Consistent Authentication in Distributed Networks

Niklas Lindskog
adi09noh@student.lu.se

Department of Electrical and Information Technology
Lund University

Advisors:
Martin Hell, LTH
&
Christoffer Jerkeby, Ericsson

April 11, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

In a time where peer-to-peer networks, often with previously unconnected devices, are increasing in relevance, new storage solutions are needed. Storage can no longer rely on a single central entity but rather needs to depend on the resources of the entire network. Such a solution is the distributed hash table (DHT) which allows distributed storage of resources, ensuring redundancy and availability of resources. Common DHT implementations have however been found to be susceptible to several attacks and therefore not suitable for security-critical data. To enable a wider use of DHT, a strengthening which can be easily implemented in existing DHT implementations have to be found.

In this thesis, the security of the Kademlia DHT, present in the Ericsson developed framework Calvin, was tested by performing a series of well-known attacks against an existing implementation. From the vulnerabilities found in these tests, security enhancements based on authentication were designed. All new functionality was designed to interfere with the original implementation as little as possible. The Kademlia DHT was strengthened with provable identities, cryptographically signed messages and a certificate distribution scheme. All of this was built on a public key infrastructure having an out-of-band certificate authority. The security enhancements were shown to both hamper known attacks and prevent outsiders from retrieving any information from the DHT. However, overhead and more complex computation were introduced into the system with the security enhancements. Further research is needed to determine if very computationally limited devices can participate or if additional functionality is needed to facilitate this.

Keywords: Authentication, Distributed Hash Table, Kademlia, Distributed Networking, Network Security, Calvin, Cryptography

Acknowledgements

I would like to sincerely thank Christoffer Jerkeby for his knowledge, ideas and for his never-ending enthusiasm. I would also like to thank Martin Hell with colleagues at Lund University for giving me the thorough understanding of computer security and cryptography needed for this thesis. Finally, I would like to thank my beloved wife Amanda and my kids for their support during my entire time of studies.

Table of Contents

1 Introduction **1**

1.1 Goal . . . 1
1.2 Delimitations . . . 2
1.3 Related work . . . 2
1.4 Outline . . . 2

2 Theory **5**

2.1 Distributed networks . . . 5
2.2 Distributed Hash Tables . . . 5
2.3 Authentication . . . 6
2.4 Kademlia . . . 7

3 Tools & Methodology **15**

3.1 Calvin . . . 15
3.2 Test-driven development . . . 16

4 Security evaluation of the DHT **17**

4.1 Identifying weaknesses . . . 17
4.2 Testing the security . . . 18
4.3 Security evaluation - Conclusion . . . 21

5 Designing a security-enhanced DHT **23**

5.1 Enable node authentication . . . 23
5.2 Securing the communication . . . 24

6 Evaluation of the security-enhanced DHT **27**

6.1 Mitigated vulnerabilities . . . 27
6.2 Attacking the security-enhanced DHT . . . 28
6.3 Overhead and increased computational load . . . 28
6.4 Evaluation . . . 29

7 Discussion **31**

7.1 Implications for DHT usage . . . 31
7.2 Adaptations to enable global availability . . . 31

7.3 Future work 32

8 Conclusion **33**

References **35**

Appendix A Network illustrations **37**

Appendix B Changes in communication flow due to security enhancements **41**

Appendix C Test case and attack methods **45**

List of Figures

2.1 A graphical representation of the identifier space in Kademlia. Circles represent nodes and triangles represent resources. 8
2.2 A graphic representation of a routing table in the Kademlia DHT. . . 9
2.3 A bucket split in Kademlia. 10
2.4 The node lookup procedure in Kademlia. 10
2.5 The value lookup procedure in Kademlia. 11
2.6 A graphical representation of the identifier space in Kademlia after a Sybil attack. The black node represent the node performing the attack and the grey nodes represents the Sybils. 12
2.7 The procedure for the Node insertion attack. 13

A.1 The topology of a normally functioning DHT network consisting of six nodes. 38
A.2 The topology of a DHT network consisting of six nodes after a successful Eclipse attack. The attacking node operates from port 57748. 39
A.3 The topology of a DHT network consisting of five nodes after a successful poisoning attack. The attacking node operates from port 55357. 40

List of Tables

4.1 Results of Sybil and Node insertion attacks 20
4.2 Results of Eclipse and Poisoning attacks 20

6.1 Comparing message lengths in non-secure and security enhanced Kademlia 29
6.2 Comparing average time for completing signing and signature verification 30

Abbreviations

BSI	Bundesamt für Sicherheit in der Informationstechnik
CA	Certificate Authority
CSR	Certificate Signing Request
DDoS	Distributed Denial of Service
DHT	Distributed Hash Table
ECC	Elliptic Curve Cryptography
IoT	Internet of Things
NAT	Network Address Translation
OSPF	Open Shortest Path First
RIP	Routing Information Protocol
RPC	Remote Procedure Call
TLS	Transport Layer Security
UDP	User Datagram Protocol
XOR	eXclusive OR

Chapter 1

Introduction

Since the late 1900s, several network solutions not relying on a centralized hierarchy has seen the light of day. Examples of such networks are ad-hoc networks[1] or the well-known Internet of Things [2]. These types of distributed networks cannot rely on single nodes* storing critical information as nodes can fail and networks partition at any given point. To counter this, distributed storage solutions has been developed to increase redundancy and availability.

One of these solutions is the distributed hash table (DHT) where resources are mapped to keys and stored on several nodes. In a DHT, the storing nodes are responsible for providing the resource upon receiving a request for that particular key. Apart from providing redundant storage, the DHT also contributes routing mechanisms simplifying the finding of stored resources. While several different paradigms exist for these routing protocols in divergent DHT implementations, they all have the feature that a node only needs to know a fraction of the network to find all values. The resources stored by previously unknown nodes are found either by forwarding messages or by doing iterative lookups.

However, while the DHT solves several storage issues in distributed networks, they also contribute with new security concerns. All nodes need to trust the resources which are returned for a certain key, without having any way of determining whether it is counterfeit or not. This issue becomes of particular interest in a context where security-critical data is to be shared between several nodes in a distributed network.

1.1 Goal

The goal of this thesis work is in the first phase to determine how vulnerable a regular implementation of a DHT† is. If potential security weaknesses are detected, well-documented attacks exploiting these are to be performed and the impact on the network documented. In the second phase, node authentication and other security functionality strengthening the DHT are to be implemented. This is intended to solve or mitigate the vulnerabilities found in the first phase. Finally, when the security features have been implemented, the chosen solution will be evaluated in terms of security and performance.

*A node is, in this instance, a participant in a distributed network

†The DHT type used in this thesis is Kademlia DHT

1.1.1 Research Questions

- Is it possible to establish authentication between nodes without abandoning the distributed properties of the network?
- Is it possible to safely expose security-critical data in a DHT?
- How is performance affected by adding authentication and improving security in the communication protocol of a DHT?

1.2 Delimitations

This thesis work will focus on protecting the communication between the members of a DHT network and the integrity of the network. The thesis work will not directly investigate the validity of the data stored in the DHT. It is therefore presumed that members of the network are not able to overwrite data stored by others in the DHT.

1.3 Related work

Extensive research has been performed in the field of attacking Kademlia DHT implementations and proposing countermeasures. The file-sharing "Kad network" (which uses the Kademlia DHT) has been the subject of several papers where vulnerabilities have been identified and attacks been performed exploiting these [3] [4].

Several attempts have also been made to create a concept securing the Kademlia DHT. A generally applicable solution is the S/Kademlia [5] where several features to secure the original protocol are proposed. The goal of [5] is similar to the goal of this thesis, but focuses on high-level concepts rather than how the protocol could be implemented. Some of the implemented features in this thesis, such as assigning secure node identifiers, originates from S/Kademlia.

An actual implementation of a security enhanced Kademlia, called "SKad" has been proposed in [6]. One of the main focus points in [6] is how to efficiently implement secure communication in a Kademlia DHT. While having a similar scope as this thesis, the solution provided is built for external communication with servers outside the DHT. This makes most of the solution proposed in [6] inapplicable for the security work in this thesis.

1.4 Outline

Chapter 2 is dedicated to the theoretical background of the thesis, giving the reader a rudimentary understanding of the concepts involved.
Chapter 3 describes the test environment and development strategies for this thesis work.
Chapter 4 examines the weaknesses found in the DHT implementation. The implementation of the well-known attacks are also evaluated in this chapter.
Chapter 5 is dedicated to the strengthening of the DHT and the security-enhancing concepts are described in detail.

Chapter 6 consists of an evaluation of the security-enhanced DHT in terms of both security and performance.
Chapter 7 handles discussion regarding usability and future work.
Chapter 8 summarizes and concludes this thesis work.
Appendix A contains illustrations of network topology before and after attacks.
Appendix B elucidates the changes in communication flow due to security enhancements.
Appendix C contains code extracts of attack methods and test cases.

Chapter 2

Theory

2.1 Distributed networks

A distributed network is a general term for networks in which data processing, storage and computation is spread out over several network members. These sorts of networks are beneficial when many members of a network are computationally weak or only sporadically available. A distributed topology, as opposed to a decentralized topology, does not have any central nodes which connects the network. Instead it relies entirely on individual peer-to-peer connection between the nodes. This avoids single points of failure in the system and creates redundancy, as a node can be reached through several different paths.

2.2 Distributed Hash Tables

A DHT is a distributed storage system based on the concept of the well-known data structure. As in a regular hash table, a key is mapped to a value, but instead of storing the value on disk, it is stored in the network. Several different nodes are responsible for storing a certain value and to be able to provide it when asked to. By storing data in several nodes, the DHT ensures availability of content in a network, even if several members should leave or fail.

