{N} \text{ then} \)
4: Add $x_1$ to \(PS(y)\)
5: end if
After node \( x_1 \) discovers that it needs to monitor node \( y \) and sends \( y \) a request to get at least \( L - 1 \) of its other monitors, Algorithm 4.4 describes how node \( y \) handles this request. If \( y \) is already familiar with enough of its monitor nodes it provides them as an answers. Otherwise, it delays the answer until it accumulates enough monitors using the discovery protocol.

**Algorithm 4.4** Node \( y \) receives a request to get at least \( L_{\text{requested}} \) of its monitors from node \( x_1 \)

1. if \( PS(y).size \geq L_{\text{requested}} \) then
2. Send \( PS(y) \) to \( x_1 \)
3. else
4. Wait for some time
5. goto Line 1
6. end if

Algorithm 4.5 shows that when node \( x_1 \) receives an answer with \( y \)'s monitors from node \( y \), it checks the hash condition for the received nodes that it is not familiar with. For each node, if the condition holds, it inserts the corresponding node to the local known \( PS(y) \). Otherwise, it does nothing.

**Algorithm 4.5** Node \( x_1 \) receives a list of \( y \)'s monitors from node \( y \)

1. for \( v \) in the received monitors do
2. if \( v \in PS(y) \) then
3. Continue
4. else if \( H(v, y) \leq \frac{K}{N} \) then
5. Add \( v \) to \( PS(y) \)
6. end if
7. end for

Algorithm 4.6, describes what node \( z \) that wishes to address \( y \)'s monitors does in order to retrieve them. If \( z \) has already obtained \( y \)'s monitors in the past, it uses them directly. Otherwise, it sends \( y \) a request to get at least \( L \) of its monitors.

**Algorithm 4.6** Node \( z \) retrieves \( y \)'s monitors

1. if \( PS(y).size \geq L \) then
2. Return \( PS(y) \)
3. else
4. Send \( y \) a request to get at least \( L \) of its monitors
5. end if

As opposed to Algorithm 4.5, in Algorithm 4.7, the receiver of \( y \)'s monitors is not a monitor of node \( y \), but rather another node in the overlay that wishes to address \( y \)'s monitors. Hence, when receiving \( y \)'s monitors, first \( z \) checks that the answer contains at least \( L \) monitors as requested (Line 1). Then, for each node that \( z \) is not familiar with, it checks that this node truly needs to monitor \( y \) (Line 5). If it does, \( z \) adds the received node to the local known \( PS(y) \) (Line 9). If one of the tests fails, \( z \) punishes \( y \)
at the application level. The application level in the context of BMON refers to any application that uses BMON, and in our case, the defense protocol against Eclipse attacks. We note that in case node $z$ fails to receive a list of $y$'s monitors from node $y$ within 2 rounds from the round that $z$ sent the request to $y$, and $y$ is already assumed to know at least $L$ of its monitors, $z$ punishes $y$ in the application level as well.

**Algorithm 4.7** Node $z$ receives a list of $y$'s monitors from node $y$

1: if number of received monitors $< L$ then
2: Punish $y$ at the application level
3: Return
4: end if
5: for $v$ in the received monitors do
6: if $v \in PS(y)$ then
7: Continue
8: else if $H(v, y) \leq \frac{K}{N}$ then
9: Add $v$ to $PS(y)$
10: else
11: Punish $y$ at the application level
12: Return
13: end if
14: end for

Algorithm 4.8 is the method that needs to be invoked whenever the application level (e.g., the Eclipse defense mechanism) of node $y$ wishes to store a new tuple at its monitors (according to the external interface as described in Section 4.2). Node $y$ sends a $Store$ message to all of its monitors that it is aware of.

**Algorithm 4.8** Node $y$ sends a $Store(<y, Key, Value, timestamp_key, >y)$ to its monitors

1: for $v$ in $PS(y)$ do
2: Send $Store(<y, Key, Value, timestamp_key, >y)$ to node $v$
3: end for

Algorithm 4.9 presents the pseudo-code that $x_1$ executes when receiving a $Store$ message from node $y$. First, $x_1$ extracts the current stored tuple matched to the $NewTuple$'s node ID and key (Line 3). Next, $x_1$ performs some sanity checks: if $x_1$ does not need to monitor $y$, or the extracted node ID from $NewTuple$ does not match the sender ID of this message, or the same tuple is already stored, or $NewTuple$ is not signed properly, $x_1$ returns without doing anything (Line 4). Then, if no stored tuple with the same key is found, or the timestamp of $NewTuple$ is larger than the old tuple's timestamp, $x_1$ stores the new tuple instead of the old one, and sends a $Forward$ message to all of the other monitors of $y$ that $x_1$ is aware of (Line 7). The $Forward$ message is composed of the new tuple and a list of all the monitors of node $y$ known to $x_1$. If $x_1$ notices that there are two tuples, signed by $y$ with the same key and timestamp, but different values, $x_1$ spreads a revocation message against $y$ providing these two tuples as evidence (Line 13).
Algorithm 4.9: Node $x_1$ receives $\text{Store}(\text{NewTuple})$ from $y$

1: $\text{NodeID} \leftarrow \text{NewTuple}.\text{NodeID}$
2: $\text{Key} \leftarrow \text{NewTuple}.\text{Key}$
3: $\text{OldTuple} \leftarrow \text{extract the current stored tuple matched to Key of NodeID}$
4: if $y \notin \text{TS}(x_1)$ or $\text{NodeID} \neq y$ or $\text{OldTuple} = \text{NewTuple}$ or $\text{NewTuple}$ is not signed properly then
5: Return
6: end if
7: if $\text{OldTuple} = \text{NULL}$ or $\text{NewTuple}.\text{timestamp} > \text{OldTuple}.\text{timestamp}$ then
8: Store $\text{NewTuple}$ instead of $\text{OldTuple}$
9: $\text{ListOfMonitors} \leftarrow \text{PS}(y)$
10: for $v$ in $\text{PS}(y)$ do
11: Send $\text{Forward}(\text{NewTuple}, \text{ListOfMonitors})$ to node $v$
12: end for
13: else if $\text{NewTuple}.\text{timestamp} = \text{OldTuple}.\text{timestamp}$ and $\text{NewTuple}.\text{value} \neq \text{OldTuple}.\text{value}$ then
14: Spread a revocation message in the overlay against $y$ providing $\text{OldTuple}$ and $\text{NewTuple}$ as evidence
15: end if

