7-2022

Asynchronous Messaging in a P2P System: Defending Against a Storage Exhaustion Attack on Kademlia DHT

Maxim Biro

Asynchronous Messaging in P2P Systems: Defending Against a Storage Exhaustion Attack on Kademlia DHT

by
Maxim Biro

Bachelor of Science
Computer Science
Florida Institute of Technology
2015

Bachelor of Science
Mathematical Sciences
Florida Institute of Technology
2015

A thesis submitted to the
College of Engineering and Science at
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Information Assurance and Cybersecurity

Melbourne, Florida
July, 2022

We the undersigned committee hereby recommend that the attached document be accepted as fulfilling in part the requirements for the degree of Master of Science in Information Assurance and Cybersecurity.

"Asynchronous Messaging in a P2P System: Defending Against a Storage Exhaustion Attack on Kademlia DHT"
a thesis by Maxim Biro

Marius C. Silaghi, Ph.D.
Professor
Computer Engineering and Sciences
Major Advisor

William H. Allen, Ph.D.
Associate Professor
Computer Engineering and Sciences

Nezamoddin Nezmoddini-Kachouie, Ph.D.
Associate Professor
Mathematical Sciences

Philip J. Bernhard, Ph.D.
Associate Professor and Department Head
Computer Engineering and Sciences

Abstract

Title: Asynchronous Messaging in a P2P System: Defending Against a Storage Exhaustion Attack on Kademlia DHT

Author: Maxim Biro

Thesis Advisor: Marius C. Silaghi, Ph.D.

An instant messaging service designed using a peer to peer distributed network architecture has many appealing properties it gets for free: high scalability, cheap operational cost and no reliance on a third party to provide the service. However, the nature of the distributed network architecture makes implementing some of the instant messaging features rather challenging, asynchronous messaging being one of them. The asynchronous messaging requires that peers store arbitrary data on behalf of other peers for prolonged periods of time, often measured in days, which, if not kept in check, can be easily abused by malicious actors by spamming the network with bogus store requests, resulting in a storage exhaustion of the network and DoS of the asynchronous messaging feature. In this thesis we present two reputation-based techniques against the storage exhaustion DoS attack – a local knowledge one, in which the reputation is tracked by a small group of peers, and a global knowledge one, in which all peers keep track of the reputation on a blockchain ledger. We discuss the security, overhead and scalability of the techniques. We implement a Kademlia

DHT that acts as an overlay network for our peer to peer distributed messaging service, we implement the defense techniques and measure their performance, and we simulate the attack on the messaging service and measure the success rate of the implemented defense techniques.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ability to send a message to another person using a computer is seen as an essential functionality of a computer nowadays. This is not surprising, given how the ability to message someone has been available for a long time, at least since the multi-user operating systems were developed in 1960s [27], which predates the creation of the Internet. With the Internet being created in 1968, a wide variety of message exchange services became available. Some of the popular message exchange services of 1960s-2000s include: UNIX talk, email, mailing lists, Internet Relay Chat (IRC), ICQ, AOL Instant Messenger (AIM) and Skype, most of which are still in use today.

Messaging services can be split into three categories based on their network architecture: centralized, decentralized (federated), and distributed.

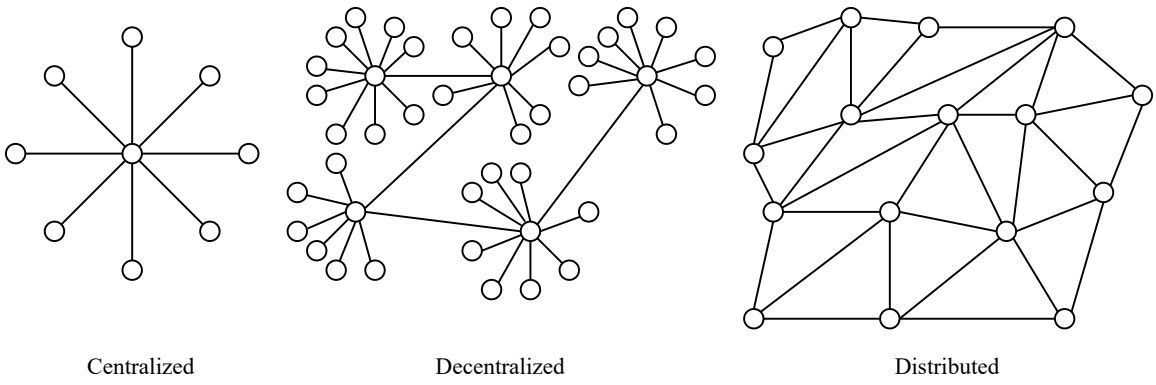

Figure 1.1: Network architectures.

1.1 Centralized Architecture

In the centralized architecture there is a central server controlled by a service provider that all users connect to in order to use the service. The central server can be a single server, or it can consist of multiple servers that act as a single one. The central server is usually used as a relay for data transmission between users: a user sends data to the server, and the server then re-sends it to the intended recipients. Therefore, the centralized architecture can be described as a client-server-client architecture.

The advantage of the centralized architecture is that it is simple – it is easy to reason about and to implement as this kind of the architecture is very common and well-understood. It also allows for easy control and moderation of the data, users and the service itself, since everything has to go through the central server that a service provider controls.

There are several drawbacks to the centralized architecture. The obvious one is its reliance on a central server, which becomes the single point of failure. The central

server becoming inaccessible is an issue – if the central server goes down, all of the users will be denied access to the service, or if the central server gets blacklisted or somehow blocked, some of the users would not be able to access it. The central server getting compromised is also an issue, as an attacker would be able to perform various man-in-the-middle attacks on all of the users, for example eavesdropping or spreading malware, without the users realizing it. Another drawback is the high cost of scaling. As a service gains more users over time, it needs more and more servers. If no servers get added, the service will degrade, unable to keep up with the new users. Also, because the number of users always fluctuates, the service needs to have server resources ready to handle spikes in user activity. Another drawback of the centralized architecture, from user's perspective, is that it allows vendor lock-in. Due to a centralized service being run and controlled by a single service provider, it is possible for them to enforce being the only provider of the service by using proprietary software and network protocols, disallowing anyone else to run the service, both technically and legally. This means that the users are at the mercy of the service provider. For example, if the service provider stops providing the service, the service would be lost to users forever. Note that a centralized service does not have to have a vendor lock-in, it is possible for it to use open standards and open source software, which would allow anyone to become a service provider, so if a service provider stops providing the service, it would be possible to find another similar service someone else is running or start one yourself.

The majority of the aforementioned services are centralized, with UNIX talk, mailing lists and IRC [17] being open standards and having open source implementations, and AIM being proprietary. It might be a bit non-intuitive as to why IRC is considered centralized, as big IRC networks usually consist of multiple servers. IRC is an example of when many physical servers act as a single abstract one, an implementation detail required for scaling to the number of users. An IRC network is still being run and controlled by a single service provider.

1.2 Decentralized Architecture

The decentralized architecture, also known as federated, is when several fully independent centralized networks (sub-networks) that run the same kind of a service are linked together to allow users of one network to communicate with users of another. In such architecture there is no central server that governs the entire network, instead there are many independent sub-networks that govern only sections of the entire network, which communicate between each other creating the entire network. The decentralized architecture is a client-server-server-client architecture when communicating between sub-networks, or a client-server-client when communicating within the same sub-network, in the case of the latter being the same as the centralized architecture.

The decentralized architecture is a little harder to implement and reason about than the centralized architecture, as it allows for users of one sub-network to commu-

nicate with users of other sub-networks. It is also impossible for a single sub-network service provider to control and moderate what happens on the entire network, as the entire network consists of independently run sub-networks and a sub-network service provider can control only what happens on their own sub-network.

The decentralized architecture makes the issue of a central point of failure less severe than in the case of the centralized architecture – if a central server becomes inaccessible, rendering a sub-network non-functional, only users of that sub-network would be denied the service, users of other sub-networks would not be affected, apart from being unable to communicate with users of the sub-network that went down. Scalability is still of concern, but in the case of the decentralized architecture, each sub-network provider is responsible only for their own scalability. This means that if the number of users in a given sub-network stays constant, the sub-network provider might not need to take any action in response to the number of users in the network as a whole increasing. This also means that if a sub-network operator has limited server resources, they could limit the number of users allowed on their sub-network without impacting the network as a whole. Also, because of the nature of the decentralized architecture, users are no longer at the mercy of a single service provider, as there are many independent sub-networks controlled by different service providers interconnected into a single network, so if one service provider decides not to provide the service anymore, there are alternative sub-networks available that users could migrate to in order to access the same network – there is no vendor lock-in.

Out of the mentioned messaging services only email uses a decentralized architecture. Email has many mail servers (sub-networks) run by many various groups, from big companies running open user registration mail servers having millions of users, to personal mail servers with closed registration used by a handful or a single user.

1.3 Distributed Architecture

The opposite to the centralized architecture is the distributed architecture. In a pure distributed architecture each network participant is equal, there is no authority such as a central server, in fact there are no servers at all, instead each user is directly connected to other users, together creating an interconnected network, and any global data that would usually require a central server to be stored on is instead distributed between every user of the network. In contrast to the centralized and decentralized architectures, a messaging service using the distributed architecture is being run directly on users' computers, instead of central servers. In that sense, a user is both a client and a server. The distributed architecture is a user-user, or the more commonly called – peer-to-peer architecture.

The distributed architecture is the hardest to implement and reason about. There is no central server that is used for user discovery, instead connections to the users that are already in the network are made and special algorithms are used to discover other users and organize the connections between them in an efficient manner. There is also no central server to store the data on, instead all the data is scattered among all

of the users, with special algorithms used to store and retrieve the data in the swarm of users. With all users being independent and the data being scattered among them all, there is no way for someone to control or moderate the data, users and the service without a consensus among the users.

The distributed architecture does not have any single point of failure as everyone in the network is equal, so if a user goes offline, the service would not be affected for everyone else. Even if several users go offline, the service would not be affected as distributed architectures are generally designed with a certain fault tolerance and redundancy in mind, such that it is unlikely to be exceeded in normal day to day operation. There is no scalability issue, as the service scales automatically with its users, since each user shares a part of the load of the service – there is no need to spend any money on hardware to run the service as it is running on users' hardware. There is also no vendor lock-in, because users are the ones providing the service for themselves. The service will be available for as long as there are users.

Out of the mentioned messaging services, there is no service that could be called purely distributed. ICQ and Skype are the closest to having a distributed network architecture – they use a hybrid approach that combines both the centralized and distributed architectures. They rely on a central server for some of their functionality, like authentication and user discovery, but the data exchanged between the users happens directly peer-to-peer, without a server in-between [2, 18]. Note that we are talking about ICQ and Skype as they were in their early days, because their

Table 1.1: Properties of messaging service network architectures.

Property	Centralized	Decentralized	Distributed
Easy to design	Yes	Maybe	No
High data control	Yes	Maybe	No
High survivability	No	Maybe	Yes
High scalability	No	Maybe	Yes
Low hardware cost	No	Maybe	Yes
Vendor lock-in impossible	No	Maybe	Yes

architectures have changed drastically since then, becoming more centralized.

1.4 Comparison of Architectures

The Table 1.1 summarizes the comparison of the messaging service network architectures. As we can see from the table, a messaging service using the distributed architecture provides many desired properties which are hard to achieve using other architectures: high survivability – there is no single point of failure and it is hard for someone to intentionally deny access to the service, high scalability – the service scales automatically with the number of users, low hardware cost – it runs on already existing users' machines, and low chance of a vendor lock-in – in a fully distributed service every user is a vendor. These properties make the distributed architecture rather attractive for a messaging service. However, these desired properties come at the cost of a messaging service being harder to design and having low data control.

1.5 Disadvantages of the Distributed Architecture

Let us take a closer look at the disadvantages of the distributed architecture: the difficulty of design and the low data control.

1.5.1 Difficulty of Design

The issue of being harder to design comes from it being hard to implement and reason about a distributed messaging service. Instead of there being just a single instance of the service that has a global view of everything going on in it, there is a local-only view instance of the service running on every user's machine, communicating and collaborating with other instances of itself, together providing an efficient and fault-tolerant distributed messaging service. Every such instance, every peer, has to keep a list of other peers, communicate with the peers, exchange information about the known peers with them, keep the peer list fresh by removing stale peers from it, replicate, verify and republish data, accommodate new peers joining and so on. The service needs to be designed to work efficiently if there are only a couple of users, as well as if there are millions of them. Such services are commonly called overlay networks.

While it is not easy to design such a service from scratch, some well-studied overlay networks exist, which can be used as a backbone and be extended upon to provide a distributed messaging service. It is common to implement an overlay network using a distributed hash table (DHT). A DHT provides an efficient and fault-tolerant way

to store data on many independent peers. It achieves this by organizing all peers in such a way that it is possible to efficiently find specific data and peers in the network, and that each data entry is stored on – and each peer is connected to – just enough other peers in order for the DHT to keep functioning while sustaining a certain fault tolerance of peers going offline. While DHTs are made with the goal of storing data, it is possible to omit implementing the data storing functionality if it is not desired, which results in an efficient and fault-tolerant way to do peer discovery – a backbone of any distributed service.

Using a DHT solves a big chunk of the issue of a distributed messaging system being harder to design, with all that is left being designing messaging features on top of the DHT, such that they work with users' local-only view of the network and scale well.

1.5.2 Low Data Control

The issue of low data control comes from the data stored in a distributed service being scattered among multiple independent users, with each user having a limited view of all the data being stored on the service, making it hard to control and moderate the data being stored on the service as a whole. Although the low data control can be seen as an advantage – everyone is free to store whatever data they want, it is in fact an issue because that makes it hard to enforce a fair use of the service – nothing stops a small number of malicious users from abusing this lack of control to significantly

degrade the quality of the service for all of the users. For example, a malicious user could launch a denial of service (DoS) attack on the service by asking it to store a huge amount of data, causing the network to run out of its capacity of storing any new data. This is not big of an issue if a distributed service allows storing only small and short-lived data, e.g. for several seconds or minutes, as it would require a lot of resources from the attacker to keep the data in the network – the attacker would need to keep republishing all the data over and over again to prevent it from expiring and getting deleted, however it becomes a major issue when long lived data storage is allowed. This makes implementing some instant messaging features that require long lived data to be stored rather challenging, one such feature being asynchronous messaging.

1.6 Asynchronous Messaging

Asynchronous messaging is the message exchange between a sender and the receiver when only one of them is online at a time. It is also commonly known as offline messaging. Offline messages must be stored in the distributed network for a relatively long period of time, often days, such that when the receiver of those messages comes online, they would be able to access the messages. As in the DoS example, an attacker could abuse this feature by requesting different peers to store bogus offline messages on the network until the network runs out of the capacity to store any new offline messages, effectively denying the feature to all other users and wasting users' resources

– disk space and bandwidth – for days. While this could be mitigated rather easily by introducing a trusted centralized service that keeps track of how many messages everyone has requested to be stored on the network, allowing the storing peers to check if they should accept the store request from a particular peer, relying on a centralized service is against the spirit of a purely distributed peer to peer architecture and would invalidate some of the mentioned desired properties of the network architecture. A purely distributed peer to peer solution for this is desired.

1.7 In This Thesis

In this thesis we present two purely distributed peer to peer reputation-based techniques against the storage exhaustion attack – a local knowledge one, in which the reputation is tracked by a small group of peers, and a global knowledge one, in which all peers keep track of the reputation on a blockchain ledger. We discuss the security, overhead and scalability of the techniques. We implement a Kademlia DHT [14] that acts as an overlay network for our peer to peer distributed messaging service, we implement the defense techniques and measure their performance, and we simulate the attack on the messaging service and measure the success rate of the implemented defense techniques.

Although the specific goal of the thesis is to make asynchronous messaging in a DHT-based peer-to-peer instant messaging system more resilient against the storage exhaustion DoS attack, it could be generalized to making Kademlia DHT more re-

silient against storage exhaustion DoS attacks, as everything described here applies to a base Kademlia DHT system as well.

1.8 Contributions

The contributions of this thesis are the literature survey, the two defense techniques against the storage exhaustion attack, and the experimental validation of the defense techniques.

1.9 Structure of This Thesis

In the next chapter we introduce Kademlia DHT and Bitcoin Blockchain, providing an overview of attacks on Kademlia DHT. In Chapter 3 we introduce a Kademlia DHT based instant messaging system, the storage exhaustion attack on it and two defense techniques against the attack. In Chapter 4 we implement a Kademlia DHT, simulate the messaging system, implement the defense techniques, simulate the techniques, measuring and comparing their performance and overhead, and simulate the storage exhaustion attack on the system, measuring the success rate of the defense techniques. In Chapter 5 we conclude the thesis.

Chapter 2

Background

This chapter reviews technologies and ideas the messaging service and defense techniques in this thesis are based on and provides an overview of attacks on Kademlia DHT.

2.1 Kademlia DHT

Kademlia [14] is one of the most popular peer-to-peer distributed hash tables. Its unique characteristic is the use of the XOR metric for calculating distance in the key space. Kademlia uses uniformly distributed random 160-bit numbers for node IDs and opaque 160-bit numbers for the keys, with the distance between any two 160-bit identifiers calculated as a bitwise exclusive or (XOR) interpreted as an integer. ⟨key, value⟩ data pairs are stored on nodes with IDs closest to the key.

2.1.1 Advantages of XOR Metric

Many of Kademlia's advantages stem from its use of the non-Euclidean XOR metric.

XOR is unidirectional. For any node x and distance $\Delta > 0$, there exists exactly one node y such that $d(x, y) = \Delta$, where $d(x, y)$ is a distance function. Meaning that lookups for the same key converge along the same path regardless of where they originate from, allowing for more efficient routing, efficient routing table structure and better data caching.

XOR is also symmetric. $\forall x, y : d(x, y) = d(y, x)$ – nodes that are close to x are also the nodes that x is close to. Meaning that a node will receive queries from the same distribution of nodes that it has in its routing table, allowing the node to keep its routing table fresh as a side-effect of receiving queries, reducing the number of manual refreshes required.

For comparison, Chord [26] is unidirectional but not symmetric, and Pastry [21] uses two metrics, where nodes closest by first could be far by the second, reducing performance.

2.1.2 Routing Table

Every node in Kademlia keeps a routing table, which is a list of 160 k-buckets, where an i-th k-bucket in the list, $0 \leq i < 160$, is a list of k nodes which are between 2^i and 2^{i+1} distance apart from the node keeping the routing table. In the other words, bucket 0 can contain only a single node – a node with 159 most significant bits of its

node ID equal to the node keeping the routing table, bucket 1 can contain only the four nodes – nodes with 158 most significant bits of their node IDs being equal to the node keeping the routing table and so on, with the last bucket containing k nodes that share no common node ID prefix with the node keeping the routing table. Nodes within a k-bucket are stored as a triple of node ID, IP address and UDP port, and are sorted by the least recently seen time. k is a replication parameter, selected such that it is very unlikely for any given k nodes to fail within 1 hour, with Kademlia suggesting to use $k = 20$.

Kademlia suggests representing the routing table as a binary tree, with leaves being the k-buckets, allocating k-buckets dynamically as needed, starting with a single k-bucket, always splitting only the k-bucket containing node's own ID into two, once it gets full.