To be able to keep track of all nodes participating in the distributed hash table, it also contains a built-in lookup protocol, which has similarities with routing protocols such as RIP[7] and OSPF[8]. The properties of the lookup protocol is implementation based and differs vastly between different DHT types.

There are several existing DHT types, all with same key-value mapping concept, but with very different concepts in terms of storage redundancy, routing and identification. Chord [9] and Kademlia are the two most common implementations and the latter was used during this thesis work. Kademlia will be described further in Section 2.4.

2.3 Authentication

2.3.1 Asymmetric encryption

Asymmetric encryption is, in contrast to its symmetric counterpart, not dependent on both parties sharing the same key. Asymmetric encryption is performed with two different but corresponding keys, where one is used to encrypt the message and the other to decrypt it. Applying the same key twice will not result in decrypting the message. Due to this property, the big upside with asymmetric encryption is that only one key needs to be secret while the other can be publicly distributed. The non-secret key is usually referred to as "public key" and its secret equivalent as "private key". This removes the problem of sharing a key between two parties, as both can distribute their keys publicly. A message encrypted by the public key can only be decrypted by the holder of the private key. A message encrypted by the private key can be decrypted by anyone. By being able to decrypt the message with the public key, the message can be proven to originate from the owner of the private key.

Asymmetric cryptography can be and is often used to exchange symmetric keys which are used during the rest of the communication session. E.g. the security protocol TLS contains this functionality [10]. The reason for not using asymmetric encryption algorithms for all communication is that they are more computationally challenging than their symmetric equivalents. Therefore they are best used for encrypting individual messages rather than an entire session.

2.3.2 Certificates and Public Key Infrastructures

To ensure the identity of someone solely asymmetric encryption is not enough. As anyone can create a public-private key pair, this is not sufficient to establish an identity. To do so one must also have a confirmation of the identity for the owner of the key pair. This can be done by consulting a trusted third party, known as a CA.

A certificate consists, in its simplest form, of four parts; information regarding the owner of the certificate, the owner's public key, information regarding the CA and a signature (see Section 2.3.4) from the CA. This certificate can be distributed to others to prove one's identity and to distribute the public key.

As it, in large systems, might not feasible to have a single CA issuing all certificates, there can exist intermediate CAs which are given the right to issue certificates on behalf of the main CA. These form a chain of trust with the issued certificate at the bottom, the intermediate in between and a trusted third party top in the chain. The top of a trust chain is referred to as a CA-root. As long as one trusts the CA-root, it is also possible to trust all certificates issued by the CA-root and subsequently any certificate with the CA-root at the end of its certificate chain. Such a system, with a tree structure of certificates originating from the CA-root, is known as a PKI.

2.3.3 Fingerprints

A fingerprint is a general term for functions mapping arbitrary large data to a fixed-width bitstring. This can e.g. be performed by a hashing algorithm. In this thesis fingerprints will be used solely for identifying certificates. If the fingerprint for a certificate is created by a cryptographically secure hash algorithm, a unique* identifier for that certificate is obtained.

2.3.4 Signatures

To cryptographically ensure that a message has been sent by the claimed sender, a signature can be used. A signature is essentially a part of a message which is hashed, asymmetrically encrypted and sent together with the message. If the receiver knows the public key of the sender, e.g. by having the sender's certificate, it can decrypt the signature. When this is done the appropriate part of the message is hashed. If it is found to be equal to the decrypted signature, the sender is declared to be legitimate.

2.4 Kademlia

The paper describing the Kademlia DHT was published in 2002 [11] and introduced several new concepts in the world of DHT. Most significantly, the routing protocol stood in contrast to many of its DHT predecessors, such as Chord. No messages are relayed in the system, instead each node requesting information will try to find the sought-for node/resource itself by doing iterative lookups. The measuring of distance is another aspect in which Kademlia took a different path than its competitors. Instead of measuring distance by counting "nodes in between" it is calculated by XOR-metric which will be explained in Section 2.4.1.

Kademlia is highly scalable, mainly due to the network-wide parameters "k" and "α". The k-parameter determines how much redundancy the net should provide and how many other nodes a single node should be aware of in the network. The ideal value is dependent on desired network size and the expected churn among the nodes participating in the network. α, on the other hand, determines how many answers one should collect when asking for a resource. Upon receiving these answers, the node will make a majority decision, i.e. deem the most common answer to be the correct one†. A high α results in impacted performance, as one has to await a higher number of answers but in return gives increased confidence that the received answer is correct.

Kademlia can be found in several real-world applications, mainly in file sharing networks. E.g. well-known BitTorrent clients uses a Kademlia implementation called Mainline DHT [13].

---

*Unique in this context is defined as something having negligible risk of having a duplicate

†Majority decisions are not a part of the original Kademlia protocol [11]. It is however implemented in [12], the Kademlia implementation used in this thesis, and is therefore here considered as a part of the Kademlia protocol.

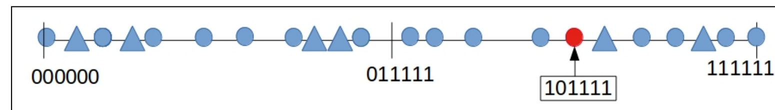

Figure 2.1: A graphical representation of the identifier space in Kademlia. Circles represent nodes and triangles represent resources.

2.4.1 Identifiers

Each node and each resource in Kademlia has a 160-bit identifier. A node chooses its own identifier when entering the network, typically a pseudo-random number. As the resources have to be identifiable, they also possess an identifier of the same type as the nodes. Their identifier is determined by e.g. the hash of the resource's name or a keyword describing the content. Due to the size of the identifier space‡, the risk of identifier collisions is negligible as long as they are somewhat randomly generated.

To determine how "far" two nodes are from each other, XOR is applied to the two identifiers and the result is set as the distance. Two nodes with a long common prefix is considered to be "close" to each other regardless of their physical location. Measuring distance with XOR results in each node having a constant and absolute distance to every other node, regardless of nodes joining and leaving. Refer to Fig. 2.1 for a graphical representation§ of the identifier space.

2.4.2 Storing / Retrieving data

As mentioned in Section 2.2, data is stored in several nodes to ensure availability and correctness of a resource. The parameter k is used to determine the least amount of nodes storing the same resource.

When a resource is to be stored in the network, the owner of the resource performs a node lookup, described in Section 2.4.2.1, to find the k nodes with least distance to the identifier of the resource. These k nodes receive a request to store the resource locally and be able to provide it upon request from the DHT. A similar process is performed when retrieving a resource from the DHT. The asking node performs a value lookup to find α of the k storing the resource. Upon receiving enough answers the node will determine the value of the resource through a majority decision.

‡The identifier space consist of 2^{160} possible identifiers

§Due to the properties of XOR, the identifier space is not actually a line but rather a system where each node has its own line. The reader should note that the illustrations is to create an understanding of concept rather than giving a correct graphical representation of the identifier space.

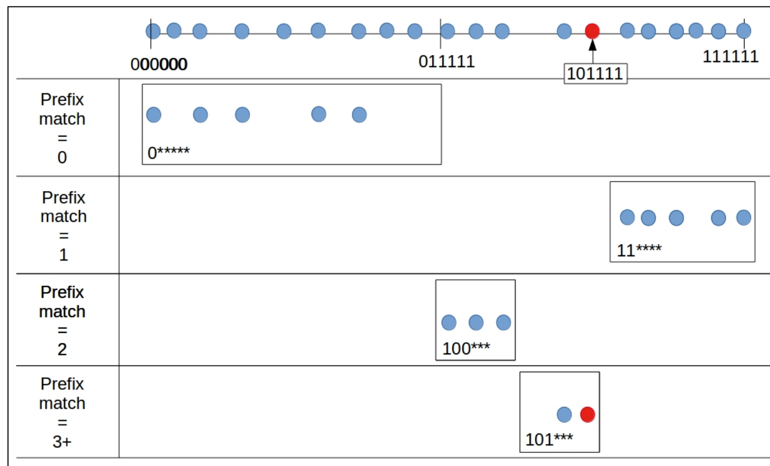

Figure 2.2: A graphic representation of a routing table in the Kademlia DHT.

2.4.2.1 Routing tables

Each node has a routing table which consists of several "buckets", as shown in Fig. 2.2. A bucket is a container in which information about nodes, having a certain identifier-prefix in their identifier, is collected. A node can only be a member of one bucket and a bucket may consist of no more than k nodes. A bucket containing the node's own identifier can be split (Fig. 2.3) into two separate buckets when the number of known nodes exceeds k. This procedure will ensure the node knowing more nodes among those close to itself than those far away. If a bucket does not contain the node's own identifier, it cannot be split.

Nodes are added to buckets upon discovery, i.e. when a node with a previously unknown identifier is found. If a bucket is full and cannot be split, an old node must be removed in order for a new node to be added to the bucket. This is done by pinging (see Section 2.4.3.3) the least-recently seen node in order to refresh the table. If the ping is not answered, the node is replaced by the new node. If the old node is present in the network, the new node is placed in a replacement buffer.