Node $x_1$ forwarded the tuple received from node $y$ to all of the other monitors of $y$ known to $x_1$. In Algorithm 4.10 we can see the pseudo-code that node $x_2$, which also monitors node $y$, executes when receiving a $\text{Forward}$ message from node $x_1$. First, $x_2$ extracts the ID of the node that created this tuple (Line 1). Next, $x_2$ verifies that it needs to monitor $\text{NodeID}$ (Line 2), and if it does, it verifies that the sender of the message is a monitor of $\text{NodeID}$ as well (Line 5). If these checks complete successfully, $x_2$ uses Algorithm 4.9 in order to store this message, as it would have done if a matching $\text{Store}$ message had been received from $\text{NodeID}$ itself (Line 12). Lastly, a $\text{Forward}$ message also includes a list of monitors of $\text{NodeID}$ known to $x_1$. For each of these monitors that node $x_2$ is not familiar with, $x_2$ checks that this node truly needs to monitor $\text{NodeID}$, and if it does, $x_2$ adds it to the local known $\text{PS(\text{NodeID})}$ (Line 13).

Algorithm 4.11 is the method that needs to be invoked whenever the application level (e.g., the Eclipse defense mechanism) of node $z$ wishes to get a stored tuple from $y$’s monitors (according to the external interface as described in Section 4.2). First, node $z$ retrieves $y$’s monitors using Algorithm 4.6. Then, $z$ creates an empty set for the answers (tuples) to this query. Finally, node $z$ sends a $\text{Get}$ message to $y$’s monitors.

Algorithm 4.12 lists the pseudo-code that node $x_1$ executes whenever it receives a $\text{Get}$ message from node $z$. The $\text{Get}$ message is a request to get a stored tuple matched to a given key, for the monitored node $y$. First, if node $x_1$ does not need to monitor node $y$, or $x_1$ does not know about at least $L - 1$ other monitors of $y$, $x_1$ returns without doing anything (Line 1). Next, $x_1$ extracts the stored tuple that matches to the given $\text{Key}$ of node $y$ (Line 4). If there is a stored tuple matching to the given $\text{Key}$ for node $y$, $x_1$ sends it to $z$ (Line 6).

Algorithm 4.13 presents the handling of node $z$ in a received tuple. First, $z$ extracts
Algorithm 4.10 Node $x_2$ receives $\text{Forward}(\text{NewTuple}, \text{ListOfMonitors})$ from node $x_1$

1: $\text{NodeID} \leftarrow \text{NewTuple.NodeID}$
2: if $\text{NodeID} \notin \text{TS}(x_2)$ then
3: Return
4: end if
5: if $x_1 \notin \text{PS(\text{NodeID})}$ then
6: if $H(x_1, \text{NodeID}) \leq \frac{K}{N}$ then
7: Add $x_1$ to $\text{PS(\text{NodeID})}$
8: else
9: Return
10: end if
11: end if
12: Use Algorithm 4.9 for a $\text{Store(\text{NewTuple})}$ from $\text{NodeID}$
13: for $v$ in the received $\text{ListOfMonitors}$ do
14: if $v \in \text{PS(\text{NodeID})}$ then
15: Continue
16: else if $H(v, \text{NodeID}) \leq \frac{K}{N}$ then
17: Add $v$ to $\text{PS(\text{NodeID})}$
18: end if
19: end for

Algorithm 4.11 Node $z$ sends a $\text{Get(y, Key)}$ to $y$’s monitors

1: $\text{YMOnitors} \leftarrow \text{Retrieve y’s monitors (Algorithm 4.6).}$
2: $\text{ReceivedTuples(y, Key)} \leftarrow \emptyset$
3: for $v$ in $\text{YMOnitors}$ do
4: Send $\text{Get(y, Key)}$ to node $v$
5: end for

Algorithm 4.12 Node $x_1$ receives $\text{Get(y, Key)}$ from node $z$

1: if $y \notin \text{TS}(x_1)$ or $\text{PS(y).size} \leq L - 1$ then
2: Return
3: end if
4: $\text{StoredTuple} \leftarrow$ get the stored tuple matched to the given $\text{Key}$ for node $y$
5: if $\text{StoredTuple} \neq \text{NULL}$ then
6: Send $\text{StoredTuple}$ to $z$
7: end if
from the tuple the ID of the node who created it, and the tuple’s key. Next, $z$ verifies that the sender of this tuple is a monitor of $NodeID$, and that it has not already received a tuple with the same $NodeID$ and $Key$ from the same sender for this specific query, and that the tuple is signed properly (Line 3). If one of these checks fails, $z$ ignores this tuple. Finally, $z$ inserts the received tuple to the matching $ReceivedTuples$ list (Line 6).

