A Byzantizer for Ad Hoc Routing Schemes

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in Computer Science

by

Yoav Reshef

Supervised by

Prof. Danny Dolev

School of Engineering and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

December 3, 2005

Acknowledgement

I would like to thank my advisor Prof. Danny Dolev, for his excellent guidance and for his great support, particularly, in the formulation of the algorithm and the proof.

I would also like to thank Ittai Abraham for his guidance, ideas and helpful suggestions.

Abstract

We consider the problem of secure routing in ad hoc networks that may have Byzantine faults. We propose a general *Byzantizer* framework. The Byzantizer is a security algorithm that transforms any ad hoc routing algorithm (conforming to some specific requirements) into a routing algorithm that is resilient to Byzantine faults. The protocol is efficient in terms of storage and computational power requirements and thus it is suitable to ad hoc networks and in particular to sensor networks.

Keywords: Distributed Algorithms, Ad Hoc and Sensor Networks, Ad Hoc Routing, Byzantine Faults, Locality Aware Algorithms.

Table of Contents

Acknowledgement **2**

Abstract **3**

1 Introduction **3**

1.1 Our Contribution 4

2 Related Work **5**

3 High Level Description **9**

4 Model **11**

5 HMACs, Messages, Variables and Timeouts **13**

5.1 HMACs 13

5.2 Messages 14

5.2.1 DATA message 15

5.2.2 ACK 18

5.2.3 FA 20

5.3 Variables 21

5.4 Timeouts 22

6 The Byzantizer Algorithm **24**

6.1 Characteristics 29

7 Analysis **30**

7.1 Definitions 30
7.2 Proof 31

8 Neighborhood Discovery **54**
8.1 Messages and Timeouts 54
8.1.1 HELLO message 54
8.1.2 FINISH message 55
8.1.3 Timeouts 56
8.2 The Neighborhood Discovery Algorithm 56
8.3 Correctness 58

9 Conclusions and Future Work **60**

Bibliography **62**

Chapter 1

Introduction

Ad hoc wireless networks have been a focus of research in recent years. These networks require no infrastructure, each node takes part in the process of forwarding packets and acts as an intermediate router. Achieving security in ad hoc networks is more challenging than in wired networks because no centralized administration or a trusted authority exist and thus nodes cannot be completely trusted. This problem is enhanced in sensor networks in which each node has a limited computational and battery power and a limited storage capability.

Under these conditions efficient and secure routing protocols are challenging to design. There are several existing routing protocols that are suitable for the dynamic nature of these networks and to the limited capabilities of the nodes. However, many of them do not consider security [11, 26]. Routing algorithms that are resilient to Byzantine faults are even more challenging, since a Byzantine node is a faulty node that can exhibit *any* arbitrary behavior either alone or in collusion with other faulty nodes.

Avramopoulos et al. [1, 2] present a routing algorithm resilient to Byzantine faults for general networks and not necessarily ad-hoc wireless networks. Their algorithm guarantees that as long as there is a non-faulty path between the source and the destination the packets will be delivered from the source to the destination. The algorithm is based on source routing and it is assumed that each node knows the global topology of the network and thus it can calculate the shortest path by using a modified Bellman-Ford shortest path algorithm. Because of these assumptions their solution is not scalable and a substantial storage capability is needed. In addition, the complexity of acquiring global network topology by each node in a way that is resilient to Byzantine faults is not discussed in the article. Further more, their fault

knowledge mechanism is limited and thus faulty nodes may still participate in paths after they have already caused faults in previous paths.

1.1 Our Contribution

We consider *ad hoc* routing algorithms that have local routing functions. Specifically, we assume each node decides to which 1-hop neighbor to forward a message only based on its 1-hop neighbors and the given message, (for example see [12, 13]). We have designed a *Byzantizer* protocol that transforms any such ad hoc routing algorithm into a routing algorithm that is resilient to Byzantine faults.

Unlike [1] which requires each node to maintain global knowledge, in our solution each node requires only local-neighborhood network topology knowledge. Hence our algorithm is scalable and its storage requirement and messages' size is suitable for ad hoc networks. Security is achieved while using only symmetric cryptography (as in [1]) and thus the Byzantizer is efficient in terms of computational and battery power.

The Byzantizer allows to isolate the faulty nodes and it guarantees that as long as there is a path between the source and the destination that is at more than $3f+3$ hops from any faulty node, where f is the maximal number of faulty nodes, the packets will be delivered from the source to the destination. Using a competitive ad hoc routing scheme, like [13], our Byzantizer obtains the first competitive ad hoc routing scheme in a Byzantine failure model.

Chapter 2

Related Work

Security challenges and solutions for routing in ad hoc networks and sensor networks were discussed in [26] and [11], respectively. Message authentication is an essential component in any security design. Three cryptographic primitives are widely used to authenticate messages, digital signature, HMAC[1] and one way HMAC key chain[2]. A digital signature is based on asymmetric cryptography, an HMAC is based on symmetric cryptography and a one way HMAC key chain is based on a one way function. Authenticated public keys can be acquired by using a Certificate Authority (CA). [28, 15] modified the traditional CA solution such that it will be suitable for ad hoc networks. In ad hoc networks nodes cannot be fully trusted and thus their solution distributes the certificate authority among several nodes. In [28], the certificate authority consists of a specific group of nodes, whereas in [15], any group of k nodes can sign a certificate. Symmetric keys can be generated with a digital signature scheme. Another approach is to use multi-space pairwise key distribution techniques ([7, 14]). In [14], the setup server generates a set of polynomials. For each node, the setup server randomly picks a subset of polynomials and assigns polynomial shares of these polynomials. Two nodes can establish a pairwise key if they share a common polynomial. [7] uses a similar idea but instead of polynomials matrices are used. It should be noted that multi-space pairwise key distribution techniques do not guarantee that any two nodes are able to generate a pairwise key. One way key chain can be used once the receiver has an authentic element of the key chain. An authenticated key can be acquired, for example, by using a digital signature or a symmetric key. Observe that

[1] To avoid ambiguity we use MAC to refer to Medium Access Control and HMAC to refer to Keyed Hash Message Authentication Code.

[2] One way HMAC key chain is also know as one way key chain

digital signatures are more expensive in terms of battery and computational power than HMAC and one way key chain.

Perrig et al. have presented TESLA [22] and later μTESLA [23] which provides an efficient authenticated broadcast primitive. The authentication is based on one-way key chain and late key disclosure rather than asymmetric cryptography. The sender generates a one-way key chain. Time is divided into uniform time intervals and each key is associated with one time interval. The HMACs of messages in the same interval will be generated with the same key. In the setup phase the receiver acquires an authenticated key of the sender. When the sender wants to send a message it first sends the message itself with an HMAC and later it sends the key for this HMAC. When the message is received it is checked that the message is *safe*, i.e. that the key has not been disclosed yet, and if it is safe it is stored. When the key is received it is authenticated by using the last authentic key. Once the key is authenticated it is used to authenticate the message. The μTESLA guarantees that if the sender is non-faulty and sends a message and a non-faulty receiver has a μTESLA authenticated key of the sender and it authenticates the μTESLA HMAC of the message, then this is the original message generated by the sender. A μTESLA authenticated key can be acquired without requiring global topology knowledge, for example, by generating a symmetric key using multi-space pairwise key distribution techniques or by using asymmetric cryptography. Authenticated public keys can be acquired, for example, by assuming that there is a setup server that has a private/public key pair. The setup server assigns each node with a private/public key pair and with a digital signature of the node's public key using the private key of the server. Moreover, each node knows the public key of the server, and thus each node can authenticate the public keys of the other nodes.

Perlman [21] overcomes Byzantine faults in wired networks by using digital signatures and by flooding the network. As stated before, digital signatures are expensive and thus the proposed protocols may not be suitable for ad hoc networks. In addition the state at each node is proportional to the number of nodes in the network, since each node knows the public keys of all the other nodes and thus such a solution is not scalable and requires global network knowledge.

Smith et al. [25] address the problem of securing distance vector routing protocols. In distance vector routing protocols each node maintains a routing table listing all possible destinations within the network and sends routing updates. Their solution overcomes Byzantine faults by using digital signatures, sequence numbers and by adding the second-to-last hop in the path to a destination in the routing table update.

Since global topology knowledge is required and messages include routing tables, this protocol is expensive in terms of node state and packet overhead. SEAD ([8]) is another example of distance vector protocol. In SEAD, authentication is achieved by using sequence numbers and one way key chain.

Bradley et al. [6] address the problem of misbehaving routers in wired network. Their solution overcomes routers that perform the selective forwarding attack or routers that mis-route packets. The first attack is countered by the use of the conservation of flow where each router validates that the number of bytes going into a neighboring router must equal the number of bytes coming out of this router. The second is countered by the use of copies of the routing tables. Each router has the routing tables of its neighbors and thus it can know whether they mis-route packets. Their solution does not work when faulty routers collude and it requires that each router must have a non-faulty neighbor.

Marti et al. [16] take advantage of the promiscuous mode in wireless networks in order to identify faults. Each node that sends a message, listens to the next node's transmissions. If the next node does not forward the message, then it is misbehaving. In such a case, a message is sent to the source notifying it of the misbehaving node. As they state, their methods for detection may fail at the presence of collisions and they do not consider a collusion of faulty nodes. Moreover, there is no way for the source, or any other node the receives a misbehavior report to validate its authenticity or correctness. As a result, the faulty node can disable the network operation.

Zapata and Asokan [27] consider the problem of incorporating security mechanisms into routing protocols for ad hoc networks. They have looked at AODV ([20]) in detail and developed a security mechanism to protect its routing information. Their solution uses digital signatures for the authentication of the non-mutable parts of the messages and one way key chains to secure the hop count information (the only mutable information in the messages). Their solution is not scalable since each node should know the public keys of all the other nodes and their technique of securing the hop count may not be applicable to other messages, where the mutable part consists of fields other then hop count.

Hu et al. [9] present the Ariadne routing protocol which is based on DSR [10] and uses TESLA. The routing begins when the source sends a request that includes a time interval, which is the pessimistic arrival time of the message at the destination, and an hash which is initialized to the hash of the non-mutable fields of the request. The nodes are accumulated along the route as in DSR. Each node appends its name, its TESLA HMAC and generates a new hash based on the current hash. The TESLA

key used is the key for the time interval specified in the request. The destination verifies the hash chain and that the keys of the TESLA have not been disclosed yet. If the destination determines that the request is valid it generates a reply to the source which is sent along the path obtained by reversing the path of the request. When a node receives the reply, it waits until it is able to disclose its key and then it adds the key to the reply. When the source receives the reply it verifies the TESLA HMACs. Source routing is expensive in terms of node state and packet overhead and thus this solution is not scalable and may require a global network knowledge.

The SRP algorithm ([18]) is another example of an algorithm that is based on source routing. In this algorithm the source and the destination have a symmetric key which is used for calculating an HMAC for the request and the reply. Intermediate nodes do not verify the message and thus the security is at the end to end level and not per hop. As a result, an adversary can change the message and only the end point (the source or the destination) can identify it. A complementary approach is taken by the SMT ([19]) protocol. Given the topology of the network, the source determines a set of diverse paths connecting the source and the destination. The source disperses each outgoing message into a number of pieces. At the destination, a dispersed message is successfully reconstructed, provided that sufficiently many pieces are received. As in the SRP protocol, it is assumed that the source and the destination have a symmetric key which is used for calculating an HMAC. Both SRP and SMT are expensive in terms of node state and thus they are not scalable.

Awerbuch et al. [5, 4] present the On Demand Secure Byzantine Routing protocol (ODSBR). Their solution involves flooding the network and using source routing and digital signatures, which are expensive as already stated. Each node maintains a *weight list* and the source includes its *weight list* in the request message that is flooded in the network. The path is accumulated in the message as the message is forwarded. Each receiving node uses the source's *weight list* to calculate the weight of the accumulated path, and only if its weight is smaller than the previously seen paths, the message is sent. The *weight list* may comprise the entire network and thus it requires a substantial storage capability. Their fault detection protocol that uses symmetric keys is started by the source upon identifying a fault. The idea behind this protocol is to use a binary search on the path in order to identify the faulty link. During the search, intermediate nodes, in addition to the destination, send acks to the source and thus when an ack is not received by the source, the faulty link can be identified. Once identified, the source's *weight list* is updated.

Chapter 3

High Level Description

We begin with a high level description of the Byzantizer. Consider a message sent from a source to a destination. We denote by *ameba* a sequence of $f + 1$ distinct nodes[1] along the path between the source and the destination. The nodes of the *ameba* are the nodes that participate in the routing at some stage of the protocol. The nodes transfer the data and decide on how to extend the path in order to reach the destination. The path of the *ameba* may fold onto itself so each node may appear more than once in the *ameba* and thus the total number of nodes in the *ameba* may be bigger than $f + 1$. The *ameba* advances by agreeing on the next node that should be included in the *ameba*. This is decided based on the routing algorithm and the network topology. Once agreement is achieved, the new node is added to the *ameba* and unnecessary nodes are removed, such that only $f + 1$ distinct nodes are included in the *ameba*.

Messages are used to synchronize between the nodes. These messages are authenticated by using keyed-hash message authentication codes. Each time a node joins the *ameba* it sends an ACK to inform the other nodes that it has joined the *ameba*. Timeouts are used to identify and handle faults. When a fault is identified a fault announcement (FA) is generated to inform the nodes in the neighborhood of the fault about its existence. Once an FA is generated the *ameba* and the local neighborhood of the generator of the FA do not participate in future routing and the current messages are lost. Each time an FA is generated it is guaranteed that at least one of the edges of a faulty node will not be used in future routings. The idea is to isolate the faulty nodes. If an FA was generated and the current message was lost, the source of the

[1] By distinct nodes we mean nodes with distinct IDs.

message sends a new message. After sending at most $\tau \cdot f + 1$ messages, where τ is the maximal degree of a faulty node, the message will reach the destination. Of course once all the faulty nodes are isolated the messages always reach the destination (if there exists a path in the remaining graph).

Designing the algorithm requires handling three main challenges. One, how to ensure that at any time the *ameba* contains $f + 1$ distinct nodes regardless of the fact that the path may include cycles. Two, how the nodes of the *ameba* can reach agreement regarding the next node in an efficient manner. Three, how to bound the radius of influence a faulty node may have. This difficulty arises since FAs are broadcasted to the local neighborhoods and non-faulty nodes do not accept these FAs at the same time; thus there is a need to coordinate these nodes such that no additional FAs will be generated.