2.4.3 Communication protocol

2.4.3.1 Node lookup

To find the IP address and port for a node not present in the routing table, a find-node request will be sent. The request is sent to the α first nodes in the bucket with longest prefix match with the sought-after node. These will either return the IP address and port of the sought-after node or return the k nodes in their routing table with the longest matching prefix. If the correct node is not found during the first attempt, the node will query the α closest nodes retrieved from the last round of find-node requests. This is done until the node is found or no closer nodes exists. (see Fig. 2.4)

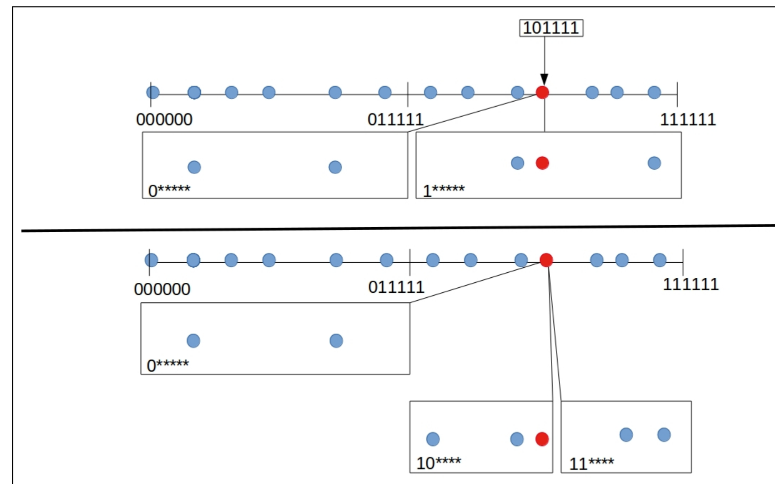

Figure 2.3: A bucket split in Kademlia.

The red node has two buckets in the upper figure. It splits the right bucket to increase the knowledge of nearby nodes. The left bucket cannot be split since it does not contain the owners identifier. The lower figure shows the buckets after the split.

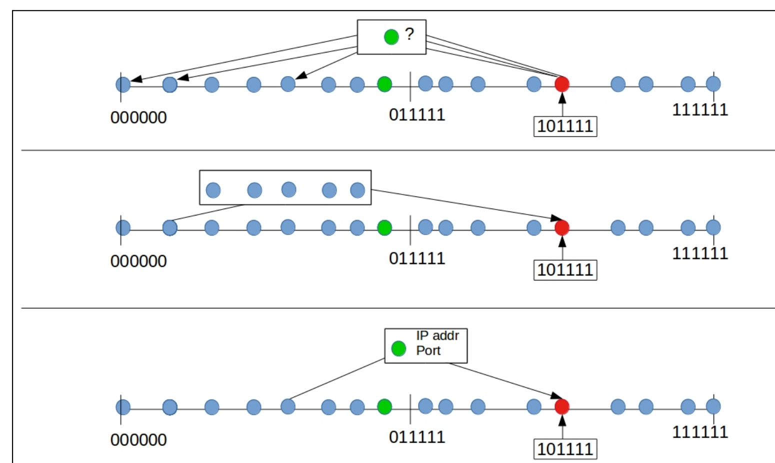

Figure 2.4: The node lookup procedure in Kademlia.

In the upper figure, the alpha closest nodes known by the red node is sent a node request. If an asked node does not know the green node, it answers with k closest nodes it knows, as shown in the middle figure. Otherwise it answers with the address and port of the green node, as shown in the lower figure.

2.4.3.2 Value lookup

The procedure in which to find a resource is in essence equal to the node lookup. However, instead of a find-node request, a find-value request is sent. When a node which is storing the sought-for resource receive such a request, it returns the actual resource rather than just pointing out where to find it. The node sending the request takes a majority decision on the received values, deciding the most common return value as the correct one. (see Fig. 2.5)

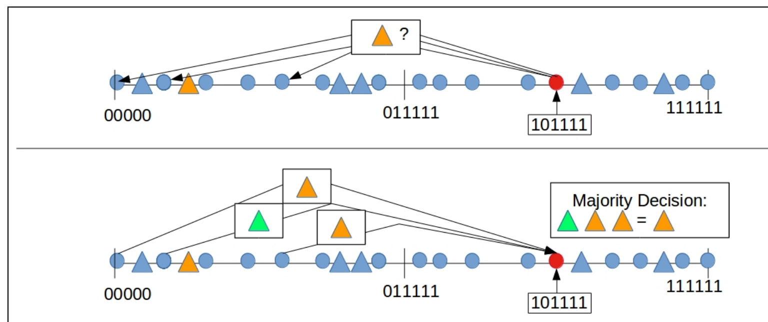

Figure 2.5: The value lookup procedure in Kademlia.

In the upper figure, the α closest nodes known by the red node is sent a value request. In the lower figure, the node receives α answers and takes a majority decision to decide the correct value.

2.4.3.3 Ping

The purpose of the ping message is to check if a known node is responsive. This is used e.g. when nodes present in the routing table has not been seen for a period of time.

2.4.3.4 Store

The store message is sent by the node wishing to store a resource in the network. The message contains a key and a value which is stored by the receiver of the message. Depending on the implementation, a store may be valid for specific period of time or removed explicitly.

2.4.4 Joining / Leaving

Joining a Kademlia DHT requires the joining node to know at least one node already participating in the network. This node is referred to as the bootstrap node. The bootstrap node provides the joining node with information regarding the existing nodes in the network and helps populate its routing table. However, bootstrapping is not used in the Kademlia implementation used in this thesis. The reader is therefore referred to [11] for more detailed information regarding

the bootstrapping procedure. The joining procedure used in this thesis will be described in Section 3.1.3.

A node does not send any specific "leave" message but will instead be removed gradually from other node's buckets as it stops responding.

2.4.5 Security

There is no functionality in Kademlia aimed solely at mitigating attacks or stopping nodes from misbehaving. However, since old nodes are not removed from routing tables as long as they are responding, the nodes are somewhat protected from injection of vast amounts of new nodes in the system. Furthermore, if a few nodes fail or stop responding, the redundant storage structure and multiple lookup paths decrease the impact on the network robustness as several nodes can provide the same information.

2.4.6 Known attacks

2.4.6.1 Sybil

As there is nothing binding a node to a specific identity, it is not possible to determine if two different identifiers are tied to the same node. Not even the IP-address can be used to differentiate two nodes as NAT [14] may cause several nodes to share IPv4-address. This enables a node to claim multiple identities and thereby gaining a disproportional amount of influence in the network. This is referred to as a Sybil attack [15] and an additional identity of a node is called a Sybil.

Sybils cause a situation where redundancy can be eliminated as the same node can claim several identifiers (see Fig. 2.6) in a particular area. By doing so it can gain control over resources, refer to Section 2.4.6.2 for more details. It can also enable eavesdropping, as one can create Sybils with identifiers close to interesting resources and record all incoming requests for these resources.

The Sybil attack is in itself passive, as the attacked network will not function differently in any way as long as the Sybils behave like legitimate nodes. It can however be a preparation for an active attack, as a large amount of influence in a network is useful when starting to behave maliciously.

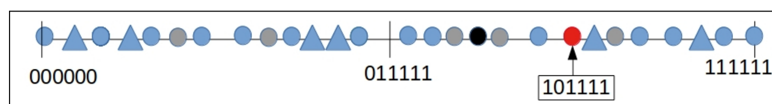

Figure 2.6: A graphical representation of the identifier space in Kademlia after a Sybil attack. The black node represent the node performing the attack and the grey nodes represents the Sybils.

2.4.6.2 Node insertion

Node insertion [3] is an attack which can be categorized as censorship. The attack aims at denying certain resources in the network to be retrieved correctly.

The attacker inserts several Sybils with identifiers very close to the targeted resource. All find-value requests received by the Sybils will be answered with the same counterfeit resource, trying to get the requester to accept this as the correct answer. Due to the fact that a majority decision is made upon retrieval of stored data, the attacker will only succeed if he receives more than half of the requests. By introducing these Sybils to as many nodes as possible in the network, the chance of getting value requests increases. See Fig. 2.7 for a graphical representation of the attack.

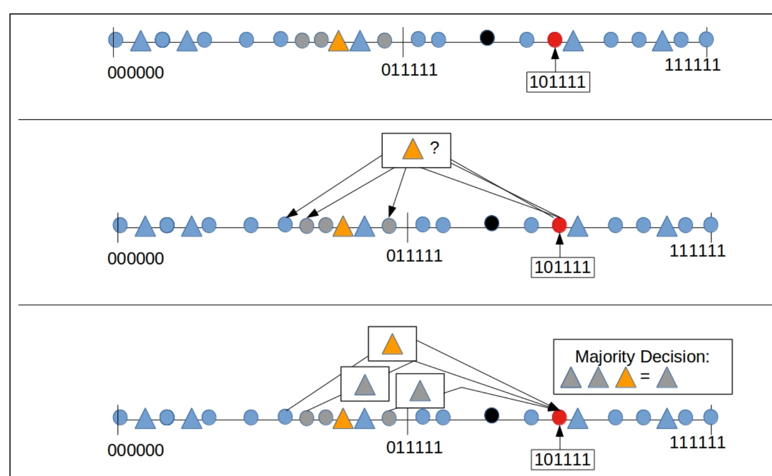

Figure 2.7: The procedure for the Node insertion attack.

In the upper figure, the black node has created three Sybils and placed them close to the yellow resource. These receive a majority of the value requests sent in the middle figure. In the lower figure, the Sybils are able to forge the resource and thereby succeeding with the attack.

2.4.6.3 Routing table poisoning

A very offensive attack strategy is to, instead of just creating new identifiers, impersonate legitimate nodes by hijacking their identifiers. This can be done in two ways; passively, by claiming to be the searched-for node when receiving a node lookup or actively, by sending out messages impersonating actual nodes. The former attack is not dependent on implementation as it is not possible to determine the owner of a previously unknown identifier. The latter is enabled due to a specific but common feature in DHTs. Nodes are often able to change their IP address and port dynamically and simply inform the network when they

are reachable in a new location. This feature increases the flexibility for nodes in the network but also enables an attacker to poison their routing tables by simply updating the routing information for legitimate nodes to the attacker's address [4].

The ultimate goal of the attack is to disrupt network communication entirely. For lasting effect, a large portion of the network must be attacked simultaneously, otherwise the communication of legitimate nodes counter the falsified routing information. The attacking node must also be able to handle vast amounts of network traffic to succeed with the attack as it otherwise very well may carry out a DDoS attack on itself.