2.1.3 RPCs

Kademlia defines four RPCs a node can call on another: PING, FIND NODE, STORE and FIND VALUE. PING tests if a node is online by requesting it to reply. FIND NODE requests a node to return k closest nodes to a given node ID. STORE requests a node to store a ⟨key, value⟩ pair. FIND VALUE requests a node to return the value for the given key if the node has it stored, or k closest nodes to the key otherwise. All RPCs include a random 160-bit ID that must be echoed back to prevent unsolicited responses.

2.1.4 k-bucket Management

Whenever a node receives any network message, it updates its routing table with the sending node's information by attempting to insert it into the appropriate k-bucket. If the sender is already in the k-bucket, the sender node is moved to the tail of it. If the k-bucket is not full, the sender node is added to the tail. If the k-bucket is full, the least recently seen node in the k-bucket is pinged. If it fails to reply, the pinged node is removed from the k-bucket and the sender node is added to the tail, otherwise the pinged node is moved to the tail and the sender node is not added. This makes Kademlia favor older nodes over newer ones, making its routing table more resilient as the nodes that have been online for a long time are more likely to keep being online than newly joined nodes. This also provides a hardening against a DoS attack of new nodes overwriting routing tables of other nodes.

To reduce the traffic caused by pinging the least recently seen node when a k-bucket is full, Kademlia suggests an optimization of delaying pinging the node until there is anything useful to be sent to it. Instead, when a k-bucket is full, the new node is placed into a replacement cache that will be used to replace stale nodes in the k-bucket the next time the node queries the k-bucket. Similarly to the k-bucket, the replacement cache is sorted by least recently seen time.

Nodes that fail to reply to RPCs must be marked as stale so that they could be replaced. To avoid removing nodes that fail to reply due to network congestion and UDP packets being dropped, a node is marked as stale only if it fails to reply to

```
Input: sender – node that has sent us a message
Procedure RoutingTableUpdateOnMessageReceived(sender):
    k_bucket = getKBucket(sender);
    if sender ∈ k_bucket then
        move sender to k_bucket tail;
    else if k_bucket not full then
        add sender to k_bucket tail;
    else
        replacement_cache = getKBucketReplacementCache(sender);
        if sender ∈ replacement_cache then
            move sender to replacement_cache tail;
        else if replacement_cache not full then
            add sender to replacement_cache tail;
        else
            remove replacement_cache head;
            add sender to replacement_cache tail;
        end
    end
```

Algorithm 1: Routing table update.

5 RPCs in a row, each with an exponentially increasing backoff interval. To avoid emptying k-buckets when our own node temporarily goes offline, stale nodes are removed only if the k-bucket is full and there are nodes in the replacement cache to replace the removed nodes.

Buckets are generally kept fresh by the RPCs being executed among the nodes. To avoid cases of buckets not being fresh, a node refreshes buckets into which it has not performed the node lookup procedure in the past hour. The refresh operation is described later in Section 2.1.6.

2.1.5 Node Lookup Procedure

The most important procedure in Kademlia is node lookup, which locates the k closest nodes to a given node ID. While RPCs are just a single request and a response back, node lookup is an iterative procedure doing multiple RPCs throughout its life-time. The procedure maintains a list of k closest nodes to the given node ID, which is initially populated with the closest nodes from the local routing table. From that list, the node picks a nodes that it has not sent the FIND NODE RPC to yet and sends it to all of them in parallel, merging the nodes it gets as a response into the list, keeping only k closest nodes in the list. The node keeps repeating this process, keeping no more than a RPCs in flight, until all the k closest nodes in the list have replied to a find node RPC, at which point the operation ends as the k closest nodes to the given node ID have been found. a is a system-wide concurrency parameter, with Kademlia suggesting 3 as the value. It trades increased bandwidth for faster lookups, as having $a > 1$ helps avoid wasting time waiting on the RPC to time out in case a node does not reply to the FIND NODE RPC.

2.1.6 Operations

Most operations in Kademlia are implemented in terms of the node lookup procedure, sometimes with slight changes.

Find Node Operation To find the k closest nodes to a given node ID, the node lookup procedure is used as it is.

```
Input: node_id, a
Result: k closest nodes to the node_id
Procedure NodeLookup(node_id):
    k_closest ⟵ GetKClosestNodes(node_id);
    SortByCloseness(k_closest, node_id);
    Loop
        foreach node in k_closest do
            if not sent FIND NODE RPC to node then
                send FIND NODE RPC to node;
            end
            if number of awaiting RPCs = a then
                break
            end
        end
        if number of awaiting RPCs = 0 then
            keep only the first k nodes of k_closest that have replied;
            return k_closest
        end
        await on any RPC reply;
        foreach replied RPC do
            add nodes from the reply to k_closest;
        end
        SortByCloseness(k_closest, node_id);
        keep the first k nodes of k_closest no matter the reply status and the
         first k nodes of k_closest that have replied;
    EndLoop
```

Algorithm 2: Node lookup.

Store Operation To store a ⟨key, value⟩ pair, a node finds the k closest nodes to the key using the node lookup procedure and sends them a STORE RPC.

Find Value Operation To find a value given a key, a node looks up the k closest nodes to the key using the lookup procedure, but instead of the procedure using the FIND NODE RPC it uses the FIND VALUE RPC, terminating as soon as a node replies with the value, caching the value on the last seen node that did not reply with the value by sending it a STORE RPC, just in case that ⟨key, value⟩ pair is popular

and this lookup is a hot path.

Refresh Operation To refresh a k-bucket, a node generates a random node ID within the k-bucket's range and does a node lookup for that ID, populating the k-bucket with fresh nodes.

2.1.7 Joining the Network

To join the overlay network, a node must know one or more other nodes that are part of the network. It adds those nodes in its routing table and does a node lookup for its own node ID, which it has randomly generated on the startup, populating the routing table with nodes closest to its node ID, as well as inserting itself into routing tables of the nodes closest to it. Finally, the node refreshes all k-buckets further away than its closest neighbor, populating k-buckets that are further from its node ID.

2.1.8 Data Management

When a node receives a STORE RPC, it stores the requested ⟨key, value⟩ pair data. The data expires 24 hours from the original publication. If the data needs to be stored for longer, the original publisher has to republish it every 24 hours.

To avoid over-caching that can be caused by many nodes looking up a popular ⟨key, value⟩ using the find value operation, resulting in the data being cached on more and more nodes, the expiration time of a data pair is exponentially inversely proportional to the number of nodes between the current node and the node whose

ID is closest to the key, which can be inferred from the routing table of the current node. This has the effect that nodes closest to the key will store the ⟨key, value⟩ for a longer time, while nodes further out will store it for exponentially shorter time.

To avoid losing the data due to nodes joining and leaving the network, the data must be republished every hour using the store operation. As an optimization, because the store operation sends a STORE RPC to the k closest nodes to the key, whenever a node receives a STORE RPC, it skips republishing the received ⟨key, value⟩ pair, assuming that the other $k - 1$ closest nodes received the STORE RPC too. This ensures that, unless nodes have the republishing time synchronized, only one node will republish the data, avoiding unnecessary traffic.

When a new node joins the network, it must store ⟨key, value⟩ pairs for which it is one of the k closest nodes. To accommodate this, any node learning of a new node issues it STORE RPCs for the relevant ⟨key, value⟩ pairs. As an optimization, a node sends a STORE RPC only if it is the closest node to the key, as otherwise the new node would receive a lot of redundant STORE RPCs.

2.2 Bitcoin Blockchain

Bitcoin [15] blockchain is one of the most popular peer-to-peer distributed public blockchain ledgers. It is the basis for Bitcoin cryptocurrency, the goal of which is to create a purely peer-to-peer digital currency free from any central authority. Bitcoin defines an electronic coin as a chain of digital signatures which represent

the transfer of a coin from one owner to the next. Such a coin is susceptible to double-spending: a coin owner can transfer the coin to one person, but then also transfer it to a different person, without them being aware that there was an earlier transfer. Double-spending is commonly resolved by relying on a trusted third party to verify all transactions, making sure a transaction for an already spent coin is not made. Bitcoin proposes blockchain – a distributed way to prevent the double-spending problem without having to rely on a trusted third party.

2.2.1 Blockchain Data Structure

Blockchain is a chain of blocks, with each block containing information on transactions made at the time the block was formed and a link to the previous block, linking the blocks all the way to the first block. Blockchain contains a list of all transactions that have ever happened, similarly to a conventional public ledger.

A block consists of a block header and a list of transactions. The block header contains a Merkle Tree root hash of all the transactions within the block, the hash of the previous block's header, timestamp of when the block was created and a nonce.

2.2.2 Operation

The only way to make sure that a coin has not been already spent in an earlier transaction is to be aware of all the transactions. In Bitcoin, all transactions are broadcast publicly and all of the peer-to-peer system participants agree on a single

history of the order in which the transactions were received, rejecting any double-spending transaction as invalid.

New transactions are broadcast to all nodes and each node works on creating a block. Once a node creates a block, it broadcasts it to all other nodes. Nodes express the acceptance of a block by creating the next block which links to the accepted one.

New transaction broadcasts do not have to reach all nodes and can be done on the best effort basis. As long as they reach enough nodes, they will eventually be included in a block. New block broadcasts do not have to reach all of the nodes either. If a node working on a block realizes that it is missing the previous block – it will request the block from other nodes.

2.2.3 Proof of work

In order for a block to be accepted by other nodes, it should create the longest chain of blocks, contain no invalid transactions, have a valid timestamp and its header hash must satisfy the proof of work. Bitcoin uses a variant of Hashcash [1] for the poof of work to make sure that a certain amount of computation was expended to generate a block. Specifically, a block header has to hash to a digest with a certain number of leading zero bits, such that when it is interpreted as a number, it is less than the current difficulty number [19]. Because hash digests are uniformly randomly distributed, requiring the hash to be under a certain value is more computationally expensive the smaller the required value is, allowing for a granular proof of work

difficulty control. A block header contains a nonce field that can be freely modified by a node creating the block in order to change the hash digest to satisfy the proof of work requirement. The proof of work difficulty is chosen such that each block is created every 10 minutes. Every 2016 blocks the network uses timestamps stored in block headers of those 2016 blocks to calculate the block creation rate and correct the proof of work difficulty to make sure that given the current block creating rate, the next 2016 blocks would be created in exactly two weeks, resulting in a single block being created every 10 minutes.

Proof of work, in combination with nodes accepting only the longest blockchain as valid, protects the blockchain against malicious modifications – as more blocks are added, it becomes more computationally expensive for an attacker to modify an earlier block, as that would change the modified block's header hash as well as header hashes of all blocks after that, requiring the attacker to re-do proof of work for all of them. Unless the attacker has more computation power than the rest of the network, this is not possible.

2.3 Attacks on Kademlia DHT

There are many ways in which a malicious Kademlia node can misbehave [9, 3, 12]. It can provide incorrect information to other nodes by return bogus node information on a FIND NODE RPC. It can also lie about the data: return bogus data on a FIND VALUE RPC or claim that a STORE RPC was successful when in fact no data was

stored. Nodes can collude and a user can run multiple nodes. All of this can be used to launch attacks and compromise the integrity of the overlay network.

In this section we provide an overview of attacks on Kademlia protocol. We omit discussion of application-dependent attacks such as the index pollution that is possible in some DHT-based file-sharing applications [22]. We also omit attacks on the underlying network level, e.g. IP address spoofing made possible by the use of a UDP-based protocol without a proper handshake [5].

2.3.1 Eclipse Attack

Eclipse attack aims to isolate a victim from the network or otherwise obfuscate victim's view of the network. It is a routing poisoning attack caused by an attacker providing information about malicious nodes to the victim. Routing table is kept fresh by receiving messages from other nodes: replies on sent RPCs and unsolicited RPC requests. An attacker can leverage this to send a victim RPCs from attacker-controlled nodes in an attempt to insert these nodes into victim's routing table. Nodes with IDs close to victim's are more likely to get added into victim's routing table, since the closest k-buckets are typically not full. Once in the routing table, the victim node might send RPCs to the malicious nodes, with the attacker controlling what the victim gets back. The attacker could always return other malicious nodes, or be more subtle by behaving correctly most of the time, hiding only certain nodes or keys.

A very powerful example of an Eclipse attack is a victim joining the network using

only attacker's nodes as the starting point. Because the victim does not have any honest node in their routing table, the malicious nodes are able to isolate the victim by returning information only about other malicious nodes controlled by the attacker, sealing the victim off in a network entirely separate from the actual network. This can allow the attacker to manipulate or trick the victim, analyze victim's traffic patterns and messages to learn some information about the victim, or man-in-the-middle the victim, proxying victim's requests to the real network.

It is also possible to apply the Eclipse attack to hide some data in DHT by creating nodes with IDs close to the target key, such that lookups for that key would be made though these nodes, and making the nodes return bogus or no data.

Kademlia favoring long-lived nodes provides resistance to the routing table poisoning, as long as the initial bootstrap nodes used to join the network are not malicious. Eclipse attack can be prevented if a node is not allowed to freely choose its node ID, e.g. by tying node ID to node's IP address [16] or ensuring that all node IDs are randomly generated.

2.3.2 Routing Attack

Routing attack aims to make a node performing the lookup procedure arrive at k closest nodes that are all attacker-controlled. It is a kind of Eclipse attack, but targeting the node lookup procedure specifically. When a honest node performs the node lookup procedure, looking for the k closest nodes to a node ID or a key, and

queries a malicious node for the k closest nodes, the malicious node can return other colluding nodes that are closer than what other honest nodes have returned, thus resulting in the node doing the lookup querying mostly the malicious nodes next. If done at an early iteration of the node lookup procedure, a malicious node could make the node query only malicious nodes next, totally isolating the node lookup from the network.

Routing attack can be partially mitigated by considering multiple disjoint paths when doing a node lookup [3]. As long as one of the paths does not contain a malicious node in it, the lookup will succeed. That is in contrast to the single path in Kademlia, where a single malicious node is enough to undermine the entire lookup.

2.3.3 Sybil Attack

Sybil attack aims to create a large number of malicious nodes. It exploits the fact that in a DHT anyone is free to generate an identity and there is no limit on how many identities one can generate. Sybil attack gives a single attacker a disproportionate presence in the network, allowing the attacker to subvert network's redundancy mechanisms. It can also be used to game various security mechanisms a DHT can have in place, as they generally assume that only a small fraction of nodes in the network are malicious. It is often used to launch other types of attacks that benefit from having multiple identities.

Sybil attack can be launched from a single computer or multiple computers po-

tentially spread around the globe, sometimes with a computer having multiple IP addresses, so it is particularly hard to detect a Sybil attack, especially if you consider that many honest nodes can have the same IP address by the virtue of being behind a NAT. Sybil attack can be prevented by having every node register their identity with a centralized authority that is able to detect Sybil attacks and revoke identity registrations. In a more distributed fashion, computational puzzles can be used to hinder the Sybil attack, under the assumption that it is hard for a single computer to solve puzzles for many identities within a limited time [3]. The disadvantage of computational puzzles is the difficulty selection for puzzles: a network might have a wide variety of devices on it with varying level of computing power, from weak mobile devices to powerful servers, so the difficulty should be selected to account for the slowest network participants, which makes the usefulness of computational puzzles questionable, especially when an attacker might be willing to splurge on a powerful computer for the attack.

2.3.4 Churn Attack

Churn attack aims to disrupt the network by having many nodes join and/or leave the network at the same time, causing network disruption and potentially some data stored in the network to be lost.

Kademlia is resistant against the Churn attack due to it favoring long-living nodes and having redundant data storage. Many new nodes joining the network will not

flush nodes' routing table as a node is unlikely to add new nodes to its routing table. Many nodes leaving the network will not flush other nodes' routing tables either, as a node would likely still have other nodes in its replacement cache, or other long-lived nodes in its routing table that it could use for bucket refresh. As for the data loss, unless an attacker can perform the Eclipse attack on the k-closest nodes to a target data, or the attacker controls a very big portion of the network as with the Sybil attack, a plain Churn attack is unlikely to result in the data being lost. Kademlia stores the data on k nodes, with the k parameter being selected specifically for the redundancy purposes – such that any k nodes are unlikely to fail within an hour of each other.

2.3.5 Denial of Service Attack

Denial of service (DoS) attack aims to use up resources of a node: bandwidth, processing power, storage, etc., making the node unable to participate in the network. For example, an attacker could ask a set of nodes to store a lot of bogus data, exhausting their storage capacity, making them unable to accept store requests from honest nodes until the stored data expires.

A node can blacklist other nodes that it thinks are misbehaving, however this should be done carefully as blacklisting nodes might skew node's view of the network and make some of the Kademlia operations less efficient, especially if it is one of the k closest nodes that is blacklisted.

Chapter 3

System Setup, Attack and Defense Techniques

This chapter introduces a Kademlia DHT based instant messaging system that allows for offline messages to be stored on the DHT for a duration of multiple days, a storage exhaustion DoS attack on the Kademlia DHT based instant messaging system preventing new offline messages from being stored, and defense techniques against the attack.

3.1 System Setup

We assume a Kademlia DHT based distributed instant messaging system. To join the DHT network, a user connects to other users that are already part of the network. Once connected, online users are able to find each other via the DHT and establish

a direct connection between each other in order to communicate. When a user goes offline and an online user wants to leave them an offline message, they store the offline message on the DHT, which the offline user would check the next time they go online. The DHT is not used for any message storage in the communication of online users, it is used only to facilitate the user discovery among online users and to store offline messages.

Each user, each DHT node, has a cryptographic key pair, the public key of which is used as the node ID. This allows nodes to sign their RPCs, providing integrity and non-repudiation, and encrypt offline messages, providing confidentiality.

The user discovery is done by users using their corresponding node IDs as user identifiers and performing Kademlia's find node operation to find the user they seek. For example, when Alice wants to communicate with Bob, she searches for Bob's node ID in the DHT, and once she finds it, she learns of the IP address and port of Bob, allowing her to connect to Bob and communicate with him directly. A user keeps a contact list of users they talk to, finding and connecting to them via this user discovery procedure on startup.

To store an offline message, the sender performs the Kademlia store operation for a key with the offline message as the value. The key is selected such that the recipient of the offline message would know to check for it. It could be either recipient's node ID or some other key that the sender and the recipient have agreed upon while being online. The offline message is stored for 3 days, after which it expires and

gets removed from the DHT. The sender can prolong this by requesting for the same offline message to be stored again, once the previously stored one expires, however doing so is discouraged under the assumption that the longer a user stays offline the more likely they are to remain offline. When the recipient goes online, they perform Kademlia's find value operation for the corresponding keys, checking if they received any messages while being offline.

Once the offline messages are received by the recipient there is no need for them to be stored on the DHT, so ideally they would be deleted right away, without having to wait for the messages to expire, but Kademlia does not support data deletion, it removes the data only on expiration. The data being replicated on multiple nodes, existing nodes going offline and new nodes joining in – all make the data deletion impossible. To delete data stored on the DHT, all nodes storing the data would need to receive the deletion request, however there is no simple way to find all of the nodes storing the data, a node might end up being missed and it would keep republishing the data, effectively reversing the deletion. Also, one of the nodes storing an offline message might happen to be offline when the delete request is sent, so it would miss the request and would continue republishing the message once it gets back online.

While it might be impossible to delete an offline message, the next best thing that could be done is to replace it with an empty message, reducing the storage burden of nodes storing the message. To prevent the received offline messages from continuing to be stored on the DHT, the recipient makes the nodes replace an offline

message with an empty message that uses the same key and has the same expiration time as the original offline message. With the empty message being signed by the recipient, and the original offline message addressed to the recipient, the nodes can use public key cryptography to verify if an offline message is allowed to be replaced by an empty message. This way the nodes store less data than if they continued storing the offline message, and it avoids the issue of nodes missing a request, as the empty message is stored for the same duration as the original offline message, so it would propagate among the nodes and nodes would not accept the original offline message when someone republishes it, with the empty offline message taking the precedence over a non-empty one.