**Algorithm 4.13** Node $z$ receives $Tuple$ from node $x_1$

1. $NodeID \leftarrow Tuple.NodeID$
2. $Key \leftarrow Tuple.Key$
3. if $x_1 \notin PS(NodeID)$ or $\exists T, (T, x_1) \in ReceivedTuples(NodeID, Key)$ or $Tuple$ is not signed properly then
4. Return
5. end if
6. Insert $(Tuple, x_1)$ to $ReceivedTuples(NodeID, Key)$

In order to decide whether to accept a tuple or not, multiple answers are required. Hence, one round after the first tuple with specific $NodeID$ and $Key$ has been received, $z$ decides whether to accept the tuple or not using Algorithm 4.14. First, $z$ extracts the tuple matches to the largest timestamp from $ReceivedTuples(NodeID, Key)$ (Line 1). Then, $z$ decides to accept $LatestTuple$ only if there are at least $f + 1$ identical tuples to $LatestTuple$ (Line 2). Otherwise, $z$ punishes $NodeID$ in the application level as $z$ did not manage to accept a tuple created by $NodeID$.

**Algorithm 4.14** Node $z$ decides whether to accept a tuple for a specific $ReceivedTuples(NodeID, Key)$

1. $LatestTuple \leftarrow$ get the tuple with the largest timestamp in $ReceivedTuples(NodeID, Key)$
2. if $ReceivedTuples(NodeID, Key)$ contains at least $f + 1$ identical tuples to $LatestTuple$ then
3. Accept $LatestTuple$
4. else
5. Punish $NodeID$ in the application level
6. end if

### 4.5 Correctness

We prove the correctness of BMON with respect to its guarantees listed in Section 4.2.

**Lemma 4.5.1.** For any cooperating monitor $y$ of $x$, at least $L - 1$ other monitors of $x$ know about $y$.

**Proof of Lemma 4.5.1.** Recall that a correct monitor $y$ of $x$ is called a cooperating monitor of $x$ iff $y$ answers all queries regarding $x$ to other nodes in the overlay and has stored at least one tuple for $x$ (according to the definition of a cooperating monitor in
Section 4.2). Also, a correct monitor of node $x$ is cooperating in monitoring node $x$ \textit{iff} it knows about at least $L - 1$ other monitors of node $x$ (according to Property 2 in Section 4.3.4). Hence, a correct monitor $y$ of $x$ is called a cooperating monitor of $x$ \textit{iff} $y$ answers all queries regarding $x$ to other nodes in the overlay and $y$ has stored at least one tuple for $x$ and $y$ knows about at least $L - 1$ other monitors of $x$. According to the protocol, when $y$ stores a tuple, it forwards the tuple to all other monitors of $x$ it is aware of. When these monitors receive the \texttt{Forward} message, they learn about $y$ being a monitor of $x$. Hence, at least $L - 1$ other monitors of $x$ know about $y$.

**Theorem 4.1.** Let $y$ be a cooperating monitor of $x$ that receives a new tuple for a specific key created by $x$ at time $t$ and the timestamp included in that tuple is the largest received by any cooperating monitor of $x$ for this key by time $t + 2$. Then, w.h.p., by time $t + 2$ all cooperating monitors of $x$ have received this tuple as well.

**Proof of Theorem 4.1.** Consider the time $t$ at which $y$ receives the new tuple $\mathcal{T}$ from $x$. If all monitors of $x$ receive $\mathcal{T}$ as well by $t$, then we are done. Hence, assume that some monitors did not receive the tuple $\mathcal{T}$ by time $t$. According to the protocol, $y$ forwards the tuple to all other monitors of $x$ it is aware of immediately after storing it, meaning that by time $t + 1$ at least $L$ monitors have received $\mathcal{T}$. Let us denote this subset of at least $L$ monitors by $M_1$ ($|M_1| \geq L$).

If by time $t + 1$ all cooperating monitors have received $\mathcal{T}$, then the theorem holds. Otherwise, consider a cooperating monitor $z$ of $x$ that has not received $\mathcal{T}$ by time $t + 1$. From Lemma 4.5.1, there are at least $L - 1$ monitors that know about $z$. Let us denote the subset that includes both $z$ and the other $L - 1$ monitors that know about $z$ by $M_2$ ($|M_2| \geq L$).

According to the requirement in Section 4.3.4, any two subsets of $L$ monitors intersect in at least one correct monitor w.h.p.. Hence, subsets $M_1$ and $M_2$ intersect in at least one correct monitor w.h.p.. This means that w.h.p., at least one of the cooperating monitors that has received $\mathcal{T}$ by time $t + 1$ knows $z$ and will forward $\mathcal{T}$ to $z$ by time $t + 2$.

**Theorem 4.2.** Provided that an arbitrary correct node $y$ in the overlay has already obtained the identities of $x$'s monitors, then whenever $y$ contacts $x$'s monitors, $x$ does not get any prior indication about this.

**Proof of Theorem 4.2.** Recall that the monitoring relationships are consistent, and do not change over time. Moreover, according to the properties of the acceptance mechanism in Section 4.3.4, $y$ caches the IDs of the monitors that it has obtained from $x$ for an unlimited amount of time. Consequently, $y$ does not contact $x$ before querying $x$'s monitors, but rather interacts with them directly. Recall that the messages' latencies are of a single round and therefore there is no danger that a Byzantine monitor would be able to inform $x$ about being contacted by $y$ before the messages of $y$ have arrived at all correct monitors.
4.6 Analysis

4.6.1 The Configuration Parameters

Recall that each node should return a set of at least $L$ monitoring nodes out of the total $K$ nodes that are assigned to monitor it w.h.p. ($L \leq K$), and that $f$ is the upper bound w.h.p. on the number of Byzantine nodes out of the $K$ monitors of each node.

First, we require that $L \geq 2f + 1$ in order to ensure that each node $z$ querying the monitors of another node $y$ will have replies from at least $f + 1$ correct monitors in order to mask erroneous replies returned by the Byzantine nodes as specified in Section 4.3.4. Next, let us denote by $M_1$ and $M_2$ two arbitrary subsets of at least $L$ monitors of the same node. To ensure that $M_1$ and $M_2$ intersect by at least one correct monitor w.h.p. (Section 4.3.4), we require that $K - L \leq f$. This way, $|M_1 \cap M_2| \geq f + 1$, meaning that indeed there is at least one correct monitor in their intersections w.h.p.. In summary, we have $K \leq 3f + 1 \land L \geq 2f + 1$.