2.4.6.4 Eclipse

Eclipse [16] is an umbrella term for several attacks with the goal of isolating one or a few nodes from the rest of the network. The attack referred to as Eclipse attack in this thesis is a combination of routing table poisoning and censorship, where the attacker aims at becoming a man-in-the-middle between one node and the rest of the of the network. The reason for choosing this version of the Eclipse attack is the research in [17]. In [17] the use of poisoning (in the paper labeled "Hijack") as Eclipse attack method is proved to be more efficient than other alternatives.

Similar to the routing table poisoning attack, the attacker will claim to possess all identifiers in the network but will only do so towards one or a few nodes. If the attack is successful, the eclipsed node(s) will only be able to contact the attacker and will therefore only receive attacker-controlled values when sending value requests. The attacker will, to all other nodes, only claim to own the eclipsed nodes identity and thereafter act legitimately.

This version of the Eclipse attack also relies on the ability to change IP address and port dynamically.

Chapter 3

Tools & Methodology

3.1 Calvin

3.1.1 General concept

Calvin[18] is an Ericsson-developed framework combining the functionality of IoT and cloud computing. In short terms Calvin provides an environment in which applications are able to run on several devices at once.

A device participating in Calvin can be anything from a smartphone to a lamp and they all have in common that they consist of one or several "runtimes". In each runtime, parts of applications, called "actors", may perform certain tasks. These tasks are defined from a set of runtime capabilities, used to inform the types of operation which may be performed on the runtime. By spreading out actors on several runtimes, intra- or inter-device, cloud functionality in an IoT environment is achieved.

Calvin uses an implementation of the Kademlia DHT to store information regarding actors and runtime capabilities. The DHT can e.g. be used to find where a certain actor is running or find out what capabilities a specific runtime possesses. Needless to say, crucial functionality in Calvin depends on correctly retrieving information from the DHT. The information stored in the DHT is thereby security-critical for Calvin.

The Kademlia implementation will be introduced further in Section 3.1.3.

3.1.2 Test environment

Due to Calvin being under development during this thesis work, it was found to be out of scope to perform any tests on actual IoT devices. Instead, the DHT functionality were extracted and tested independently.

To simplify control of the DHT nodes and to shorten the running time of the test cases, all tests were carried out on a single machine.* All nodes in the DHT used the local loopback interface to communicate, simulating actual network traffic.

*Machine specifications: Ubuntu 14.04, 16 GB RAM, 8 Core 3.5 GHz processor

3.1.3 Kademlia in Calvin

Calvin uses the Kademlia implementation created in Python by Brian Muller[12]. The original version follows the Kademlia specification with the exception that it only takes one unanswered message for a node to get removed from routing tables. The communication between nodes in the DHT is built on UDP messages, containing either a request or a response. The requests consist of a RPC indicating which action it wishes the responding node to perform. An example of the communication flow can be found in Appendix B.

To further fit Calvin's needs and usage of DHT, the joining procedure has been changed and differs vastly from the procedure described in Section 2.4.4. Instead of using an existing node to bootstrap from, all nodes participating in the network are registered in a multicast group. A new node, wishing to join the network sends a join announcement to the multicast group which will reach all participants.[†] All nodes receiving this announcement responds with their IP-address and port. The joining nodes sends out a ping message to all responders and adds them to its routing table upon receiving an answer.

3.2 Test-driven development

As it, previous to the start of this master thesis, existed general purpose test cases for the Calvin DHT, these became an excellent starting point for creating the attack scenarios. This led to the decision to apply test-driven development as technique for developing both the attack scenarios and later also the security enhancements for the DHT implementation. This type of development favors small iterations which are all evaluated by a test suite. All functionality not tested by the test suite is considered unnecessary which increases the possibility of the code becoming both simpler and cleaner than in other development techniques [19].

[†]The participants have to be a part of the same domain as the joining node to reach the announcement. In thesis however, all nodes are considered to be in the same domain.

Chapter 4

Security evaluation of the DHT

4.1 Identifying weaknesses

To find weaknesses in the applied Kademlia implementation, papers attacking Kademlia implementations such as [3][4], were used as inspiration. By using the concluded exploits in these papers, the source code for the implementation[12] in this thesis was scanned for similar weaknesses. The three weaknesses found to be most exploitable are presented below.

4.1.1 Multiple identities

As mentioned in Section 2.4.1, all nodes are identified by a unique 160-bit identifier. The identifier can either be randomly generated or set to a specific value. Other than this, there is nothing identifying a node, as an IP address is not enough to provide a unique identifier. As there is no way of proving whether an identity is valid or not, a node can easily send out messages with new identifiers and thereby create several identities. This enables the use of Sybils which can be used in attacks to faster gain influence in the network and thus increasing attack efficiency.

4.1.2 Non-random identifiers

As a node can create new identifiers at any time and also decide the value of the identifier, censorship can easily be accomplished. When a resource is posted, an evil node can announce new Sybil identities very close to the resource. When a node asks for the value, it takes a majority decision if several different values are received. Therefore, α well-established Sybils can censor a resource for the entire network as long as they answer with a unilateral but false value.

4.1.3 Dynamically changing IP/Port

Due to the lack of provable identities, a node can impersonate other nodes by claiming they have switched address and are now present at a destination controlled by the attacker. For each message, the receiving node will update the node's address. This feature enables anyone to be impersonated, no matter if they are known in the network on forehand or not.

4.1.3.1 Active impersonation

An active impersonation can be performed via a spoofed message sent from the attacker claiming to be a legitimate node. The receiver will update the claimed senders address in its routing table to the address of the attacker.

4.1.3.2 Passive impersonation

A passive impersonation is carried out by responding to find-node requests with nodes having existing identifiers but falsified addresses. The receiver will contact the nodes on their new falsified addresses and will update its routing table when receiving a response.

4.2 Testing the security

4.2.1 Attack scenarios

4.2.1.1 General scenario

The tests begins with a network consisting of 16 legitimate nodes. The parameters are set to $k = 5$ and $\alpha = 3$ to prevent all nodes from knowing all other nodes in the network. When the DHT is stable, i.e. when all nodes have started and joined the DHT, three nodes sends out a store request. When the store requests has been confirmed, one of the nodes which has not sent a store request will turn evil. Turning evil is in this instance defined as starting to break Kademlia protocol in order to execute a particular attack.

The attacking node gets 15 seconds to execute the attack before the three nodes storing values try to fetch the original value from the DHT. The attacking node will also create a predetermined number of Sybils in order to increase the efficiency of the attack. When the 15 seconds have passed, the storing nodes send out value requests according to procedure (i.e. to the α closest nodes they are aware of). All value requests reaching the attacker will be answered with a forged value. Lastly the storing nodes evaluates if the received value differs from the stored value.

4.2.1.2 Sybil attack scenario

When executing the Sybil attack, the attacking node will start to introduce 30 different Sybils in the network. The Sybils are given random identifiers and will therefore likely distribute somewhat evenly in the identifier space. The attacking node responds with the identifiers of its Sybils when answering find-node requests. By including the Sybils in find node-responses, the new identities will spread throughout the network, as legitimate nodes without hesitation add these to their routing tables.

The attack is considered successful if all storing nodes receive a forged value.

4.2.1.3 Node insertion attack scenario

The node insertion scenario is very similar to a Sybil attack but with the important exception that the Sybils introduced are placed very close to the stored resources in the network. By placing α Sybils with identifiers having very long matching prefixes with the resources (152+ bits), it is very likely that the Sybils will get a majority of the find-value request and therefore be able to forge the response.

The attack is considered successful if all storing nodes receive a forged value.

4.2.1.4 Eclipse attack scenario

The eclipse attack starts with the attacking node selecting one of the storing nodes in the network which will become the victim of the attack. To the eclipsed node, it will claim to own all identifiers in the network. To all other nodes it will only claim to own the identifier of the eclipsed node. This gives a scenario where, if the attack is successful, the evil node becomes a man-in-the-middle and isolates the eclipsed node from the rest of the network.

The attack is considered successful if the eclipsed node receives a forged value and it is not aware of any legitimate node at the end of the test. An example of the topology after a successful Eclipse attack can be seen in Fig. A.2.

4.2.1.5 Poisoning attack scenario

In the poisoning scenario, the attacking node will claim to own the identifiers for all nodes in the network by both active and passive impersonation. If successful, all nodes in the network will only be able to contact the attacking node, which has the ability to cease all communications or to return forged information. The attack is carried out simultaneously on all nodes in the network to prevent them from countering the attack by sending legitimate messages.

The attack is considered successful if all storing nodes receive a forged value and if none of the nodes are aware of any legitimate node at the end of the test. An example of the topology after a successful poisoning attack can be seen in Fig. A.3.

4.2.2 Results

4.2.2.1 Sybil

The success factor of a Sybil attack is a question of sheer numbers as the number of Sybils determines the likeliness of the success of the attack. Only if a Sybil is closer to a resource than one of the α closest legitimate nodes will it likely receive a find-value request. As the identifiers were randomly generated, the attack had a very varying success rate.

However, by only claiming multiple identities, the attacker managed to forge more than half of the value responses in the network (see Table 4.1). This gives an indication of the multiple-identities weakness being a serious threat to the security of the DHT.

Table 4.1: Results of Sybil and Node insertion attacks

	Sybil	Node Insertion
Attack vectors exploited	Multiple identities	Multiple identities Non-random identifiers
Number of Sybils	30	9
Results		
Forged values	97	150
Correct values	53	0
Test w. all values forged	26	50
% Successful attacks	52%	100%

4.2.2.2 Node insertion

In contrast to the Sybil attack, the node insertion attack was extremely effective in fulfilling its purpose. The inserted nodes, due to their closeness to the resources, were almost guaranteed to receive all value requests and thereby able to falsify the resources. As all resources were forged during this attack (see Table 4.1), it is proved that resources can easily be censored and forged by just claiming to own several identities close to a specific resource.