In the case the recipient does not receive the offline messages, e.g. due to being offline long enough for the messages to expire, they would receive the messages when both the recipient and the sender are online. The sender keeps a local copy of the offline messages sent, so once they are both online, the sender can send the missed offline messages to the recipient.

An optimization could be made where a user that goes online does not check the DHT for offline messages right away, but instead only after discovering and connecting to all the users in their contact list, which is something they would do anyway, so that they get the missed offline messages directly from the users that are currently online, which is a very cheap operation, performing the more expensive operation of checking DHT for the remaining offline messages only for users that are currently

offline.

The system is resistant to Eclipse attacks, as nodes are not able to freely choose their node IDs, given the random nature of public keys. It is not, however, Eclipse-proof, as an attacker could pre-generate enough node IDs to launch an Eclipse attack, especially if the DHT is of small size. We assume that Sybil attacks are not possible, or at least are very hard to perform, with a single attacker not being able to create too many identities. This could be achieved, for example, using computational puzzles discussed in Section 2.3.3.

To simplify things, we ignore the privacy aspect of the system. For example, the user discovery is done using user IDs, with nodes IDs acting as user IDs, which is a very simple and effective way to do this, but it allows for nodes in the DHT to build social graphs of users as they can see that one user ID is trying to discover another user ID. This could be avoided by decoupling user IDs from node IDs and using onion routing for DHT lookups [10], making DHT lookups anonymous, hiding who communicates with whom.

For simplicity, the system is kept as close to Kademlia as possible, with node IDs being 160-bit numbers, the replication parameter $k = 20$ and the data (the offline message) republishing being done every hour. One of the differences with Kademlia is that the data is stored for 3 days instead of just 1, as a single day storage of offline messages is rather short. Another difference is that the data does not get cached on other nodes during Kademlia's find value operation – because an offline message has

just a single recipient, an offline message becoming popular among many nodes is not something that would happen, making caching pointless. Node ID is also a bit different from Kademlia – while it is still 160-bit, a node cannot choose its node ID freely as it is a public key now. The 160-bit key part of the ⟨key, value⟩ pair used to store offline messages, however, can still be chosen arbitrarily.

While the system is kept as close to Kademlia as possible in this thesis, in the real world application it might make sense to consider different constants for Kademlia, for example 256-bit node IDs that are Ed25519 [4] signing keys, which can also be used as Curve25519/X25519 [29] keys for authenticated encryption [6], providing 128-bits of security. The replication parameter k is another constant to re-consider – BitTorrent uses a Kademlia-based DHT with bucket size $k = 8$ [11], so using $k = 20$ might be excessive.

3.2 Attack

The described instant messaging system is vulnerable to a storage exhaustion DoS attack. An attacker can abuse the offline messaging feature by requesting different nodes to store bogus offline messages on the network until the network runs out of the capacity to store any new messages, denying new offline messages from being stored and wasting users' resources – disk space to store those bogus messages and bandwidth to periodically republish them – for days.

To launch the attack, an attacker joins the network by generating a key pair, using

the public key as their node ID and connecting to the nodes that are already part of the network. Once joined, the attacker continuously crawls the DHT to find more and more nodes. A simple way to crawl the DHT is to do breadth first search and iterative queries: querying the already known nodes to discover new nodes [24, 25], eventually finding all the nodes in the network. As the attacker crawls the network discovering new nodes, the attacker requests the nodes to store bogus offline messages until the nodes are no longer able to store them.

While such an attack is very comprehensive, as the attacker targets every node in the network, it is also very expensive for the attacker, as they need to expand a lot of bandwidth. They would need to expand as much bandwidth as the amount of data the network (all nodes) can hold, plus the DHT protocol and crawling overheads. The cost of the attack can be greatly reduced by targeting only a portion of the nodes in the network. Because nodes republish their data to other k nodes closest to the data key at regular intervals, it is enough for the attacker to target just a few nodes for k nodes to be affected. The targeted nodes would act as forwarders for the attacker, attacking other k-closest nodes on attacker's behalf. That way, the attacker only needs to target just a $\frac{1}{n}$ fraction of the network, where n is from 1 to k, greatly reducing the cost of the attack. Obviously, targeting the smallest number of nodes – only $\frac{1}{k}$ of the network, i.e. only one node in every k-closest nodes, is not very reliable, as that node needs to stay online long enough to republish the data on other nodes, so the bigger fraction of the network is targeted, the more reliable is the attack. This

optimization trades off attack reliability and the time (as it takes time for nodes to republish the data) for the amount of bandwidth required to successfully launch the attack.

The attack could be optimized a bit further by optimizing the crawling algorithm. When doing the iterative breadth first search crawl, the attacker asks known nodes about nodes closest to some ID. The way to pick that ID varies by implementation, for example, it could be randomly generated, or continuously increasing, or derived by observing the gaps in the key space of the list of known nodes. A better, more deliberate way of picking the lookup ID can improve the performance of the crawl. A better ID can be picked if the attacker can estimate the density of the network – how far node IDs are spread out apart in the key space. The attacker can do so by measuring the amount of peers in small portions of the network, results of which can be extrapolated to the whole network as node IDs are uniformly randomly distributed among the key space [28]. Knowing how far node IDs are spread out on average, the attacker can estimate which node ID to do a lookup for such that the lookup result would yield as few of the already known nodes as possible, while finding all the nodes the attacker wants to target. Doing a node lookup and getting back as a result nodes that are already known to the attacker is wasteful and redundant, which is the inefficiency that this deliberate picking of node ID prevents. If the attacker wants to target all the nodes, then using this optimization the attacker will be able to crawl the entire DHT picking lookup IDs spaced apart just enough to avoid getting too many

of the known nodes as the result of the lookup, saving on bandwidth. However, if the attacker wants to target only a few nodes per each k-closest nodes group instead, relying on the targeted nodes to forward the attack to the nodes within that k-closest group of nodes, the lookup ID could be picked such that the crawling would be done with some omissions, avoiding having to crawl all of the nodes, saving on bandwidth. There is also another benefit to doing this optimization: knowing the density of the key space, the attacker can estimate the size of the network, the total number of nodes in it [28], which would give the attacker a rough estimate of the amount of resources needed to attack the network before even starting the attack.

So far we have mentioned that an attacker, instead of fully exhausting the storage of all nodes, could fully exhaust the storage of a few nodes out of k-closest nodes, which would then republish all the bogus offline messages on the k-closest nodes on their own, but it does not necessarily has to be done this way and might be too obvious of an attack. Instead, the attack could be spread out among multiple peers. For example, instead of fully exhausting the storage of n peers, the attacker could exhaust $\frac{1}{n}$ portion of the storage of k peers, where n is from 1 to k. This is equivalent in cost as it requires the same amount of bandwidth, but it would make it less obvious to the peers that they are under an attack, as a node might not suspect that something is wrong when a peer asks it to store just a fraction of its storage capacity. Only after the nodes republish would they notice that they have ran out of space.

Furthermore, an attacker could perform the attack from several different machines

under different identities. While we assume that Sybil attacks are not possible or hard to perform, that concerns running multiple nodes on a single machine, nothing stops an attacker from renting multiple servers to perform the attack from. In fact, an attacker would likely want to perform the attack from multiple servers because the attack is primarily limited by attacker's network speed and allotted bandwidth, as both the DHT crawling and offline message sending are highly parallelizable.

3.2.1 Cost Estimation

To illustrate the feasibility of the storage exhaustion DoS attack, let us estimate the upper and lower bounds of the egress bandwidth required to storage exhaust a system with $1,000,000$ online users, each able to store 100 MiB of offline messages, with each offline message being at most 1024 *bytes*. Such a network has $1,000,000 \cdot 100 = 100,000,000\ MiB \approx 95.37\ TiB$ of raw storage capacity, but because the same offline message is duplicated on $k = 20$ nodes, it can store only $95.37 \div 20 = 4.77\ TiB$ of offline messages. For the sake of the estimation, we assume the worst case of there being no offline messages stored on the system and no one but the attacker storing them, i.e. the attacker must exhaust all the storage on their own.

The upper bound is the attacker crawling the entire DHT and attacking every node they find. In such case, the attacker would need to expand bandwidth on sending offline messages equal to the entire storage capacity of the system – $95.37\ TiB$, plus the DHT protocol overhead, i.e. the combined overhead of looking up nodes as part

of crawling and the overhead of STORE RPCs on top of the offline message being sent.

To estimate the crawling overhead, let us assume the worst case scenario that from every node lookup the attacker learns of just 1 new node among all the nodes returned, i.e. all other nodes are already known. While unrealistic, that would require the attacker to do $1,000,000$ node lookups to crawl the entire network. In Kademlia, a node lookup takes at most $log_2 n$ hops to perform [14], meaning there are at most $\log_2(1,000,000) \approx 20$ hops, i.e. 20 FIND NODE RPCs, needed per lookup. A FIND NODE RPC request contains 3 IDs: the node ID of a node being looked for, the node ID of a sender and a random RPC ID that must be echoed back by the receiver. That is $3 \cdot 160/8 = 60$ *bytes* of data. Let us double that to account for any data encoding information, encryption, and other overhead – 128 *bytes* per a FIND NODE RPC request. That is $128 \cdot 20 = 2,560$ *bytes* for a whole node lookup procedure with 20 hops, and $1,000,000 \cdot 2,560 \approx 2.38\ GiB \approx 0.0023\ TiB$ to have it repeated for every single node, to crawl the entire network.

To estimate the STORE RPC overhead, let us assume that offline messages are limited to 1024 *bytes* per message, which would allow storing 1024 1-byte or 256 4-byte UTF-8 characters, and that each message has 256 *byte* overhead for encoding format, included message timestamps, encryption, etc. That results in additional $\frac{1}{4}$ overhead in STORE RPC on top of the offline message content being sent, i.e. $95.37 \cdot \frac{1}{4} \approx 23.84\ TiB$.

The upper bound of the egress traffic adds up to $95.37 + 0.0023 + 23.84 = 119.21\ TiB$.

The lower bound of the egress traffic is estimated similarly, adding up to $4.77 + 0.0023 + 1.19 = 5.96\ TiB$. That is for the case of an attacker storing the data on just 1 node out of every k-closest, while still crawling the entire DHT as the bandwidth cost of that is negligible.

With commodity virtual servers offering 1 *Gbps* uplink ($\approx 120\ MiB/s$) connections and the attack being limited by the network speed, it will take the attacker between $5.96 \cdot 1024^2 \div 120 = 52,079\ seconds \approx 14.47\ hours$ to $119.21 \cdot 1024^2 \div 120 = 1,041,673\ seconds \approx 289.35\ hours \approx 12.06\ days$ to perform the attack from a single server. Performing the attack from multiple servers reduces the attack time greatly.

To estimate the cost of the attack in USD, let us use DigitalOcean virtual server provider as an example. It provides 5 $USD/month$ virtual servers with 1 *Gbps* uplink and 1 TiB bandwidth limit [7]. At the lower bound estimate of 5.96 TiB the attack would use 6 such servers, costing $6 \cdot 5 = 30\ USD$ and taking $\frac{14.47}{6} \approx 2.41\ hours$. At the upper bound estimate of 119.21 TiB the attack would use 120 such servers, costing $120 \cdot 5 = 600\ USD$ and taking $\frac{289.35}{120} \approx 2.41\ hours$.

The cost of the attack in USD can be reduced by using a virtual server provider with higher bandwidth/USD value, allowing for reducing the number of servers used, increasing the time it takes to launch the attack as a trade-off. With server providers offering unlimited bandwidth it is possible to continuously DoS the system, render-

ing the offline messaging inoperable for indefinite amount of time, instead of DoS attacking it just once and denying the service for 3 *days*. For example, Hetzner offers dedicated servers with 1 *Gbps* uplink and unlimited bandwidth for as low as 34 $EUR/month$ [8], or approximately 37 $USD/month$. Using 5 of such servers, it is possible to perform the attack at the upper bound estimate of 119.21 TiB for 185 USD/month in $\frac{289.35}{5} \approx 57.87$ $hours \approx 2.41$ *days*, and because the bandwidth is unlimited, keep it running afterwards for as long as the attacker desires: days, weeks, months.

With this rough estimate we have shown that it is very feasible and not that expensive for an attacker to exhaust the storage capacity of the network of $1,000,000$ nodes for 3 *days*, costing the attacker anywhere from roughly 30 to 600 USD and taking under 3 *hours*, or continuously exhaust it for 185 $USD/month$.

There are some nuances to this estimation that we gloss over for the sake of simplicity. For example, a server provider might monitor its users for spam or any other sort of malicious activity. Utilizing all available network bandwidth non-stop by continuously sending UDP datagrams to multitude of different IP addresses might trigger an automatic abuse activity detection alert, resulting in the service suspension. It might also be hard to rent 120 servers without some extended verification or a big upfront payment, as cloud providers typically limit the number of servers one can use to avoid abuse. As an alternative, a smaller number of servers with increased bandwidth limit could be used, e.g. instead of using 120 servers with 1 TiB bandwidth

limit, 30 servers with 4 TiB limit could be used, but that will also slow down the attack 4 times as the uplink speed per server typically remains the same, so the combined uplink speed of 30 servers would be 4 times smaller than that of 120 servers. Another thing to note is that, unlike dedicated servers, virtual servers often have uplink speed shared with other tenants on the same physical server, so not the entire 1 *Gbps* might be available to them.

3.3 Defense Techniques

To defend against the storage exhaustion attack, honest nodes should be able to identify attacker-controlled nodes and deny them the service of storing offline messages. Taking a closer look at the attacker, they are able to attack from multiple servers, with each server running a single or a few nodes, they have limited number of servers/IP addresses as those are expensive, they are able to change node IDs of their nodes at will, they can make the attack less obvious by making the nodes attack the network in an uniformly random manner and they have to finish the attack in under 3 days for it to be fully effective. Based on that, attacker nodes can be identified by looking for nodes sending disproportionally more offline messages than the rest of the network, as that is the only thing that sets them apart from honest nodes. However, because they can easily change their node IDs, they should be uniquely identified not by their node ID but rather by their IP address, which the attacker is not able to change as easily. We present two defense techniques against the attack: a DHT-based

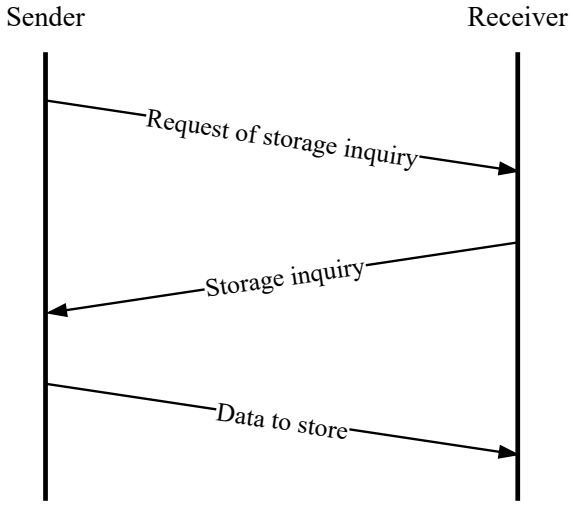

Figure 3.1: Message exchange during SECURE STORE RPC.

technique and a blockchain-based technique.

Both of the proposed techniques aim to limit how many offline messages an IP address can send by keeping track of the already sent messages by an IP address, with the difference being that one keeps track of that information on the DHT, while another on a blockchain, resulting in two different defense techniques.

As the validity of an IP addresses is paramount in defending against this attack, both techniques add a new RPC to the DHT called SECURE STORE RPC, which is an IP address forgery resistant version of STORE RPC. Instead of the regular request-response data flow, where it is not possible to tell if the request comes from the IP address it claims to be coming from, it does request-inquiry-response data flow – a node wanting to store some data sends another node a request for a store inquiry, then the receiving node replies back asking the first node about what it wants to store – the store inquiry, to which the first node finally responds with the data. Figure 3.1

shows this message exchange. The initial message may include some metadata about the data that the sender wants to store, e.g. its size, to which the receiving node might respond not with an inquiry but with a storage denial, e.g. because it does not have enough free space to store the data.

3.3.1 DHT-based Technique

In the DHT-based technique, we propose keeping track of how many offline messages were sent from an IP address by storing that information on the nodes closest to the hash of the IP address, the IP-closest nodes. Nodes receiving offline message store requests are able to check that information and make a decision on whether an offline message should be stored or not. If an IP address sends too many offline messages, nodes will stop accepting offline messages from that IP address. It can be thought of as an IP address reputation – the more messages are sent from an IP address, the spammier the nodes sharing that IP address are considered to be. However, in practice, the reputation is rather binary than a gradation – an IP address is not considered spammy until it reaches the limit of offline messages it is allowed to send. Note that instead of limiting the number of offline messages an IP address can send, we could limit it by how much data it can send, by counting the sizes of offline messages in bytes.

To keep track of offline messages sent, we add a new type of data that can be stored on the DHT – an offline message receipt. A sender generates an offline message and

the receipt together, with the receipt containing metadata on the sender and the offline message being sent. An offline message receipt is a combination of the sender's signing key (node ID public key), sender's IP address, timestamp of when the offline message was sent that matches the timestamp included with the offline message, hash of the key the offline message is to be stored under, the size of the offline message data in bytes and a hash of the offline message, all signed by the initial sender of the offline message. It serves as a proof of the sender having sent the offline message. Each node has two separate data stores: each node stores offline messages and the corresponding receipts – for keys closest to node's node ID, and, completely separately from the offline messages and corresponding receipts, each node also stores receipts used for the IP reputation – for IP addresses the hash of which is the closest to node's node ID. Because the receipts are used to limit the number of offline messages an IP address may send, which in turn also limits the number of receipts an IP address may send as it matches the number of offline messages, the receipts are not susceptible to the storage exhaustion attack and thus do not require any extra spam prevention techniques.

An attacker might try to manipulate the reputation of their IP addresses by generating k node IDs closest to an attacker controlled IP address, thus owning the IP-closest nodes which keep track of how many messages the attacker's IP address has sent. We modify the instant messaging system to use a hash of the IP address as the node ID prefix [16], restricting attacker's ability to choose their node ID even

further than it already is, making this attack harder to perform.

While the system being Sybil-resistant prevents an attacker from running multiple nodes on a single machine, the attacker could instead run multiple machines behind a single IP address, allowing multiple nodes to run under a single IP address, sharing the same node ID prefix, making it easier to generate node IDs that are close to each other. To prevent the attacker from doing so, whenever a node sends requests to a group of nodes, if more than one node has the same IP address, only the node with the highest node ID for that IP address is considered.

Storage Process

The process of a node storing its offline message is as following.