Ensuring that the *ameba* contains $f + 1$ distinct nodes is done by allowing the *ameba* to expand or shrink as needed. An efficient agreement is achieved by assuming that each node knows its $O(f)$-hop local neighborhood and thus it can identify the $f + 1$ distinct nodes that follow it along the path. A message includes a list of nodes and in particular it includes f distinct next nodes. When a node receives a message, it first verifies the path in the message (the *ameba*), i.e. it verifies that its calculation of the f distinct next nodes agrees with the list of nodes in the message. If the verification succeeds the node determines the $f + 1$st next node by adding it to the message so when its next node receives the message, the message will include f distinct next nodes.

Coordination among the non-faulty nodes is achieved in the following way. Once an FA is accepted and a node is suspected, it does not immediately stop participating in the routing. Instead this node still participates in the routing until it can assume that all nodes that should accept this FA have accepted it and only then the suspected node stops participating in the routing. This idea is implemented by the "gray list" and "black list" as explained in the next sections.

Chapter 4

Model

We consider an n node synchronous distributed system connected via a network $G = (V, E)$, in which each node has a unique ID. An edge between two nodes indicates that they have a direct bidirectional FIFO communication channel between them. We consider Byzantine faults in which a faulty node is an authenticated node that can exhibit any arbitrary behavior either alone or in collusion with other faulty nodes, and that its behavior may not conform to the protocol. We assume there are at most f faulty nodes and denote $F \subseteq V$ the set of faulty nodes. In addition we assume that the maximal degree of a faulty node is τ.

A neighborhood discovery algorithm (described in Chapter 8) is performed before the routing starts. At the end of the neighborhood discovery each node x knows its local neighborhood up to σ hops away, where $\sigma = 2f + 2$. It also knows for each neighbor y in its σ-hop neighborhood the information needed for the routing and for verifying y's authenticated messages. Keyed-Hash Message Authentication Code (denoted by HMAC) is used throughout the algorithm to authenticate the received messages.

We assume that there is a bound on the length of each μTESLA ([23]) interval and denote this bound by μ. In Chapter 8 we prove that if a node x does not include in its local neighborhood a node y that is at most σ hops away, or x does not know the correct information (basic key of μTESLA, μTESLA schedule, etc.) of y, then there must be a faulty node w on the shortest path between x and y.

The Byzantizer uses a routing algorithm as a building block. The routing algorithm and the topology of the network should apply to the following requirements. A node decides to which 1-hop neighbor (denoted by 1HN) to forward a message only

based on its 1-hop neighbors and the given message. In addition, a node x should be able to determine for each of the next nodes on the path that are in its local neighborhood to which node they should forward the message. Examples for such routing algorithms include geometric routing algorithms such as GPSR [12] and GOAFR [13] and source routing algorithms in which the whole path is included in the message. The path defined by the routing algorithm may fold onto itself so nodes can be included in the path several times. We assume that any part of the path that includes $f + 1$ distinct nodes has a limited length. We denote this length by θ. It is obvious that $f \leq \theta$, since the length of the minimal path that includes $f + 1$ distinct nodes is f. The number of nodes in each such part is of course at most $\theta + 1$. In addition we assume that the maximal length of a path between two nodes is known. We denote this maximal path length by PATH-MAX-LEN.

Chapter 5

HMACs, Messages, Variables and Timeouts

In this chapter we describe the building blocks that are used throughout the Byzantizer algorithm. We describe how messages are authenticated, which messages are used, which information is stored at each node and how timeouts are used to identify and handle faults.

5.1 HMACs

The HMACs are generated in an onion like style as done in [1] for the previous or next $f+1$ distinct nodes on the path. There is a symmetric key between the generator of the HMAC and each of its recipients. Let the path be $x_h, \ldots, x_{i-1}, \boldsymbol{x_i}, x_{i+1}, \ldots, x_{i+k}$ and let the current node be x_i. Both $x_h, \ldots, x_{i-1}$ and $x_{i+1}, \ldots, x_{i+k}$ consist of $f+1$ distinct nodes. Assume that x_i generates HMACs for $x_{i+1}, \ldots, x_{i+k}$. The computation of the HMAC for node $x_j, i+1 \leq j \leq i+k$, receives as input both the message and the HMACs for nodes $x_{j+1}, \ldots, x_{i+k}$, i.e. the HMACs are computed sequentially from the last node (x_{i+k}) to the first node (x_{i+1}). Let $K_{u,v}$ denote the symmetric key between u and v, let $K_{u,v}(msg)$ denote the generation of the HMAC on msg by the key $K_{u,v}$ and let $\text{HMAC}_{u,v}$ denote the HMAC generated by u for v. Next we give an example of the onion HMAC x_i generates for $x_{i+1}, \ldots, x_{i+k}$. $\text{HMAC}_{x_i,x_{i+k}}$ is generated first. $\text{HMAC}_{x_i,x_{i+k}} = K_{x_i,x_{i+k}}(msg)$. $\text{HMAC}_{x_i,x_{i+k-1}} = K_{x_i,x_{i+k-1}}(msg||\text{HMAC}_{x_i,x_{i+k}})$, where $||$ denotes concatenation, and in general $\text{HMAC}_{x_i,x_{i+l}} = K_{x_i,x_{i+l}}(msg||\text{HMAC}_{x_i,x_{i+k}}|| \ldots ||\text{HMAC}_{x_i,x_{i+l+1}})$. Finally x_i sends to x_{i+1} the following message $< msg, \text{HMAC}_{x_i,x_{i+k}}, \text{HMAC}_{x_i,x_{i+k-1}}, \ldots,$

$\text{HMAC}_{x_i,x_{i+1}} >$. When x_{i+1} receives the message it authenticates its HMAC, removes it from the message and sends $< msg, \text{HMAC}_{x_i,x_{i+k}}, \text{HMAC}_{x_i,x_{i+k-1}}, \ldots, \text{HMAC}_{x_i,x_{i+2}} >$ to x_{i+2}.

Each receiving node may generate an onion HMAC for the next nodes and thus each message may also include several onion HMACs of up to $f+1$ distinct nodes. The path may fold onto itself and thus it may happen that several onion HMACs are generated by the same node. As a result, each receiving node may verify a variable number of HMACs. Each onion HMAC is completely separated from the other onion HMACs, i.e. it is not based on the other onion HMACs. Assume that x_{h+1} is the first node that has generated an HMAC for x_{i+1}. Next we show an example of a full message for x_{i+1}.

$$
\begin{aligned}
&< msg, \\
&\text{HMAC}^1_{x_{h+1},x_{i+1}} \\
&\text{HMAC}^2_{x_{h+2},x_{i+2}}, \text{HMAC}^2_{x_{h+2},x_{i+1}}, \\
&\ldots \\
&\text{HMAC}^{i-(h+1)}_{x_{i-1},x_{i+k-1}}, \text{HMAC}^{i-(h+1)}_{x_{i-1},x_{i+k-2}}, \ldots, \text{HMAC}^{i-(h+1)}_{x_{i-1},x_{i+1}}, \\
&\text{HMAC}^{i-h}_{x_i,x_{i+k}}, \text{HMAC}^{i-h}_{x_i,x_{i+k-1}}, \ldots, \text{HMAC}^{i-h}_{x_i,x_{i+1}} \\
&>
\end{aligned}
$$

where $\text{HMAC}^j_{x_u,x_v} = K_{x_u,x_v}(msg||\text{HMAC}^j_{x_u,x_{v+m}}|| \ldots ||\text{HMAC}^j_{x_u,x_{v+1}}), h+1 \leq u \leq i, i+1 \leq v \leq i+k, 0 \leq m \leq k-2, 1 \leq j \leq i-h$. The receiving node verifies the last HMAC of each one of the onion HMACs and afterwards it removes these HMACs from the message. Observe that each onion HMAC is generated for a different sequence of nodes and thus each onion HMAC contains different number of HMACs when it is received.

The message may include parts that are updated along the path, for example the hop count or the list of nodes. Therefore, each HMAC is computed while taking it into consideration so when the message is received by node x, x is able to verify the HMAC. An example is given in the DATA and ACK sections below.

5.2 Messages

The algorithm is comprised of DATA, acknowledgement (ACK) and fault announcement (FA) messages. Their structure and usage is described next. The field types are defined in Table 5.1.

Field Types	
NUMBER	An integer number
ID	A node's ID
SEQUENCE	A sequence of IDs, each ID may appear more than once in the sequence
SET	A set of unique NUMBERs or unique IDs
STRING	Any sequence of characters

Table 5.1: The types of the messages' fields

5.2.1 DATA message

DATA messages include the payload the source wants to send to the destination and the routing information needed in order to do it. The format of the DATA message is depicted in Table 5.2.

Invariants

The order of the nodes in the *listOfNodes* is defined by the routing algorithm and the network topology. *pastNodes* include $f+1$ distinct past nodes, *currNode* is a single node and *futureNodes* include f or $f+1$ distinct future nodes. The total number of nodes in *pastNodes* may be bigger than $f+1$ since each node may appear more than once in the list, but by the definition of θ, it may be at most $\theta+1$. This observation also applies to *futureNodes*. Apart from the above there is no limitations on the list and in particular each part may contain nodes from the other parts, e.g. *currNode* can be one of *pastNodes* or nodes of *futureNodes* may have already been included in *currNode* or *pastNodes*.

Each node may appear several times in the path so S , D, *seqNum* and *hopCount* are the unique DATA message identifiers. The *hopCount*, *listOfNodes* and *hmacs* are changed as the DATA message is being relayed. In addition the *payload* may also change since the routing algorithm payload may also change along the path.

Handling a DATA message

Upon receiving a DATA message by node x, x verifies the HMACs of *pastNodes*. As explained later, each HMAC may be based on a different sequence of nodes. In addition, x chooses the nodes that follow it on the path as would have been chosen by the routing algorithm. Node x verifies that *futureNodes* contain f or $f+1$ distinct nodes and that the future nodes equal its chosen nodes. Each node x that accepts a DATA message, must make sure that when the next node receives the DATA message there are $f+1$ distinct nodes in *pastNodes*, there is a current node and at least f distinct

DATA Message Format

S	*D*	*seqNum*	*payload*	*hopCount*	*listOfNodes*	*hmacs*

Field	**Description**
ID *S*	The source's ID
ID *D*	The destination's ID
NUMBER *seqNum*	A unique sequence number generated by *S*
STRING *payload*	The payload of the routing algorithm message which also includes the information the source wants to send to the destination
NUMBER *hopCount*	The number of hops the DATA message has been relayed
SEQUENCE *listOfNodes*	A sequence of nodes that is comprised of the past nodes, the current node and the future nodes, where the past nodes and the future nodes are SEQUENCEs and the current node is an ID. The current node is the node that receives the DATA message, the past nodes are the nodes that precede the current node on the path and the future nodes are the nodes that follow the current node on the path. The past, current and future nodes are denoted *pastNodes*, *currNode* and *futureNodes*, respectively. We use the explicit notation DATA.*listOfNodes.pastNodes* and .*currNode* and .*futureNodes*, respectively, if it is not clear from the text to which list of nodes we refer to. In addition we denote by *firstNode* and *lastNode* the first and last node in *listOfNodes*, respectively.
SET *hmacs*	A set of onion HMACs for the next nodes

Table 5.2: DATA message format

nodes in *futureNodes*. Nodes are added and removed from the list as needed in order to apply to this requirement. It may happen that *futureNodes* already includes $f+1$ distinct nodes; in such a case x does not add new nodes to the list. Otherwise, x should add the next $f+1^{st}$ distinct node. Depending on the routing algorithm and the network topology the path may fold onto itself. So before adding the next $f+1^{st}$ distinct node to *futureNodes*, node x may add several nodes that are already included in *futureNodes*. Before adding the nodes, x adds itself to the list as the one that is responsible for adding the nodes.

Let P be the path of the DATA message and let x be one of the nodes on P. For each appearance of x in P, we denote by *prev-nodes* (*next-nodes*), the $f+1$ distinct nodes that come before (after) this appearance of x in P, respectively. Obviously, the prev-nodes (next-nodes) include less than $f+1$ distinct nodes at the beginning (end) of the path, respectively. The prev-nodes of a DATA message of x are the *pastNodes* of this DATA message. The next-nodes of a DATA message of x include the *futureNodes* of this DATA message and the nodes that were added by x to *listOfNodes* before sending the updated DATA message to the next node.

After updating the list of nodes *hopCount* is increased by 1. Finally HMACs are generated in an onion like style for the next-nodes. Each HMAC is generated based on the DATA message that will be received when this HMAC is verified. *hopCount* and *listOfNodes* are always updated along the path. *payload* may be updated and it depends on the routing algorithm. Each node calculates its HMACs according to the *listOfNodes* of the DATA message it **sends** and *listOfNodes.lastNode* of the sent message is the last node these HMACs are based on. Since *listOfNodes* is updated and past nodes are removed from it, each HMAC is calculated while taking it into consideration. This calculation guarantees that when a node verifies the HMACs of the prev-nodes, all the HMACs of the prev-nodes are (at least) based on *listOfNodes.firstNode*, ..., *listOfNodes.currNode*. Observe that nodes are added to the list and thus when a next node verifies an HMAC it must calculate what is the last node in the list each HMAC is based on. After generating the HMACs, the updated DATA message is sent to the next node.

An example

We give an example of how the list of nodes of a DATA message is updated and how the HMACs are calculated. We assume that $f = 3$ and let the path be

$$\begin{array}{c} \ldots \leftarrow x_{10} \leftarrow x_9 \\ \ldots \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \overset{\searrow}{\rightarrow} x_5 \overset{\leftarrow}{\rightarrow} x_6 \overset{\leftarrow}{\rightarrow} \boldsymbol{x_7} \overset{\leftarrow}{\rightarrow} x_8 \end{array}$$

Assume the DATA message includes the following list of nodes $(x_1)x_5, (x_2)x_6, (x_3)x_7, (x_4)x_8, (x_5)\boldsymbol{x_7}, x_6, x_5, (x_8)x_9$. We denote the second appearance of a node by adding ^{2}, so x_5^2 is the second appearance of x_5 and in the same manner x_7^2 and x_6^2. The nodes in parentheses are the nodes that have added the following nodes to the list, e.g. x_1 has added x_5 and x_5 has added x_7, x_6 and x_5^2. x_5, x_6, x_7 and x_8 are the $f+1$ past distinct nodes, x_7^2 is the current node, i.e. the node that receives the message (it is in bold) and x_6^2, x_5^2 and x_9 are the f future distinct nodes. It should be noted that

x_6 and x_7 do not add any nodes to the *listOfNodes*, since the *futureNodes* already includes $f+1$ distinct nodes when they receive the message (x_6 and x_7 do not appear in parentheses and after (x_5) appears (x_8)).