Table 4.2: Results of Eclipse and Poisoning attacks

	Eclipse	Poisoning
Attack vectors exploited	Multiple identities Active impersonation Passive impersonation	Multiple identities Active impersonation Passive impersonation
Number of Sybils	10	30
Results		
Forged values	50	145
Correct values	0	5
Tests w. all values forged	N/A*	47
Tests w. complete isolation	44	35
% Successful attacks	88%	70%

4.2.2.3 Eclipse

It was shown to be relatively easy to target a specific node and to be able to forge all values this node requests from the DHT (see Table 4.2). It can however be

*The goal of the Eclipse attack is only to forge a single value

harder to ensure complete isolation of the node over time, as joining nodes can gain knowledge of the nodes existence from the multicast. In the test network, where most nodes have already joined before the attack has started, the attack proved to be very efficient. The eclipsed node was completely isolated from the network in a majority of the tests and in the six remaining cases only knew a single legitimate node. Worth noticing is the two other storing nodes always receiving the correct value and thereby not detecting any anomalies in the network.

4.2.2.4 Poisoning

The poisoning attack were very ambitious as the attacking node needed to convince all nodes it owns every identifier in the network. This inevitably led to light overload of the attacking node as almost every message in the network were sent to it. A slight deterioration of attack efficiency could thus be seen in some of the tests. Certain nodes were able to keep several connections to legitimate nodes simply due to the attacking node not being able to respond to all messages. Despite this aggravating circumstance, the attack was generally very successful and took down the entire network in a majority of the tests (see Table 4.2).

4.3 Security evaluation - Conclusion

All of the attacks were possible to perform and reached their goal in a majority of the test cases. Especially the node insertion and Eclipse attacks were very successful which proves the possibility for a single node to compromise the functionality of the DHT while leaving most nodes untouched.

As the identified weaknesses were successfully exploited it is clear the current implementation is not suited for storing any sort of security-critical data. The performed attacks do have the multiple identity exploit in common, it should however be noted that the Eclipse and poisoning attack are able to function without the use of multiple identities but they will take longer time to perform. Therefore it is not a viable solution to only fix the identity exploit to stop all four attacks. In other words, to secure the DHT, all of the weaknesses must be addressed properly.

Chapter 5

Designing a security-enhanced DHT

5.1 Enable node authentication

5.1.1 Establishing authority

To be able to create a secure version of DHT, a trust concept must first be introduced into the DHT. Despite researching alternative authentication paradigms, e.g. using pre-shared keys or security tokens, no easily implemented solution for authentication not relying on a CA was found for distributed networks. Therefore, the security enhancement requires an out-of-band CA which can provide a signature trusted by all nodes in the network. All nodes will store the CA certificate upon joining the network. This is necessary to not introduce a single-point-of-failure * in the DHT. If all nodes store the CA certificate, the CA will only have to be present when nodes perform first-time joining of the network.

Before joining, each node must perform an authentication procedure. This starts with the node wishing to join the network providing a CSR to the CA. This request could also include device information regarding e.g. its type and serial number. The CA will either by human interaction or by a predetermined set of rules decide whether the node is allowed to enter the network or not. If the CA decides to allow it, the new node is issued a certificate signed by the CA.

5.1.2 Connecting the node identifiers to the certificate

One of the main problems in the original Kademlia implementation is the fact that it is very difficult to determine whether an identifier actually exist in the network. Nor is it possible to know if an identifier belongs to the address its messages are sent from. One suggestion would be to disallow all address updates for nodes, thereby mitigating e.g. the poisoning attack. But apart from limiting flexibility in the network, it does not protect against node insertion and Sybil-based attacks. Therefore, if the identifier is bound to a certificate, for which a signature can be provided, changing addresses dynamically is no longer an issue. Also, if the identifier cannot be known beforehand by a joining node, one cannot ensure closeness to a certain resource with intention of executing a node insertion attack.

*A single function which the entire system is dependent on to operate properly

The identifier is, as mentioned in Section 2.4.1, a 160-bit number. If this is set to the 160 least significant bits of the certificate's fingerprint, a forgeable identifier is created. A node claiming to own a certain identifier needs to be able to provide a correct signature for the certificate with the corresponding fingerprint. The risk for identifier collisions in a small network is negligible. One can therefore safely say that identifiers based on the certificate's fingerprint can be regarded as unique.

5.1.3 Distributing the certificates

Each node will, as mentioned previously, store the CA certificate in order to be able to identify valid CA signatures on other nodes' certificates. Each will also store all certificates of the nodes present in its routing table. It may also store certificates for nodes currently not present in the network to facilitate their return. The CA signature must be checked before accepting a node's certificate into storage.

All communication requires both nodes to have each other's certificate, which will be shown in Section 5.2. Thus, there must exist a certificate distribution mechanism which gives a joining node as many certificates as possible without relying on a single distributor.

By taking advantage of Calvin's bootstrapping mechanism, using multicast instead of conventional bootstrap-by-node, each node can provide their own certificate in the multicast response. The certificate is sent as a base64-encoded string and stored by the joining node, given that it was found to be valid and signed by an appropriate CA. The node which has sent the response is not accepted into the routing table at this point. It will need to provide a signature on the ping response to prove ownership of the corresponding private key. For obvious reasons, the joining node will also provide its own certificate and a signature to prove it. This is done by sending a ping with explicit certificate, explained further in Section 5.2.3.3.

When this is done, a joining node will have received several certificates from nodes present in the DHT. There is no guarantee all certificates will have been received as the joining node only listens to multicast responses for a limited period of time.

5.2 Securing the communication

5.2.1 Signatures and verification

As identities for each node have been established, the next step is to bind each message to an identity by adding a signature to the message. The receiver will check the signature with the public key of the certificate having a fingerprint corresponding to the sender's identifier. If the signature is valid, the packet is determined to be legitimate. If found to be correct, the receiving node will reply in the same manner, i.e. by signing the response message to confirm that the origin is indeed from the intended node. If the signature was not found to be valid, the node will instead reply with a signed NACK. This results in a communication protocol where a node not having a valid certificate is not able to communicate with the DHT, as all messages will be met with a NACK.

NACK is explained in Section 5.2.3.1 and the necessity of sending a NACK instead of not responding at all will be discussed in Section 5.2.4.

5.2.2 Replay protection

The signatures are in themselves not enough to determine the sender of a message. Therefore, each message must consist of something unique for the current conversation between two nodes. To solve this, each request is sent with a 64-bit random challenge, which is to be signed by the responding node and included in the response.

Due to the fact that the randomness is chosen by the sender, the request signature cannot solely depend on the challenge value. Signing only the challenge would allow a request to be resent by any node, enabling it to impersonate the sender. However, by adding the identifier of the recipient in the signature, the messages cannot be replayed to any other node than the one intended. I.e. an evil node will not be able to reuse the request for malicious purposes.

It should be noted that the replay protection does not protect against an attacker which is able to intercept messages it is not the receiver of.

5.2.3 New message types

5.2.3.1 Not acknowledged

A node, henceforth called B, receives a signed message from node A, whose certificate it does not possess. B must now inform A that it is not able to verify the signature. This is done by sending back a NACK message[†], informing A that B does not possess A's certificate and that any communication have to be preceded by A sending a ping with explicit certificate, which is described in Section 5.2.3.3. The same procedure is followed if B receives an unsigned message from a previously unknown node.

5.2.3.2 Certificate request

If B receives a find node-response containing a previously unknown node, it needs to confirm the claimed identity and address. For this reason a find value-message is sent including B's certificate, a challenge and a signature. The key asked for belongs to the certificate of the unknown identity from the response. If the previously unknown node responds with a valid certificate and a correct signature on the challenge for the sent certificate, B can safely store the certificate and add the node to its routing table.

It is worth noticing that nodes whose certificates we do not posses are excluded from the node lookup and will thus not receive a find-node request. If a request would have been sent, the response from the unknown node would have to be handled with a NACK and is therefore of no use.

[†]A NACK response is used to indicate that the received message was not accepted in the way the sender intended

5.2.3.3 Ping with explicit certificate

If node A possesses the certificate of node B but has been made aware that B does not possess A's, it needs to send over a signed message including the certificate. The significance of the signature is that it is a proof of A also possessing the private key for the certificate. The message sent is a ping with a signature and with the certificate in base64 format. Upon retrieval, B verifies the signature and the certificate and, if found correct, stores it.

Ping with explicit certificate is also sent as a part of the joining procedure.

5.2.4 Avoiding deadlocks

There are certain situations in which certificate-protected communication in a distributed network can get caught in deadlocks if one is not careful when implementing the communication protocol. The main problem is that each node only knows which certificates itself possesses, having no possibility to find out which nodes in the network are storing its certificate.

A common situation is that Node A receives a request from Node B but does not possess B's certificate. B is not aware of this fact. This could clearly be solved by including a certificate in every message sent. But as it is not desirable to introduce a severe overhead in the network, a more refined solution is needed.

The second option would be for A to just ignore B's message, but the silence does not give B any information other than A is not responding and should be removed from B's routing table.

A could send a certificate request to B, to ask for the missing certificate. But as all certificate requests has to include the senders certificate, this would open up the DHT for an amplification attack. An outside node would be able to send a small unsigned message resulting in a large signed message containing the certificate every time.

The best option found was A informing B through a signed NACK message. By doing so, B is informed that B does not possess its certificate and can now send it via a ping with explicit certificate. A non-certified node gets no information by sending unsigned requests and does not cause an amplification attack as NACK messages are roughly the same size as a request. The only drawback is the signing which is necessary to avoid spoofed NACK messages.