Step 1 The node creates an offline message receipt for the offline message it wants to store. The node then stores the receipt on its IP-closest nodes, using Kademlia's store operation with SECURE STORE RPC. The IP-closest nodes verify the receipt: that the receipt is indeed coming from the same IP address as written in the receipt, that the timestamp is recent enough, that the signing key matches sender's node ID key and that the receipt is signed with sender's node ID key. If everything verifies and the number of messages sent from the IP address within 3 days does not exceed the limit, which can be found by counting the number of stored receipts with the matching IP address, the IP-closest nodes store the receipt, counting it towards sent offline messages of that IP address. Otherwise, the receipt does not get stored. The

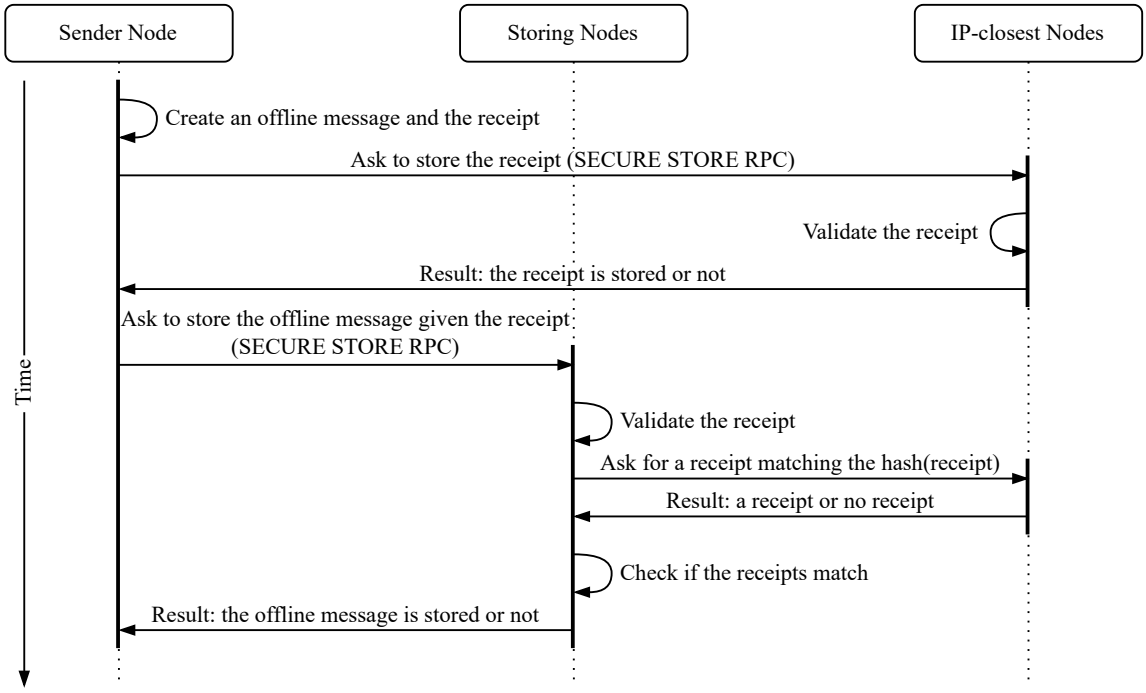

Figure 3.2: Storage process of the DHT-based technique.

IP-closest nodes then reply back to the node with the result of the operation, allowing it to abort the storage process early in case the receipt did not get stored.

Step 2 The node proceeds with storing the offline message. It does Kademlia's store operation with SECURE STORE RPC, storing both the offline message and the receipt. The nodes receiving the store request verify the receipt: that the receipt is indeed coming from the same IP address as written in the receipt, that the timestamp is recent enough and matched the one included in the offline message, that the signing key matches sender's node ID key, that the receipt is signed with sender's node ID key and that the hashes of offline message's key and the offline message itself included in the receipt match those of the offline message. If the verification fails, then the

nodes delete the offline message and the receipt, stopping processing the request. Otherwise, the nodes keep the offline message and the receipt, but do not mark the offline message as being stored yet, preventing it from being sent to other nodes or being republished.

Step 3 The storing nodes send the hash of the receipt received in Step 2 to sender's IP-closest nodes. The IP-closest nodes then check if they have the matching receipt, which they should have received in Step 1. If they find it, they reply back with the original sender's signature of the receipt as a proof. Otherwise, they reply that they do not have such receipt, which can mean that either they never received such a receipt, or received it but did not store as sender's IP address has exceeded the limit of offline messages sent.

Step 4 The storing nodes process the replies received form sender's IP-closest nodes. Replies with a receipt signature that does not match the received receipt signature in Step 2 are removed from the consideration. Out of the remaining replies, the majority is considered. If the majority of the replies did not find the receipt, then the storing nodes delete the offline message and the receipt, stopping processing the store request. Otherwise, the nodes mark the offline message as being stored, storing the receipt along with it. The storing nodes reply back to the sender node with the result of the operation.

Republish Process

At some point after an offline message is stored, the offline message, as well as the receipts associated with it, need to be republished. In Kademlia a node republishing data is not different from a node storing data, however in the case of this DHT-based technique, the modified storage process is unsuited for republishing – the IP address of a node republishing its stored offline messages would not match the IP addresses in the corresponding receipts and we do not want the republished data to count against republishing node's offline message sending limit. As such, there needs to be a separate republishing process defined.

Offline Message and Receipt Republishing Offline messages and the associated receipts are republished to the k-closest nodes to the offline message key using Kademlia's store operation. Whenever a node receives a republished offline message that it does not yet store, it checks if the offline message is stored by a majority of the nodes responsible for storing this offline message. Typically those would be the nodes closest to the current node, making this operation rather efficient. The node sends the hash of the receipt to the k-closest nodes of the offline message key, which then check if they have the matching receipt. If they find it, they reply back with the receipt's signature as a proof. Otherwise, they reply that they do not have such receipt. The node processes the replies, removing the ones with a receipt signature that does not match the receipt. Out of the remaining replies, a majority is considered. If the majority of the replies did not find the receipt, then the node does not store

the offline message and the receipt. Otherwise, the node marks the offline message as being stored, storing the receipt along with it.

IP-closest Receipt Republishing Republishing of receipts received as part of being an IP-closest node is done similarly to republishing of receipts received as part of offline messages, additionally accounting for the offline message limit. The receipts are republished to the IP-closest nodes of the hash of the IP address included in the receipt using Kademlia's store operation. Whenever a node receives a republished receipt that it does not yet store, it counts the local receipts to check if the number of offline messages sent by that IP address has reached the limit. If it did, and the received receipt's timestamp is newer than any local receipt for that IP address, then the received receipt does not get stored. Otherwise, the node checks if the receipt is stored by a majority of the nodes responsible for storing that receipt. Typically those would be the nodes closest to the current node, making this operation rather efficient. The node sends the hash of the receipt to the IP-closest nodes of the IP address included in the receipt, which then check if they have the matching receipt. If they find it, they reply back with the receipt's signature as a proof. Otherwise, they reply that they do not have such receipt. The node processes the replies, removing the ones with a receipt signature that does not match the receipt. Out of the remaining replies, a majority is considered. If the majority of the replies did not find the receipt, then the node does not store the receipt. Otherwise, the receipt gets stored, and if the IP address in that receipt has already exceeded the limit of sent offline messages, the

newest receipt for that IP address, according to the timestamp within the receipt, gets discarded, so that the receipts stay within the limit. Nodes effectively favor storing receipts of older offline messages.

Optionally, nodes could periodically check their stored offline messages against a majority of IP-closest nodes, to make sure those offline messages are still counted against senders' IP addresses, i.e. that their receipts were not removed by a majority of IP-closest nodes, removing the associated offline messages if they were. However, this is not strictly necessary, would generate a lot of traffic and would allow attacker-controlled majority of IP-closest nodes to delete previously sent offline messages.

Security Considerations

This defense technique successfully prevents the described storage exhaustion attack by limiting how many offline messages a node can send, however, like any purely peer-to-peer technique without a central trusted authority, it can be bypassed by an attacker given enough resources. In this section we discuss the security of the system using this technique: ways an attacker may try to bypass the technique to achieve the original goal of the storage exhaustion attack, ways an attacker can misbehave and abuse the defense technique, how secure the system using this defense technique is against the churn, privacy aspects of this technique and considerations for multiple users sharing the same IP address. We show that this defense technique makes it harder to perform the storage exhaustion attack, requiring more resources from the attacker than on a system not using the defense technique.

k-closest Nodes Majority The defense technique relies heavily on an attacker not being able to generate $\frac{k}{2}+1$, i.e. a majority, of node IDs that are closest to some key. In other words, it relies on a majority of IP-closest nodes for an IP address and a majority of storage nodes for an offline message key being honest nodes. While an attacker could not pick node IDs freely before this technique, due to them being public keys, the attacker could keep generating them until eventually finding $\frac{k}{2}+1$ node IDs that would be the closest to some key, given the key space density of the DHT at the time. This defense technique restricts attacker's ability to generate node IDs even further by requiring the prefix of a node ID to be a hash of node's IP address, meaning that an attacker would need to go through many IP addresses to find $\frac{k}{2}+1$ of them that hash to the same or a close enough digest, in addition to generating key pairs such that the rest of the node ID is also as close as possible to some DHT key. Going through IPs makes the attack a lot more expensive to perform, as virtual server providers typically do not allow its users to choose the IP address freely, they typically assign an unused IP address at the virtual server creation time, so the attacker would need to keep re-creating virtual servers, keeping only the ones with IP addresses that hash the closest to some target key, or instead of having a target key, create many virtual servers at the same time and figure out which key would they hash the closest to and target that one. Aside from being expensive in the sense of costing more money, this restriction also slows down attacker's node ID generation rate considerably, as it takes some time for a virtual server to be created

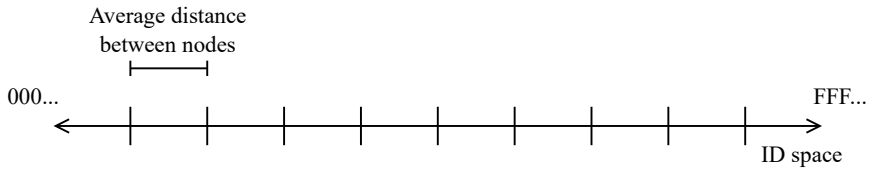

Figure 3.3: Uniformly randomly distributed node IDs.

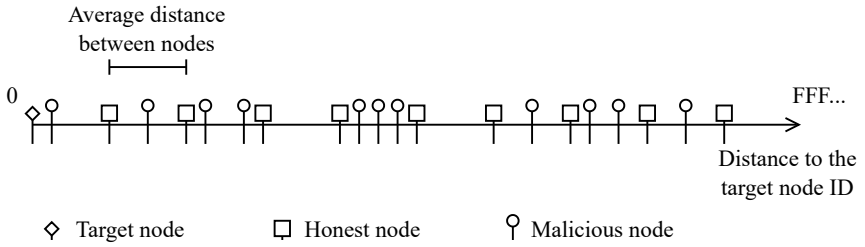

Figure 3.4: Attacker controlling the majority of nodes with node IDs closest to a target node ID (key).

and virtual server providers limit how many simultaneous virtual servers a user can have. Granted, if an attacker owns their own IP block and does not rely on a virtual server provider to provide the IP address, this may not slow down the attacker, but owning an IP block is still expensive and unless that is a large block, the attacker would have limited choice for node ID prefixes.

Probability of Generating a Majority of Non-IP-prefixed Node IDs Let us estimate the probability of an attacker generating a majority of node IDs closest to a key using regular, non-IP-prefixed node IDs. Let b be the number of bits in a node ID and N be the number of nodes in the DHT network. Given that node IDs can take on a value from 0 to 2^b, there are N nodes with node IDs in the DHT and the node IDs are uniformly randomly distributed – each of the N nodes in the DHT

should be on average $\frac{2^b}{N}$ IDs apart in the ID space, as illustrated in Figure 3.3. The attacker-generated majority of node IDs, i.e. $\frac{k}{2}+1$ node IDs, should be no further to the target key than the $\left(\frac{k}{2}-1\right)$-closest non-attacker-controlled node ID to the key. In other words, they should be within $\frac{2^b}{N}\left(\frac{k}{2}-1\right)$ distance from the target key, as illustrated in Figure 3.4. Note that as mentioned in Section 2.1.1, because Kademlia uses XOR distance, which is unidirectional, there exists exactly one node y such that $d(x,y)=\Delta$, where $d(x,y)$ is a distance function, in contrast to the two-dimensional Euclidean distance, where exactly two nodes, y and z, could be equal space apart: $d(x,y)=\Delta$ and $d(x,z)=-\Delta$. So once a target key is chosen, there are only $\left(\frac{2^b}{N}\left(\frac{k}{2}-1\right)\right)-\frac{k}{2}$ possible unused node IDs that the attacker could generate to be one of a majority of node IDs closest to the key. The $-\frac{k}{2}$ part is due to there already being $\frac{k}{2}$ non-attacker-controlled IDs in that interval. After generating the first such node ID, there would be $\left(\frac{2^b}{N}\left(\frac{k}{2}-1\right)\right)-\frac{k}{2}-1$ unused node IDs left in that interval that the attacker could generate, and so on.

The probability of an attacker randomly generating $\frac{k}{2}+1$ node IDs to a target key is:

$$\prod_{i=0}^{\frac{k}{2}} \frac{\frac{2^b}{N}\left(\frac{k}{2}-1\right)-\frac{k}{2}-i}{2^b} \tag{3.1}$$

If the target key is an attacker-controlled node ID, then the attacker has to gen-

erate one node ID less, so the last iteration of the product is unnecessary:

$$\prod_{i=0}^{\frac{k}{2}-1} \frac{\frac{2^b}{N}\left(\frac{k}{2}-1\right)-\frac{k}{2}-i}{2^b} \tag{3.2}$$

Given $b = 160$, $k = 20$ and $N = 1,000,000$, the probability of an attacker controlling a majority node IDs for a hash of a target IP address via randomly generating node IDs is $\prod_{i=0}^{10} \frac{\frac{2^{160}}{1,000,000}\cdot 9-10-i}{2^{160}} \approx 3.14 \times 10^{-56}$. In other words, the attacker would need to try 3.18×10^{55} different node IDs before finding a majority closest node IDs to the IP address. If an attacker is able to generate a node ID every nanosecond, that will take the attacker approximately 7.2×10^{28} ages of the Universe (14 billion years) to complete the attack.

Note that 3.18×10^{55} node IDs is a lot more node IDs than are possible in the 160-bit space: $log2\,(3.18 \times 10^{55}) \approx 184.38$, so that is a bit over 2^{184} node IDs vs 2^{160}. The attacker would have a better chance if instead of randomly generating node IDs they did so systematically, for example, starting with a private key that is all zeros and incrementing it by one until it is all Fs in hexadecimal. The resulting public key and thus node IDs they generate would still be uniformly randomly distributed, however the attacker would be limited to generating 2^{160} node IDs and there would be no duplicates. For the comparison, by generating a bit over 2^{184} random node IDs an attacker would be able to control a majority of k-closest nodes to some set target key, when by generating a lot less node IDs systematically, 2^{160}, they could control any node ID they want, not just a majority of k-closest nodes to some node ID, but

all of k-closest nodes to any node ID.

Let us calculate how many node IDs an attacker would need to generate to get a majority of node IDs closest to a target key if they generated them sequentially instead of randomly. Each time an attacker generates a node ID, the number of remaining possible node IDs decreases, e.g. $2^{160} - 1$, then $2^{160} - 2$, and so on. First we need to find how many node IDs the attacker needs to generate to find the first of a majority of node IDs closest to a target key, which can be done by solving the following summation for the n_1:

$$\sum_{i=0}^{n_1} \frac{\frac{2^b}{N}\left(\frac{k}{2} - 1\right) - \frac{k}{2}}{2^b - i} = 1 \tag{3.3}$$

To find how many nodes the attacker has to generate to find the second node ID of a majority of node IDs closest to a target key, we need to solve the following summation for n_2:

$$\sum_{i=0}^{n_2} \frac{\frac{2^b}{N}\left(\frac{k}{2} - 1\right) - \frac{k}{2} - 1}{2^b - i - n_1} = 1 \tag{3.4}$$

and so on, with the last one, $n_{\frac{k}{2}+1}$, being:

$$\sum_{i=0}^{n_{\frac{k}{2}+1}} \frac{\frac{2^b}{N}\left(\frac{k}{2} - 1\right) - \frac{k}{2} - \frac{k}{2}}{2^b - i - \sum_{j=1}^{\frac{k}{2}} n_j} = 1 \tag{3.5}$$

Adding up all n, i.e. $\sum_{j=1}^{\frac{k}{2}+1} n_j$, would tell us how many node IDs an attacker would need to generate to find a majority of node IDs closest to a target key. Assuming

$\frac{2^b}{N}\left(\frac{k}{2}-1\right)-\frac{k}{2} \gg \frac{k}{2}$, i.e. assuming that the difference in the numerator between each summation is negligible, instead of solving the $\frac{k}{2}+1$ summations for their iterators, the result can be approximated by solving just one summation for n:

$$\sum_{i=0}^{n} \frac{\frac{2^b}{N}\left(\frac{k}{2}-1\right)-\frac{k}{2}}{2^b-i} = \frac{k}{2}+1 \tag{3.6}$$

Given $b = 160$, $k = 20$ and $N = 1,000,000$ and solving the equation 3.6 for n, we get $\sum_{i=0}^{n} \frac{\frac{2^{160}}{1,000,000}\cdot 9-10}{2^{160}-i} = 11$ or $n \approx 1,222,221$, i.e. an attacker has to go through approximately $1,222,221$ node IDs to find a majority node IDs for a key if they were to sequentially try them out. $log2\,(1,222,221) \approx 20.22$, so that is about 2^{20} node ID tries, in contrast to 2^{184} needed when randomly generating private keys corresponding to node IDs (private keys).

Probability of Generating a Majority of IP-prefixed Node IDs In the case of node IDs prefixed with a hash of an IP address, to generate a node ID closest to a key, an attacker would first need to make the prefix part of the node ID be closest to a target key by finding an IP address that creates the closest IP hash prefix, and then finding the rest of the node ID that would make it closest to the target key. The same equations apply as for non-IP-prefixed node ID, just with b set to the size of the IP prefix hash in bits, p, for calculating the prefix part, and b set to the remaining of the node ID bits for calculating for the rest of the node ID.

With Node ID prefix bits being the most significant bits of a node ID, in the case

of $N < 2^p$, i.e. with every node having mostly unique node ID prefixes with little to none collisions due to the pigeonhole principle – N pigeons and 2^p holes – it is enough for the attacker to find an IP address that hashes close enough to a target key's prefix bits in order to be able to generate a majority k-closest node ID to the key, as the rest of the node ID is a lot less significant and could be generated easily.

Given a 32 *bit* prefix, i.e. $b = p = 32$, $k = 20$ and $N = 1,000,000$, using Equation 3.6, the attacker would need to go through $\sum_{i=0}^{n} \frac{\frac{2^{32}}{1,000,000} \cdot 9 - 10}{2^{32} - i} = 11$ or $n \approx 1,222,364$ unique IP addresses to find a majority of node IDs closest to a key by the node ID prefix. This is similar to $1,222,221$ of node IDs the attacker would need to go though when systematically trying them out, however, it is not as easy as incrementing the private key, as it requires that attacker to be in control of the corresponding IP addresses. This is equivalent to an attacker owning /11 IPv4 or /43 IPv6 blocks, and the attack only increases in difficulty as N, the number of nodes in the system, increases, requiring the attacker to own more IPs. While technically an attacker does not have to own an entire IP block, they only need to control a few specific IP addresses within it, e.g. they could hash all the IPs addresses, find which IPs addresses would result in the closest node ID prefixes to a target key and gain access to only those IPs addresses, but there is no practical way of gaining access to specific IP addresses, which is why we consider the attacker owning an IP block.