As can be easily concluded from the list of nodes, x_5 has generated HMACs for its next-nodes: $x_6, x_7, x_8, x_7^2, x_6^2$ and x_5^2. Node x_6 has generated HMACs for x_7, x_8, x_7^2, x_6^2 and x_5^2. Node x_7 has generated HMACs for x_8, x_7^2, x_6^2 and x_5^2 and x_8 has generated HMACs for x_7^2, x_6^2, x_5^2 and x_9. Node x_5 generates an HMAC for x_7^2 based on $(x_1)x_5, (x_2)x_6, (x_3)x_7, (x_4)x_8, (x_5)x_7, x_6, x_5$. x_6 and x_7 based their HMACs on the same *listOfNodes* since they have not added or removed any nodes from it. Node x_8 generates an HMAC for x_7^2 based on $(x_1)x_5, (x_2)x_6, (x_3)x_7, (x_4)x_8, (x_5)x_7, x_6, x_5, (x_8)x_9$. Node x_7^2 verifies the HMACs of the prev-nodes, i.e. x_5, x_6, x_7 and x_8. In order to do it, x_7^2 calculates for each HMAC what is the last node the HMAC is based on. In addition, x_7^2 verifies that x_6^2, x_5^2 and x_9 are the same next nodes it would have chosen. It should be noted that all the prev-nodes have calculated their HMACs for x_7^2 while considering what should be the hop count of the DATA message when it is received by x_7^2. Of course if the payload is changed as well, the HMACs are calculated while taking it into consideration.

After verifying the DATA message, x_7^2 adds nodes to DATA.*listOfNodes* such that when the message is received by the next node, DATA.*listOfNodes.futureNodes* includes at least f distinct nodes. Node x_7^2 must make sure that there are $f+1$ distinct nodes after itself in the DATA message it sends. Assume that the next $f+1^{st}$ distinct node is x_{10} and that it comes immediately after x_9 on the path. So the list of nodes is $(x_1)x_5, (x_2)x_6, (x_3)x_7, (x_4)x_8, (x_5)x_7, \boldsymbol{x_6}, x_5, (x_8)x_9, (x_7)x_{10}$. Node x_7^2 increases the hop count by 1 and generates an onion HMAC for its next-nodes, i.e. x_6^2, x_5^2, x_9 and x_{10}. x_5, x_6, x_7 and x_8 are the $f+1$ past distinct nodes, x_6^2 is the current node and x_5^2, x_9 and x_{10} are the f future distinct nodes. As can be seen, no nodes were deleted from the list by x_7^2, since x_7 appears twice and x_5 is needed for the $f+1$ past distinct nodes. This process is being done at each node. For example, when x_5^2 accepts the DATA message, before sending it to x_9, it removes nodes from the list. x_5, x_6 and x_7 are removed and the past nodes become x_8, x_7^2, x_6^2 and x_5^2.

5.2.2 ACK

ACKs are a confirmation that a specific DATA message was accepted by the generator of the ACK. Each node generates an ACK for *listOfNodes.pastNodes* of the corresponding DATA message it has accepted. The format of the ACK is depicted in Table 5.3.

ACK Format

gen	*S*	*D*	*dataSeqNum*	*dataHopCount*	*hopCount*	*hmacs*

Field	Description
ID *gen*	The generator of the ACK
ID *S*	The source of the corresponding DATA message
ID *D*	The destination of the corresponding DATA message
NUMBER *dataSeqNum*	A unique sequence number of the corresponding DATA message
NUMBER *dataHopCount*	The number of hops the corresponding DATA message has been relayed when it has been accepted by ACK.*gen*
NUMBER *hopCount*	The number of hops the ACK has been relayed
SET *hmacs*	A set of HMACs generated by ACK.*gen*

Table 5.3: ACK format

Invariants

Each time a node generates an ACK, it calculates an onion HMAC for the *pastNodes* of the DATA message it has accepted. The path of the ACK is from *listOfNodes.currNode* to *listOfNodes.firstNode* of the corresponding DATA message. The number of nodes that should accept this ACK are at most $\theta + 1$, i.e. the maximal number of *pastNodes* of a DATA message. The hop count is changed as the ACK is being relayed and the HMACs of the ACK are calculated while taking it into consideration.

Handling an ACK

A node x should accept ACKs from the next-nodes of the DATA message it has accepted. There are in total $f+1$ distinct nodes from which an ACK should be accepted, **but** the total number of ACKs may be bigger (but no more than $\theta + 1$), since each node may generate several ACKs. Once an ACK is accepted, its hop count is increased by 1 and the ACK is sent to the previous node, unless the receiving node is the last node that should accept this ACK.

An example

We return to the previous example, where the list of nodes was $(x_1)x_5, (x_2)x_6, (x_3)x_7, (x_4)x_8, (x_5)x_7, \boldsymbol{x_6}, x_5, (x_8)x_9, (x_7)x_{10}$ and the current node is x_6^2. Node x_6^2 generates an ACK for the *pastNodes*, i.e. $x_5, x_6, x_7, x_8, x_7^2$. It generates the HMACs based on the hop count of the ACK that will be when the ACK is received, i.e. when x_7^2 receives the ACK the hop count is 1, for x_8 it is 2, etc. When x_7^2 accepts the ACK it increases the hop count by 1 and sends it to the previous node on the path, i.e. x_8. When x_5 accepts the ACK it does not sent it backwards since x_5 is the first node of x_6's *pastNodes*.

5.2.3 FA

FAs are an indication that a fault has occurred and that the generator of the FA and possibly its local neighborhood should not participate in the routing anymore. The format of the FA is depicted in Table 5.4. A node x generates an FA^{1st} or an FA^{2nd} in order to notify its local neighborhood that it should stop participating in the routing. By sending an FA^{1st}, x also notifies each node y in its local neighborhood that y should generate an FA^{2nd} and by sending these FA^{2nd}s the local neighborhood of x will stop participating in the routing as well.

FA Format

gen	*type*	$\mu TESLAInterval$	$\mu TESLAHmac$

Field	Description
ID *gen*	The generator of the FA
NUMBER *type*	The type of the FA. It may be either FIRST or SECOND. If the type is FIRST this FA is denoted by FA^{1st} and if the type is SECOND this FA is denoted by FA^{2nd}. FA^{1st}.*gen* and FA^{2nd}.*gen* are the generators of this FA^{1st} and FA^{2nd}, respectively
NUMBER $\mu TESLAInterval$	The interval at which the key for this FA will be disclosed
NUMBER $\mu TESLAHmac$	A μTESLA HMAC generated by FA.*gen*

Table 5.4: FA format

Invariants

The FA must be received before its μTESLA key is disclosed. It should be noted that the μTESLA keys are sent periodically ([23]).

Handling an FA

The FA is broadcasted to the local neighborhood of the FA.*gen*. The FA can only be verified by the nodes that can verify the μTESLA HMAC, and thus it can only be verified by the σ local neighborhood of the FA.*gen*.

Generating and verifying a μTESLA HMAC is described in [23]. First an FA is generated and sent. Upon receiving the FA, the receiving node makes sure that the structure is correct and that the μTESLA key has not been disclosed yet, i.e. that the FA is *safe*. If it is safe, the FA is broadcasted and stored. Later the μTESLA key is sent and upon receiving it the μTESLA HMAC is verified. Once verified, the μTESLA key is broadcasted and the FA is processed.

FAs are flooded to the local neighborhood of the generator of the FA. A non-faulty node broadcasts each FA it accepts. When an FA is accepted, it is stored until it is not safe, and if the same FA is received for a second time it is ignored, so each FA is broadcasted only once. There is an exception to this rule that will be explained later.

From now on we do not distinguish between the FA and the μTESLA key, so when it is said that an FA was sent or received, it means that both the FA and its μTESLA key were sent or received, and accepting an FA means that the FA has been verified after the μTESLA key was received and the μTESLA HMAC was verified. When we say that the FA was accepted t time units after it was sent, we mean after the FA itself was sent.

5.3 Variables

Each node x stores the following variables.

- ID id - the unique identifier of x.
- SET *blackList* - a set of nodes that is denoted by the BL of x or in short BL(x). If $y \in$ BL(x) then x will not accept any message that includes an HMAC or a μTESLA HMAC generated by y and it will not send a message that includes an HMAC for y.
- SET *grayList* - a set of nodes that is denoted by the GL of x or in short GL(x). This set includes the nodes that are about to be added to the BL and are handled

sometimes as the nodes in the BL and sometimes as not. When a node sends a DATA message and adds nodes to the message's list, it does not consider a node that is neither in its BL nor in its GL, BUT if a node receives a DATA message that has already included nodes that are in its GL, this is legal, and it calculates its next nodes while considering these nodes.

- NUMBER *dataSeqNum* - a unique sequence number for each DATA message generated by x, i.e when x generates a new DATA message, it increases *dataSseqNum* by 1 and sets DATA.*seqNum* to be *dataSeqNum*.
- NUMBER *isGoingToDeactivate* - this is a flag that is set to TRUE upon the activation of a final timeout (See next section).
- The i-hop neighborhood of x consists of nodes which are at most i hops away from x and is denoted by $\text{LN}(x, i)$. For each node $y \in \text{LN}(x, f+1)$, x stores the symmetric key of x and y, the 1HNs of y and the μTESLA information of y, i.e. an authenticated key of μTESLA and μTESLA schedule of y. For each node $y \in \text{LN}(x, \sigma) \backslash \text{LN}(x, f+1)$, x stores only the μTESLA information of y. Each time an FA that was generated by $y \in \text{LN}(x, \sigma)$ is accepted by x, y's μTESLA information that is stored by x may be updated as explained in [23].
- Node x may appear in the path of a DATA message several times. For **each** appearance in the path (instance of the DATA message) it stores the following:
  NUMBER *ackNext* - the number of hops the next ACK should be relayed before being received, i.e. when the next ACK is received, ACK.*hopCount* should equal *ackNext*.
  SEQUENCE *addedNodes* - the nodes x has added to the DATA message (if any).

5.4 Timeouts

Timeouts are used throughout the routing to identify and handle faults. The structure of a timeout is depicted in Table 5.5. There are several types of timeouts; each one has a different role as explained next.

The final timeout is a timeout of type T-FINAL and is set upon the generation of an FA. Upon the activation of this timeout, *isGoingToDeactivate* is set to TRUE. Upon its expiration, the node stops sending and receiving messages. The final timeout is chosen such that a non-faulty node does not deactivate itself before sending all the messages it is expected to send.

The global timeout is a timeout of type T-GLOBAL and is set by the source of

Timeout Structure

type	*id*

Field	Description
NUMBER *type*	The type of the timeout, can be one of the following: T-FINAL, T-GLOBAL, T-BL and T-ACK
ID *id*	This field is set only in case the type is T-BL. In such a case, it equals the ID of the node because of which this timeout was set

Table 5.5: Timeout structure

a DATA message upon sending a DATA message. Upon its expiration, a new DATA message is sent. The global timeout is proportional to the maximal round trip time between the source and the destination so a non-faulty source sends a new DATA message only if a fault has occurred.

The BL timeout is a timeout of type T-BL and is set upon adding a node y to the GL. Upon its expiration y is moved from the GL to the BL. The BL timeout is part of the coordination mechanism between non-faulty nodes. It takes into consideration the maximal difference in the time that non-faulty nodes accept FAs, such that a non-faulty node will not generate an additional FA because of this difference.

The ACK timeout is a timeout of type T-ACK and is set upon sending a DATA message. Upon its expiration an FA^{1st} is generated. The ACK timeout takes into consideration the maximal time it would have taken to accept an ACK or an FA from the next-nodes. In addition this timeout ensure that if two consecutive non-faulty nodes u and v are on a path and v comes immediately after u and both have set an ACK timeout, then u is able to authenticate a message sent by v before its ACK timeout expires.

The global timeout and the ACK timeout are similar to the timeouts described in [1]. Each timeout is set for a different period of time. Let T-FINAL-TIME, T-GLOBAL-TIME, T-BL-TIME and T-ACK-TIME denote the values of the final timeout, the global timeout, the BL timeout and the ACK timeout, respectively. The values themselves will be defined later.

Chapter 6

The Byzantizer Algorithm

Next we give a detailed description of the Byzantizer algorithm. The algorithm is depicted in Figures 6.1 and 6.2. The algorithm begins when the source sends a message to the destination (lines A1, J1-J8). The source chooses the nodes that will follow it on the path, i.e it chooses its next-nodes, as would have been chosen by the routing algorithm such that its next-nodes contain $f + 1$ distinct nodes. Since the path may fold onto itself, each node may be included in the next-nodes more than once. The next-nodes must not be in the GL or the BL of the source. The source sets DATA.*listOfNodes* to be the concatenation of the source itself and its next-nodes (In Figure 6.1, line J3, the concatenation is denoted by ||). The HMACs are generated in an onion like style for the next-nodes. The HMACs are computed while taking into consideration that the DATA message is changed along the path, so when the next-nodes receive the DATA message, they are able to authenticate the message. (See Section 5.2.1). An ACK timeout is set for accepting an ACK from each one of the next-nodes. *ackNext* is set to 1, since the next ACK should be generated by the next node on the path.

The *ameba* contains all nodes that received the DATA message and set their ACK timeouts, and have pending ACK timeouts. Thus, initially the *ameba* contains only the source.