4.6.2 Initialization Time

Recall that $t_c$ is the amount of time after which each node is assumed to know at least $L$ of its monitoring nodes. Once the system has been operating for this amount of time, if $x$ fails to return a complete answer in a timely manner that contains at least $L$ of its monitors, this serves as an indication that $x$ is Byzantine.

Let us analyze $t_c$ as the initialization time of the system. We denote by $F$ the number of Byzantine nodes in the overlay. We assume that Byzantine nodes do not participate in the discovery protocol, but their monitoring relationships can be discovered by correct nodes in the overlay. Recall that $N$ represents the total number of nodes in the overlay. We denote the $K$ monitors of node $x$ by $m_1, m_2, \ldots, m_K$ and remind the reader that the hash condition $H$ holds for all the pairs $(m_1, x), (m_2, x), \ldots, (m_K, x)$.

Looking at the monitoring relationship between $m_1$ and $x$ reveals that this monitoring relationship is discovered for the first time when some node (not necessarily $m_1$ or $x$) checks the hash function for this pair. Based on the discovery protocol described in 4.3.2, this discovery can occur when some node in the overlay obtains a new sample from Brahms that contains both $m_1$ and $x$. The probability that a node $y$ in the overlay will obtain a sized $s$ sample from Brahms that contains both $m_1$ and $x$ is: $\frac{s}{N} \cdot \frac{s-1}{N-1} \geq \frac{(s-1)^2}{N^2}$. Thus, the probability that $(m_1, x)$ is not checked by node $y$ in this particular sample is: $\leq \left(1 - \frac{(s-1)^2}{N^2}\right)$. Since all the correct nodes in the system participate in the discovery protocol, the probability that the pair $(m_1, x)$ is not checked by any correct node in a
specific discovery protocol round is:

\[
\leq \left(1 - \frac{(s - 1)^2}{N^2}\right)^{(N-F)}
\]

\[
\leq \left(1 - \frac{(s - 1)^2}{N^2}\right)^{\frac{N^2}{(s-1)^2}} \cdot \left(\frac{(s-1)^2}{N^2}\right)^{(N-F)}
\]

\[
\leq (e^{-1}) \frac{(s-1)^2(N-F)}{N^2}
\]

\[
\leq e^{-\frac{(s-1)^2(N-F)}{N^2}}
\]

In the third inequality we relied on the well known result that as \( M \) approaches \( \infty \), \( (1 - \frac{1}{M})^M \leq e^{-1} \).

The probability that \((m_1, x)\) is not checked by all the correct nodes for \( R \) rounds is:

\[
\leq \left(e^{-\frac{(s-1)^2(N-F)}{N^2}}\right)^R
\]

\[
\leq e^{-\frac{(s-1)^2(N-F) \cdot R}{N^2}}
\]

The probability that \((m_1, x)\) is checked by at least one correct node during \( R \) rounds is:

\[
\geq 1 - e^{-\frac{(s-1)^2(N-F) \cdot R}{N^2}}
\]

Given that, the probability that at least \( L \) out of the \( K \) pairs - \((m_1, x), \ldots, (m_K, x)\) are each checked by at least one correct node during \( R \) rounds is:

\[
\approx \sum_{i=L}^{K} \binom{K}{i} \cdot \left(1 - e^{-\frac{(s-1)^2(N-F) \cdot R}{N^2}}\right)^i \cdot \left(e^{-\frac{(s-1)^2(N-F) \cdot R}{N^2}}\right)^{(K-i)} \tag{4.1}
\]

The probability shown in equation 4.1 has been calculated for an arbitrary node \( x \) in the overlay, and thus it holds for any node in the overlay.

In order to measure the amount of time \( t_c \) after which each node is assumed to know at least \( L \) of its monitoring nodes, we analyze the probability shown in equation 4.1 as a function of the discovery protocol round. The results are found in figure 4.1, where \( N \) denotes the overlay size. From the results we can learn that in an overlay with 100,000 nodes, after 35 discovery protocol rounds, the probability that each node in the overlay knows at least \( L \) of its monitoring nodes is greater than 0.99. When the overlay size is 10,000 nodes, the convergence is more than twice as fast. The conclusions from this analysis is that for a stable system, the discovery protocol can be stopped after a relatively small number of rounds, or at least its frequency can be significantly decreased, thereby decreasing the protocol’s overhead.
Figure 4.1: The probability that an arbitrary node in the overlay knows at least $L$ out of its $K$ monitors, as a function of the discovery protocol round, where $L = 7$, $K = 10$, $s = 2\sqrt{N}$, and the fraction of Byzantine nodes in the system is 0.2.

4.7 Adding Churn

In this section we discuss the robustness of BMON to churn. In a system subject to churn, nodes can join, leave, or crash dynamically. A node that has joined and did not leave or crash is called active. We assume that the churn rate of the system is below some threshold, and that the number of active nodes in the system is always between $N - \varepsilon$ and $N + \varepsilon$, corresponding to the finite arrival model [Agu04].

In a system where nodes leave, or crash dynamically, the requirement described in Section 4.3.4 may not hold. I.e., the intersection of two arbitrary subsets of monitors of the same node whose original size was $L$ might not include even one correct monitor under churn. The shared correct monitor between the two sets might crash (or leave), leaving the two sets with $L - 1$ nodes each, and without any way to forward tuples between the sets. The described situation breaks the correctness of BMON as proven in Section 4.5.

In order to handle churn, we assume the existence of a failure detector, which determines which of the correct nodes is active and which is not [CT96]. Recall that our model is synchronous (Chapter 3), in which the existence of a perfect failure detector is a reasonable assumption as shown in [CT96].