Majority Attacks An attacker controlling a majority of IP-closest nodes for some IP address can fully control the reputation of that IP address, circumventing the

defense technique.

An attacker controlling a majority of the IP-closest nodes for some IP address could misbehave by always storing receipts, even if the IP address has exceeded its limit of offline messages sent, allowing nodes with IP addresses these IP-closest nodes are responsible for to send as many offline messages as they want. This can be used to spam the network with offline messages, making the storage exhaustion attack possible again. For such an attack to be effective, however, an attacker would need to control not just a majority of the IP-closest nodes for some IP address, but a majority of the IP-closest nodes specifically for an IP address the attacker controls, which increases the difficulty of the attack. Furthermore, unless the attacker has enough bandwidth and high enough uplink speed to perform the storage exhaustion attack from a single IP address or a virtual server, the attacker would need to perform the attack from multiple IP addresses, which would require the attacker to control the majority of the IP-closest nodes for those IP addresses too, raising the attack difficulty even more. In addition to that, the attacker-controlled IP-closest nodes would need to have enough storage capacity to store the receipts for all of the spammed offline messages the attacker sends, as well as enough bandwidth to keep replying with them, as the storing nodes would ask for receipt signatures when receiving the spammed offline messages. Compared to the original storage exhaustion attack, this storage exhaustion attack is harder to perform.

An attacker controlling a majority of the IP-closest nodes for some IP address

could misbehave by not storing any receipts, preventing the users with IP addresses these IP-closest nodes are responsible for from being able to store any offline messages. This is a very targeted attack as it affects only the IP addresses the IP-closest nodes are responsible for. In order for it to be extended to cause storage exhaustion of the network as a whole, the attacker would need to control a majority of nodes in all IP-closest groups of nodes in the network, which would require the attacker nodes to account for over half of all the nodes in the network.

Another way an attacker controlling a majority of IP-closest nodes for some IP address could misbehave is by generating bogus valid receipts for the IP addresses the IP-closest nodes are responsible for and republishing them. Since during the republishing process there is no requirement that a receipt should come from the original sender's IP address that is written in the receipt, attacker-controlled nodes can create receipts with any IP addresses they want, including any other fields: signing key, timestamp, etc. and the honest IP-closest minority nodes would trust those republished receipts, storing them, as they are stored by the attacker controlling majority. The attacker can utilize this to flush the honest IP-closest minority nodes' receipts by republishing receipts for IP addresses these nodes already store the receipts for, marking them as being sent almost 3 days ago, making the honest nodes replace their stored receipts with these older ones that would expire in several seconds or minutes. Once the receipts expire and honest nodes' receipt store is flushed, the attacker could then fill up the honest nodes' receipt store for the maximum duration

by republish receipts with the current timestamp, until the limit of offline messages for the IP addresses included in those receipts is reached. This would effectively make the honest nodes not store any receipts for those IP addresses for 3 days. This attack might seem a little pointless, given the attacker already holds all the power by controlling a majority of the IP-closest nodes, so it should not matter what a minority of the nodes do, however, this could be used by the attacker as an anti-churn measure – due to the churn, an attacker previously controlling a majority of IP-closest nodes might find themselves controlling a minority of them, so having the honest minority nodes agree with attacker's minority might form a majority again, and while the attacker would not control a majority, this might be good enough for whatever they are trying to achieve. This attack is a very targeted attack as it affects only the IP addresses the IP-closest nodes are responsible for. In order for it to be extended to cause storage exhaustion of the network as a whole, the attacker would need to control a majority of nodes in all IP-closest groups of nodes in the network, which would require the attacker nodes to account for over half of all the nodes in the network.

Instead of controlling a majority of IP-closest nodes, an attacker could control a majority of storing nodes for a key. An attacker controlling majority of storing nodes for some key can affect the offline messages stored under that key.

An attacker controlling a majority of storing nodes for some key could misbehave by always storing offline messages, without checking with the IP-closest nodes if

the IP address has exceeded its limit of offline messages sent. The minority of honest nodes would not store the offline messages exceeding the limit, but since the attacker-controlled majority stores them, the attacker could republish them on the honest minority nodes, making them store those messages that way. This is then no different from the republishing attack that we discuss a bit later.

An attacker controlling a majority of storing nodes for some key could misbehave by not storing any offline messages and associated receipts while reporting that they were successfully stored. This would affect the offline message republishing, as any node being republished to that has not seen the offline messages before would try to check if the offline messages are stored by a majority of the k-closest nodes, and seeing that they are not, they would not store the offline messages either. Meaning that while initially only the honest minority stored the offline messages the the attacker-controlled majority decided not to store, over time, due to republishing caused by churn, the old honest nodes storing the offline message would be replaced with new honest nodes that would not store these offline messages, with the offline messages the attacker decided not to store disappearing from the network. This is a very targeted attack as it affects only offline messages stored under keys the attacker controls a majority of nodes closest to. The offline message keys are typically randomly generated and exchanged between two users while online, so the attacker would typically not be able to target a particular recipient or sender of such offline messages, aside from one exception: users that have not yet exchanged offline message keys. Such users

use recipient's public key as the offline message key, so the attacker could target such offline messages intended for a particular user. However, that would be a difficult attack to perform, as the attacker would need to control not just a majority of nodes closest to some key, but a majority of nodes closest to a target key. The attack also does not lead to the storage exhaustion.

Another way an attacker controlling a majority of storing nodes for some key could misbehave is by generating bogus offline messages, along with the receipts for them, for the keys the storing nodes are responsible for, and republishing them. The honest minority nodes then would have to store these offline messages and receipts as they are also stored by the attacker-controlled majority. The attacker can use this to exhaust the storage of the honest minority nodes, making then unable to store any new offline messages. Note that in contrast to the IP-closest majority republishing attack, the attacker cannot flush minority nodes' store, as nodes do not remove offline messages once stored until they expire. So if the minority of the nodes had some offline messages stored already, then the attacker might not be able to exhaust their storage for the full 3 days, as those previous offline messages might expire sooner, freeing some of storage space. This is a very targeted attack as it affects only offline messages stored under keys the attacker controls a majority of nodes closest to. An attacker could use it to prevent offline messages for certain keys from being stored, however to do so the attacker would need to control a majority of nodes closest to the target key, which is very difficult to do. In order for it to be extended to cause

storage exhaustion of the network as a whole, the attacker would need to control a majority of nodes in all k-closest nodes, which would require the attacker nodes to account for over half of all the nodes in the network.

Churn The majority requirement for republishing makes it so that the system cannot tolerate losing a majority of nodes within the republishing period. If $\frac{k}{2} + 1$ of the IP-closest nodes for an IP address are lost within the republishing period, either due to many going offline or due to many new nodes appearing that are closer to the IP address, any receipts that were stored for that IP address would be lost as republishing the receipts requires a majority of IP-closest nodes to have the receipts. Similarly with offline messages – losing a majority of k-closest nodes to some key within a republishing period would result in those offline messages being lost. This is in contrast to Kademlia, which tolerates the loss of $k - 1$ nodes, i.e. all but one in a k-closest group. In Kademlia, a node could be offline for some time, come back online and republish its data successfully. With this defense technique, even if a majority of nodes go offline and later comeback online, they might no longer be the closest nodes for the keys they have stored, so when they republish their data – it will not get stored by other nodes for this reason.

The majority requirement for republishing is needed to prevent republishing attacks. Having no requirement for republishing, i.e. allowing any node in the network to republish any data, like in Kademlia, makes the original storage exhaustion attack possible again. Republishing and initial publishing are both used to store data on the

network, and if there is a limit to how many offline messages an IP address can publish but no limit when it comes to republishing, then the attacker can bypass the defense technique completely by republishing. Restricting the number of times a node can republish by counting, similar to how the offline messages are restricted, makes little sense, as we end up with receipts for republishing, which also need to be republished, etc. Having a requirement that the republished data will be stored only if one or more of the k-closest nodes already have the data makes more sense, because for the k-closest nodes to already have that data would mean that it was initially published on there i.e. it was stored by the original sender, with the IP address verified and the offline message limit checked, bringing these checks into republishing. However, if the requirement would be for just one of the k-closest nodes to have the data, an attacker could easily republish any data into any k-closest set of nodes the attacker has a node in. Two nodes would make it harder, three even more so, etc. Since this defense technique depends on an attacker not being able to control a majority of k-closest nodes, it makes sense for the requirement for republishing to be that a majority of the k-closest nodes must already store the data being republished.

To lessen the effects of churn on republishing, the system could be modified a little. To prevent many new nodes joining the network from making a k-closest majority of nodes storing the data from becoming a minority and thus breaking republishing and resulting in the data loss, newly joined nodes could identify themselves as such and not be considered a part of the k-closest majority nodes until some period of time

passes, e.g. a single republishing period, by the end of which the established nodes would have republished the data on the newly joined nodes, with the newly joined nodes verifying it against established majority nodes. To prevent many nodes leaving the network from making a k-closest majority of nodes become a minority and thus breaking republishing and resulting in the data loss, instead of republishing the data on the k-closest nodes to the key, nodes could republish it on more nodes, e.g. the $2k$ closest nodes, while still requiring a majority of just the k closest nodes to already store it. For this to work, the expiration time of the data stored on the DHT would need to be adjusted from Kademlia's default to allow the data stored on $2k$ closest nodes not to expire as quickly.

Other modifications that could help include more frequent republishing and relaxing the majority requirement. The majority requirement could be relaxed in the entire defense technique to require only n nodes to respond with a receipt, instead of a majority, where n is between 2 and $\frac{k}{2}$, both inclusively. Lowering the majority number makes the system more susceptible to the majority attacks, so n must be chosen such that an attacker would not be able to control n of the k-closest nodes. This might be feasible with the number of nodes in the system being large. For example, we know from Section 3.3.1 that with $b = p = 32$, $k = 20$ and $N = 1,000,000$, using Equation 3.6, the attacker would need to go through $n \approx 1,222,364$ unique IP addresses to find a majority of node IDs closest to a key by the node ID prefix. With the number of nodes in the system increased to $10,000,000$, that would become $n \approx 12,236,500$

IP addresses, making the attack a lot harder to perform as an attacker would need to own a lot larger block of IP addresses. With the majority requirement relaxed to requiring only 5 nodes instead of 11, the calculation changes to $\sum_{i=0}^{n} \frac{\frac{2^{32}}{10,000,000} \cdot 15-16}{2^{32}-i} = 5$ or $n \approx 3,340,330$ IP addresses. Depending on the security requirement, this could be considered secure enough while reducing the effect of the churn.

Privacy The technique is privacy-preserving as only the IP-closest nodes learn of how many offline messages an IP address has sent, when they were sent and what size they are, and it is not possible to extract that data from them. IP-closest nodes republish the data only on IP-closest nodes, and other nodes cannot access this information as querying the IP-closest nodes results only in a binary answer of whether they store a receipt with a provided hash value or not.

Storing nodes know a little more than they have to, specifically the IP address of the original sender and sender's node ID key used for singing included in the receipt. The system could be modified such that the sender could sign the receipt using any signing key, not necessarily the one used as part of the node ID, and the IP address could be removed by the storing nodes once they verify that the sender is allowed to store that offline message, as the storing nodes have no use of knowing the IP address after that.

Multiple Tenants Sharing IP The technique could be modified to accommodate multiple tenants sharing the IP address. In addition to having a limit of how many

offline messages an IP address is allowed to send, there could be smaller limits enforced on different node IDs under the IP address. For example, if the IP address limit is $1,000$, the node ID limit could be 100, allowing up to 10 different node IDs under that IP address to reach their sending limit. This would not prevent an attacker, as they could easily bypass this by changing their node ID, but it might be useful for preventing honest users from unintentionally using up the entire offline message allotment of their IP address, denying the users sharing the same IP address from sending any.

Overhead Cost Estimation

In this section we discuss how much networking and storage overhead the technique adds in comparison to the system not using any defense technique.

Networking Overhead Offline message storing takes $k + 2$ store or lookup operations as opposed to just 1 without the defense technique: the sender node does one store operation on the IP-closest nodes, another on the k-closest nodes to the offline message key, and each of those k-closest nodes does a lookup operation against the IP-closest nodes. Data republishing takes in the worst case k extra lookups: when a single node republishes the data no other k-closest node knows about, each of those nodes does a lookup to check if the majority of the k-closest know about it, eventually discarding the data as they do not know about it. The republishing lookups are rather inexpensive as nodes store keys they are the k-closest nodes to, so the republishing

is typically done against the k-closest nodes, most, if not all, of which are already in node's routing table. The SECURE STORE RPC that in some cases is used instead of the regular STORE RPC to prevent IP address spoofing adds an extra message exchange.

Storage Overhead Let us use the the same example network as in Section 3.2.1 – the network with $1,000,000$ online users, each able to store 100 MiB of offline messages, with each offline message being at most 1024 *bytes* and the raw storage capacity of 95.37 TiB. Each unique offline message has $2k$ receipts – k stored on the IP-closest nodes, and k stored on the k-closest nodes to the offline message key. Let us assume that a receipt containing the signing key, IP address, timestamp, hash of the offline message key, size of the offline message, hash of the offline message, and the receipt's signature takes $160/8 + 16 + 8 + 160/8 + 2 + 160/8 + 160/8 = 106$ *bytes*, let us round that up to 128 *bytes* for encoding overhead. Each unique offline message is duplicated k times, so for every offline message stored on the system there are $\frac{2k}{k} = 2$ receipts stored, i.e. each 1024 *bytes* requires additional $2 \cdot 128 = 256$ *bytes* of storage, i.e. $\frac{1024}{256} = \frac{1}{4}$ of the total storage, i.e. the system needs additional $\frac{95.37}{4} \approx 23.84$ TiB of raw storage in order to store the receipts, which is just $\frac{100}{4} = 25$ MiB per node.

Scalability

The technique scales very well with the number of nodes increasing. The number of network requests stays the same, as well as the storage requirement of each node

stays the same. A node is allowed to store only a certain number of offline messages on the network, say l, which would use at most $\left(l \cdot k + \frac{l \cdot k}{4}\right) \cdot 1024$ bytes of the system. As long as the node provides as much storage back to the system, it balances out anything it could store. For example, for $l = 500$ that would be 12.21 MiB the node would need to provide, which is rather reasonable. The security of the technique also scales well – the defense technique becomes less susceptible to majority attacks as the number of nodes increases.

3.3.2 Blockchain-based Technique

A blockchain can be used as a distributed append-only database, storing various data the network agrees on. While a DHT is a localized storage solution, where only a group of nodes closest to the data key learns of the data being stored, a blockchain is global store, as every node learns of what is being stored on the blockchain. All nodes knowing of all the data can be useful, it can allow keeping track of node ID reputation, e.g. newer, never before seen, node IDs could start with lower reputation, while established node IDs could have their reputation raised. It also could be used to keep track of node interactions in order to help identify nodes that a misbehaving node colludes with. A DHT solution for this would require many node lookups for different keys with a long wait time for them to complete, when with a blockchain solution this would be a quick local data lookup. The downside of using a blockchain, however, is that every node has to store a copy of the entire blockchain, so the amount

of data stored on it must be kept to a minimum. As such, storing information on how well established a node is, or storing the history of node interactions, or even storing offline messages on it is not practical, as that would take too much space. To keep the data stored to a minimum while still defending against the storage exhaustion attack, the blockchain-based technique restricts how many offline messages an IP address can send by keeping track of the number and sizes of offline messages each IP address sends, storing that data in a blockchain.

Every node in the DHT participates in creating the blockchain. For every offline message sent, some metadata about the offline message, an offline message receipt, gets stored on the blockchain: sender's IP address, timestamp offset of when the offline message was sent relative to the previous block's creation timestamp, size of the offline message and hash of the key the offline message was stored under. The blockchain used is similar to the Bitcoin blockchain, with blocks being created every 10 minutes using the proof of work. In contrast to the Bitcoin blockchain, it keeps only blocks that were made within 3 days, plus one block header before that for its timestamp. All blocks older than that get discarded as offline messages stored on the DHT expire after 3 days and storing information about expired offline messages on the blockchain serves no purpose.

Note there are two kinds of offline message receipts used in this technique. One that the original offline message sender creates, which gets passed among nodes, containing all the information required for nodes to include it in a block. It is further

described in Section 3.3.2. Nodes work on adding the information from the receipt into a block, however in order to keep the blockchain small is size, they strip all of the unnecessary information from the original receipt, keeping only the bare minimum, which we also call a receipt. It contains just enough information to be able to count how many messages IP addresses has sent and their sizes. The unnecessary information stripped is the information nodes need in order to verify the receipt for it to be included in a block, which is of no use for nodes using the blockchain to check on how many offline messages an IP address has sent. For example, the receipt stored on the blockchain does not include the signature, because for it to be present in the blockchain the nodes would have already verified its signature and formed a consensus about its validity. The contents of the receipt included in a block are described later in Section 3.3.2.

Storage Process

The process of a node storing its offline message is as following.

Step 1 The node creates a signed receipt containing node's signing key (node ID public key), its IP address, port, offline message timestamp, hash of the key the offline message is to be stored under and the size of the offline message data in bytes. It then does the Kademlia's store operation with SECURE STORE RPC, storing both the offline message and the receipt. Nodes verify the receipt: that it is indeed coming from the same IP address as written in the receipt, that the timestamp is recent enough

and matches the one in the offline message, that the hash of the offline message key matches the one in the receipt, that the offline message sizes match, that the signing key matches sender's node ID key and that the receipt is signed with sender's node ID key. If everything verifies and the sender's IP address has not exceeded the limit of offline messages sent in the last 3 days, which can be checked by consulting the blockchain data, the nodes store the offline message, marking it as unverified (not present in the blockchain yet), passing the receipt to other nodes in their routing tables, and work on including it in a future block.

Step 2 The nodes that have received the passed receipt in Step 1 have no way of knowing if the IP address in the receipt is made up. They first consult the blockchain to see if the IP address has not exceeded the limit of offline messages sent in the last 3 days. If it has not exceeded the limit, they verify the IP address by sending the hash of the receipt to the IP address and port written in the receipt, which is the original sender's address, to which the sender replies whether they have sent the receipt or not. If the sender confirms that they have sent the receipt, the nodes pass the receipt to other nodes in their routing tables and work on including it in a future block. These other nodes the receipt was passed to do the actions described in Step 2 upon receiving it.

Mining Process

Nodes work to include an offline message receipt in a block only if they verify that the receipt was sent by the IP address written in it, the receipt was created less than the DHT republishing period minutes ago according to the offline message timestamp in the receipt, the offline message sender's IP address has not exceeded the limit of offline messages sent in the last 3 days and it is not already included in the blockchain. Otherwise they discard the receipt. The reasoning for discarding the receipt once it is older than the DHT republishing period is because without it being in the blockchain, the nodes storing the offline message will not republish it, the offline message will be stored only on those nodes, and we cannot rely on them being online for any longer than the republishing period, so there is a good chance the offline message would disappear from the DHT after that, at which point adding the receipt to the blockchain would be rather pointless.

Republish Process

Nodes keep track of verified and unverified status of offline messages, i.e. whether the offline message's receipt is in the blockchain or not. Only verified offline messages get republished on other nodes. When a node receives a block, it checks if any of unverified offline message's receipts were included in it, marking the offline messages as verified if a sufficient number of blocks were made after that block. If an unverified offline message has been stored for longer than twice the republishing period, it gets

discarded. This is in-line with the mining process discarding receipts older than the republishing period, with the extra time added for the node to discover the block containing the receipt and rule out other blockchain forks.