It may happen that a DATA message may not reach its destination because of faults. For this reason a global timeout is set and upon its expiration the source sends a new DATA message (lines J6 and I1). It's exact value (T-GLOBAL-TIME) is defined in Theorem 7.7.

```
Upon generating a DATA message by the source S to the destination D
(A1)   generateSendData(S, D, payload)

Upon receiving a data by u
(B1)   if verifyData(data) == TRUE
(B2)     generateSendAck(data);
(B3)     if reached D
(B4)       generateSendData(D, S, payload');
(B5)     else
(B6)       store(data);
(B7)       if data.listOfNodes.futureNodes include f distinct nodes
(B8)         addedNodes = chooseAdditionalFutureNodes(data);
(B9)       else
(B10)        addedNodes = NULL;
(B11)      updateListOfNodes(addedNodes, data);  data.hopCount++;
(B12)      hmacs = generateDataHmacs(data);  add hmacs to data;
(B13)      ackNext = 1;  setTimeouts(T-ACK, data);
(B14)      send data to nextNode(data);

Upon receiving an ack by u
(C1)   data = getData(ack);
(C2)   if data ≠ NULL && validateHmacs(ack) == TRUE
(C3)     if isLast(ack.gen, data)
(C4)       removeData(data);
(C5)     else
(C6)       ackNext = getAckNext(data);
(C7)       removeTimeout(ackNext, data);  ackNext++;
(C8)     if u ≠ firstPrevNode(ack.gen, data)
(C9)       ack.hopCount++;  send ack to prevNode(data);

Upon the expiration of timeout with timeout.type == T-ACK
(D1)   removeAllData();  generateSendFa(FIRST);
(D2)   if isGoingToDeactivate == FALSE
(D3)     setTimeout(T-FINAL);  isGoingToDeactivate = TRUE;

Upon accepting an fa by u
(E1)   if fa.gen ∉ blackList ∪ grayList && fa.gen ∈ LN(u, f + 1)
(E2)     add fa.gen to grayList;  setTimeout(T-BL, fa.gen);
(E3)   if fa.type == FIRST
(E4)     generateSendFa(SECOND);
(E5)     if isGoingToDeactivate == FALSE
(E6)       setTimeout(T-FINAL);  isGoingToDeactivate = TRUE;
(E7)     ∀ stored data, s.t. fa.gen ∈ activeNodesData(data)
(E8)       removeData(data);

Upon the expiration of timeout with timeout.type == T-BL
(F1)   move timeout.id from graylist to blacklist;

Upon the expiration of timeout with timeout.type == T-FINAL
(G1)   stop sending and receiving messages;

Upon accepting data' by S
(H1)   data = getOriginalData(data');
(H2)   removeData(data);  removeTimeout(T-GLOBAL, data);

Upon the expiration of timeout with timeout.type == T-GLOBAL
(I1)   generateSendData(S, D, payload);
```

Figure 6.1: The Byzantizer algorithm

```
generateSendData(S, D, payload)
(J1)   dataSeqNum++;   hopCount = 1;
(J2)   nextNodes = chooseNextNodes(D);   addedNodes = NULL;
(J3)   data = <S, D, dataSeqNum, payload, hopCount, S ||nextNodes>;
(J4)   hmacs = generateDataHmacs(data);   add hmacs to data;
(J5)   ackNext = 1;   setTimeouts(T-ACK, data);
(J6)   setTimeout(T-GLOBAL, data);
(J7)   storeData(data);
(J8)   send data to nextNode(data);

verifyData(data)
(K1)   return (     validateHmacs(data) == TRUE
(K2)           && validateListOfNodes(data) == TRUE
(K3)           && validateSeqNumAndHopCount(data) == TRUE);

generateSendAck(data)
(L1)   hopCount = 1;
(L2)   ack = <id, data.S, data.D, data.seqNum, data.hopCount, hopCount>;
(L3)   hmacs = generateAckHmacs(data, ack);   add hmacs to ack;
(L4)   send ack to prevNode(data);

getData(ack)
(M1)   ∀ stored data s.t. data.S == ack.S && data.D == ack.D && data.seqNum == ack.dataSeqNum
(M2)      ackNext = getAckNext(data);
(M3)      if ackNext == ack.hopCount && data.hopCount + ackNext == ack.dataHopCount
(M4)         return data;
(M5)   return NULL;
```

Figure 6.2: The Byzantizer algorithm (cont). Italic font as in *ackNext* is used for variables. Roman (TypeWriter) font as in verifyData (`setTimeout`) is used for functions that their implementation is (is not) depicted in the figures, respectively.

Once a DATA message is received by a node x, it is verified (lines B1, K1-K3). Observe that x was chosen by the nodes of the *ameba* to be added to the *ameba*. Node x checks that the structure of the message is correct and the HMACs of DATA.*listOfNodes.pastNodes* (or in short *pastNodes*). Node x verifies that there are $f+1$ distinct nodes in *pastNodes*. If the source S is the first node in DATA.*listOfNodes*, then there may be less than $f+1$ distinct nodes. In addition, x verifies that there are f or $f+1$ distinct nodes in DATA.*listOfNodes.futureNodes* (or in short *futureNodes*). If the destination D is the last node of DATA.*listOfNodes*, then there may be less than f distinct nodes. Let α be the number of distinct nodes in *futureNodes* (f, $f+1$ or less than f nodes, respectively). Node x chooses the nodes that follow it on the path as would have been chosen by the routing algorithm such that they include α distinct nodes. These nodes are denoted by *chosen-nodes*. The chosen-nodes must not be in BL(x) and must not be in GL(x) unless *futureNodes* already include a node(s) u that is in GL(x); in such a case x should choose its chosen-nodes while considering u. Node x verifies that the chosen-nodes equal *futureNodes*. If they are equal it means that x agrees with the nodes of the *ameba* regarding the future nodes. If x has already stored

a DATA message with the same *S, D* and *seqNum* it ensures that the DATA.*hopCount* is different from the *hopCount* of the already stored DATA messages.

Node x processes the DATA message after verifying it (lines B2-B14). Observe that now x is part of the *ameba*. An ACK which is authenticated by an onion HMAC is generated for DATA.*listOfNodes.pastNodes* (lines B2, L1-L4). The DATA.*S*, DATA.*D*, DATA.*seqNum* and DATA.*hopCount* are copied to the ACK so each node that accepts the ACK is able to know to which instance of the DATA message it relates. By sending the ACK, x notifies the nodes of the *ameba* that it has joined the *ameba*. If x is the destination it sends a reply (a new DATA message) to the source, otherwise the message is stored. Observe that there may be cycles in the path so x may store several instances of the same DATA message and each instance is managed separately of the other instances. The DATA message is updated before being sent to the next node on the path (lines B7-B12). If DATA.*listOfNodes.futureNodes* include only f distinct nodes and D is not the last node of *listOfNodes*, x adds nodes to *listOfNodes* according to the routing algorithm. So when the next node receives the updated DATA message, the number of distinct nodes in *futureNodes* will be at least f, i.e. x adds nodes to *listOfNodes* such that there are $f+1$ distinct nodes after x in the list. The added nodes except the last added node, are of course among the nodes that are already included in *futureNodes*. The last added node, i.e. the $f+1^{st}$ distinct node, is not one of *futureNodes* and it cannot be in GL(x) or BL(x). The nodes that x adds to the *listOfNodes* are stored in *addedNodes*. In addition, *pastNodes* of the updated message should include only $f+1$ distinct nodes so nodes may be removed from the beginning of *listOfNodes* in order to apply to this requirement. The addition and removal of nodes ensure that the *ameba* contains $f+1$ distinct nodes even if the path folds into itself. Finally, an onion HMAC is generated for the next-nodes.

Assume x **receives** a DATA message m_1 and **sends** an updated DATA message m_2. As stated before, prev-nodes of m_1 of x are the nodes that precede x, i.e. m_1.*listOfNodes.pastNodes*. Next-nodes of m_1 of x are the nodes that follow x, i.e. the nodes that follow x in m_2.*listOfNodes*. These nodes consist of the nodes of m_1.*listOfNodes.futureNodes* and the nodes of *addedNodes* x has added to m_1. Obviously prev-nodes and next-nodes are calculated per (instance of) DATA message x receives.

Upon receiving an ACK by node x, the ACK is verified (lines C1-C2, M1-M5). The corresponding DATA message is one of the instances of the DATA messages that are stored and has the same ACK.*S*, ACK.*D* and ACK.*dataSeqNum*. Each instance has its own *ackNext* (lines C6, M2) and *hopCount*. The right instance is chosen by

checking that *ackNext* equals the ACK.*hopCount*, and since the path may repeat on the same cycle several times we also check that the DATA.*hopCount* + *ackNext* = ACK.*dataHopCount* (line M3). After the correct instance of the DATA message is found node x knows its next-nodes. The source of the ACK should be the *ackNext-th* node of the next-nodes. Node x verifies that the structure of the ACK is correct and the HMAC of the ACK.

Once the ACK is verified by x (lines C3-C9) the corresponding instance of the DATA message is analyzed. If ACK.*gen* is the last node of next-nodes of this instance of DATA message, no more ACKs should be accepted for it and this instance of DATA message, its timeouts and variables are removed, i.e. x removes itself from the *ameba*. Otherwise the ACK timeout for that ACK is removed and *ackNext* is increased by 1. If x is not the first node of ACK.*gen*'s prev-nodes then ACK.*hopCount* is increased by 1 and the ACK is sent to the previous node on the path.

Upon the expiration of the ACK timeout at node x (lines D1-D3), all the DATA messages are removed, i.e. the messages themselves, their timeouts and their variables are removed, and an FA^{1st} is generated. This FA^{1st} is broadcasted and is authenticated by using μTESLA and thus it can only be authenticated by the σ neighborhood of FA^{1st}.*gen*. The key used for this FA^{1st} is the key that its μTESLA interval starts at least σ time units after sending the FA^{1st}. By sending an FA^{1st}, x notifies $\text{LN}(x, \sigma)$ that a fault has occurred and that x should not participate in the routing anymore. Moreover, each node y in $\text{LN}(x, \sigma)$ should generate an FA^{2nd}, i.e. y should not participate in the routing anymore as well. Upon sending the corresponding μTESLA key, a final timeout is set if it is not already set. It's exact value (T-FINAL-TIME) is defined in Lemma 7.13. It should be noted that a node may generate several FAs and the final timeout is only set upon the generation of the first FA.

Accepting an FA is described in Section 5.2.3. Remember that a non-faulty node accepts each FA only once. The exception to this rule is the case where an FA^{1st} is received from the immediate next (respectively, previous) node on a path of some DATA message(s) and FA^{1st}.*gen* is one of the active nodes and one of the next-nodes (respectively, prev-nodes) of this DATA message(s). In such a case, if this FA^{1st} is verified it is still accepted (despite being already stored). The active nodes of a DATA message are calculated individually by each node x and they are the nodes in next-nodes or prev-nodes of x, including x, that should accept the same ACK x is waiting for. Once accepted, the FA is broadcasted and processed (lines E1-E8). As stated before, node x may update the μTESLA information of FA.*gen* that it stores. If y the FA.*gen* is not in $\text{GL}(x)$ and not in $\text{BL}(x)$ and it is in $\text{LN}(x, f + 1)$ then it is added to

GL(x) and a BL timeout is set. It's exact value (T-BL-TIME) is defined in Lemma 7.16. Upon the expiration of this timeout (line F1), the FA.*gen* is added to BL(x) and no more messages that include its HMAC will be accepted and no DATA message will be accepted if the FA.*gen* $\in$ DATA.*listOfNodes*. The GL and BL mechanism is used to synchronize between non-faulty nodes and it ensures that non-faulty nodes will not generate additional FAs as a result of not receiving FAs at the same time. If the accepted FA is an FA^{1st}, an FA^{2nd} is generated. The key used for this FA^{2nd} is the key that its μTESLA interval starts at least $f+1$ time units after sending the FA^{2nd}. By sending an FA^{2nd}, x notifies LN($x, f+1$) that a fault has occurred and that x should not participate in the routing anymore. If FA^{1st}.*gen* is one of the active nodes of a current DATA message(s) then this DATA message(s) is removed.

Once the destination has accepted the DATA message it sends a new DATA message, DATA', to the source (line B4) as a reply. DATA' includes DATA.*seqNum* so when the source accepts the reply it can know to which DATA message it has sent it relates (line H1) and thus it can be sure that its message has been accepted by the destination. In such a case the original DATA message and the global timeout are removed at the source (line H2).

6.1 Characteristics

The Byzantizer algorithm has the following characteristics. All routing decisions are based on the $f+1$ local neighborhood. DATA, ACKs and FAs are authenticated and thus if a faulty node alters the messages they will not be accepted. A faulty node can only generate new messages, but it cannot generate messages as if they have originated from another non-faulty node, since it cannot generate an HMAC or a μTESLA HMAC of another non-faulty node. After receiving ACKs from the next $f+1$ distinct nodes, a node can be sure that the DATA message has been passed on, since a non-faulty node has accepted it and thus the faulty node cannot perform the selective forwarding attack and the black hole attack. Observe that μTESLA authentication is only used to verify FAs and thus no delays are caused to the routing when there are no faults.

Encryption can be used in addition to authentication and this is an algorithm parameter. Each pair of nodes can generate a key for encryption in addition to the key for authentication and particularly, the source may have a symmetric key with the destination and the message may be encrypted with that key.

Chapter 7

Analysis

In this chapter we prove that if the source sends a DATA message to the destination along a path P and P is more than $\sigma+\mu+1$ hops away from any faulty node, then the DATA message will be accepted by the destination. In addition we bound the number of DATA and ACK messages that will be sent until the DATA message is accepted by the destination. Finally we bound the total number of FA s that may ever be sent.

7.1 Definitions

We begin the analysis by presenting the definitions that will be used in the proofs.

Definition 7.1.1. *Suspected node: A node v is suspected if exists a non-faulty node u such that $v \in \text{BL}(u)$.*

Observe that a non-faulty node may be suspected.

Definition 7.1.2. *$d(u,v)$: The number of hops in the shortest path between u and v.*

Definition 7.1.3. *$d_{path}(u,v)$: The number of hops in the specified path between u and v. It should be noted that v comes after u in the path.*

Definition 7.1.4. *$d^{unq}_{path}(u,v)$: The number of distinct nodes between u and v along the specified path.*

It is clear that $\forall u, v \text{ and path},\ d(u,v) \leq d^{unq}_{path}(u,v) \leq d_{path}(u,v)$.

Definition 7.1.5. *$d(X,Y)$: The minimal number of hops between any node $x \in X$ and any node $y \in Y$, i.e for $X \subseteq V$ and $Y \subseteq V$ let $d(X,Y) = \min_{x \in X, y \in Y} d(x,y)$*

Definition 7.1.6. *A node v accepts message m: v accepts m if it has received and verified m.*

7.2 Proof

We begin the proof by stating the main lemma (the proof of this lemma is given later in Lemma 7.21). Based on that lemma we bound the radius of influence a faulty node has.

Lemma 7.1. *If u has generated an* FA^{1st} *then there exists w s.t. w is faulty and $d(u, w) \leq f + 1$.*

The next lemma is similar to the previous lemma and it proves that when an FA^{2nd} is generated a faulty node must be near by.

Lemma 7.2. *If node u has generated an* FA^{2nd} *then there exists w s.t. w is faulty and $d(u, w) \leq f + 1 + \sigma$.*

Proof. If u is faulty we are done; so consider the case that u is non-faulty. Observe that a non-faulty node u generates an FA^{2nd} only as a result of accepting an FA^{1st} (See Figure 6.1, lines E3-E4). By Lemma 7.1, if an FA^{1st} was generated by a node v then $d(v, w) \leq f + 1$. Since the range of the FA^{1st} is σ, we can conclude that if u has accepted an FA^{1st} then $d(u, w) \leq f + 1 + \sigma$. □

The theorem below bounds the radius of influence a faulty node has.

Theorem 7.3. *If u is suspected then there exists w s.t. w is faulty and $d(u, w) \leq f + 1 + \sigma$.*

Proof. Observe that a node u becomes suspected by a non-faulty node v only if an FA was accepted by v and FA.*gen* is u (See Figure 6.1, lines E1-E2 and F1). The theorem holds by Lemmas 7.1 and 7.2 in which it was proven that if u has generated an FA then $d(u, w) \leq f + 1 + \sigma$. □

So far we have proved the bound on the maximal distance from a faulty node in which faults may occur. Next we bound the number of messages that should be sent in the presence of faulty nodes. The next lemma proves that an FA^{1st} must be generated if a DATA message is not accepted by its destination.