First, in a system subjected to churn, we require $f + \alpha$ identical answers from a node’s monitors ($\alpha \geq 1$), instead of the $f + 1$ identical replies described in Section 4.3.4. The reason for this is that under churn the number of Byzantine monitors out of the $K$ monitors of each node might increase beyond $f$ for some time. Thus, we need to increase the number of identical replies that are sufficient in order accept an answer from the monitors. I.e., we require that $L \geq 2f + \alpha$.

Next, let us denote by $M_1$ and $M_2$ two arbitrary subsets of at least $L$ monitors
of the same node. Under churn, the intersection of $M_1$ and $M_2$ must include at least $\beta$ correct monitors w.h.p. ($\beta \geq 1$) in order to compensate for crashing (or leaving) monitors. To that end, we require that $K - L \leq f + \alpha - \beta$. This way, $|M_1 \cap M_2| \geq f + \beta$, meaning that indeed there is at least $\beta$ correct monitors in their intersections w.h.p.. To conclude, we have $K \leq 3f + 2\alpha - \beta \wedge L \geq 2f + \alpha$.

The above changes, in addition to the existence of a failure detector, allows us to strengthen the definition of a cooperating monitor made in Section 4.3.4 and adjust it to a churn subjected environment. Now, a correct monitor of node $x$ is cooperating in monitoring node $x$ iff it knows about at least $L - 1$ other active monitors of node $x$. Whenever a monitor of $x$ discovers, using its failure detector, that the number of active monitors of $x$ known to it has decreased below $L - 1$, it stops cooperating in monitoring node $x$. Recall that a non-cooperating monitor of $x$ does not answer requests from nodes in the system regarding $x$. The correct but non-cooperating monitor of $x$ starts cooperating again when it knows about at least $L - 1$ other active monitors of node $x$.

Notice that both $\alpha$ and $\beta$ are parameters that grow proportionally with the churn rate of the system, and present a tradeoff for the system. A larger value of them makes the system more robust to churn. Yet, an increased $\alpha$ and $\beta$ also yields more overhead, both computational and network, as the number of monitors of each node increases too. In the rest of this work, and in Chapter 5 in particular, we assume that our system is stable, i.e., without churn, as described in our model in Chapter 3.

### 4.8 Oral Messages vs. Signed Messages

Oral messages, as opposed to signed messages, do not require the use of asymmetric keys, thereby could lower the computational overhead of BMON. Due to the current design of BMON, using oral messages only, without any signed messages is not possible. The main reason for that is the forwarding between the monitors of the same node that is being used as part of the acceptance mechanism (Section 4.3.4), where the authenticity of the forwarded tuple is needed. In all other cases, there is no need for authenticity of messages, and thus signing additional messages will not reduce communication overheads.

The other direction of changing BMON so it will use oral messages only at the cost of additional communication, e.g., using Bracha’s uniform delivery protocol [Bra84], remains for future work. In particular, solving this issue despite churn is a non-trivial issue.
Chapter 5

Eclipse Attacks Defense Protocol

5.1 Overview

In this chapter we present our protocol for defending against Eclipse attacks in unstructured overlays. Intuitively, the defense works as follows: Our goal is to limit the indegree and outdegree of each node in the system, as this would prevent Byzantine nodes from eclipsing too many correct ones [SCDR04]. Yet, we need to be able to enforce this in a decentralized manner. We obtain this by relying on BMON - a Byzantine Monitor Service that was presented in Chapter 4.

Using BMON’s external interface (as described in Section 4.2), whenever a node $x$ in the overlay changes its neighbor set or reverse neighbor set, it uses Algorithm 4.8 to send to its monitoring nodes a new $\text{Store}$ message of the form $\text{Store}(x, \text{NS}(x), t_{s_{ns}} > x)$ or $\text{Store}(x, \text{RNS}(x), t_{s_{rns}} > x)$ respectively. Recall the signed tuple structure found in Section 4.3.3. Also, recall that each of the correct monitors of $x$ stores a maximum of 2 tuples of $x$ at any given time: one tuple for the $\text{NS}$ key and one tuple for the $\text{RNS}$ key. These are the tuples with the largest corresponding timestamp value.

With this in place, auditing the indegree of nodes is very simple. At unpredictable times, each node $x$ in the overlay audits the indegree of the nodes that are found in its $\text{NS}(x)$. For every node $y \in \text{NS}(x)$, $x$:

1. Sends to each of $y$’s monitors a $\text{Get}(y, \text{RNS})$ message (Algorithm 4.11).

2. After accepting a $\text{RNS}(y)$ answer using BMON (Algorithm 4.14), $x$ removes $y$ from its $\text{NS}(x)$ if $x \notin \text{RNS}(y)$ or $\text{RNS}(y)$ is greater than the indegree bound.

In case one of the inner Algorithms of BMON has failed to achieve its purpose, and a punishment at the application level is needed, $x$ removes $y$ from its $\text{NS}(x)$. We note that the defense protocol against Eclipse attacks is activated after a sufficient warm up time ($t_c$ in Section 4.3.3) after which each node in the overlay is assumed to know at least $L$ of its monitors. An analysis of the initialization time of BMON can be found in Section 4.6.2.
A similar check applies to the auditing of nodes’ outdegree as well. At unpredictable times, each node $x$ in the overlay audits the outdegree of the nodes that are found in its $RNS(x)$. For every node $y \in RNS(x)$, $x$:

1. Sends to each of $y$’s monitors a $\text{Get}(y, \text{NS})$ message (Algorithm 4.11).

2. After accepting a $\text{NS}(y)$ answer using BMON (Algorithm 4.14), $x$ removes $y$ from its $RNS(x)$ if $x \notin NS(y)$ or $NS(y)$ is greater than the outdegree bound.

In case one of the inner Algorithms of BMON has failed to achieve its purpose, and a punishment at the application level is needed, $x$ removes $y$ from its $RNS(x)$.