Verified offline messages are republished on the k-closest nodes to the offline message key using Kademlia's store operation. Whenever a node receives a republished offline message that it does not store yet, it checks if it is included in the blockchain. If it is included in the blockchain, the node stores the offline message and marks it as verified. Otherwise, the offline message does not get stored.

Effectively, the only unverified offline messages a node stores are the ones that it receives directly from their original sender. While unverified offline messages are not republished on other nodes, they are still sent to nodes requesting for them, as an offline message can remain unverified for a long enough time that the recipient of the offline message might come looking for it before it gets verified.

Joining the Network

When a node joins the network, it does not have the blockchain data yet, the node has to download it first. Without the blockchain data the node cannot check if an IP address has exceeded the limit of offline messages sent or if an offline message receipt was recorded in the blockchain, which are the checks a node would do when storing offline messages or republishing them. Depending on the size of the blockchain and node's download speed, it might take some time before the node downloads the blockchain and can perform these checks, from minutes to even hours. Meanwhile,

the node should still be discoverable by other online users via the DHT, as well as being able to discover others, and be able to store offline messages and retrieve offline messages others sent for it, so it cannot be cut off from the DHT, but on the other hand it also cannot preform the storage and republishing DHT processes under this technique until it downloads the blockchain. Since the node is taking a part in the DHT, it has to participate in the storage and republishing processes too, as otherwise there would be patches of semi-functioning nodes in the DHT, which would cause reliability issues leading to potential loss of offline messages. In Kademlia, the k parameter is selected such that at least one node out of k would remain after the republishing period of time elapses, so that if $k-1$ nodes leave, the k-th node would manage to republish the data, preventing it from being lost, however if that k-th node is a new node that is downloading the blockchain and thus does not store any messages yet and neither republishes any, then that will lead to the higher data loss rate. As such, it is better for a node to participate in the storage and republishing processes even if it cannot check IP address reputation without the blockchain.

When a node joins the network and does not have the blockchain data yet, it works on downloading the blockchain data. While the blockchain is being downloaded, the node acts as it normally would, except with the blockchain checks disabled. It accepts offline message store requests and verifies the receipt as usual, but it does not check the IP address reputation of the sender, proceeding with storing the message and passing down the receipt to other nodes to be included in a future block. Similar

with republishing: the node republishes all of its offline messages without verifying them since it is not able to, relying on the nodes these offline messages are being republished to do the verification. When receiving a republishing offline message, it stores it without verifying it either. The node verifies offline messages it stores as it downloads the blockchain, such that by the time it is fully downloaded, offline messages that have not made it into the blockchain are removed, leaving only verified offline messages and unverified offline messages that are pending a verification.

The node is unable to create blocks until it fully downloads the blockchain, but since other nodes send it receipts for inclusion in a future block, the node keeps them until they expire for the purpose of mining a block, so that when it fully downloads the blockchain, it can start working on creating a block right away, without having to wait to receive receipts.

Security Considerations

The blockchain-based defense technique successfully prevents the described storage exhaustion attack by limiting the number of offline messages a node can store, however, just like with any purely peer-to-peer technique without a central authority, it can be bypassed by an attacker with enough resources. In this section we discuss the security of the system using this technique: ways an attacker may try to bypass the technique to achieve the original goal of the storage exhaustion attack, ways an attacker can misbehave and abuse the defense technique, how secure the system using this defense technique is against the churn, privacy aspects of this technique and con-

siderations for multiple users sharing the same IP address. We show that this defense technique makes it harder to perform the storage exhaustion attack, requiring more resources from the attacker than on a system not using the defense technique.

DHT Attacks An attacker can exploit the delay between nodes storing an offline message and its receipt appearing on the blockchain. An attacker could use this time window to launch the storage exhaustion attack on the network. The small time window makes this attack rather impractical, as it can be as short as a few minutes in the case the receipt appears on the blockchain right away, but it will typically be under the republishing period, i.e. 60 minutes – the latest a receipt can be included in the blockchain, after which nodes would stop working on including it and would discard it. In case something abnormal happens and the receipt does not get included in the blockchain, it can be 120 minutes at the longest, as that is the longest time nodes would store any offline messages with a receipt not on the blockchain. That is in contrast to the 3 day time window the original storage exhaustion attack has. Thus this defense technique makes the storage exhaustion attack harder to perform.

Blockchain Attacks The blockchain-based technique is vulnerable to the same attacks as the Bitcoin blockchain. Primarily, an attacker with more compute resources than the rest of the network is able to beat honest nodes in the proof of work computation race, allowing the attacker to modify past blocks and create new ones with any content they like, constructing the longest blockchain fork, which will be preferred

by honest nodes over other forks. Such an attack would allow the attacker to freely remove existing receipts or add new, possibly bogus, receipts to the blockchain.

Removing receipts from the blockchain results in nodes not counting the associated offline messages towards the limit. This can be used by an attacker to reset the offline message count of attacker-controlled nodes, allowing them to launch the storage exhaustion attack, sending offline messages without a limit. However, removing receipts from the blockchain also affects offline message storing and republishing processes. If the attacker removes receipts from the blockchain too fast, nodes storing attacker's offline messages would not verify them and they would get discarded in under 120 minutes, making the storage exhaustion attack harder to perform. An attacker waiting for the nodes to verify offline messages before removing their receipts would slow down the attack considerably, as the attacker would have to wait for around 30 up to 120 minutes before removing them, long enough for the nodes to discover and download the block containing the receipts and then a few more blocks after that, again making the storage exhaustion attack harder to perform. What is more, these offline messages would not survive churn as they would not get republished due to not having their receipts in the blockchain.

An attacker could remove receipts for someone else's offline messages, which, as described above, would make these offline messages more likely to disappear from the network. If the attacker removes everyone's receipts from the blockchain, then no one would be able to store messages for longer than 120 minutes.

An attacker could add bogus receipts for an IP address to the blockchain, which would prevent nodes on that IP address from sending offline messages. If done for all IP addresses in the network, that would prevent anyone in the network from sending offline messages.

Churn The blockchain-based technique does not modify storage and republishing DHT processed very much, keeping them close to Kademlia's, so it provides similar churn resilience guarantees Kademlia does, i.e. stored offline messages would be retained in the system as long as at least one node out of k-closest nodes remains online after the 60 minute republishing period. The blockchain is not affected by churn.

Privacy The technique provides no privacy as every node can see the number of offline messages every IP address has sent, their size, timing and the hash of the key they are stored under. Storing the hash of an IP address instead of the actual IP address might provide some improved privacy, if only a little, and then only assuming IPv6 addresses, as the IPv4 address space is rather small and any privacy hashing provides would be easily defeated by constructing a rainbow table. Even with IPv6 addresses hashed IP addresses would not provide much privacy because an attacker could crawl the DHT, collecting all IP addresses in the network and calculating their hashes.

Multiple Tenants Sharing IP The technique could be modified to provide a better support of multiple tenants sharing the same IP address by making each node ID have a smaller limit of the number of offline messages they can send than the global IP address limit. That way, when multiple honest nodes are sharing an IP address, one honest node will not accidentally use up all the offline message slots allocated for the IP address, leaving another honest node without any. Note that this does not prevent an attacker from using up all the offline messages, as they could keep generating new node IDs until reaching IP address's limit of offline messages sent. As such, this is primarily a measure for honest nodes. For this to be possible, the blockchain receipts would need to have sender's node ID included in it, which takes a lot of space – 20 bytes, and is a privacy issue. As such, instead of recording the node ID as it is, it would be better to record a small hash of it, e.g. 4 or 8 byte hash, salted with other data from the receipt. Such small hash should be enough to identify the handful of nodes that could be sharing the same IP address and because it would be hashed with the message size and the timestamp, among other things, it would be different for every offline message, so everyone reading the blockchain will not be able to tell if there is a single node under the IP address or multiple of them.

Overhead Estimation

In this section we discuss how much networking and storage overhead the technique adds in comparison to the system not using any defense technique.

Networking Overhead The DHT under this technique uses the same number of operations for storing and republishing as Kademlia. The only difference is the SECURE STORE RPC used, which has an extra request in comparison to the regular STORE RPC, so this technique adds very little network overhead to the DHT.

The blockchain, on the other hand, adds a lot of network overhead. The blockchain requires that nodes share among each other the data to be included in a block as well as the created blocks themselves, so nodes would be passing receipts between each other and asking other nodes for new blocks, downloading them. This data exchange is typically done on the best effort basis, so it should at least add a few requests per node for each offline message receipt to be included in a block and each block created.

The biggest number of blockchain related requests a node receives comes from it sending an offline message. Nodes that receive an offline message receipt indirectly have to verify if the IP address did indeed send the receipt by querying it, which means that once a node sends an offline message (along with the receipt), it would be contacted by many nodes in the network. How many depends on how well the receipt gets shared among nodes, but it could be a sizable portion of the network, given that the more nodes know of the receipt, the more likely it to be included in a block.

Storage Overhead There is no storage overhead added to the DHT, it uses the same amount of storage as without the technique. All of the storage overhead of this technique is in the blockchain.

The blockchain stores a receipt for every single offline message sent within the

last 3 days and every node stores a full copy of the blockchain. To reduce the storage burden a receipt is made to be as small as possible. As mentioned before, the receipt contains sender's IP address, timestamp offset of when the offline message was sent relative to the previous block's creating timestamp, size of the offline message and the hash of the key the offline message was stored under. To minimize the amount of the data being stored on the blockchain, only the 64 bit network prefix of the IPv6 is stored, the timestamp is stored as a 7 bit signed integer representing an offset in minutes from previous block's timestamp – enough to represent a value between -64 and 63 minutes, the size is stored as 10 bit unsigned integer – enough to represent the maximum offline message size of 1024 bytes, and the offline message key is hashed to a 64 bit value. So each 1024 *byte* offline message sent results in the blockchain gaining $64 + 7 + 10 + 64 = 145$ *bits* or 18 *bytes* and 1 *bit*.

Using the example network from Section 3.2.1 – the network with $1,000,000$ online users, each able to store 100 MiB of offline messages, with 95.37 TiB of raw storage capacity, the resulting blockchain receipts will take at most $95.37 \div 20 \cdot \frac{145}{1024 \cdot 8} \cdot 1024 \approx 86.43$ GiB of storage. Meaning that each node would need to store 86.43 GiB of blockchain data in addition to the 100 MiB of offline message DHT data.

Using a more conservative example of a network of $1,000,000$ online users, with 1024 *byte* offline messages, in which an IP address is limited to sending at most 300 offline messages within 3 days, the network would need to have $1024 \cdot 300 \cdot 20 \cdot 1,000,000 \div 1024^4 \approx 5.59$ TiB of raw storage capacity, with each node storing

$1024 \cdot 300 \cdot 20 \div 1024^2 \approx 5.86\ MiB$ of offline messages and a $\frac{145}{8} \cdot 300 \cdot 1,000,000 \div 1024^3 \approx 5.06\ GiB$ copy of the blockchain. While less than in the example network from Section 3.2.1, the blockchain still takes a lot of space, even considering that these are maximum sizes and in practice they would be smaller as not all nodes would send the maximum number of offline messages they are allowed to.

Scalability

The technique scales poorly with the number of nodes in the network increasing. The number of nodes increasing makes the blockchain increase proportionally in size, requiring nodes to store and transfer more data. Newly joined nodes are unable to verify offline messages until they download the whole blockchain, so with the blockchain size increasing, new nodes would have to download more data and would be unable to verify messages for longer. If the time it takes to download the blockchain grows to the point that it becomes longer than the average time users keep the messenger software open, it will result in a big portion of the network not using the technique. It would also result in fewer nodes working on creating blocks, which might allow an attacker to launch the majority attack on the blockchain, artificially inflating it in size so that most nodes would not be able to download it and fooling nodes that manage to download it by modifying the blocks, defeating the technique. Also, with the number of nodes increasing, whenever a node sends an offline message it would get contacted by more and more other nodes, which could result in the node experiencing DDoS.

Chapter 4

Simulation Implementation and Experimental Simulations

This chapter introduces the simulation implementation, specifically implementations of the simulator framework, the Kademlia DHT based messaging system, the storage exhaustion attack, and the defense techniques. It then describes two simulated experiments. The first experiment is simulating three systems: one not using any defense techniques, one using the DHT-based defense technique and one using the blockchain-based defense technique, measuring the overhead defense techniques add in comparison to a system not utilizing any defense techniques. The second experiment is simulating the same there systems but with them being under an attack, showing the effectiveness of defense techniques against the attack.

4.1 Simulator Implementation

Simulations are implemented in the Java programming language, and instead of running on a real network in real-time, they run in a simulator. The simulator is an abstraction layer providing system services. The simulator accepts one or more nodes that are to be simulated and provides them with an abstraction for networking, event scheduling, timer service, current time and random number generation. The primary benefit of using the simulator is that it allows for simulation fast-forwarding by skipping time periods when nothing happens, saving on hours of time in comparison to running simulations in real-time. It also allows nodes to pass Java objects in network messages directly, without having to serialize and deserialize them, and it makes simulations independent of the particularities of the networking stack and other aspects of the operating system they run on, as it abstracts the operating system services nodes need to use. Ideally it would also allow for the simulations to be reproducible by having the simulator and nodes use the same seeded random number generator, however due to the way the simulator measures time it is not possible to have reproducible simulations.

At simulator initialization time, each node is assigned a network address, an integer starting at zero and incrementing for every node. Each node is also assigned a network latency to every other node. The network latency is assigned at random within $[50, 250)$ ms interval. To keep the latencies between the nodes realistic and prevent situations where $latency\,(a, b) > latency\,(a, c) + latency\,(c, b)$, the latency

graph is processed by the Floyd-Warshall algorithm. This latency is constant, once it is calculated by the simulator at the start of a simulation, it does not change. In addition to that latency, whenever a node sends a network message to another node, a random $[0, 50)$ ms congestion latency is added on top of the constant latency. The congestion latency is not constant and is randomly generated for every network message sent. There is no packet loss or fragmentation simulated. Network messages are not byte arrays, but Java objects that are passed from one node to another. It is possible for network messages to arrive out of order due to the congestion latency, e.g. the first message a node sends could have the congestion latency of 49 ms but the second message could have the congestion latency of 0 ms, resulting in the second message arriving first.

Once network addresses are assigned and network latencies are calculated, the simulator sets the simulation time to zero and calls into nodes, indicating that the network connection is established, passing them their network address and a reference to the simulator. The nodes use this opportunity to send any network messages and set any timers, populating simulator's event queue. By setting a timer, a node requests the simulator to send it a timer event at the specified time in the future, which is useful for scheduling Kademlia RPC timeouts, republishing, bucket updates and other events.

The simulator keeps track of network and timer events, current time and the availability time of each node – the time at which a node becomes available for

processing events. The simulator is event-driven and runs for as long as there are events to be processed or until the simulation reaches a specified time. After the simulator is initialized, network addresses are assigned, latencies are calculated, nodes are notified of being connected and the event queue is populated with events, the simulator is ready to be run. When the simulator is run, it pops the earliest scheduled event from the event queue, which can be either a network message one node sent to another or a self-set timer event, sets the current simulation time to the time the event should be received at by the recipient node, effectively skipping all the time in-between, and calls into the node the event is intended for, providing the network message or the timer information to its event handler. Whenever the simulator calls into a node, it measures how long the node runs for before giving the control back to the simulator by measuring node's run-time time in nanoseconds. Once the node gives the control back to the simulator, the simulator updates node's availability time to be equal to the time the event was scheduled for, plus the run-time of the node. This has the effect that the simulator will not process any events for this node until the simulation time catches up with node's availability time. Because nodes are single-threaded, it makes sense that they cannot process any events during the time they are already processing events. If the simulator processes an event for a node that is busy to accept an event, the simulator re-schedules the event to be run at the time the node becomes available again, putting it back into the queue with the updated time. This way, if there are multiple events for a node to process, they get queued

up and are processed by the node one by one in the order they were scheduled to be received.

To illustrate how event re-scheduling works, let us imagine two nodes: node A and node B, with node A sending to Node B two network messages. At time 0, the simulator tells both nodes that they are connected. Node A spends 1 second to send node B two network messages which will arrive at 2 seconds of the simulation time. Node A returns the control to the simulator, which records that node A ran for 1 second and thus will not be available until $0 + 1 = 1$ seconds. It adds the two network message events to the event queue and then pops the earliest arriving one, setting simulation's current time equal to the event arrival time, i.e. 2 seconds, processing the event, seeing that it is a network message intended for node B, calling into node B and passing it the network message. Node B takes 10 seconds to process the network message and then returns control back to the simulator. The simulator records that Node B ran for 10 seconds and will not be available until the simulation time is $2 + 10 = 12$ seconds. The simulator then pops the next earliest event, which is another network message Node A sent to Node B, and sets the simulation's time equal to event's time – 2 seconds, and processes the event. The simulator sees that it is a network message intended for Node B, but it cannot call into Node B at this time as it is not available yet, at time 2 seconds it is still processing the previous network message, it will become available only at 12 seconds. (Technically, since the entire code is single-threaded and the simulator called into node B and got the control

back, the node B has already processes the event and is not actually running at the moment, but to keep the illusion of time, node B is considered as processing the event at this time.) The simulator reschedules the event to be run at 12 seconds instead, adding it back to the event queue with the new time. It then pops the earliest event from the event queue, which is the rescheduled network message, sets the simulation time to event's arrival time, i.e. 12 seconds, and tries to process it. It sees that the event is a network message intended for Node B, which is now available, so it calls into Node B, passing it the network message to process.

Simulations are not reproducible due to the simulator measuring nodes' run-time for the node availability time, which varies from a simulation run to a simulation run due to CPU jitter, JVM deciding when to garbage collect memory in the middle of a node running, etc., causing nodes to become available sooner or later, which changes timings and the execution order of simulation events. The simulator keeps track of all times in nanoseconds, as that is the most precise way to measure time in Java and once Java bytecode is JIT-compiled by the JVM, it can take less than a millisecond for a node to process an event. Some events can take very short time to process – a node might ignore an event and immediately return the control to the simulator, while other events might take long time due to a node performing calculations, sorting data-structures, and sending many network messages in response to an event. The simulator is single-threaded, processing only one event at a time, executing only one node at a time. Each node's time advances only when the simulator calls into the

node, executing node's code, and the simulation time advances only when processing events, setting the simulation time to the time of when the event should be received. The time is effectively stopped when the simulator does not execute a node, so all the bookkeeping the simulator does in between calls to nodes does not count towards the simulation time. This means that the simulation can be paused between processing events, the state of the simulator and nodes can be inspected and the simulation can be resumed as if nothing has happened, without the time having advanced. Incidentally, the bookkeeping the simulator does when a node calls into the simulator to send a network message or schedule a timer is also not counted towards node's run-time, as it increases with with the number of nodes in the simulation increasing, which we do not want to be reflected in node's run-time.