Lemma 7.4. *If there is a path from the source to the destination and the source has sent a* DATA *message to the destination and the message has not been accepted by the destination, then an* FA^{1st} *would have been generated.*

Proof. The path is chosen exactly as would have been chosen by the routing algorithm and it is guaranteed that if there is a path to the destination the routing algorithm finds it, so the destination must accept the message if there are no faults.

Next we prove that if there were faults an FA^{1st} must have been generated. Each non-faulty node participates in the routing until it accepts an ACK from its next-nodes, so at least $f+1$ distinct nodes participate in the routing at any single moment and thus there is at least one non-faulty node u among these nodes. It is guaranteed that either an FA^{1st} is accepted by u (remember that an FA^{2nd} does not stop the ACK timeout) or u's ACK timeout expires and it generates an FA^{1st} and the lemma is proven. □

In the following two lemmas we bound the time since an FA^{1st} is accepted or generated by a non-faulty node u, until a non-faulty node v that is a 1HN of a faulty node (u may be v) deactivates itself.

Lemma 7.5. *If a faulty node w generates an* FA^{1st} *and at least one non-faulty node accepts this* FA^{1st}*, then at least one edge of a faulty node will never be used in any subsequent path $f+1+\mu+$*T-FINAL-TIME *time units after the* FA^{1st} *has been accepted.*

Proof. Node w generates an FA^{1st} and a non-faulty node x accepts it and thus x generates an FA^{2nd}. Let the path between w and x be $w, x_i, \ldots, x_k, x$. It is clear that $d(w,x) \leq f+1$ since there are at most f faulty nodes. If $w, x_i, \ldots, x_k$ are faulty we are done, since x will stop sending and receiving messages and thus the edge between x and x_k will never be used in subsequent paths. If there is a non-faulty node(s) among $w, x_i, \ldots, x_k$, then let y be the first non-faulty node among these nodes. Node y must accept this FA^{1st}, since otherwise it would not have sent it to x. So y generates an FA^{2nd} and the same analysis done for x can be applied to y, and thus the edge between y and the faulty node that precedes it on the path will never be used in subsequent paths.

Next we prove that one of the edges of a faulty node will not be used $f+1+\mu+$T-FINAL-TIME after the FA^{1st} has been accepted. Assume the FA^{1st} has been accepted at time t_0 by a non-faulty node x that is a 1HN of a faulty node. Node x sends an FA^{2nd} and its corresponding key by $t_1 = t_0 + f + 1 + \mu$ and it deactivates itself by

t_1+T-FINAL-TIME= $t_0 + f + 1 + \mu$+T-FINAL-TIME. So the edge between the faulty node and x will not be used $f + 1 + \mu$+T-FINAL-TIME after accepting the FA^{1st}. □

Lemma 7.6. *If a non-faulty node u generates an* FA^{1st}*, then at least one of the edges of a faulty node will never be used in any subsequent path $\sigma + 2\mu + 2f + 1$+*T-FINAL-TIME *time units after the* FA^{1st} *has been sent.*

Proof. The proof is by analyzing the possible distances between u and the faulty node w because of which the FA^{1st} was generated. An FA^{1st} is generated upon the expiration of the ACK timeout and by Lemma 7.1 it was proven that $d(u, w) \leq f + 1$.

If $d(u, w) = 1$, i.e. u is a 1HN of the faulty node, then since u stops receiving and sending messages the edge between the u and w will not be included in any subsequent path.

If $1 < d(u, w) \leq f + 1$, then let the shortest path between the u and w for which $d(u, w) \leq f + 1$ be $u, \ldots, x, w$. If the nodes $u, \ldots, x$ are non-faulty then all of them and particularly x accept this FA^{1st} and generate an FA^{2nd}, and thus x will stop sending and receiving messages and the edge between x and w will not be used in any subsequent path. If there is a faulty node on the path between u and x, then let w' be the faulty node and let the path be $u, \ldots, y, w'$. The same analysis done for x can be applied to y and thus the edge between y and w' will not be used in any subsequent path.

Next we prove that one of the edges of a faulty node will not be used $\sigma + 2\mu + 2f + 1$+T-FINAL-TIME after the FA^{1st} has been sent. Assume the FA^{1st} is sent at time t_0. The μTESLA key is sent by $t_1 = t_0 + \sigma + \mu$. The non-faulty node that is a 1HN of w (u, x or y in the analysis above) is at most f hops from u (when the non-faulty node is x and $d(u, w) = f + 1$), and thus it accepts the FA^{1st} by $t_2 = t_1 + f$. This non-faulty node deactivates itself since it sends an FA^{1st} or an FA^{2nd}. The worst case is when an FA^{2nd} is generated by the furthest node, i.e. by x. Node x sends an FA^{2nd} and its corresponding key by $t_3 = t_2 + f + 1 + \mu$ and it deactivates itself by t_3+T-FINAL-TIME= $t_0 + \sigma + \mu + f + f + 1 + \mu$+T-FINAL-TIME= $t_0 + \sigma + 2\mu + 2f + 1$+T-FINAL-TIME. So the edge between the faulty node and the non-faulty node will not be used $\sigma + 2\mu + 2f + 1$+T-FINAL-TIME after sending the FA^{1st}. □

Is should be noted that the range of an FA^{1st} is $\sigma = 2(f + 1)$ and not $f + 1$, in order to isolate the faulty nodes more quickly. Since $d(\text{FA}^{1st}.gen, w) \leq f + 1$ it is possible that at least the $f + 1$ local neighborhood of the faulty node w accept this FA^{1st} and generate an FA^{2nd} and thus the faulty node will be isolated. The $f + 1$ local

neighborhood of w generate FAs in order to try to prevent from w to be included in any path even if there are more faulty nodes near w.

In the following two theorems we bound the number of messages that are sent before a DATA message is accepted by the destination.

Theorem 7.7. *If the source S and the destination D are non-faulty and S sends a* DATA *message(s) to D and there is a path P which goes from S to D and back to S for which $d(P,F) > f+1+\sigma$, then D accepts the* DATA *message after S sends at most $\tau \cdot f + 1$* DATA *messages.*

Proof. The proof follows from Theorem 7.3 and Lemmas 7.4, 7.5 and 7.6. It was proven by Lemma 7.4 that once a DATA message was sent and it has not been accepted by the destination an FA^{1st} is generated. This conclusion also applies to the DATA message the destination sends to the source as a reply to the DATA message it has accepted from the source.

Upon the generation of an FA^{1st} the non-faulty nodes remove the DATA message, and since the global timeout at S expires, a new DATA message with increased sequence number is sent (See Figure 6.1, line I1). Next we prove that if T-GLOBAL-TIME is set to 2·PATH-MAX-LEN$+(2+\sigma+\mu)(\theta+1)+\mu+2f+1+$T-FINAL-TIME, then by the time the new DATA message is sent, a non-faulty node that is a 1HN of a faulty node has deactivated itself. 2·PATH-MAX-LEN is the maximal round trip time between S and D. $(2+\sigma+\mu)(\theta+1)+\mu+2f+1+$T-FINAL-TIME is the maximal time since a non-faulty node u accepts a DATA message until a non-faulty node v, which is a 1HN of a faulty node (v may be u), generates an FA^{1st} or an FA^{2nd} and deactivates itself. A non-faulty node u generates an FA^{1st} upon the expiration of its ACK timeout which can happen at most $2(\theta+1)+(\sigma+\mu)\theta$ (See Lemma 7.14) after accepting the DATA message. This is also the maximal time till which u may accept an FA^{1st} and generate an FA^{2nd}. So By Lemmas 7.5 and 7.6 we can conclude that the time since u accepts a DATA message until v deactivates itself is at most $2(\theta+1)+(\sigma+\mu)\theta+\sigma+2\mu+2f+1+$T-FINAL-TIME $= (2+\sigma+\mu)(\theta+1)+\mu+2f+1+$T-FINAL-TIME.

It is clear that since an FA^{1st} was generated, it must have been generated by 2·PATH-MAX-LEN$+2(\theta+1)+(\sigma+\mu)\theta$ since the DATA message has been sent, so by at most T-GLOBAL-TIME a non-faulty node has deactivated itself. Since the new DATA message is sent **after** the global timeout has expired, by the time the new DATA message reaches v (if at all), it has already deactivated itself and thus the path of the new DATA message does not go through v, i.e. one of the edges of a faulty node is not used any more. It is obvious that after at most $\tau \cdot f$ DATA messages are sent, all

the faulty nodes are isolated and do not participate in any path.

Next we prove that since there exists such a path P for which $d(P, F) > f+1+\sigma$, the $\tau \cdot f + 1^{st}$ DATA message is accepted by the destination. A node x becomes suspected only if $d(x, F) \leq f+1+\sigma$. Since $d(P, F) > f+1+\sigma$, all the nodes on the path are non-faulty and not suspected so they act according to the protocol and do not delete any DATA message or deactivate themselves. The nodes agree on the list of nodes, since no node is suspected and since their views of the local neighborhoods are consistent by the assumptions on the neighborhood discovery algorithm (Chapter 4). The nodes are chosen exactly as in the routing algorithm and since it is guaranteed that the routing algorithm reaches the destination, the $\tau \cdot f + 1^{st}$ DATA message is accepted by the destination.

Since D is non-faulty, it sends a reply to S which must be accepted by S for the same reasons that the $\tau \cdot f + 1^{st}$ DATA message has been accepted by D, and thus the global timeout at S is deleted and no more DATA messages are generated by it. □

Some remarks about the previous theorem. It is obvious that the ith DATA message, $i < \tau \cdot f + 1$, may be accepted by D and D's reply may be accepted by S and it depends on the location and behaviour of the faulty nodes. Assume that this ith DATA message was sent along a path Q. It may happen that $d(Q, F) \leq f + 1 + \sigma$. Observe that if the jth DATA message, $j < \tau \cdot f + 1$, has been accepted by D but the reply of D has not been accepted by S by the expiration of the global timeout at S, S will send another DATA message to D.

Using a competitive underlining ad hoc routing scheme with our Byzantizer obtains the first competitive ad hoc routing scheme resilient against Byzantine failures. The exact form of competitiveness is given in the following theorem.

Theorem 7.8. *If the source S and the destination D are non-faulty and S sends a* DATA *message(s) to D and there is a path P of length Ψ which goes from S to D and back to S for which $d(P, F) > f + 1 + \sigma$, then D accepts the* DATA *message after at most $O(\Psi^2 f(\tau \cdot f + 1))$* DATA *and* ACK *messages are sent.*

Proof. The proof is by Theorem 7.7 and by assuming that the underlining ad hoc routing algorithm is GOAFR ([13]), which is competitive when there are no failures. GOAFR traverses at most $O(l^2)$ edges where l is the number of edges of the optimal path. Observe that at most $\tau \cdot f + 1$ messages are sent and each message traverses a different path. Let l_i be the optimal path for message i, $1 \leq i \leq \tau \cdot f + 1$. It is easy to see that $l_i \leq l_{i+1}$, since the $i + 1^{st}$ message was generated because the global timeout

at S has expired which must be the result of the generation of an FA^{1st} while message i was relayed. By the time the $i+1^{st}$ message has reached LN(FA^{1st}.*gen*, σ), nodes were deactivated and thus the optimal path l_{i+1} can only be longer than l_i. Assume the kth DATA message, $1 \leq k \leq \tau \cdot f + 1$, is the last DATA message that was sent by S, i.e. this message was accepted by D and D's reply was accepted by S. And thus, $l_1 \leq l_2 \leq \ldots \leq l_k$ so $O(l_1^2) \leq O(l_2^2) \leq \ldots \leq O(l_k^2)$.

P must be the longest path of a DATA message, since $d(P, F) > f + 1 + \sigma$ and thus a DATA message that is sent along P must reach D and D's reply must reach S and no additional DATA messages will be sent. Obviously it may happen that the last DATA message is relayed and accepted along a path Q for which $d(Q, F) \leq f + 1 + \sigma$. Consequently we can conclude that $l_k \leq \Psi$. Each edge costs $O(f)$ messages, since an ACK is sent for the previous (up to) $\theta + 1$ nodes. So each DATA message costs at most $O(\Psi^2 f)$ messages, where by DATA message we mean the DATA message the source sends to the destination and the reply message the destination sends to the source. Since there are at most $\tau \cdot f + 1$ DATA messages, the total number of DATA and ACK messages is $O(\Psi^2 f(\tau \cdot f + 1))$ □

Once all the faulty nodes are isolated, no FAs are generated and all the DATA messages are accepted by their destinations, as long as there is a path to the destination in the remaining graph. If there is such a path of length Ψ, after sending at most $O(\Psi^2 f)$ DATA and ACK messages, the DATA message is accepted by the destination.

Observe that there are at most $\tau \cdot f$ FA flooding phases regardless of the number of DATA and ACK messages. If we assume that the graph is a Unit Disk Graph and the $\Omega(1)$-model as in [13], we can conclude that in LN(FA.*gen*, i) there are $O(i^2)$ nodes and $O(i^2)$ edges. Each FA flooding consists of one FA^{1st} and the FA^{2nd}s that are generated because of it. FA^{1st} is flooded to LN(FA^{1st}.*gen*, σ) and each FA^{2nd} is flooded to LN(FA^{2nd}.*gen*, $f + 1$). Since the number of FA^{2nd}s dominates the total number of FAs in each FA flooding, we can conclude that the total number of FAs in each FA flooding is $O(f^2 \cdot f^2) = O(f^4)$. It should be noted that most of the FAs are sent and received by nodes that are in LN(FA^{1st}.*gen*, σ), i.e. by nodes that are about to deactivate themselves anyway.

Now we return to the proof of the main lemma (Lemma 7.21). We begin by proving Lemmas 7.9, ..., 7.20, which are used in the proof of the main lemma. The first lemma, i.e. Lemma 7.9, proves that if a non-faulty node v accepts a message of another non-faulty node u, then this message is the original message generated by u.

Lemma 7.9. *If a* DATA*, an* ACK *or an* FA *was generated by a non-faulty node u and was accepted by a non-faulty node v, then v has accepted the original message generated by u.*

Proof. First is should be noted that a DATA message or an ACK are changed along the path, e.g. the hop count in increased, so by original message we mean the unchanging parts of the message. We begin by proving the lemma for a DATA message. It should be noted that if v has accepted the message then it has verified the (up to) $f+1$ HMACs of distinct nodes, where each HMAC is generated by the symmetric key of the generator of the HMAC and the receiving node. Let S be the source of the message. If $d^{unq}_{path}(S,v) \leq f+1$, then v has verified the HMAC of the source of the message, and thus this is the original DATA message. If $d^{unq}_{path}(S,u) > f+1$, then there is at least one non-faulty node x among DATA.*listOfNode.pastNodes* that has generated the HMAC and x has already verified the message and thus this is the original DATA message.