The remainder of this Chapter is organized as follows: Section 5.2 presents a proof of correctness of our defense protocol in terms of preventing Eclipse attacks, and tolerance against attacks on the defense protocol itself. In Section 5.3 we analyze the protocol’s overhead.

5.2 Correctness

We divide the discussion about the correctness of this defense protocol into 3 parts. First, we prove in Theorem 5.1 that a node $x$ that repeatedly does not include one of its correct neighbors (or reverse neighbors) $z$ in its reports to its monitors will be eventually detected w.h.p.. Next, we prove in Theorem 5.2 that the correct neighbors (or reverse neighbors) of a correct node $x$ that knows at least $L$ of its monitors will accept the last neighbor (or reverse neighbor) set that $x$ has sent to all of its known monitors as the real neighbor (or reverse neighbor) set of $x$. We prove these Theorems only for the neighbors of node $x$. Similar proofs also hold for the reverse neighbors of node $x$. Finally, we focus on why this defense protocol indeed prevents Eclipse attacks by Byzantine nodes.

Notice that if a Byzantine node tries to mount an Eclipse attack, if it reports all of its neighbors to its monitors, it will be immediately detected as having too many neighbors (according to the defense protocol described in this chapter). Hence, to avoid detection, the neighbor sets reported by such a Byzantine node must omit some of its neighbors. To that end, we have the following theorem.

**Theorem 5.1.** A node $x$ that repeatedly does not include one of its correct neighbors $z$ in its reports to its monitors will be eventually detected w.h.p..

**Proof of Theorem 5.1.** Suppose that node $z$ has already obtained $x$’s monitors in the past.

By combining Theorem 4.2 with the fact that the degree auditing in the Eclipse attacks defense mechanism is being done in unpredictable times, we establish that $x$ does not get any prior indication about queries that node $z$ sends to $x$’s monitors.

If $x$ omits $z$ from all of its reports, according to Theorem 4.1 all the cooperating monitors will get these reports w.h.p., and the next time $z$ contacts the monitors of
it will detect this omission w.h.p.. Hence, suppose that node $x$ fails to include $z$ in its reported neighbor set occasionally. This means that either $x$ creates (and spreads) multiple tuples with the same timestamps, some of which include $z$ and some do not, or that $x$ includes $z$ in some of its tuples and does not include $z$ in others in an alternating fashion.

In the former case, according to Theorem 4.1, all the cooperating monitors will get these reports w.h.p., and one of the correct monitors will detect w.h.p. that $x$ has created multiple tuples with the same timestamp and will have a proof for this. Consequently, we focus on the latter case in which $z$ is included in some reports and excluded from others. Since $x$ does not know when $z$ will contact $x$’s monitors, there is a probability (upper bounded by) $p$ that when $z$ contacts the monitors of $x$, the tuple stored there will include $z$ and $1 - p$ that $z$ will not be in the stored tuple. In other words, after $r$ instances in which $z$ has contacted $x$’s monitors, $z$ will detect $x$’s misbehavior with probability $1 - p^r$, which approaches 1 as $r \to \infty$.

**Theorem 5.2.** The correct neighbors of a correct node $x$ that knows at least $L$ of its monitors will accept the last neighbor set that $x$ has sent to all of its known monitors as the real neighbor set of $x$.

**Proof of Theorem 5.2.** Suppose that node $x$ is a correct node that knows at least $L$ of its monitors and sent its last neighbor set to these monitors.

Being the most up-to-date neighbor set of node $x$, $x$ assigns this tuple a larger timestamp than all the timestamps it has given before to tuples with the same key. Hence, all the monitors that $x$ is aware of and sent them the last tuple will receive, and thus store, this tuple, as it matches the largest timestamp available for this key.

Recall that according to BMON (Section 4.3.3), the Get and the Store messages are sent to the monitors in different rounds. Hence, starting from the next Get round (after the round in which $x$ has stored this tuple at its monitors), any correct neighbor of $x$ that queries at least $L$ monitors of $x$ (as BMON requires) will get at least $f + 1$ identical answers from the correct monitors of $x$. Due to the fact that all the correct monitors of $x$ have received and stored this new tuple, and the fact that no node but $x$ can produce a newer tuple with a larger timestamp for the same key, these $f + 1$ identical answers will include the last neighbor set that $x$ has sent them. According to the properties of the acceptance mechanism in Section 4.3.4, this neighbor set that matches to the largest timestamp available for this specific key will be accepted by $x$’s neighbors.

5.2.1 The Prevention of Eclipse Attacks

According to the insights that were made in [SCDR04, SNDW06], the indegree of an attacker who tries to mount an Eclipse attack is likely to be higher than the average indegree of correct nodes in the system. The reason for this is that an attacker that tries
to eclipse correct nodes will try to become a neighbor of as many nodes as it can, hence its indegree is likely to be higher than usual. Relying on Theorem 5.1 that claims that node $x$ cannot omit neighbors (or reverse neighbors) when it reports its neighbor sets (or reverse neighbor sets) to its monitors without being detected, we conclude that an attacker $x$ will have up to $i_t$ nodes that $x$ is their neighbor, and it will try to consume up to $o_t$ other correct nodes’ indegree, in order to prevent other correct nodes from neighboring them. The notations $i_t$ and $o_t$ stand for indegree threshold and outdegree threshold respectively.

Recall that $F$ is the number of Byzantine nodes in the overlay, and $N$ is the total number of nodes in the overlay. Now, we would like to calculate the number of nodes that can be eclipsed in the worst case scenario when using our defense. The total indegree of the malicious collaborating nodes in the system is $F \cdot i_t$. When exploring the worst case scenario, we consider a perfect attacker, which can successfully manipulate the neighbor selection mechanism of attacked correct nodes so they will only select nodes that are found under the attacker’s control as their neighbors. The maximum number of correct nodes that can be eclipsed by such an attacker is $F \cdot o_t$. I.e., all of the neighbors of the eclipsed correct nodes are selected from a subset of nodes that are found under the control of the attacker.