4.2 Kademlia DHT Implementation

For the purpose of simulating the messaging system, a Kademlia DHT was implemented. It is a complete and fully functional implementation of the Kademlia DHT without corners being cut for the purpose of simulation. The implementation follows the Kademlia paper [14] closely, with one exception being the routing table – instead of using a binary tree for the routing table, where the routing table starts with a single k-bucket, which grows and splits into more k-buckets as more nodes are discovered, the implementation uses a flat array containing 160 k-buckets. The reasoning for using the flat array data-structure is that the flat array routing table

is easier to implement and reason about, while not affecting the correctness of the Kademlia algorithm. The Kademlia paper describes the routing table as a flat list of 160 k-buckets when describing the node state, introducing the binary tree based routing table only later in the paper, and the popular pre-proceedings version of the Kademlia paper [13] describes it as a flat list as well, without the tree data-structure existing in that version of the paper. With a flat array routing table there is no need to split buckets based on complex rules, no care should be taken to avoid the tree from becoming unbalanced, every bucket covers a well-defined range, with bucket i containing nodes that are 2^i to 2^{i+1} distance away, and the index of the bucket a node will fall into can be easily calculated by finding the length of the common bit prefix of the current node's and the target node's node IDs. The downsides of using a flat array routing table are negligible for the purpose of the simulation: with 160 buckets pre-allocated the flat array routing table takes more space than the tree would, and most of the buckets are empty, as there are not be enough nodes in the network to fill the buckets closest to the node. Having empty buckets that no node in the network falls into also leads to unnecessary bucket refreshes, however, as described later in Section 4.6, this is not an issue for the simulation.

The Kademlia implementation supports all RPCs: PING, FIND NODE, STORE and FIND VALUE, with requests including a random 160-bit ID that is echoed back in the response. It also implements all of the operations: find node, store, find value and refresh. The find value operation can also optionally cache the data found on the

closest node to the value's key that did not reply with the data.

4.2.1 Finding Closest Nodes

Finding k nodes that are closest to a given node ID in one's routing table is one of the most important algorithms in Kademlia. The simplest way of finding them would be calculating the distance of every node in the routing table to the given node ID and taking k node IDs that resulted in the the smallest distances. However, given how frequently a node has to run this algorithm, something more optimized is desired. The Kademlia implementation finds the nodes closest to a given node ID by calculating the distances between the node ID ranges buckets represent and the given node ID, visiting the buckets in the order of increasing distance from the given node ID until k nodes are found. As shown by Spori [23], this can be done by a single XOR operation – calculating the distance between node's node ID and the given node ID, as node IDs the buckets represent are derived directly from node's own node ID. By visiting buckets with the indexes of the set bits in the resulting 160-bit distance number in the order from the most significant to the least, and then visiting the buckets with the indexes of unset bits in the opposite order – from least significant to the most, the node will visit the buckets in the increasing order of closeness to the given node ID.

To demonstrate this, let us assume 8-bit node IDs, with the current node having node ID 0001_0100. This results in the node's routing table having 8 k-buckets

that contain nodes with node IDs starting with the following prefixes: 0001_0101, 0001_011x, 0001_00xx, 0001_1xxx, 0000_xxxx, 001x_xxxx, 01xx_xxxx and 1xxx_xxxx. Let us find closest nodes to a node ID 1010_0010. The distance between the current node and the given node is 0001_0100_2 XOR $1010\_0010_2 = 1011\_0110_2 = 182_{10}$. By following the algorithm above, we should first visit the buckets with indexes corresponding to the indexes of the set bits in the XOR result in the left to right order: 8th, 6th, 5th, 3rd and 2nd. Then we should visit buckets with indexes matching unset bits in the XOR result in the right to left order: 1st, 4th and 7th. To confirm this, we calculate the distance between the given node ID and each bucket, replacing "x" with "0", we get: 183, 180, 178, 186, 162, 130, 226, 34. If we sort them by the distance, we can see that we get the same order of bucket indexes: 8th (34), 6th (130), 5th (162), 3rd (178), 2nd (180), 1st (183), 4th (186) and 7th (226).

4.2.2 Data Expiration

The Kademlia paper omits a few implementation details, leaving it up to the implementers. Notably, it says that all ⟨key, value⟩ pairs stored on a node have the expiration time that is "exponentially inversely proportional to the number of nodes between the current node and the node whose ID is closest to the key ID", without clarifying on how to estimate the said number of nodes aside from saying that "the number can be inferred from the bucket structure of the current node". The Kademlia implantation calculates the expiration time as described by the XLattice Project [20].

Specifically, to calculate the expiration time of a ⟨key, value⟩ pair, the node finds the index i of the bucket the key would fall into, counts all nodes C_a in $[0, i)$ buckets and then, in the final bucket i, it counts only nodes that are closer to the current node than the key is. Given $C = C_a + C_b$ and t_r being the remaining time of the data pair, the expiration time t_e is calculated as following:

$$t_e = \begin{cases} t_r, & C < k \\ t_r \cdot \exp\left(-C/k\right), & \text{otherwise} \end{cases} \tag{4.1}$$

In other words, if the node is a k-closest node to the key, the data is stored for the full remaining time, otherwise it is stored for an exponentially less time the more nodes there are in-between. With $k = 20$, $C = 20$ and $t_r = 24$, the data would be stored for just 8 hours, with $C = 40$ for 3 hours and with $C = 100$ for under 10 minutes. Any data that expires too soon, under 10 minutes, is not stored by the node.

The Kademlia paper also does not say anything about checking the data for expiration or updating expiration times. Every 10 minutes the Kademlia implementation recalculates the expiration times of all data pairs stored and removes the expired data. Recalculating expiration times is important to account for churn: due to nodes leaving the network we might become the k-closest node for a data pair, in which case we would want to store it for the full time, or due to new nodes joining, some cached data we were storing is now considered as expired, in which case it should be

removed.

On Different Ways To Calculate Data Expiration

Some Kademlia implementations calculate the number of nodes between the current node and the key, used to calculate the data expiration time, differently. For example, Spori [23] in their work calculate the number of nodes between the current node and the key differently, which we believe to be incorrect. What the Kademlia implementation calculates is the number of nodes with the distance to *the current node* lower than the distance between the key and the current node, but Spori [23] calculates the number of nodes with the distance to *the key* lower than the distance between the current node and the key. While both methods calculate the number of nodes between the current node and the key, we believe that the way it is calculated in our Kademlia implementation to be the correct one. Due to the structure of the routing table, a node has more knowledge about the nodes that are closer to it rather than about the nodes that are closer to the key, allowing the current node to better estimate the number of nodes between itself and the key when counting nodes that are closer to it.

To illustrate this, let us assume 8-bit node IDs, with the current node having node ID 0000_0000. This results in the node's routing table having 8 k-buckets that contain nodes with node IDs starting with the following prefixes: 0000_0001, 0000_001x, 0000_01xx, 0000_1xxx, 0001_xxxx, 001x_xxxx, 01xx_xxxx and 1xxx_xxxx. Let us find the number of nodes between the current node and the key 1000_0000.

The distance between the current node and the key is 0000_0000_2 XOR $1000\_0000_2 = 1000\_0000_2 = 128_{10}$. Note how this is half of the maximum possible distance, i.e. there ought to be many nodes in-between, and also note how the key falls into the last bucket that the current node has a poor knowledge of. Following the algorithm described in Spori [23], only nodes in the last bucket would be counted, as only that bucket would contain nodes closer than 128 to the key. That means that the current node would count at most k nodes between itself and the key, incorrectly deducting that there are few nodes in-between, storing the key for a long period of time, if not even the full remaining time, while the current node might be far from the k-closest node to the key.

The algorithm used by our Kademlia implementation would count all nodes in buckets 1-7, plus the nodes in the bucket 8 that are less than 128 distance away to the current node. (As the bucket 8 contains only nodes from 1000_0000 to 1111_1111 and the key is 1000_0000, there would be no such nodes found in that bucket, so technically only nodes in buckets 1-7 would be counted). It will correctly count many more nodes, resulting in a better estimation of the nodes between the current node and the key, as well as in the proper calculation of the data expiration time.

4.2.3 Optimizations

The Kademlia implementation supports the efficient republishing as described in the Kademlia paper, that is, when a node receives a STORE RPC for some data, it

assumes that other $k-1$ nodes have also received it and thus does not republish it for the next hour. Unless all k nodes are synchronized, only one node would republish a data pair. The implementation also republishes the stored data on new nodes. Whenever a new node is discovered, the current node sends it all the data pairs with keys that are closer to the new node than to itself.

The Kademlia implementation supports the optimized contact accounting as described in the Kademlia paper: storing nodes in the replacement cache and marking unresponsive nodes as stale. Whenever a node receives a request coming from an unknown node that falls in a k-bucket that is full, the last recently seen node in that bucket must be sent a PING RPC to decide if it should be replaced by the new node, however by placing the new node in the replacement cache this can be avoided. This reduces the traffic Kademlia generates and removes all of the uses of the PING RPC in Kademlia. Kademlia also counts RPCs as failed if they are not replied to within 5 seconds and after 5 consecutive failed RPCs the node that is failing to reply is marked as stale. A stale node is replaced with a node from the replacement cache if the bucket containing the stale node is full.

4.2.4 Parameters

The Kademlia implementation uses the same parameters as Kademlia, and for things Kademlia leaves out the details it tries to use sensible defaults: 160-bit node IDs, $k = 20$, concurrency parameter $\alpha = 3$, considers RPCs failed if not replied within

5 seconds, nodes stale after 5 failed RPCs, refreshes stale buckets every 60 minutes, republishes data every 60 minutes, sets data expiration to 24 hours, does not accept data that has the remaining time less than 10 minutes, updates the remaining time of stored data and removes expired data every 10 minutes. It also allows to limit how much data a node can store and how big a single stored value can be, rejecting store requests once the limit is reached, which can be set by the user depending on the expected storage requirements.

4.2.5 Interface

Kademlia DHT is implemented to run in the simulator and allows a user to store a ⟨key, value⟩ pair on the DHT, look up a value given a key, and access extensive operational statistics on the node, such as the breakdown of network messages sent by type, total number of network messages sent, number of the data pairs received, number of data pairs stored, number of data pairs republished, number of failed RPCs, number of stale nodes replaced, number of failed STORE RPCs with the breakdown of the reasons why the data was not not stored: invalid data, already expired, expires too soon, the data is bigger than allowed and not enough storage, and so on.

4.3 Messaging System Implementation

The messaging system is implemented as a slightly modified Kademlia DHT. Because this thesis concerns primarily with storing offline messages, no online messaging com-

munication, peer finding, offline message deletion, etc. was implemented. The messaging system does not employ encryption or digital signatures, everything is done in plain text. Offline message storage is done by Kademlia DHT storing a ⟨key, value⟩ pair.

The system is kept as close to Kademlia as possible. There only notable change done to Kademlia is disabling of data caching. The data caching during Kademlia's find value operation is disabled, as it is a measure to reduce the load for popular data lookups, but in the messaging system an offline message is intended for a single recipient, with the expectation that only one node would look it up – an offline message becoming popular among many nodes is not something that would happen.

4.4 DHT-based Defense Technique Implementation

For the purpose of the simulation, the DHT-based defense technique was implemented. It is implemented as a modification of the base Kademlia DHT-based messaging system. Because we control all of the nodes in the simulation and can make sure that nodes do not do certain things, the implemented technique makes use of that assumption and omits implementing certain checks that would be redundant in this case. Only redundant checks and things of no consequence for the simulation were omitted, it still implements all the core checks required for the technique to

work and sends the same number of network messages as it otherwise would.

A separate data store was added to Kademlia DHT to keep track of IP-closest receipts. The implementation uses a simpler version of an offline message receipt that contains only the hash of sender's network address and the key an offline message is stored under, as that is enough to simulate the technique. At no point the receipts are verified by any node, as no node in the simulation is going to fake or modify a receipt. SHA-1 hashing algorithm is used for hashing sender's network address, which generates 160-bit digests, making it convenient to use as a node ID when looking up IP-closest nodes. The receipt data store republishing is handled the same way as the regular data republishing. They both republish on the same 60 minute timer, and they both republish entries that they have not received a store request for in the last 60 minutes, as per Kademlia's efficient republishing optimization.

Node IDs used are random 160-bit numbers, they are not prefixed with the hash of a network address, as no node in the simulation is going to attempt the majority attack. Also, because every node in the simulation is guaranteed to have an unique network address, the implementation does not check for nodes sharing the same network address in order to exclude them from majority lookups.

To be able to query k-closest nodes to a key and consider a majority of their replies, the implementation adds a new find value closest operation to Kademlia. The operation does a lookup for the k-closest nodes to the key using FIND NODE RPC first, and then send them FIND VALUE RPC, capturing all the responses. This

is in contrast to Kademlia's find value operation, which does a lookup for the k-closest nodes to the key sing FIND VALUE RPC, which returns immediately as soon as a single, potentially not even k-closest, node replies with the value.

The implementation adds the SECURE STORE RPC to Kademlia, which is a 3 message version of the STORE RPC. While no node in the simulation is going to forge message's network address, simulating this RPC is important for getting the accurate number of network messages the technique would generate.

The implementation follows the technique closely. To store an offline message, a node sends the corresponding offline message receipt to its IP-closest nodes using the SECURE STORE RPC. The IP-closest nodes store the receipt, as long as sender's network address has not exceeded the maximum number of allowed offline messages sent, and reply back to the sender with the storage result. If the result indicates majority success, the node proceeds by storing the offline message and the receipt on the k-closest nodes to the offline message's key using the SECURE STORE RPC. Upon receiving the store request, the storing nodes contact sender's IP-closest nodes to check if the sender has exceeded the maximum number of offline messages sent. The storing nodes decide on whether or not to store the offline message depending on the majority response from the IP-closest nodes, responding back to the original sender about the success of the operation.

When a node receives a republished offline message that the node does not store, it checks if the k-closest nodes to offline message's key store the offline message, storing

it depending on the majority response. Similarly when a node receives a republished IP-closest receipt that it does not store – it checks if the network address included in the receipt has exceeded the maximum number of offline messages sent and checks with the IP-closest nodes responsible for that receipt, storing it depending on the majority response.

4.5 Blockchain-based Defense Technique Implementation

For the purpose of the simulation, the blockchain-based defense technique was implemented. It is implemented as a modification of the base Kademlia DHT-based messaging system. Because we control all of the nodes in the simulation and can make sure that nodes do not do certain things, the implemented technique makes use of that assumption and omits implementing certain checks that would be redundant in this case. Only redundant checks and things of no consequence for the simulation were omitted, it still implements all the core checks required for the technique to work and sends the same number of network messages as it otherwise would.

The implementation uses a simpler version of an offline message receipt that contains only the hash of sender's network address, the key an offline message is stored under and the timestamp of when the offline message was created, as that is enough to simulate the technique. SHA-1 hashing algorithm is used to for hashing sender's

network address. The receipts that nodes pass to each other and the ones stored on the blockchain are the same in this implementation.

The implementation adds the SECURE STORE RPC to Kademlia, which is a 3 message version of the STORE RPC. While no node in the simulation is going to forge message's network address, simulating this RPC is important for getting the accurate number of network messages the technique would generate.

The implementation follows the technique closely. To store an offline message, a node sends the offline message and the corresponding offline message receipt to the k-closest nodes to the offline message's key using the SECURE STORE RPC. Receiving nodes check the blockchain to see if sender's network address has exceeded the maximum number of allowed offline messages sent. If it has not, they store the offline message and the receipt, reply back to the sender with the operation result, work on including the receipt in a future block and forward the receipt to other nodes so that they would also work on including it in the blockchain. The nodes receiving the forwarded receipt check if they have not seen the receipt yet and if it is not stored on the blockchain already, after which they contact the original sender, to make sure they indeed sent the receipt, and if so they work on including it in a future block and forward it to further nodes. While checking if the original sender did indeed sent the receipt is unnecessary in the simulation, after all there are not be fake receipts, it is important for estimating the number of network messages the technique sends when implemented with all checks present.

The way a node selects which nodes to forward a receipt to is important, as the goal of the forwarding is for all nodes to receive the receipt and work on including it in a future block. The way the implementation does this is by forwarding the receipt to the k-closest nodes of the current node, then finding all non-empty buckets that are further than the furthest k-closest node and forwarding the receipt to a random node in each such bucket. Forwarding to the k-closest nodes exploits node's good knowledge of the local ID space and forwarding on a single node of each of the further buckets spreads the receipt further down the ID space. When simulating the technique, this algorithm resulted in all nodes learning of the receipt.

The implementation modifies Kademlia's republishing process to make sure that only messages included in the blockchain are republished. Additionally, any offline messages receipts of which have not made it in the blockchain within twice the republishing period, i.e. 120 minutes, are removed from the data store, as they are too old to appear in the blockchain at any later time.

The implementation is not using an actual blockchain, but instead the blockchain is simulated as three hash tables that are shared among all nodes. One hash table represents the blockchain – it contains all the receipts that were added to the blockchain, another hash table, staging hash table, represents receipts that every node is working towards including in a future block – it contains all the receipts that were created that have not made it in the blockchain yet, and the last hash table is the staging counter hash table – it counts how many nodes have received each of the receipts in

the staging hash table, or how many nodes are "working" on including each receipt in the blockchain. Whenever a node receives a receipt, it makes sure that it is not in the blockchain, adds it to the local list of received receipts so that it does not process it again if some other node forwards it again, the receipt is stored in the shared staging hash table and the counter for the receipt is incremented in the shared staging counter hash table, indicating that one more node is working on including the receipt in a future block. Every 10 minutes "a new block" is created by some of the receipts from the staging hash table being moved to the blockchain hash table. Which receipts to move in the blockchain is decided at random, with older receipts and the receipts that many nodes are working towards inclusion of having higher odds. The probability $p(r)$ of a receipt r being included in the blockchain is calculated as the following:

$$p(r) = \begin{cases} \frac{including(r)}{N} \cdot 0.25 \cdot \left\lfloor \frac{age(r)}{10} \right\rfloor, & age(r) < 50 \\ 1, & \text{otherwise} \end{cases} \tag{4.2}$$

Where $including(r)$ is how many nodes are working on including the receipt in a block, N is the total number of nodes in the simulation and $age(r)$ is the age of the receipt in minutes. Given that all receipts are received by all other nodes in the simulation, this makes $\frac{including(r)}{N} = 1$ and effectively means that $[0, 10)$ minutes old receipts have 0% chance of inclusion, $[10, 20)$ have 25% chance, $[20, 30)$ have 50% chance, $[30, 40)$ have 75% chance and $[40, 50)$ have 100% chance. If a receipt does not have all nodes working on it, it might not get included in the blockchain in time, so

there is the special fail-safe case for receipts 50 minutes and older, effectively $[50, 60)$ due to blocks being created every 10 minutes, to have 100% chance of inclusion. The implementation assume that all qualifying receipts are included in the blockchain before they are 60 minutes old, i.e. there is no receipt loss simulated. The floor operation is the effect of using the integer division. Right before a receipt is to be added to the blockchain, it is checked if the network address the receipt belongs to exceeded the maximum number of allowed offline messages sent, in which case the receipt gets dropped.

Nodes can access the newly added to the blockchain receipts immediately, there is no block discovery delay simulated. Although not intentionally made that way, receipts younger than 10 minutes not being added to the blockchain could be thought of as a such delay.

Every republishing period, i.e. 60 minutes, a node checks the local list of received receipts and removes all receipts older than 60 minutes as they should no longer be forwarded by anyone.

4.6 Simulation Setup

This section describes the simulation setup and parameters common to all messaging system simulations.

Nodes are assigned a unique network addresses, with no two nodes sharing the same address. The system has no churn, all nodes connect at the start of a simulation

and remain online throughout the entire simulation. Nodes connect to the network sequentially, with 1 second delay between each other to avoid synchronization issues. Once connected, nodes bootstrap off the first node. To avoid synchronization issues when republishing, as all nodes republishing at the same time would defeat the efficient republishing optimization, the nodes republish at a $[0.85, 1]$ of the republishing time, i.e. $[51, 60]$ minutes, at random. Refreshing of stale buckets is disabled, as it is not useful when there is no churn – it results in only empty buckets being refreshed, which nodes have many of, just to find that there are no nodes that fall into these buckets.