An ACK is authenticated by an HMAC that is generated by the symmetric key of the source of the ACK and the receiving node and thus this is the original ACK. Finally, an FA is authenticated by μTESLA HMAC which was generated by the source of the FA. μTESLA relies on one-way function and thus this is the original FA ([22, 23]). □

It should be noted that a faulty node x may generate and send a message on behalf of another faulty node y or even change the message of y and generate a new HMAC while forwarding it. In such a scenario, y whose key was used, is still considered the source and if a non-faulty node v accepts the message, then as far as v is concerned this is the original message generated by y.

In the following lemmas we assume that if a non-faulty node accepts a message of another non-faulty node, then this message is the original message.

Next we prove that if a non-faulty node fails to verify the HMAC of a DATA or of an ACK, then there is a faulty node among the nodes on the path between the generator of the HMAC and the receiving node. Observe that if a node u verifies an HMAC of an ACK, then the generator of the HMAC is one of the next-nodes of the corresponding DATA message of the ACK, and if u verifies an HMAC of a DATA message, then the generator of the HMAC is one of the prev-nodes of that DATA message.

Lemma 7.10. *If a non-faulty node u has received a* DATA *(respectively, an* ACK*) and u has failed to verify the* HMAC*(s), then there is a faulty node w among the previous (respectively, next) $f+1$ distinct nodes along the path, i.e. $d(w,u) \leq f+1$.*

Proof. Node u has failed to verify the HMAC(s) and thus there is at least one invalid HMAC. There are two options, either the generator of the HMAC has generated invalid HMAC or a node on the path has altered the message. Since HMACs are generated by the prev-nodes or by the next-nodes and these nodes are in $\text{LN}(u, f+1)$, option 1 implies that the faulty node w must be among the prev-nodes or the next-nodes and $d(u, w) \leq f + 1$. Regarding option 2, the faulty node must be among the nodes that have relayed the HMAC, i.e. the prev-nodes or the next-nodes. Each HMAC is relayed by nodes that are at most f hops away from the receiving node and thus $d(u, w) \leq f$ and the lemma is proven. □

It should be noted that since onion HMACs are used, once a faulty node x alters the message, the node after it y cannot verify the message.

In the following two lemmas we bound the time an FA should be accepted at the presence of only non-faulty nodes.

Lemma 7.11. *Let u and v be non-faulty nodes that have a non-faulty path P of length $l \leq \sigma$ between them. If v generates an FA^{1st} then u accepts it after at most $\sigma + \mu + l$ time units since it was sent.*

Proof. Node v is non-faulty so it generates a valid FA^{1st} that is flooded to $\text{LN}(v, \sigma)$, so u should accept it. Since all the nodes in P are non-faulty u receives the FA^{1st} at most l time units after it has been sent. Node v sends the corresponding μTESLA key at most $\sigma + \mu$ after sending the FA^{1st}, and thus by the same logic, u should receive it at most $\sigma + \mu + l$ time units after it has been sent. Node u accepts the FA^{1st} since v generates a valid FA and all the nodes are non-faulty, so they do not alter the messages and u receives the valid messages generated by v. □

By Lemma 7.11 we can conclude that if u is a non-faulty node and $d(u, v) \leq \sigma$ and u does not accept an FA^{1st} generated by v by $\sigma + \mu + d(u, v)$ since it was sent, then there is a faulty node w on the shortest path between u and v and $d(u, w) \leq d(u, v)$. This conclusion holds even if a faulty node has generated and sent the FA^{1st} on behalf of v, since in such a case, v is considered a faulty node. Observe that if v is non-faulty, no other node can generate an FA on behalf of it. We define T-FA-FIRST-LIMIT(i) to be $\sigma + \mu + i$, which is the time till which $\text{LN}(\text{FA}^{1st}.gen, i)$ should accept the FA^{1st} after it was sent.

Lemma 7.12. *Let u and v be non-faulty nodes that have a non-faulty path P of length $l \leq f + 1$ between them. If v generates an FA^{2nd} then u accepts it after at most $f + 1 + \mu + l$ time units since it was sent.*

Proof. The proof is similar to the proof of Lemma 7.11 but the nodes that should accept the FA are the nodes of LN$(v, f+1)$ instead of LN(v, σ), and the μTESLA key is disclosed at most $f+1+\mu$ time units instead of $\sigma+\mu$ time units after sending the FA. □

By Lemma 7.12 we can conclude that if u is a non-faulty node and $d(u,v) \leq f+1$ and u does not accept an FA^{2nd} generated by v by $f+1+\mu+d(u,v)$ since it was sent, then there is a faulty node w on the shortest path between u and v and $d(u,w) \leq d(u,v)$.

We define T-FA-SECOND-LIMIT(i) to be $f+1+\mu+i$, which is the time till which LN(FA^{2nd}.*gen*, i) should accept the FA^{2nd} after it was sent. In addition we define T-FA-LIMIT(i) to be T-FA-FIRST-LIMIT(i) in case of FA^{1st} and T-FA-SECOND-LIMIT(i) in case of FA^{2nd}, and it is the time till which LN(FA.*gen*, i) should accept the FA.

Next we define the values of T-FINAL-TIME and T-ACK-TIME. The ACK timeouts are set for the next-nodes upon accepting a DATA message. The function `setTimeouts`(T-ACK, *data*) (Figure 6.1 line B13 and Figure 6.2 line J5) sets these timeouts. The timeout for a next node x which is i-hops away along the path is $2i+(\sigma+\mu)(i-1)$. In Lemma 7.14 we prove that this calculation of the ACK timeout ensures that if two consecutive non-faulty nodes u and v are on a path and v comes immediately after u and both have set an ACK timeout, then u is able to authenticate a message sent by v before its ACK timeout expires.

The final timeout is set upon generating an FA. A node x that has generated an FA may be in the GL of the other nodes and may still be included in a path. Node x may be added to the GL of the other nodes before it was added to the path or while it was part of the path. The first case may happen because it takes time to flood FAs and thus a remote node may add x to the path while at nodes that are near x the FA was already accepted. The second case may happen when several paths are used simultaneously and on one path an FA^{1st} is generated. Next we prove that if there are no faults, a non-faulty node deactivates itself only when it is not and never will be included in any path.

Lemma 7.13. *Let* T-FINAL-TIME *be* $2(\sigma+\mu+2)\theta+f+5$ *and* x *be a non-faulty node that generates an* FA *and all non-faulty nodes in* LN$(x, f+1)$ *accept this* FA *within* T-FA-LIMIT$(f+1)$ *since it was sent. If a* DATA *message is sent in* LN$(x, f+1)$ *only by non-faulty nodes after the* FA *is accepted, then* x *is not included in* DATA.*listOfNodes after deactivating itself.*

Proof. Node x sets the final timeout upon sending an FA, and when it expires, x deactivates itself (See Figure 6.1, lines D1-D3, E4-E6 and G1). We are going to prove that T-FINAL-TIME is longer than the time x participates in the path, i.e. by the time the final timeout expires, no non-faulty node is waiting for an ACK or FA^{1st} from x. Remember that x is non-faulty so it generates all the messages correctly and on time. Since all the non-faulty nodes in $\text{LN}(x, f+1)$ accept this FA within $\text{T-FA-LIMIT}(f+1)$ since it was sent, x is added to the GL and the BL of these nodes and thus it is not added to DATA.*listOfNodes* by a non-faulty node after the FA is accepted. It should be noted that even though v has generated an FA, it may continue to participate in the routing and it does not matter which FA v has generated, either an FA^{1st} or FA^{2nd}.

Each non-faulty node participates in the path until it has accepted ACKs from all the next-nodes, or until its ACK timeout expires, or until an FA^{1st} is accepted for an active node on the path, (See Figure 6.1, lines C3-C4, D1 and E7-E8, respectively). The maximal time x may be relevant for one path is the time since it is included in DATA.*listOfNodes*, until all the nodes preceding x have stopped participating in the path. Let the path P be $u, \ldots, v, x, \ldots, y$. Node x is the FA.*gen* and v is the node that immediately precedes it. Node u has added x to the DATA message before the FA was accepted. It is obvious that $d_P(u, v) \leq \theta$. Node y is the last node from which v should accept an ACK and thus $d_P(v, y) \leq \theta + 1$. So x should set a timeout that its length it at least the time it takes a DATA message to be relayed from u to v and from v to y plus the time it takes v to accepts y's ACK or an FA^{1st}. This time is at most the sum of the ACK timeout u activates for its last next node and the ACK timeout v activates for its last next node. This sum equals $2[2(\theta + 1) + (\sigma + \mu)\theta] = 2(\sigma + \mu + 2)\theta + 4$.

Since u, the node that has added x to the path, may be at most $f+1$ hops from x, it may take the FA to reach u at most $f+1$ time units after x has sent it. So x may have generated the FA at most $f+1$ time units before u has added it to the DATA message, and thus $f + 1$ should be added to the previous calculated time. Consequently we can conclude that x can deactivate itself after at most $2(\sigma + \mu + 2)\theta + 4 + f + 1 = 2(\sigma + \mu + 2)\theta + f + 5 =$ T-FINAL-TIME. □

In the following lemmas we assume, unless stated otherwise, that nodes do not deactivate themselves before sending all required messages.

Next we prove that the ACK timeout is set such that a non-faulty node receives an ACK or an FA^{1st} before its ACK timeout expires. Remember that the ACK timeout for a next node x which is i-hops away along the path is $2i + (\sigma + \mu)(i - 1)$.

Lemma 7.14. *Let u and v be non-faulty nodes such that v comes immediately after u on a path P of some* DATA *message. If u sends this* DATA *message to v and v accepts the message, then u receives an* ACK *or an* FA^{1st} *from v before its* ACK *timeout expires, where this* ACK *was generated by one of u's next-nodes and the* FA^{1st} *was generated by one of the active nodes of u.*

Proof. Observe that each node on the path can generate an FA^{1st} and an ACK, and relay an FA^{1st} or an ACK. Node v is non-faulty and thus it generates an ACK upon accepting a DATA message and an FA^{1st} upon the expiration of its ACK timeout. In addition, upon accepting an ACK or an FA^{1st}, it sends them to u. Remember that FA^{2nd} does not cause the removal of a DATA message and its ACK timeout and thus it is not considered here.

ACKs can be verified once accepted. FAs are verified by μTESLA and thus they can be verified after the μTESLA key is received. As a result, it takes to verify an FA the longest time and in particular an FA^{1st}, since the μTESLA keys are disclosed at most $\sigma + \mu$ after the corresponding FA^{1st} is sent. We are going to prove that the difference between the values of the ACK timeouts of u and v allows u to verify an FA^{1st} before its ACK timeout expires.

We are going to analyze all the possible values of u's *ackNext*. Observe that all the ACK timeouts for the next-nodes are set upon accepting the DATA message. By the lemma's assumption, v comes immediately after u on the path. If u's *ackNext* is 1, then it has not accepted v's ACK yet and it has an active ACK timeout of 2 time units. Only an ACK can be received from v because of this DATA message since an FA^{1st} can be generated only after the ACK timeout has expired, and thus if the ACK timeout has expired at v, v must have already accepted a DATA message and sent an ACK to u. So by 2 time units an ACK must be accepted, since v has accepted the DATA message and must have sent a valid ACK. It should be noted that v may have sent an FA to u because of another path. By the lemma's assumption, v does not deactivate itself before sending all the messages to u and in particular, it sends an ACK to u upon accepting the DATA message. It should be noted that until u accepts v's ACK, v is not among u's active nodes, and thus if u accepts an FA^{1st} that was generated by v before accepting v's ACK, u does not delete the DATA message.

Next we prove that the difference between the values of the ACK timeouts that u and v set for the same next node is $1 + \sigma + \mu$ and this difference allows accepting an FA^{1st} before the ACK timeout at u expires. If u's $ackNext = 2$ then it has an active ACK timeout of $2ackNext + (\sigma + \mu)(ackNext - 1) = 4 + \sigma + \mu$. Node v has an active ACK

timeout of 2 time units, since his *ackNext* is 1, and thus the difference between the values of their ACK timeouts is $2+\sigma+\mu$. Node v starts its ACK timeout upon accepting the DATA message sent by u and thus v's ACK timeout starts 1 time unit after u starts its ACK timeout. As a result the actual difference between the values of their ACK timeouts is $1+\sigma+\mu$. At the worst case v's ACK timeout expires and it generates an FA^{1st}. Since v's ACK timeout has expired after 2 time units and it takes the message to be relayed 1 time unit and the key is disclosed after at most $\sigma+\mu$ since the FA^{1st} was sent, the FA^{1st} can be verified by u after at most $1+2+\sigma+\mu+1=4+\sigma+\mu$, which is u's ACK timeout, and thus the FA^{1st} can be verified by u before its ACK timeout expires.

Next we consider the case where u's *ackNext* > 2. Since all the ACK timeouts for the next-nodes are set upon accepting the DATA message, v has set an ACK timeout for each next node 1 time unit after u has. u's *ackNext* is bigger by 1 than v's *ackNext*, so the ACK timeout of u is bigger by $2+\sigma+\mu$ than v's ACK timeout and thus the actual difference is $1+\sigma+\mu$. By the same analysis as done in the previous paragraph, we can conclude that u can verify an FA^{1st} before its ACK timeout expires. □

The next lemma proves that at the presence of only non-faulty nodes; if a non-faulty node accepts an FA^{1st} and as a result deletes a DATA message then all its prev-nodes will delete the DATA message before their ACK timeouts expire.

Lemma 7.15. *Let u be a non-faulty node that has either generated or accepted an FA^{1st} and as a result deleted a* DATA *message(s). If the prev-nodes and the next-nodes of u of this* DATA *message are non-faulty, then this deletion does not cause the generation of another FA^{1st} by a non-faulty node that is among the prev-nodes of u.*

Proof. Upon generating or accepting an FA^{1st} by a non faulty node u, all the relevant DATA messages are deleted (See Figure 6.1, lines D1, E7-E8). These DATA messages are the DATA messages that $\text{FA}^{1st}.gen$ is one of the active nodes of u, i.e one of the nodes of next-nodes or prev-nodes of u, including u, that should accept the same ACK u is waiting for. If u is the $\text{FA}^{1st}.gen$, it deletes all its DATA messages. Each node calculates the active nodes of each (instance of) DATA message it has stored. The active nodes are updated as ACKs are accepted.