Chapter 6

Evaluation of the security-enhanced DHT

6.1 Mitigated vulnerabilities

6.1.1 Multiple identities

As each node's identity is now bound to a certificate, a node cannot claim to possess several identifiers. There will not exist any Sybils as it is not possible to provide a valid certificate or a valid signature for the Sybils. Protection against attacks aiming at providing several CSR for the same device to the CA and thereby getting multiple identities is out of scope for this master thesis work.

6.1.2 Non-random identifiers

Every identifier is now defined as the 160 least significant bits from the fingerprint of the node's certificate. As the joining node only provides a CSR which is signed by the CA and sent back, the fingerprint of the certificate cannot be predicted.

6.1.3 Dynamically changing IP/Port

6.1.3.1 Active impersonation

Attempting to impersonate another node actively, i.e. by sending messages claiming to be from their identifier, will fail as one cannot provide a signature with the private key of that certificate. Nor is it possible to replay messages from other nodes other than under very special circumstances. Questions can only be repeated if another node is given the same identifier as a previously existing node. Answers can only be repeated if the challenge happens to be identical to the challenge of a previously received request. According to the birthday paradox, one must have observed $6.07 * 10^7$ messages* to have 0.1% chance of finding a single collision [20].

6.1.3.2 Passive impersonation

If a malicious node returns a find node-response consisting of legitimate identifiers but false addresses, the receiver of the response will continue as usual, knowing that only the rightful owner of the identifier will be able to provide a correct

*Expressed in power of two: $2^{25,85}$ messages

signature for the answer. The receiver will not update the address information until it has received a signed confirmation from the new address. If the response contains previously unknown nodes, such as Sybils, a certificate request will be sent to the given address. And as mentioned in Section 6.1.1, a node without a valid certificate will not be able to verify its identity.

6.2 Attacking the security-enhanced DHT

6.2.1 Without a valid certificate

There are several possible scenarios in which a node should not be permitted to communicate with the network. It could be a node not having a certificate at all, a node having an invalid certificate or a node not possessing the private key to a certificate. In all of these cases the node should not be allowed to retrieve any values from the DHT and no member of the network should add the node to its routing table.

A few test iterations using the test scenario described in Section 4.2 confirmed this behavior. The non-certified node was not able to communicate with the legitimate nodes. Nor was the attacking node able to extract resources from the DHT, as it did not get any responses to its find-value requests.

6.2.2 With a valid certificate

A node with a valid certificate is allowed to communicate with all other nodes and therefore able to carry out the attacks mentioned in Chapter 4. However, as all of the attack vectors were mitigated, an evil node cannot severely disturb the communication between the nodes in the network. With a certificate a node is able to reply with false values and hampering the finding of nodes and values. But as it cannot impersonate or use multiple identities, the distributed and redundant properties of the DHT will cancel out the malicious behavior of an evil node.

50 test iterations were carried out with the test scenario where a certified node tests all vulnerabilities using a combination of node insertion and poisoning. All test cases were successfully performed without receiving a single false value. As expected, the evil node did receive some find-value requests (since it has a certificate and thereby exists in routing tables). The counterfeit values in the responses were not able to impact the test cases due to the majority decision.

6.3 Overhead and increased computational load

6.3.1 Asymmetric encryption method

Two different digital signature algorithms RSA[21] and ECC[22] were used to test the performance of the security enhancements. The choice of method does not have any impact on the functionality but can differ vastly in both overhead and computational load. In order to get a fair comparison between the two, the level

of security was set to be as equal as possible. According to BSI[†] the minimum secure key length for 2016 is 2048 bit for RSA and 256 bit for ECC.[‡] [23] These key lengths were used in the performance comparison.

6.3.2 Comparison

Comparing the message lengths, it is clear the overhead introduced by the security enhancements is substantial when it comes to certificate exchange. It requires the sending of approximately 2000 additional bytes for ECC and 3000 for RSA for each node pair communicating for the first time. When the certificates have been exchanged, only the 8-byte challenge and the signature cause additional overhead. As seen in Table 6.1, the overhead for the standard Kademlia messages is 88-90 bytes for ECC and 274 bytes for RSA.

Looking at the additional time required for the cryptographic operations, shown in Table 6.2, RSA can be concluded to be suitable for a protocol where verification is more common than signing.[§] In Kademlia's case however, the ratio between signing and verification is almost 1:1. This, and the lesser overhead makes ECC the suitable signature type for the security-enhanced implementation.

Table 6.1: Comparing message lengths in non-secure and security enhanced Kademlia

Message type	Non-secure	ECC	RSA
Certificate send-over	-	1041	1503
Multicast Response	399	1293	1559
Find node/Find value	119	208	393
Value response	78	167	352
Ping	92	181	366
Certificate length	-	879	1155
Signature length	-	70-72	256

All values are shown in bytes.

6.4 Evaluation

With the introduction of authentication and a certification-before-inclusion strategy for node and value lookups, all vulnerabilities identified in Section 4.1 have been removed. In the situation where an evil node is a part of the network, i.e. has a valid certificate, the negative impact on network robustness has been heavily reduced. The evil node can still break the protocol but with the important

[†] Federal Office for Information Security in Germany

[‡] The curve "NIST P-256" were used for ECC.

[§] This due the fact that the public exponent is often much smaller than the private, thus being easier to calculate.

Table 6.2: Comparing average time for completing signing and signature verification

	ECC	RSA
Verify operation	$4.0 * 10^{-4}$	$8.5 * 10^{-5}$
Signing operation	$1.7 * 10^{-4}$	$2.1 * 10^{-3}$
Total	$5.7 * 10^{-4}$	$2.2 * 10^{-3}$

All values are shown in seconds.

difference that it cannot increase its influence, reduce the redundancy or prevent nodes from communicating in the network.

Nodes outside the network can no longer communicate with nodes in the DHT if they have not been certified to do so. A non-certified node will only be able to get a NACK message in return regardless of message sent. This will also aggravate external DDoS attacks as amplification is not possible to achieve.

As mentioned in Section 6.3.2, the increased overhead is substantial when new nodes enter the DHT, due to the added certificate exchange procedure. This can however be seen as a one-time cost in a small DHT network, as certificates are stored after the first contact. The remaining overhead, consisting of signatures and challenges is no more than 88-90 bytes per message, using ECC. Given a network not receiving members frequently, the total amount of overhead per message over time will likely not exceed 100 bytes. The increased computational load was quite limited in the test environment, with ECC it ended in a very modest sum of $5.7 * 10^{-4}$ seconds of crypto operations per node and message. Neither of these factors had an observable negative effect in any of the test cases. One could however notice that due to certificate exchanges, the network needed almost double amount of time before most of the nodes had added each other to their routing tables.

The security enhancements proposed in this thesis were found to be a good fit for Calvin. The implementation was merged into Calvin and is now an optional security feature which can be used to protect the distributed storage.

Chapter 7
Discussion

7.1 Implications for DHT usage

While DHT is a very convenient storage solution for distributed systems, it has been shown in Section 4.2 that a plain implementation of Kademlia easily can be compromised or taken over by a single malicious node. Such weaknesses make DHT unsuitable for storage of anything security-critical as one simply cannot trust that the received value for a key is correct. Adding security enhancements to Kademlia can increase the possible usage of the DHT. With the new functionality, attackers cannot compromise the DHT significantly and can neither increase their influence nor reduce redundancy.

An even more significant improvement is that non-authorized nodes no longer can fetch any information from the DHT, as only certified nodes may send DHT-messages. But, contrary to the defense against the attacks described in Section 2.4.6, which also defends against certified attackers, this feature depends solely on the CA. Thus, the mechanism certifying nodes must either be manually controlled or strictly regulated. If any node can get a certificate, the security solution is of very little use as it presuppose only a few nodes with certificate will misbehave.

To summarize, a distributed network can benefit greatly from having a security layer and allow more sensitive data to be stored. But the security is also dependent on taking precautions to not letting illegitimate nodes enter the network.

7.2 Adaptations to enable global availability

All systems in which a single or few CA can be used for node initialization can make use of this security model. However, in a distributed network with several thousands or even millions of members, this is not a scalable solution. There will likely need to be multiple CAs, because of geographical reasons, but also due to the need of sub-domains. A sub-domain could e.g. be a company, a geographical area or a unit type which need a certification from a specific CA. Having multiple CAs will, with the current solution, force all nodes including computationally limited ones to store vast amounts of CA certificates (as is done in e.g. web browsers). This is not an optimal solution as the overhead for both verifying and storing certificates with many different CAs will likely decrease the performance greatly.

For a global DHT to work, different strategies regarding certificate storage and verification have to be applied. One idea is to use the same paradigm as in this thesis but let computationally limited devices use more powerful devices as proxies. I.e. by letting a more powerful device store the certificates and verify the signature, the limited devices can still participate in the DHT, but route all traffic through a single node. One could think of this as a voluntary Eclipse attack, as the rest of the DHT will regard the stronger node as the owner of the weaker node's identity. However, while this solves one problem, it also creates several others, the biggest being the DHT de facto no longer being entirely distributed. The powerful nodes will gain more influence in the network and can therefore do more damage if they turn evil or malfunction.

7.3 Future work

To improve the security enhancements implemented in this thesis, the focus should be on the protection of the actual data in the DHT. With the current implementation it is not possible for a node to verify the correctness of received data. This could be achieved by storing every resource together with a signature of the storing node. With such a solution, non-repudiation would be assured and values without valid signatures would simply be ignored.

To further benefit from the signed resource feature and also the security enhancements described in Section 5.2, some sort of incident report system should be implemented. In this kind of system, protocol breaking activity is to be reported. This can give an indication of an attack taking place and possibly also let the nodes take evasive actions. Such an action could for example be revoking a misbehaving node's certificate and distributing a revocation list in the network. However, this would also increase the complexity in the network as one or several nodes need to log events and if certificate revocation is desired these events will also need to reach the CA. Further study is needed to determine if the possibility to identify attacks counterbalance the increased complexity.