The simulations simulate 24 hour scenarios, with offline messages expiring after 24 hours. The system allows for a single node to store at most 100 offline messages, which is enforced by nodes refusing to store more than $100 \cdot k = 100 \cdot 20 = 2000$ offline messages and by defense techniques limiting the number of offline messages a network address can store to 100.

To simulate users storing offline messages, nodes store offline messages at random times such that on average each node would store at most 70 offline messages in the 24 hour period. Users are assumed to be well-behaving and not exceeding the 100 offline message limit. To achieve this, nodes store messages in accordance with the Poisson distribution. The Poisson distribution describes the number of times an independent event occurs within a fixed interval of time given a constant mean rate of occurrence, which fits the description of users sending offline messages well. Every minute a node

samples from the Poisson distribution $P(\lambda)$ with $\lambda = 70/(24 \cdot 60)$, storing as many offline messages as the number of events the Poisson distribution returns. In a rare occasion a node might attempt to store more than 100 offline messages. To prevent such cases, a node stops storing offline messages after it has stored 100 of them.

The simulations are done with either 100, 200 or 300 nodes. While it is possible to simulate a system with no defense techniques with many nodes for long periods of time, e.g. 500 nodes running for 72 hours, defense techniques add enough computational and memory overhead to make simulations with such big numbers very slow and take long time, so a lower number of nodes and a shorter simulated time was used.

4.7 Simulation 1: Defense Technique Overhead and Scalability Comparison

To compare the performance and overhead of defense techniques, the three systems were simulated: one not using any defense technique, one using the DHT-based defense technique and one using the blockchain-based technique. The systems were simulated with different number of nodes: 100, 200 and 300, to measure their scalability with the number of nodes increasing. The rest of the simulation is the same as described in Section 4.6: it ran for 24 hours of the simulated time, with honest nodes storing on average 70 offline messages throughout the entire simulation, with-

out exceeding the 100 offline message limit. There was no attack simulated. The simulations measured the breakdown of the average number of RPCs a node has sent by the RPC type and the average number of RPCs a node has sent in total, shown in Table 4.1, and the average run-time of a node as measured by the simulator, shown in Table 4.2.

4.7.1 Networking Overhead

The Table 4.1 summarizes the networking overhead of the simulated systems by counting the number of RPCs processed. As it can be seen from the table, both techniques add a substantial overhead in terms of RPCs a node processes in comparison to a system not utilizing any defense technique. The DHT-based technique has the least amount of overhead, with a node processing on average 3.8 times more requests than a system without any defense technique. A notable difference between the techniques is that the DHT-based technique scales well with the number of nodes increasing in the network – it uses the same number of RPCs no matter the network size, while the blockchain-based technique scales poorly as it uses more RPCs the more nodes there are in the network. This is in-line with the salability discussions of the DHT-based technique in Section 3.3.1 and the blockchain-based technique in Section 3.3.2.

RPCs

Both techniques use the SECURE STORE RPC for storing an offline message, with the DHT-based technique using double the number the blockchain-based technique

Table 4.1: The average number of RPCs a node processed.

Technique	DHT-based			Blockchain-based		
Number of nodes	100	200	300	100	200	300
FIND NODE	74,047	72,589	74,279	19,628	19,002	19,433
FIND VALUE	33,628	31,927	31,933	6,546	13,260	20,415
STORE	34,037	33,822	34,359	169,898	330,901	509,767
SECURE STORE	2,755	2,762	2,782	1,425	1,383	1,403
Total	144,467	141,100	143,353	197,497	364,546	551,018

Technique	None		
Number of nodes	100	200	300
FIND NODE	19,128	19,361	19,096
STORE	18,522	18,911	18,581
Total	37,650	38,272	37,677

does as it performs a secure store operation twice: to the IP-closest nodes and to the nodes that would store the offline message.

Both techniques use the STORE RPC for republishing. The blockchain-based technique additionally uses the STORE RPC to forward a receipt to further nodes, so that they would work on including it in a future blockchain block. The nodes forward the receipt in such a way that all the nodes in the network would eventually receive it, which is why the blockchain-based technique does so many STORE RPCs and why they increase with the number of nodes in the network increasing.

The FIND VALUE RPC is used by the DHT-based technique to query a node for a receipt when receiving the STORE RPC or a SECURE STORE RPC for a receipt or an offline message that the node does not store yet, which happens during republishing and when a node stores an offline message. In the blockchain-based technique the FIND VALUE RPC is used by nodes that received a forwarded receipt via the STORE RPC, they use FIND VALUE RPC to query the original sender of

the receipt to check if they indeed sent the receipt. The disparity between the number of STORE RPCs and FIND VALUE RPCs in the blockchain-based technique shows the redundancy of the receipt forwarding, that a node has a receipt forwarded to it many more times than it checks the receipt with the original sender (it checks the receipt only the first time it is received).

The FIND NODES RPC is used by the DHT-based technique alongside all other RPCs, as most actions in the DHT-based technique require finding the k-closest nodes to a key or a node ID. In contrast, the blockchain-based technique does not use the FIND NODE RPC for anything aside from finding nodes to store an offline message on and republishing – the same actions a system without any defense technique uses the FIND NODE RPC for, which is why its number of FIND NODE RPCs processed matches the one of the defenseless system.

The system without a defense technique does not use SECURE STORE and FIND VALUE RPCs. It lacks the SECURE STORE RPC and the use of the FIND VALUE RPC is not simulated – the FIND VALUE RPC is something a user would trigger when looking up an offline message, but the simulation does not simulate users looking up messages, it has users only storing them.

4.7.2 Computational Overhead

The Table 4.2 shows the average measured run-times of a node in a simulation. With the DHT-based technique system and the defenseless system each processing the same

Table 4.2: The average run-time of a node in milliseconds.

Technique	None			DHT-based		
Number of nodes	100	200	300	100	200	300
DHT (ms)	612	696	772	2,019	2,776	3,421

Technique	Blockchain-based		
Number of nodes	100	200	300
DHT (ms)	1,123	2,423	4,219
Blockchain (ms)	43,200,000	43,200,000	43,200,000

number of RPCs regardless of the network size, it is expected that a node in these systems would spend on average the same amount of time on processing them, but that does not appear to be the case. The run-time of these two systems increases with the number of nodes increasing. Since the run-time is measured in real time, it could reflect side-effects outside the simulation, for example the JVM pausing a node in the middle of it processing an RPC to perform garbage collection, which is more likely to happen the more nodes there are in the system, as more nodes storing more data increases the memory pressure. As such, this data might not be very telling of the actual performance.

What is telling, however, is blockchain technique's absurd run-time caused by running a proof of work algorithm. Since the simulation does not use an actual blockchain running a proof of work algorithm, but rather simulates its properties using a shared hash table, the run-time of a proof of work blockchain algorithm was taken to be the half of the block creation time – 5 minutes, which is 12 hours, or $43,200,000$ milliseconds, when aggregated over 24 hours of the simulation. The 5 minutes were picked somewhat arbitrarily, thinking that surely no one would want to

dedicate more than half of their CPU time on a proof of work for offline messages. It is hard to pick a non-arbitrary estimation of proof of work run-time, as it is highly user-adjustable: one user could dedicated multiple threads of their CPU running it, while another could throttle it down heavily. Regardless, there is a non-zero run-time cost to it, which greatly outweighs the run-time of the DHT. This makes the blockchain-based technique the most computational demanding technique.

4.8 Attack Implementation

For the purpose of the simulation, a storage exhaustion attack was implemented. The attack simulates a single attacker node storing bogus offline messages on the network until no more offline messages can be stored.

The attack is implemented as a single attacker node storing offline messages sequentially on all other honest nodes. The attacker does not store offline messages randomly, they are addressed directly to specific nodes and use the STORE RPC or the SECURE STORE RPC instead of the regular store operation. Using RPCs allows one to target specific nodes at specific network addresses, which makes the attack more efficient than storing an offline message on the k-closest nodes to a randomly generated key, as by targeting nodes directly the attacker can avoid storing offline messages on nodes that are already full. Furthermore, sending just an RPC per an offline message makes it so that the offline message would get republished by honest nodes on k other nodes closest to the offline message's key on attacker's

behalf, without the attacker having to spend resources sending all of the k RPCs themselves. This is possible mainly due to there being no churn in the simulated system, guaranteeing that a node receiving an offline message would republish it on the k closest nodes.

The attacker uses the STORE RPC for attacking a system without any defense, as that is the only kind of a store RPC it supports, and the SECURE STORE RPC for other systems, as they require using it in order for offline messages to get stored. For the DHT-based technique the attacker also stores the corresponding receipt on its IP-closest nodes, as otherwise its offline messages would not get stored at all. Similarly, the attacker cooperates in the blockchain-based technique by replying with a confirmation to the nodes asking if the attacker indeed sent the offline message, as otherwise it would expire and get removed from the network in 120 minutes.

We assume that the attacker has already crawled the network and knows of all the nodes in it, as such the crawling of the DHT is not simulated, the attacker makes use of the data available in simulation that list all of nodes' network addresses. Every minute, the attacker stores a certain number of offline messages on honest nodes. The attacker stores them by contacting the nodes sequentially, from network address 0 to N, one offline message per node at a time. This is done so that no single node is overloaded by the store requests, spreading out the store requests among all the nodes, as constantly sending them to a single node could DoS it for a short time.

The attacker keeps track of the responses nodes send on the store requests. If

a node indicates that an offline message could not be stored due to the receiving node reaching the maximum number of offline messages it can store (the no technique system) or due to the sender node exceeding the maximum number of offline messages it is allowed to store (DHT and blockchain based techniques), the attacker remembers the node and does not store any more offline messages on it. Eventually all of the honest nodes respond with errors, at which point the attack is complete.

The offline messages are stored under keys that are close to the target nodes in order to avoid republishing issues. Specifically, if a node stores offline messages that it is not the k-closest node to, it would never receive republishing requests for them and would keep republishing them every time until they expire, causing inefficiencies in the simulated DHT network. To avoid this, the attacker generates keys that share the 96 bit prefix with the node ID of a node the offline message is to be stored on, making the remaining 64 bits of the key random.

4.9 Simulation 2: Defense Technique Effectiveness Against the Attack

To test the effectiveness of the defense techniques against the storage exhaustion attack, the three systems were simulated: one not using any defense technique, one using the DHT-based defense technique and one using the blockchain-based technique. The systems were simulated with 200 nodes, under the attack from a single node

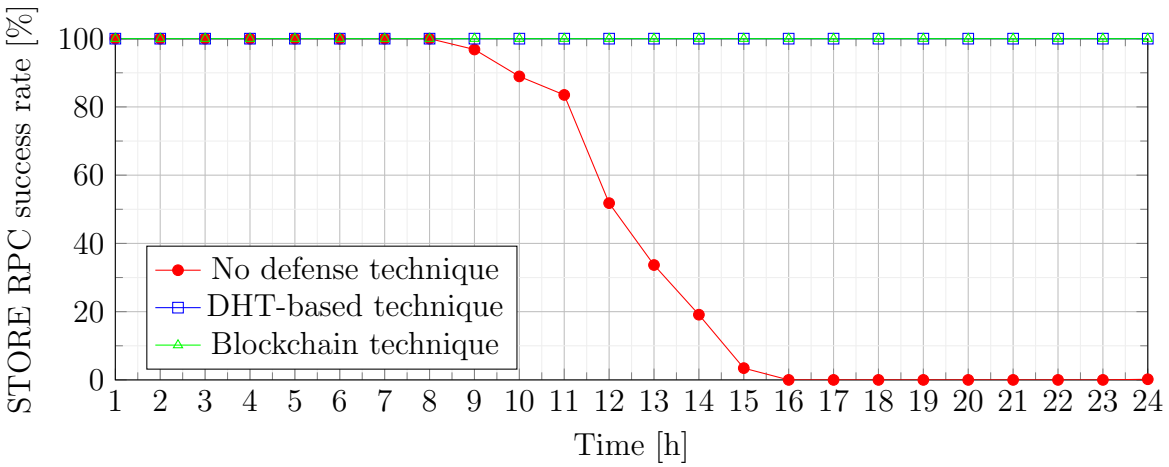

Figure 4.1: STORE RPC / SECURE STORE RPC success rate of different defense techniques over time while under an attack.

storing 20 offline messages per minute. The rest of the simulation setup is the same as described in Section 4.6 and Section 4.8: the simulation lasted for 24 hours, with offline messages expiring in 24 hours, with honest nodes storing on average 70 offline messages throughout the entire simulation, without exceeding the 100 offline message limit. The success rate of honest node's STORE RPCs was measured every hour.

As it can be seen in the Figure 4.1, the attack succeeded in the system with no defense technique, making nodes unable to store offline messages after the 15 hour mark. The system's storage capacity is $100 \cdot 20 \cdot 200 = 400,000$ offline messages – 200 nodes storing at most $2,000$ offline messages each. In 15 hours, the attacker sending 20 offline messages per minutes would have sent $20 \cdot 60 \cdot 15 = 18,000$ offline messages, with each receiving node republishing them on 20 other nodes, resulting in the system storing approximately $18,000 \cdot 21 = 378,000$ of offline messages in total. With not only the attacker storing offline messages but also the honest nodes storing

their messages on the system, the storage capacity of the system gets exceeded at 15 hour mark and nodes refuse to store ant new messages.

The attack failed in systems using defense techniques, with nodes having 100% success rate of storing offline messages throughout the entire simulation. Both of the techniques made the attacker give up on the attack in under an hour, as nodes were refusing to store attacker's offline messages with the attacker having exceeded the maximum number of offline messages a network address is allowed to send.

Chapter 5

Conclusion

We have presented two reputation-based defense techniques against the storage exhaustion attack on Kademlia DHT: the DHT-based technique that stores the IP address reputation information on the DHT, locally on a few nodes, and the blockchain-based technique, which stores the information globally on the blockchain, shared by all of the nodes. Due to an attacker being able to easily change their DHT identities, making it hard to identify the attacker, the techniques use IP addresses as a way to identify the attacker. The techniques prevent the storage exhaustion attack by keeping track of how many offline messages were stored on the DHT by an IP address, restricting the number of offline messages an IP address can store. Both techniques are effective against the storage exhaustion attack, as shown in the simulated attack in Section 4.9, however some have more advantages than the others. The blockchain technique uses too many network requests on receipt forwarding and receipt verification, too much processing power on the proof of work, and too much storage on

the blockchain, to the point that a node stores a blockchain that is a thousand times larger in size than the storage allocated for offline messages which the blockchain protects against the storage exhaustion attacks on. Meanwhile, the DHT-based technique is shown to have lower networking, processing and storage overheads, while also scaling well, making the DHT-based technique the better choice out of the two. The DHT-based technique does suffer from not being very churn resistant, with high churn rate resulting in republishing not functioning and the data loss, which would not happen in a base Kademlia DHT, but as mentioned in Section 3.3.1, there are ways this can be mitigated.

References

[1] Adam Back. Hashcash - a denial of service counter-measure. `http://www.hashcash.org/papers/hashcash.pdf`, August 2002. Accessed: 26 February 2022.

[2] Salman Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–11, April 2006.

[3] Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure key-based routing. *2007 International Conference on Parallel and Distributed Systems*, 2:1–8, 2007.

[4] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Cryptographic Hardware & Embedded Systems - CHES 2011*, pages 124 – 142, 2011.

[5] Liu Bingshuang, S. Berg, J. Li, Wei Tao, Zhang Chao, and Han Xinhui. The store-and-flood distributed reflective denial of service attack. *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, pages 1 – 8, 2014.

[6] Frank Denis and libsodium documentation contributors. Ed25519 to curve25519. `https://libsodium.gitbook.io/doc/advanced/ed25519-curve25519`, August 2014. Accessed: 14 April 2022.

[7] DigitalOcean. Droplet pricing. `https://www.digitalocean.com/pricing/droplets`. Accessed: 26 May 2022.

[8] Hetzner Online GmbH. Dedicated root server hosting. `https://www.hetzner.com/dedicated-rootserver`. Accessed: 26 May 2022.

[9] Guillaume Pierre Guido Urdaneta and Maarten Van Steen. A survey of dht security techniques. *ACM Computing Surveys*, 43(2), 2011.

[10] Masaki Kondo, Shoichi Saito, Kiyohisa Ishiguro, Hiroyuki Tanaka, and Hiroshi Matsuo. Bifrost : A novel anonymous communication system with dht. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 324–329, 2009.

[11] Andrew Loewenstern and Arvid Norberg. Dht protocol. `http://bittorrent.org/beps/bep_0005.html`, January 2008. Accessed: 14 April 2022.

[12] Giancarlo Ruffo Luca Maria Aiello, Marco Milanesio and Rossano Schifanella. Tempering kademlia with a robust identity based system. *2008 Eighth International Conference on Peer-to-Peer Computing*, pages 30 – 39, 2008.

[13] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. `http://www.scs.stanford.edu/~dm/home/papers/kpos.pdf`. Pre-proceedings version. Accessed: 5 July 2022.

[14] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2429:53–65, 2002.

[15] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, February 2008. Accessed: 26 February 2022.

[16] Arvid Norberg. Dht security extension. `https://www.bittorrent.org/beps/bep_0042.html`, January 2014. Accessed: 1 March 2022.

[17] Jarkko Oikarinen and Darren Reed. Internet relay chat protocol. RFC 1459, RFC Editor, May 1993. `http://www.rfc-editor.org/rfc/rfc1459.txt`.

[18] Dimitry O. Photo. Icq: 20 years is no limit! `https://medium.com/@Dimitryophoto/icq-20-years-is-no-limit-8734e1eea8ea`, November 2016. Accessed: 26 February 2022.

[19] Bitcoin Project. Block chain - bitcoin developer guides. `https://developer.bitcoin.org/devguide/block_chain.html`. Accessed: 26 February 2022.

[20] The XLattice Project. Kademlia: A design specification. `http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html`. Accessed: 5 July 2022.

[21] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[22] Kyuyong Shin and Douglas S. Reeves. Winnowing: Protecting p2p systems against pollution through cooperative index filtering. *Journal of Network and Computer Applications*, 35(1):72 – 84, 2012.

[23] Bruno Spori. Implementation of the kademlia distributed hash table. *Master's thesis, Swiss Federal Institute of Technology Zurich*, 2006.

[24] Moritz Steiner and Ernst W. Biersack. Crawling azureus. *Research report RR-08-223*, 2008.

[25] Moritz Steiner, Ernst W Biersack, and Taoufik En-Najjary. Actively monitoring peers in kad. In *IPTPS 2007, 6th International Workshop on Peer-to-Peer Systems, February 26-27, 2007, Bellevue, USA*, 2007.

[26] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[27] Tom Van Vleck. The history of electronic mail. `https://www.multicians.org/thvv/mail-history.html`, February 2001. Accessed: 23 January 2022.

[28] Liang Wang and Jussi Kangasharju. Measuring large-scale distributed systems: case of bittorrent mainline dht. *2013 IEEE Thirteenth International Conference on Peer-to-Peer Computing (P2P)*, pages 1 – 10, 2013.

[29] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin, and Daniel J. Bernstein. *Curve25519: New Diffie-Hellman Speed Records*, pages 207 – 228. 2006.