First we prove that all the active nodes, as calculated by u, are in the range of this FA^{1st}, and more precisely, they are at most f hops away from $\text{FA}^{1st}.gen$. Remember that the range of an FA^{1st} is $\sigma \geq f$. Since each node sends its ACK for its prev-nodes, at most $f+1$ distinct nodes should accept the same ACK and thus there are at most

$f+1$ distinct active nodes (as calculated by u) and the distance between any two active nodes is at most f. FA^{1st}.*gen* is among the active nodes of u and thus all the active nodes are at most f hops away from the FA^{1st}.*gen*. Observe that all the prev-nodes and next-nodes are non-faulty and the FA^{1st}.*gen* is one of the active nodes of u, and thus the FA^{1st} is received by each active node from the immediate next node or previous node on the path within T-FA-FIRST-LIMIT$(f+1)$ since it was sent.

Next we prove that all the active nodes accept this FA^{1st} and delete the DATA message(s) before their ACK timeouts expire. Since u has deleted a DATA message(s) as a result of generating or accepting an FA^{1st}, it must have accepted the DATA message(s) and it is only waiting for ACKs from the next-nodes, and thus the nodes that precede u on the path are only waiting for ACKs as well. There are two cases to consider, either u and its immediate previous node v are waiting for an ACK of the same node or u is waiting for an ACK of some node z and v is waiting for an ACK of some node y. Obviously in the latter case, since u comes immediately after v on the path it must have accepted the ACKs before sending them to v and thus z must follow y on the path. In addition, since the network is synchronous, z must be the immediate next node of y.

W.l.o.g. let the current path P be $p, q, \ldots, r, u_1, v, u_2, \ldots, x, y, z$ and assume $p, \ldots, v$ are waiting for an ACK of y and $u_2, \ldots, y$ are waiting for an ACK of z, i.e. the ACK generated by y was accepted by $u_2, \ldots, y$ and not by $p, \ldots, v$. In addition, we assume that z is among the next-nodes of $p, \ldots, y$, since if it does not, some nodes will never include it in their active nodes and thus upon accepting z's FA^{1st}, they never delete their DATA message(s) and so there is no issue of synchronizing the deletion of DATA message(s) between immediate neighbors. The same applies for the prev-nodes, so we assume that p is among the prev-nodes of $q, \ldots, z$.

There are two cases to consider, either $u = u_1$, i.e. u and r are waiting for an ACK of y or $u = u_2$, i.e. u is waiting for an ACK of z and v is waiting for an ACK of y. Assume that $u = u_1$. If FA^{1st}.*gen* $\in \{p, \ldots, x\}$, then $p, \ldots, x$ delete the DATA message upon accepting the FA^{1st}, since FA^{1st}.*gen* is among the active nodes of $p, \ldots, x$. Observe that if y would have been the FA^{1st}.*gen*, then since it is not among the active nodes of $p, \ldots, v$, they would not have deleted the DATA message upon accepting its FA^{1st}. Node u has generated or accepted the FA^{1st} and the prev-nodes of u receive this FA^{1st} from u or from another node and accept it within T-FA-FIRST-LIMIT$(f+1)$ since it was sent. If r has not accepted this FA^{1st} by the time it receives it from u, then the FA^{1st} is *safe*, and since r is in the range of the FA^{1st} and u has accepted it, v accepts it as well. It is clear that the other option is that r has already accepted this FA^{1st} from

another node which is among r's 1HNs by the time r receives the FA^{1st} from u. So we can conclude that at the latest, r accepts the FA^{1st} when it is received from u and by Lemma 7.14, node r accepts it before its ACK timeout expires. The same reasoning applies to the other prev-nodes of u as well. All the prev-nodes are non-faulty, so at the latest they accept this FA^{1st} $f+1$ time units after u has generated or accepted it. By Lemma 7.14 and since they are non-faulty, the difference between the values of the ACK timeouts of any node g that is among the prev-nodes of u (except v) and u is $\alpha(2+\sigma+\mu)-\alpha > \sigma+\mu+f+1$, where $1 < \alpha = d_P(g,u) \leq \theta$. So the prev-nodes accept this FA^{1st} before their ACK timeouts expire.

Assume that $u = u_2$. The case where FA^{1st}.*gen* $\in \{p,\ldots,x\}$ was already proven in the previous paragraph, so we assume that FA^{1st}.*gen* is y (node z is not one of the active nodes so if it is the FA^{1st}.*gen*, no node will delete a DATA message as a result of accepting its FA^{1st}). Node y is among the active nodes of $u,\ldots,x$ and not among the active nodes of $p,\ldots,v$, and thus upon accepting the FA^{1st}, $u,\ldots,x$ delete the DATA message(s) and $p,\ldots,v$ do not (if y is non-faulty it deletes all its DATA messages when it generates an FA^{1st}). Node y is not among the active nodes of v, so v has not accepted y's ACK sent by u. When v accepts y's ACK, node y becomes one of its active nodes. Node u sends y's FA^{1st} **after** sending y's ACK. So when v receives y's FA^{1st} from u, v accepts this FA^{1st} for either one of the following reasons. If v has not accepted this FA^{1st} yet, then since u is non-faulty and since this FA^{1st} is accepted within T-FA-FIRST-LIMIT$(f+1)$ since it was sent, v accepts it and thus v deletes the DATA message. If v has already accepted this FA^{1st}, then since all the prev-nodes and next-nodes of u are non-faulty, this FA^{1st} sent by u is received within T-FA-FIRST-LIMIT$(f+1)$ since it was sent. Observe that this FA^{1st} is received before v's ACK timeout expires by Lemma 7.14. Node u is the immediate next node and y is among the active nodes and next-nodes, so v will accept this FA^{1st} and delete the DATA message. $p,\ldots,u_1$ are non-faulty so the analysis done for v also applies for them with the relevant changes, in a similar way done in the previous paragraph.

All the active nodes and prev-nodes of u accept this FA^{1st} before their ACK timeouts expire and while FA^{1st}.*gen* is among their active nodes, so all of them delete the relevant DATA messages and their corresponding ACK timeouts. Since the ACK timeouts are removed, the prev-nodes do not wait for an ACK(s) and thus do not generate an FA^{1st} because of these DATA messages. □

Next we prove that at the presence of only non-faulty nodes the difference in the timings of adding a node to the GL or BL does not cause the verification of a DATA message to fail at a non-faulty node. We denote by *new-node*, a node that is not

included in DATA.*listOfNodes*. Remember that a non-faulty node u does not add a new-node x to DATA.*listOfNodes* if $x \in \text{GL}(u)$ or $x \in \text{BL}(u)$. In addition u does not accept a DATA message if $x \in$ DATA.*listOfNodes* and $x \in \text{BL}(u)$ or if it thinks that x should have been included in DATA.*listOfNodes* but it does not, i.e. $x \notin \text{GL}(u)$ and $x \notin \text{BL}(u)$ but $x \notin$ DATA.*listOfNodes*. Lemma 7.16 proves that if u includes x in the DATA message path by adding it to DATA.*listOfNodes*, then x is not in the BL of the non-faulty nodes when they receive the message. Lemma 7.18 proves that if u does not add x to the path of the DATA message by not adding it to DATA.*listOfNodes* because $x \in \text{GL}(u)$ or $x \in \text{BL}(u)$, then x is either in the GL or in the BL of the non-faulty nodes when they receive the message. We assume in Lemmas 7.16 and 7.18 that the local neighborhoods of the non-faulty nodes are consistent. So if the GLs and BLs are ignored and only the local neighborhoods are taken into consideration, then if u thinks (does not think) that a node x should be added to DATA.*listOfNodes* and sends the message to v, then v thinks the same.

Lemma 7.16. *Let u and v be non-faulty nodes and x be any node such that x has generated an* FA*, $u \in \text{LN}(x, f+1)$ and $v \in \text{LN}(x, f+1)$ and all the non-faulty nodes in $\text{LN}(x, f+1)$ accept this* FA *within* T-FA-LIMIT$(f+1)$ *since it was sent. Assume u sends a* DATA *message along a path P and adds a new-node x to* DATA.*listOfNodes. In addition, assume that $d_P^{unq}(u, v) < d_P^{unq}(u, x) \leq f+1$ and that v receives the message. If all the nodes on P between u and v are non-faulty then $x \notin \text{BL}(v)$ when v receives the message.*

Proof. The proof follows from the timeout mechanism and the handling of the GL and BL. First it should be noted that since x is among the next-nodes of u and $d_P^{unq}(u, v) < d_P^{unq}(u, x)$, x is among the next-nodes of v. In addition, u has added x to DATA.*listOfNodes* and thus $x \notin \text{GL}(u)$ and $x \notin \text{BL}(u)$ at the time u sends the message. If the FA has not been accepted by v by the time the DATA message has, we are done, since x is not in $\text{GL}(v)$ and not in $\text{BL}(v)$ when the DATA message is received.

Next we consider the case where v has accepted the FA by the time it has received the DATA message. Observe that since $d_P^{unq}(u, v) \leq f$ we can conclude that $d_P(u, v) \leq \theta$. We denote the time the FA has been accepted by v as t_v^{FA}. We define T-BL-TIME to be the same as T-FINAL-TIME (Defined in Lemma 7.13) and thus T-BL-TIME $\gg f+\theta$, is the timeout v sets for the nodes to be removed from the GL and added to the BL. Of course t_v^{FA}+T-BL-TIME is the time x is added to the BL of v.

When v accepts the FA at t_v^{FA} it sets a BL timeout. All the nodes in $\text{LN}(x, f+1)$ accept the FA by $t_v^{FA} + f$, by the lemma's assumption and Lemmas 7.11 and 7.12, and

particularly, u accepts this FA. It should be noted that for routing decisions only the $f+1$ neighborhood is used and thus we only consider the $f+1$ local neighborhood of x here. From the above we can conclude that the DATA message was sent by u at the latest at $t_v^{FA} + f$, v's time.

It takes the message to be relayed from u to v at most θ, since $d_{path}(u,v) \leq \theta$ and all the nodes between u and v are non-faulty, and thus the DATA message reaches v at the latest at $t_v^{FA} + f + \theta < t_v^{FA} +$ T-BL-TIME. So we can conclude that at the time v receives the message x is not in BL(v). □

It should be noted that since T-BL-TIME = T-FINAL-TIME and a non-faulty node sets a BL timeout upon accepting an FA, then it is guaranteed that if the FA.*gen* is non-faulty, it sets the final timeout **before** the FA is accepted and thus it deactivates itself **before** the BL timeouts of the non-faulty nodes that have accepted the FA expire, i.e. it stops sending messages **before** being added to these nodes BLs.

The next lemma is an auxiliary lemma and is used by Lemma 7.18.

Lemma 7.17. *Let u be a non-faulty node that has sent a* DATA *message along a path P. If v should have been added by u to* DATA.*listOfNodes and since v is in* GL(u) *or* BL(u) *it has not been added, then all the next-nodes of u, maybe except the last node of these nodes, are at most $f+1$ hops from v.*

Proof. The next-nodes of u consists of $f+1$ distinct nodes. Let these nodes be $x_1, x_2, \ldots, x_{f+1}$, where x_{f+1} is the last node of next-nodes. u has considered v and thus we define $\hat{d}_P(u,v)$ to be the distance between u and v on P if v would have been included in P and similarly $\hat{d}_P^{unq}(u,v)$. We prove the lemma by considering the following cases, $\hat{d}_P(u,v) = 1$ (i.e. $\hat{d}_P^{unq}(u,v) = 1$) and $1 < \hat{d}_P(u,v) \leq \theta + 1$ (i.e. $1 < \hat{d}_P^{unq}(u,v) \leq f+1$). Observe that $d(u,v) \leq \hat{d}_P^{unq}(u,v) \leq \hat{d}_P(u,v)$.

If $\hat{d}_P(u,v) = d(u,v) = 1$ then since $d(x_i,u) \leq f, 1 \leq i \leq f$ we can conclude that $d(x_i,v) \leq d(x_i,u) + d(u,v) \leq f+1$.

Next we consider the case where $1 < \hat{d}_P(u,v) \leq \theta + 1$. Let $x_j, 1 \leq j \leq f$ be the node before v in P if v would have been included in P, i.e. if v was not in GL(u) or BL(u), then x_{j+1} would have been v, and thus $d(x_j,v) = 1$. Since $d(x_j,x_k) \leq f, 1 \leq k \leq f+1$ we can conclude that $d(x_k,v) \leq d(x_k,x_j) + d(x_j,v) \leq f+1$. □

Lemma 7.18. *Let u and v be non-faulty nodes and x be any node such that x has generated an* FA*, $u \in \text{LN}(x, f+1)$ and $v \in \text{LN}(x, f+1)$ and all the non-faulty nodes in $\text{LN}(x, f+1)$ accept this* FA *within* T-FA-LIMIT$(f+1)$ *since it was sent. Assume u accepts a* DATA *message* ***after*** *accepting this* FA *and x is not among* DATA.*listOfNodes and u thinks that x should be added to* DATA.*listOfNodes according to its local neighborhood. In addition, assume that u sends this* DATA *message along a path P and this* DATA *message is received by v and $d_P^{unq}(u, v) \leq f+1$. If all the nodes on P between u and v are non-faulty then $x \in \text{GL}(v)$ or $x \in \text{BL}(v)$ when v receives the message.*

Proof. The proof follows from the handling of FA s and the assumption that the first message sent is received first (FIFO). Observe that $x \in \text{GL}(u)$ or $x \in \text{BL}(u)$ when u sends the DATA message, since it has accepted the FA before sending the DATA message. Node u thinks that x should be added to DATA.*listOfNodes* according to its local neighborhood, but it does not add x to DATA.*listOfNodes* when it sends the message since $x \in \text{GL}(u)$ or $x \in \text{BL}(u)$.

Next we prove that $x \in \text{GL}(v)$ or $x \in \text{BL}(v)$ when v receives the message by induction on the number of hops between u and v in P. Observe that since $d_P^{unq}(u, v) \leq f+1$, we can conclude that $d_P(u, v) \leq \theta + 1$. The base case is $d_P(u, v) = 1$. When u accepts the FA it broadcasts it and thus v must receive it. Remember that the FA is received within T-FA-LIMIT$(f+1)$ since it was sent, by the lemma's assumption, so if v has not accepted this FA by the time it receives it from u, then the FA is *safe* and since v is in the range of the FA and u has accepted it, v accepts it as well. It is clear that the other option is that v has already accepted this FA from another node y which is among v's 1HNs by the time v receives the FA from u. So we can conclude that at the latest, v accepts the FA when v receives it from u. It should be noted that since by FA we mean both the FA itself and its corresponding μTESLA key, it may happen that the FA has been received from u and its corresponding μTESLA key from y or vice versa. Since u broadcasts the FA before sending the DATA message, when v receives the DATA message it has already accepted the FA and thus $x \in \text{GL}(v)$ or $x \in \text{BL}(v)$ when the DATA message is received by v.