A factor which has not been considered in this thesis is the computational power of nodes of a DHT. As all tests were performed on a computer, the values in Table 6.2 are not necessarily valid for IoT devices. There may be devices which are not able to handle asymmetric cryptography at all and therefore cannot participate in the secured DHT. The proxy solution mentioned in Section 7.2 is a possible method of including such devices in a security-enhanced DHT, possibly combined with symmetric encryption between the proxy and the node.

Chapter 8
Conclusion

In this master thesis, the Kademlia DHT implementation used in Calvin was studied and found to contain severe security weaknesses. By exploiting these vulnerabilities, several well-known attacks were able to severely hamper the functionality of the DHT. With these results it could be concluded that the current implementation is not suitable for storing security-critical information.

To strengthen the security of the DHT, a concept of trust was introduced into the system, where each node needs a signed certificate by a trusted CA to participate in the DHT. A part of the fingerprint for the certificate were set as identifier for the node owning the certificate. By doing so a node is only able to prove ownership of an identifier by having the private key of the certificate. With the introduction of this functionality, nodes are able to authenticate other legitimate nodes in the DHT. Furthermore, the authentication procedure can be performed without breaking the distributed properties of the network.

Several changes were also made to the communication protocol. All messages now contain a signature from the sender and all requests must also include a challenge which is to be signed by the responding node. Support was also added for certificate exchange and a NACK message for informing a node that it will have to provide its certificate in order to be able to communicate further.

The security enhancements successfully stopped all of the well-known attacks and also prohibited uncertified nodes from retrieving any data from DHT. Adding security did however cause a relatively large overhead, especially due to the exchanging of certificates. By choosing ECC as encryption method, the overhead will likely not exceed 100 bytes per message over time as certificate exchanges can be considered a one time cost. The performance of the DHT did not observably decrease in the performed tests.

By using the security enhancements in the DHT, it is possible to safely expose security-critical data without risk of information leakage. The solution is however not optimal for large-scale DHT, where having a single CA is simply not feasible. Further study is also needed on how to include devices which are not able to compute signatures and validations due to computational limitations.

References

[1] Charles E Perkins. *Ad hoc networking*. Addison-Wesley Professional, 2008.

[2] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[3] Thomas Locher, David Mysicka, Stefan Schmid, and Roger Wattenhofer. Poisoning the kad network. In *Distributed Computing and Networking*, pages 195–206. Springer, 2010.

[4] Peng Wang, James Tyra, Eric Chan-Tin, Tyson Malchow, Denis Foo Kune, Nicholas Hopper, and Yongdae Kim. Attacking the kad network. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, page 23. ACM, 2008.

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

[6] Philipp C Heckel. *Blacklisting Malicious Web Sites using a Secure Version of the DHT Protocol Kademlia*. PhD thesis, Citeseer, 2009.

[7] Charles L Hedrick. Rfc2453: Routing information protocol. 1988.

[8] John Moy. Rfc2178: Ospf version 2. 1997.

[9] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[10] Tim Dierks. The transport layer security (tls) protocol version 1.2. 2008.

[11] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[12] Brian Muller. Kademlia. `https://github.com/bmuller/kademlia`, 2015.

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

[14] Kjeld Egevang and Paul Francis. Rfc1631: The ip network address translator (nat). Technical report, 1994.

[15] John R Douceur. The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002.

[16] Atul Singh et al. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer, 2006.

[17] Daniel Germanus, Robert Langenberg, Abdelmajid Khelil, and Neeraj Suri. Susceptibility analysis of structured p2p systems to localized eclipse attacks. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 11–20. IEEE, 2012.

[18] Ericsson Research. Calvin. `https://github.com/EricssonResearch/calvin-base`, 2015.

[19] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[20] `http://preshing.com/20110504/hash-collision-probabilities/`. Accessed: 2016-04-10.

[21] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[22] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[23] M Margraf. Kryptographische verfahren: Empfehlungen und schlüssellangen. *Technische Richtlinie TR-02102, Bundesamt fur Sicherheit in der Informationstechnik*, 2008.

Appendix A

Network illustrations

This appendix consists of illustrations of the network topologies in different scenarios. The pictures are to be interpreted as follows:

- The circles represent nodes in the DHT and the number which port they are operating from (as mentioned in Section 3.1.2, all nodes run on the same computer and thereby share the same address).
- Each line represents one entry in the node's routing table and the numbers next to it represents the last four digits for the identifier of this routing table entry.

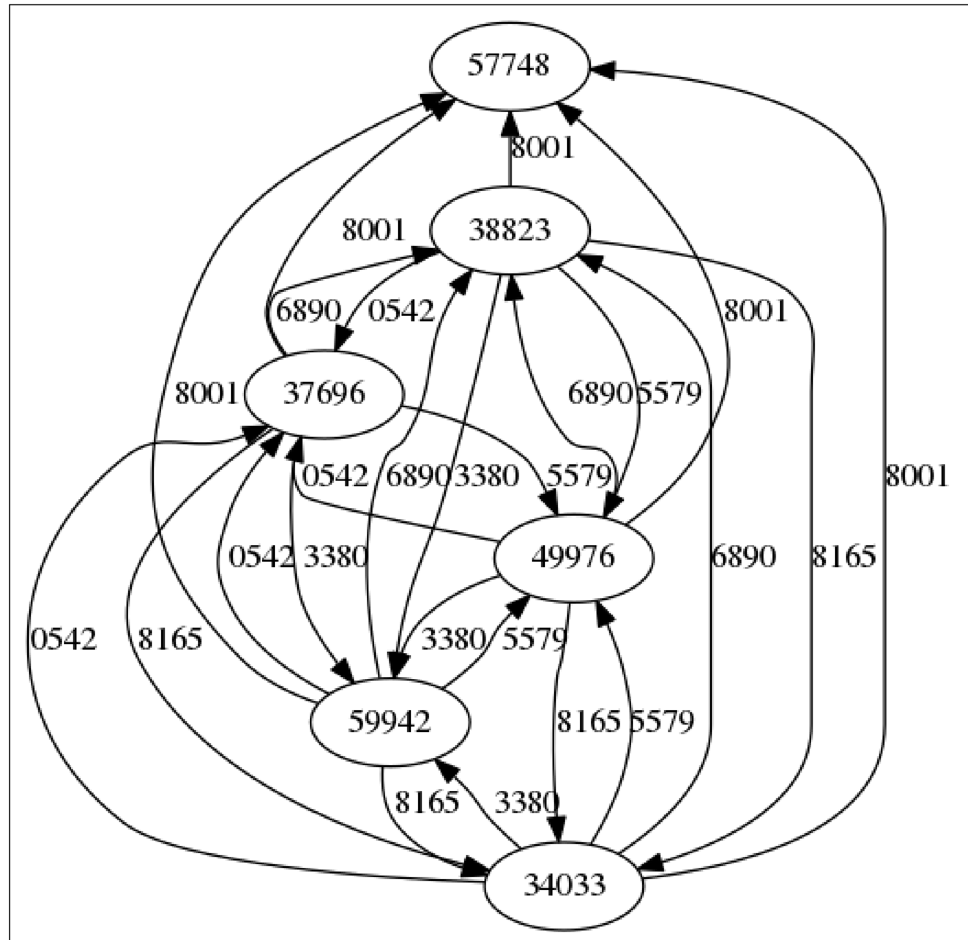

Figure A.1: The topology of a normally functioning DHT network consisting of six nodes.

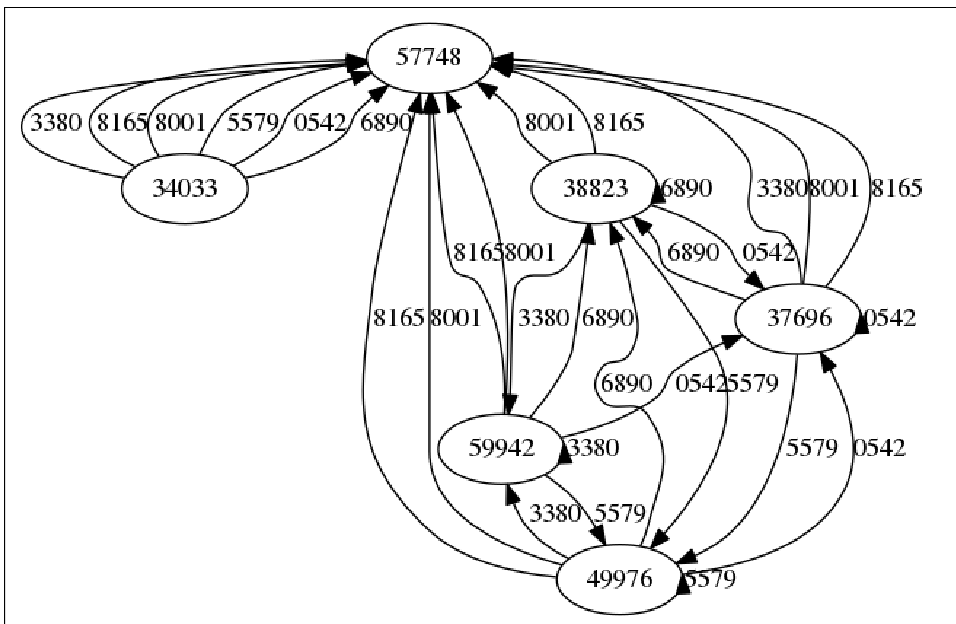

Figure A.2: The topology of a DHT network consisting of six nodes after a successful Eclipse attack. The attacking node operates from port 57748.