Lastly, a perfect attacker will also try to use its outdegree links in order to attack other correct nodes in the system. Due to the common fact that nodes do not really use their reverse neighbors to gain new information at the application level, and only answer reverse neighbors when probed by them, we do not consider these nodes as eclipsed nodes. However, as already mentioned in [SCDR04], without the outdegree bounding of nodes in the system, an attacker would be able to consume the indegree of all the correct nodes in the overlay, and theoretically attack the defense protocol suggested in this chapter. Thus, the bounding of the outdegree is also needed although it has no direct effect on the maximum number of possible eclipsed nodes.

To conclude, using degree bounding, an attacker that controls $F$ nodes in the overlay can eclipse a maximum of $F \cdot \frac{i_t}{o_t}$ correct nodes. Considering an overlay with $N = 100,000$, $F = 5,000$ and $i_t = o_t = 4$, a perfect attacker would be able to eclipse up to $\frac{5,000 \cdot 4}{4} = 5,000$ nodes out of 100,000 nodes in the overlay. Essentially, the message is that whenever $i_t \approx o_t$, then the maximal number of eclipsed nodes is similar to the number of Byzantine nodes in the system. This is in contrast to the fact that without our degree bounding defense, an attacker would be able to eclipse all the correct nodes in the overlay.

Due to the fact that an attacker can still eclipse correct nodes, one might wish to use a complementary defense mechanism at the application level. For example, when a correct node notices at the application level that it has been eclipsed (e.g., in a live streaming video application, not receiving video chunks for some time can be a sign for an Eclipse attack), it can disconnect from its neighbors and reconnect to other ones using a decentralized and secure mechanism such as Brahms. Obviously, this complementary
defense mechanism is slower than our defense, as it involves the detection of being
eclipsed at the application level, but can still provide an effective defense for modest
number of nodes that are being eclipsed by Byzantine nodes.

5.3 Overhead

In this section we analyze the overhead of our suggested defense protocol, including the
overhead of BMON. We distinguish between the computational overhead on each of the
correct nodes in the overlay, and the total network overhead of the defense protocol. For
simplicity of analysis, we assume bidirectional links between the nodes in this section.
I.e., if node $x$ is a neighbor of node $y$, then node $y$ is a neighbor of node $x$ as well.

5.3.1 Computational Overhead

Let us calculate the computational overhead on each of the correct nodes in the system.
When analyzing the computational overhead on correct node $x$, we distinguish between
3 kinds of computational operations:

1. Signing operation using a private key.

2. Verification of a signature using a public key.

3. Using the hash function $H$ - used by all the nodes in the system when they need
to verify a monitoring relationship.

The signing operation using a private key is used by $x$ only when it creates new
tuples. Each change in $x$’s neighbor set requires it to sign on a new tuple containing
the new neighbor set. Hence, the number of signing operations that node $x$ performs is:
$r_c$, where $r_c$ is the average rate in which a node changes its neighbor set.

The verification of a signature using a public key is being used by $x$ whenever it
verifies tuples which have been created by its neighbors or its monitored nodes. In
particular, as a monitor of other $K$ nodes, $x$ receives signed tuples from its monitored
nodes each time they change their neighbor sets and needs to verify the signatures.
Notice that the same tuple can be received up to $K$ times at node $x$ due to the forwarding
of tuples between the monitors, but it has no effect on the computational overhead,
as duplicate messages are thrown away before verifying the signature (Algorithm 4.9,
Line 4). Moreover, in unpredictable times, $x$ audits its neighbors degree as part of the
defense against Eclipse attacks. For each neighbor it audits, it gets $K$ responses from
that neighbor’s monitors, and it needs to verify that all of them are signed properly
by that neighbor. Hence, the number of verification of signature operations that node
$x$ performs is: $Kr_c + Knr_a$, where $r_c$ is the same as before, $r_a$ is the average rate in
which a node audits its neighbors, and $n$ is the degree bound (a node in the overlay
cannot have more than $n$ neighbors).
The hash function $H$ is being used by $x$ whenever it verifies monitoring relationships. As a participant of the discovery protocol of the monitoring relationships in the system (Section 4.3.2), $x$ uses $H$ for each ordered pair of nodes in each sample of nodes it obtains from Brahms. Let us note the sample size obtained from Brahms in $s$. Hence, $x$ uses the $H$ function for this matter $s(s-1)r_d$, where $r_d$ is the rate in which nodes in the system obtain samples from Brahms and use them. As a participant of the defense protocol, $x$ needs to verify the monitors of each new neighbor it obtains. Hence, $x$ uses the hash function $Kr_n$ times for this matter, where $r_n$ is the average rate in which a node obtains a new neighbor. Lastly, in a stable system the number of times that $x$ is using the hash function $H$ in order to discover its monitored nodes and its monitors is bounded by a constant $c$, as the monitoring relationships are consistent. To conclude, the total number of times that the hash function is being used by $x$ is:

$$s(s-1)r_d + Kr_n + c.$$ 

Let us note that signing new tuples is restricted to the rate in which nodes create their tuples, and in the context of Eclipse attacks, to the rate they change their neighbor sets. This way, Byzantine nodes that create tuples in a high rate and try to manipulate their neighbors suffer from a major computational overhead, whereas correct nodes that follow the protocol, do not.

### 5.3.2 Network Overhead

In contrast to the computational overhead, we define the network overhead as the addition in messages caused by our defense protocol. Due to the fact that our protocol does not generate large messages, we consider any sending of a message as a single message.

Let us describe the network overhead on a node $x$ in the overlay. Each change in $x$’s neighbor set requires it to send a new signed tuple to each of its $K$ monitors, hence it involves sending $K$ messages. In turn, $x$ receives signed tuples from its $K$ monitored nodes, each time they change their neighbor sets. If a tuple of one of its monitored nodes $y$ is accepted, $x$ forwards this tuple to all the other monitors of $y$ it is aware of.