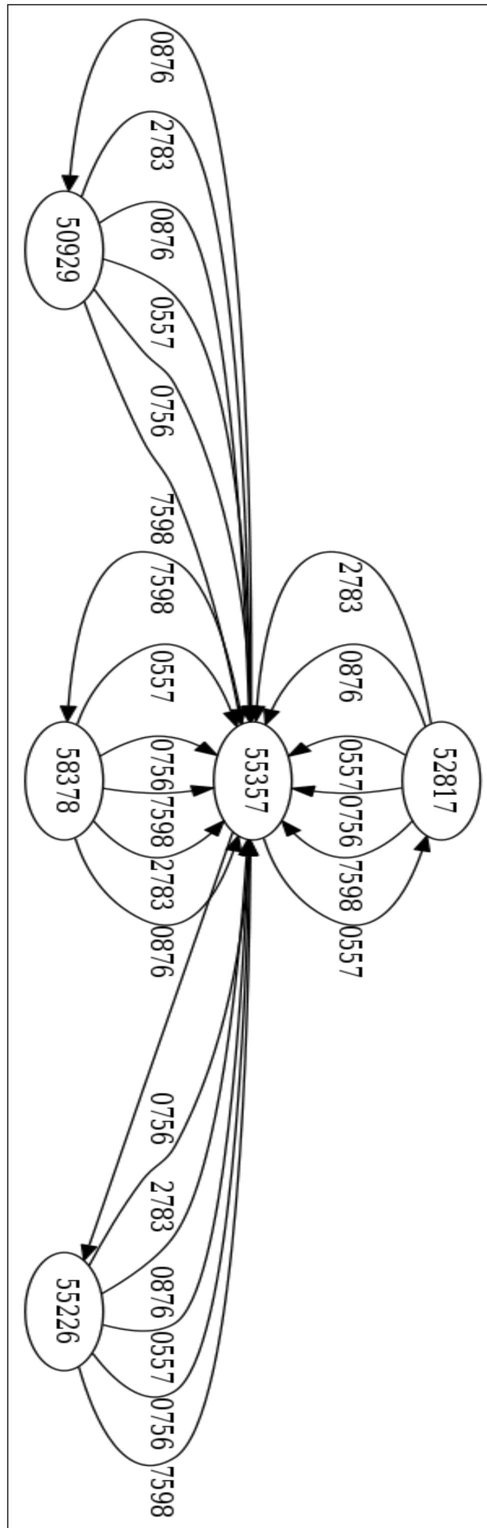

Figure A.3: The topology of a DHT network consisting of five nodes after a successful poisoning attack. The attacking node operates from port 55357.

Appendix B

Changes in communication flow due to security enhancements

This appendix aims at giving the reader an understanding of how the security concept have been implemented in Calvin's Kademlia DHT. The comparisons shown is intended to show to additional functionality which is added in the chain Request -> Response -> Response handling. While only the flow for the store request is shown, the find node, find value and ping requests have similar albeit more complicated communication flows. The entire implementation of the code was, at the time of the publication of this thesis, found at
`https://github.com/EricssonResearch/calvin-base/tree/develop/calvin/runtime/south/plugins/storage/twistedimpl/securedht`

Listing B.1: Comparison of the store request methods.
An eight byte challenge and a signature of the receiver's identifier has been added to the request.

```
#NEW
def callStore(self, nodeToAsk, key, value):
    address = (nodeToAsk.ip, nodeToAsk.port)
    challenge = os.urandom(8).encode("hex")
    private = OpenSSL.crypto.load_privatekey(OpenSSL.crypto.FILETYPE_PEM,
        self.priv_key, '')
    signature = OpenSSL.crypto.sign(private, nodeToAsk.id.encode("hex").
        upper() + challenge, "sha256")
    d = self.store(address, self.sourceNode.id, key, value, challenge,
        signature)
    return d.addCallback(self.handleSignedStoreResponse, nodeToAsk,
        challenge)

#OLD
def callStore(self, nodeToAsk, key, value):
    address = (nodeToAsk.ip, nodeToAsk.port)
    d = self.store(address, self.sourceNode.id, key, value)
    return d.addCallback(self.handleCallResponse, nodeToAsk)
```

Listing B.2: Comparison of the store response methods.
First, the identifier of the sender is matched against known certificates. If the certificate is found, the signature is validated, otherwise a signed NACK is returned. Assuming the signature is correct, the value is put into storage with the identifier "key" and a signature on the challenge is returned

```
#NEW
def rpc_store(self, sender, nodeid, key, value, challenge, signature):
    source = Node(nodeid, sender[0], sender[1])
    certificate = self.searchForCertificate(nodeid.encode('hex').upper())
    if certificate == None:
        try:
            private=OpenSSL.crypto.load_privatekey(OpenSSL.crypto.
                FILETYPE_PEM, self.priv_key, '')
            signature = OpenSSL.crypto.sign(private, challenge, "sha256")
        except:
            return None
        logger(self.sourceNode, "Certificate for {} not found in store".
            format(source))
        return {'NACK' : None, "signature" : signature}
    else:
        try:
            OpenSSL.crypto.verify(certificate, signature, self.sourceNode.id
                .encode('hex').upper() + challenge, "sha256")
        except:
            logger(self.sourceNode, "Bad signature for sender of store
                request: {}".format(source))
            return None
        self.router.addContact(source)
        self.storage[key] = value
        try:
            private=OpenSSL.crypto.load_privatekey(OpenSSL.crypto.
                FILETYPE_PEM, self.priv_key, '')
            signature = OpenSSL.crypto.sign(private, challenge, "sha256")
        except:
            logger(self.sourceNode, "Signing of rpc_store failed")
            return None
        return signature

#OLD
def rpc_store(self, sender, nodeid, key, value):
    source = Node(nodeid, sender[0], sender[1])
    self.welcomeIfNewNode(source)
    self.log.debug("got a store request from %s, storing value" % str(sender
        ))
    self.storage[key] = value
    return True
```

Listing B.3: Comparison of the store response handler methods. The response handler is now specific for each message type and support for handling the NACK message have been added. The returned signature must also prove to be correct for the function to return true. "True" in this instance is a confirmation that the value has been stored in the DHT by the sending node.

```
#NEW
def handleSignedStoreResponse(self, result, node, challenge):
    if result[0]:
        if "NACK" in result[1]:
            return self.handleSignedNACKResponse(result, node, challenge)
        cert = self.searchForCertificate(node.id.encode('hex').upper())
        if cert == None:
            logger(self.sourceNode, "Certificate for sender of store
                  confirmation: {} not present in store".format(node))
            return None
        try:
            OpenSSL.crypto.verify(cert, result[1], challenge, "sha256")
            self.router.addContact(node)
            return (True, True)
        except:
            logger(self.sourceNode, "Bad signature for sender of store
                  confirmation: {}".format(node))
            return None
    else:
        logger(self.sourceNode, "No store confirmation from {}, removing
             from bucket".format(node))
        self.router.removeContact(node)
    return None

#OLD
def handleCallResponse(self, result, node):
    if result[0]:
        self.log.info("got response from %s, adding to router" % node)
        self.welcomeIfNewNode(node)
    else:
        self.log.debug("no response from %s, removing from router" % node)
        self.router.removeContact(node)
    return result
```

Appendix C

Test case and attack methods

This appendix contains an extract of a test case and the active attack methods for the eclipse and poisoning attack. It is not a complete description of the attacks nor the test cases but shows key functionality giving the reader insight into how the tests have been carried out.

Listing C.1: Extract from the test case
One node posts a resource to the DHT, checks its availability and checks this again approximately 15 seconds later. The topology of the network is depicted three times during the test (see Appendix A).

```
key = "KANIN"
value = "morot"
set_def = servers[0].set(key=key, value=value)
set_value = yield threads.defer_to_thread(set_def.wait, 10)
assert set_value
get_def = servers[0].get(key="KANIN")
get_value = yield threads.defer_to_thread(get_def.wait, 10)
assert get_value == "morot"

drawNetworkState("nice_graph.png", servers, amount_of_servers)
yield threads.defer_to_thread(time.sleep, 7)
drawNetworkState("middle_graph.png", servers, amount_of_servers)
yield threads.defer_to_thread(time.sleep, 7)
drawNetworkState("end_graph.png", servers, amount_of_servers)

get_def = servers[0].get(key="KANIN")
get_value = yield threads.defer_to_thread(get_def.wait, 10)
assert get_value == "morot"
```

Listing C.2: Active impersonation in the poisoning attack.
The node chooses a random member of its routing table and sends ping requests to the chosen node impersonating all other nodes present in the routing table.

```
def poison_routing_tables(self):
    self.neighbours = map(tuple, self.router.findNeighbors(Node(hashlib.sha1(str
        (random.getrandbits(255))).digest()),k=20))
    my_randoms = random.sample(xrange(len(self.neighbours)), 1)
    for nodeToAttack in my_randoms:
        for nodeToImpersonate in range(0, len(self.neighbours)):
            if nodeToImpersonate != nodeToAttack:
                self.ping((self.neighbours[nodeToAttack][1], self.neighbours[
                    nodeToAttack][2]), self.neighbours[nodeToImpersonate][0])
```

Listing C.3: Active impersonation in the Eclipse attack.
The node applies the same procedure as in C.2 towards the eclipsed node (here known as closest_neighbour). Towards all other nodes however, it only sends one ping requests claiming to be the eclipsed node.

```
def eclipse(self):
    self.neighbours = map(tuple, self.router.findNeighbors(Node(hashlib.sha1(str
        (random.getrandbits(255))).digest()),k=20))
    for nodeToAttack in range(0, len(self.neighbours)):
        self.ping((self.neighbours[nodeToAttack][1], self.neighbours[
            nodeToAttack][2]), self.closest_neighbour[0][0])
        self.ping((self.closest_neighbour[0][1], self.closest_neighbour[0][2]),
            self.neighbours[nodeToAttack][0])
```