Periodically, $x$ audits its neighbors’ degrees as part of the defense against Eclipse attacks. For each neighbor it audits, it sends $K$ requests to the monitors of that neighbor. As a monitor of $K$ other nodes, $x$ receives requests to get certain tuples matched to specific keys from its monitored nodes’ neighbors. Node $x$ answers each request with a single message containing the requested tuple if it exists.

Lastly, as a participant in the discovery protocol of monitoring relationships in the system (Section 4.3.2), $x$ sends Notify messages whenever it finds an ordered pair of nodes that satisfies the hash function $H$. For each pair $(y,z)$ that satisfies $H$, $x$ sends a Notify$(y,z)$ message to both $y$ and $z$.

To conclude, the number of messages each correct node sends is: $Kr_c + K^2r_c + Knr_a + Knr_a + 2r_h$, where $r_c$ is the average rate in which a node changes its neighbor.
set, $r_a$ is the average rate in which a node audits its neighbors, $n$ is the degree bound, and $r_h$ is the average hit-rate of the discovery protocol. Multiplying the number of messages that each node sends by the number of nodes in the system gives the total network overhead of our defense: $N(Kr_c + 2Kn r_a + K^2 r_c + 2r_h)$

Recall that according to our model, described in Chapter 3, Byzantine nodes do not follow the protocol, and can send whatever they want, whenever they want. Hence, they can increase the network overhead, a fact that also applies to every environment with Byzantine nodes. On the other hand, detecting at the network level nodes that try to send a significantly larger number of messages than what is expected can be used to detect them as likely to be Byzantine.
Chapter 6

Discussion

In this work, we have presented a defense mechanism against Eclipse attacks in unstructured overlays. To the best of our knowledge, our work is the first solution to this problem. Our defense is based on limiting the degree of nodes, which by definition prevents Byzantine nodes from eclipsing a large number of correct ones. At the heart of our enforcement mechanism is BMON, a decentralized Byzantine monitor service that can detect nodes that have a large number of neighbors, or try to lie about their neighborhoods. This information can be retrieved by neighbors and potential neighbors of such nodes, which then avoid neighboring with them.

Some of the appealing properties of BMON include being fully decentralized, and that each node can verify locally whether a given node $x$ should monitor another node $y$ or vice versa. BMON’s correctness relies on probabilistic guarantees, which implies that our system as a whole provides its guarantees in a probabilistic manner. The work also includes an analytic study of the computational and communication cost of our solution, as well as a probabilistic analysis of its warmup time.

Looking into the future, the immediate goal is to relax the synchrony assumption. Also, we would like to implement and perform an empirical performance evaluation study of our solution. Finally, when exploring the general availability problem in distributed systems [MG09], we believe that BMON can play a central role in designing a Byzantine tolerant availability monitoring service. To the best of our knowledge, currently, there is no availability monitoring service that can be used in adversarial environments.
Bibliography


[DdVP+02] Ernesto Damiani, De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati, and Fabio Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In


To summarize, our new network defense system is effective in defending against attacks on networks lacking structure. From the Warm up time protocol and its observations and calculations, we can analyze and learn from our system. In the future, we plan to implement the attack on networks lacking structure to study its impact. Additionally, we plan to implement the BMON protocol (incorporated into the model) to understand the attack on the network being defended. Moreover, we seek to implement a new protocol that synchronizes our model with the necessary changes in our protocol to work in a non-synchronized model.

The model, validated by various protective protocols, is based on the network defense defined in our model.
Let $H(x, y) \leq K$ be satisfied. For a fixed $x$, if $y$ is chosen uniformly at random from $\{0, 1\}$, then with probability at least $1 - \alpha$, we have $H(x, y) \leq K$. Hence, we can use $H$ as a hash function to distribute the nodes uniformly.
swers ב(factor) המאוחזים ב(kho), או במספר תחומים נוספים

ולפי,涅ץ, ר. "עקרונות התוכנית המאוונת" (1980), הוצאת ר. י. ידיעות אוניברסיטת

המעון

ולפי,涅ץ, ר. "עקרונות התוכנית המאוונת" (1980), הוצאת ר. י. ידיעות אוניברסיטת

המעון

ולפי,涅ץ, ר. "עקרונות התוכנית המאוונת" (1980), הוצאת ר. י. ידיעות אוניברסיטת

המעון

ולפי,涅ץ, ר. "עקרונות התוכנית המאוונת" (1980), הוצאת ר. י. ידיעות אוניברסיטת
A summary of large-scale P2P networks, where centralized overlays are typically used. In most cases, overlays are implemented in the network architecture and are maintained by centralized nodes.

Centralized overlays, such as Chord and Pastry, maintain a fixed topology of nodes to facilitate communication. However, this can lead to vulnerabilities, such as single points of failure.

Constraint Routing Tables (CRT) are used to enforce policies on the network. CRTs are maintained by each node and are updated as needed.
המחקר בוצע בהנחייתו של הפרופסור רועי פריידמן, בכוללה של מדעי המחשב. זה עלה תמיכה כספית הנדיבת בהשתלמותו.

אני מודה למל ברליק, למשרדי המודע ולטכניון על התמיכה והיכולת hendיבתי בخصوصה בבעלותם.
הגנה מפני התקפות האפלת
ברששות חסרות מבנה

ינויו על מחקר

לשם مليולי חלקי שלHDRישות לקבלת התואר
מיסטר למדעיimiento המחשב

עידו גונן

ה瘁 של הטכניון – מכון טכנולוגי לישראל
שבט החשיע"ה חפחת
פברואר 2015
הגנה מפניע
התקפות האפליה
ברשתות השרותים מובננים

עידו גוון