Assume that the induction hypothesis is correct for $d_P(u, v) = i$ where $1 \leq i \leq \theta$ and we prove it for $i+1$. Let the path P be $u, y, \ldots, z, v$. Observe that $y, \ldots, z$ are non-faulty by the lemma's assumption. In addition, they are in the range of this FA by Lemma 7.17, so we can conclude that they have accepted the FA within T-FA-LIMIT$(f+1)$ since it was sent. By the induction hypothesis, $x \in \text{GL}(z)$ or $x \in \text{BL}(z)$ when z receives the DATA message, so z broadcasts the FA before sending the DATA message to v. This case is similar to the base case with z playing the role of u in the

base case and thus x must be in GL(v) or BL(v) when it receives the DATA message from z. □

Lemma 7.19 uses Lemmas 7.16 and 7.18 to prove that in the presence of only non-faulty nodes, the difference in the timings of adding a node to the GL or BL does not cause the verification of a DATA message to fail at a non-faulty node. We assume in the following lemma that the local neighborhoods of the non-faulty nodes are consistent.

Lemma 7.19. *Let v, x and y be non-faulty nodes such that y has generated an* FA*, $v \in$ LN$(y, f+1)$, $x \in$ LN$(y, f+1)$ and all the non-faulty nodes in* LN$(y, f+1)$ *accept this* FA *within* T-FA-LIMIT$(f+1)$ *since it was sent. Assume that x sends a* DATA *message to v along a path P such that $d_P^{unq}(x, v) \leq f+1$ and all the $f+1$ distinct nodes preceding v on P are non-faulty. If v receives the message then this* FA *does not cause the verification of the* DATA *message at v to fail.*

Proof. First it should be noted that all the $f+1$ distinct nodes preceding v on P are non-faulty and thus all the prev-nodes of v are non-faulty. The prev-nodes are non-faulty so they generate valid messages and thus the only reason because of which v fails to validate the DATA message is that its chosen-nodes differ from DATA.*listOfNodes.futureNodes*. So we prove that the FA generated by y will not cause the verification of DATA.*listOfNodes.futureNodes* to fail. The local neighborhoods of the prev-nodes and v are consistent by the lemma's assumption. If according to the local neighborhoods y should not follow v on P then both x and v do not consider y and we are done, so we prove that the lemma holds when y **should come** after v according to the local neighborhoods (regardless of the GLs and BLs of the nodes).

We prove the lemma by analyzing the DATA message v receives. There are two options: either $y \in$ DATA.*listOfNodes.futureNodes*, i.e. when v receives the message y is among the future nodes of DATA.*listOfNodes*, or $y \notin$ DATA.*listOfNodes.futureNodes*, i.e. when v receives the message y is not among the future nodes of DATA.*listOfNodes*. Both v and x are in the range of the FA and, by the lemma's assumption, x and v accept y's FA by T-FA-LIMIT$(f+1)$ time units since y has sent it.

If $y \in$ DATA.*listOfNodes.futureNodes*, then it is either because x has added y to DATA.*listOfNodes* or because another node u that precedes v has added y to DATA.*listOfNodes*. Node u is among the prev-nodes of v and thus it is non-faulty. Since u has added y to DATA.*listOfNodes* it must be in LN$(y, f+1)$ and thus it has accepted y's FA within T-FA-LIMIT$(f+1)$ time units since y has sent it. By

Lemma 7.16, where x or u play the role of u and y plays the role of x in the lemma, we can conclude that $y \notin \text{BL}(v)$ when v receives the DATA message and thus this FA does not cause the verification of the DATA message at v to fail (Obviously, if u precedes x, then x accepts the DATA message by Lemma 7.16 as well).

If $y \notin$ DATA.*listOfNodes.futureNodes*, then either x thinks that y should be in DATA.*listOfNodes* according to its local neighborhood but since $y \in \text{GL}(x)$ or $y \in \text{BL}(x)$ it does not add y to DATA.*listOfNodes* or another node u that precedes v has not added y to DATA.*listOfNodes* because $y \in \text{GL}(u)$ or $y \in \text{BL}(u)$. As before, u must be non-faulty and in $\text{LN}(y, f+1)$ and thus it has accepted y's FA by T-FA-LIMIT$(f+1)$ time units since y has sent it. By Lemma 7.18, where x or u play the role of u and y plays the role of x in the lemma, we can conclude that $y \in \text{GL}(v)$ or $y \in \text{BL}(v)$ and thus this FA does not cause the verification of the DATA message at v to fail (Obviously, if u precedes x, then x accepts the DATA message by Lemma 7.18 as well). □

Next we prove that if a non-faulty node receives a DATA message but fails to verify it and all its prev-nodes are non-faulty, then there must be a faulty node near by. In the next lemma we do not assume that the local neighborhoods of the non-faulty nodes are consistent.

Lemma 7.20. *Let v be a non-faulty node that has received but not accepted a* DATA *message that was sent along some path P. If all the ($f+1$ distinct) nodes preceding v on P are non-faulty then node u, the node that immediately precedes v on P, must generate an* FA^{1st} *and there is a faulty node w s.t. $d(u,w) \leq f+1$.*

Proof. Node v has received a DATA message. Let DATA.*listOfNodes.pastNodes* be $x_i, x_{i+1}, \ldots, x_{i+l-1}, x_{i+l}$, $f \leq l \leq \theta$. *pastNodes* are the prev-nodes of v and they consist of $f+1$ distinct nodes. x_{i+l} is the node that immediately precedes v so x_{i+l} is u. Let DATA.*listOfNodes.futureNodes* be $z_i, z_{i+1}, \ldots, z_{i+m}$. *futureNodes* may be comprised of f, $f+1$ or even less than f distinct nodes. If *futureNodes* are comprised of f distinct nodes then $f-1 \leq m \leq \theta-1$, if *futureNodes* are comprised of $f+1$ distinct nodes then $f \leq m \leq \theta$ and if *futureNodes* are comprised of less than f distinct nodes then $f \leq m < \theta-1$. Observe that the proof for all cases is the same.

The prev-nodes are non-faulty and they have verified the message and agreed that v should follow u ($= x_{i+l}$) on the path, since if they have not agreed the verification of the DATA message would have failed before reaching v. Thus, they have generated a valid DATA message for v.

The verification at v has failed and thus its chosen-nodes $z_i^v, z_{i+1}^v, \ldots, z_{i+m}^v$ differ from the *futureNodes* $z_i, z_{i+1}, \ldots, z_{i+m}$. Let $y \in \{z_i, z_{i+1}, \ldots, z_{i+m}\}$ and $y^v \in$

$\{z_i^v, z_{i+1}^v, \ldots, z_{i+m}^v\}$ be the first nodes for which $y \neq y^v$. Let z_r be the immediate node that precedes y (z_r may be v). Since y comes after v on the path it is not included in the next-nodes of some of v's prev-nodes and thus only some nodes among $x_i, x_{i+1}, \ldots, x_{i+l}$ have agreed that y should be after z_r. Let $x_k, x_{k+1}, \ldots, x_{i+l}$ be these nodes. It is clear that x_k has added y to DATA.*listOfNodes*.

So the nodes are ordered in the following way:

$$x_k, x_{k+1}, \ldots, x_{i+l}, v, z_i, \ldots, z_r \nearrow \begin{matrix} y \\ \\ y^v \end{matrix} \searrow$$

From the above we can conclude that: $d_P(z_r, y) = d_P(z_r, y^v) = 1$, $d_P^{unq}(x_k, y) = f+1$, since it is the last node of the next-nodes of x_k. $d_P^{unq}(x_k, z_r) = f$ since $d_P^{unq}(x_k, z_r) < f+1$ (otherwise x_k would not have added y) and $d_P^{unq}(x_k, z_r) \geq f$ (z_r is a 1HN of y) and thus $d_P^{unq}(x_k, y^v) \leq d_P^{unq}(x_k, z_r) + d_P(z_r, y^v) = f+1$. In the same manner $d_P^{unq}(x_{i+l}, z_r) \leq f$, $d_P^{unq}(x_{i+l}, y) \leq f+1$, $d_P^{unq}(x_{i+l}, y^v) \leq f+1$, $d_P^{unq}(v, z_r) \leq f$, $d_P^{unq}(v, y) \leq f+1$ and $d_P^{unq}(v, y^v) \leq f+1$.

Since the prev-nodes of v and v are non-faulty and choose the nodes according to the routing protocol they should have chosen the same nodes and thus there are the following three options.

1. The local neighborhoods of $x_k, \ldots, x_{i+l}$ and v are consistent and x_k has added y to DATA.*listOfNodes* and for $x_{k+1}, \ldots, x_{i+l}$ node y was not in their BLs when they have received the message but y was in BL(v) when v has received the message so it has chosen y^v instead.

2. The local neighborhoods of $x_k, \ldots, x_{i+l}$ and v are consistent and y^v should have been added to DATA.*listOfNodes* by x_k but since $y^v \in$ GL(x_k) or $y^v \in$ BL(x_k), x_k has added y instead of it and for $x_{k+1}, \ldots, x_{i+l}$ node y^v was in their GLs or BLs when they have received the message but y^v was not in GL(v) or BL(v) when v has received the message, so v has included y^v in its chosen-nodes.

3. The local neighborhoods of $x_k, \ldots, x_{i+l}$ and v are not consistent and $x_k, \ldots, x_{i+l}$ think that y should be after x and v think that y^v should be after x.

We begin with the first option. First we analyze the nodes that consist of the path P, i.e. $x_k, x_{k+1}, \ldots, x_{i+l}, v, z_i, \ldots, z_r, y$. $d_P^{unq}(x_k, y) = f+1$ so there is a path Q of length $f+1$ between x_k and y. The path between x_k and v is part of Q and

it is non-faulty. $y \in \text{BL}(v)$ so v has accepted an FA for which y is the FA.*gen*. By Lemma 7.19, where x_k, v and y are playing the role of x, v and y in the lemma, we can conclude that if x_k would have accepted this FA within T-FA-LIMIT$(f+1)$ since it was sent, then this FA would not have caused the verification at v to fail. The verification at v has failed so there must be a faulty node w on the path between x_k and y that has prevented from x_k to accept this FA within T-FA-LIMIT$(f+1)$ since it was sent. $x_k, \ldots, v$ are non-faulty so $w \in \{z_i, \ldots, z_r, y\}$, since otherwise, there is a path of length $f+1$ between x_k and y which is comprised entirely of non-faulty nodes, so x_k would have accepted the FA within T-FA-LIMIT$(f+1)$ since it was sent. $z_i, \ldots, z_r, y$ are among the next-nodes of x_{i+l}, so we can conclude that $d(x_{i+l}, w) \leq f+1$.

The proof of the second option is similar to the previous option. Node x_k has not added y^v to the DATA message since it was in its GL or BL, so it has accepted an FA for which y^v is the FA.*gen*. By Lemma 7.19, where x_k, v and y^v are playing the role of x, v and y in the lemma, we can conclude that if v would have accepted this FA within T-FA-LIMIT$(f+1)$ since it was sent, then this FA would not have caused the verification at v to fail. The verification at v has failed so there must be a faulty node w on the path between v and y^v that has prevented from v to accept this FA within T-FA-LIMIT$(f+1)$ since it was sent. Similarly to the previous option, $w \in \{z_i, \ldots, z_r, y^v\}$ and we can conclude that $d(x_{i+l}, w) \leq f+1$.

The last option implies that the local neighborhoods of x_{i+l} and v are not consistent. Remember that $\forall p, q$ and $path, d(p,q) \leq d^{unq}_{path}(p,q)$ so $d(v,y) \leq f+1$, $d(v,y^v) \leq f+1$, $d(x_{i+l}, y) \leq f+1$ and $d(x_{i+l}, y^v) \leq f+1$. It is clear that either v or x_{i+l} does not include a node (i.e. y or y^v, respectively) that it should in its local neighborhood or does not know the correct information of that node. If it is x_{i+l}, then by the assumption on the neighborhood discovery algorithm (See Chapter 4), we can conclude that $d(x_{i+l}, w) \leq d(x_{i+l}, y) = f+1$. If it is v, then again by the same assumption, the faulty node is on the path between v and y and we prove that w is at most $f+1$ hops from x_{i+l}. There are two options, either y is the faulty node or another node w for which $d(v,w) \leq f$ is faulty. If it is y then we are done since $d(x_{i+l}, y) \leq f+1$, otherwise $d(x_{i+l}, w) = d(x_{i+l}, v) + d(v, w) \leq f+1$.

So we have proved that $d(x_{i+l}, w) \leq f+1$ and remember that $x_{i+l} = u$. Since the verification at v has failed it drops the message and the ACK timeout at u expires and thus it generates an FA^{1st}. □

Finally we prove the main lemma.

Lemma 7.21. *If u has generated an* FA^{1st} *then there exists w s.t. w is faulty and $d(u, w) \leq f + 1$.*

Proof. If u is faulty we are done; so consider the case that u is non-faulty. Observe that a non-faulty node u generates an FA^{1st} only upon the expiration of its ACK timeout (See Figure 6.1, line D1). It should be noted that u has accepted a DATA message and its ACK timeout has expired and thus it has generated an FA^{1st}. Let the path P of this DATA message be $\ldots, z, \ldots, u, v, \ldots$. Node v is the node that comes immediately after u on the path and z is the node that has added v to DATA.*listOfNodes*. It is obvious that $d(z, v) \leq f + 1$, i.e. $z \in \text{LN}(v, f + 1)$. If v is faulty we are done; so consider the case that it is non-faulty. We should consider the following cases depending whether v has accepted the DATA message or not.

We consider the case where v has accepted the DATA message sent by u to v. Node u has not accepted an ACK or an FA^{1st} since its ACK timeout has expired. Node v is non-faulty and thus it verifies, updates and generates valid messages for u. If v has deactivated itself before sending an ACK or FA^{1st} to u then either, by Lemma 7.13, z is faulty, since it has added v to the path though $v \in \text{GL}(z)$ or $v \in \text{BL}(z)$, or there must be a faulty node that has prevented from z to accept v's FA within T-FA-LIMIT$(f + 1)$ since it was sent. So the faulty node must be on P between v and z and we can conclude that $d(w, u) \leq f$.

We have covered the case that v deactivates itself before sending all the expected messages to u, so we assume now that v has sent all the expected messages to u. The ACK timeout at v has not expired, since if it had, v would have generated a valid FA^{1st}. By Lemma 7.14 and since u and v are non-faulty, the FA^{1st} generated by v would have been accepted by u before the expiration of u's ACK timeout and thus u would have deleted the DATA message and would not have generated an FA^{1st}. So v must have accepted an ACK or an FA^{1st} before its ACK timeout has expired and this ACK or FA^{1st} must have caused v to remove its ACK timeout. In addition, this ACK or FA^{1st} was sent to u and by Lemma 7.14, u receives them before its ACK timeout expires. Node u has received the message from v but not accepted it, since otherwise, u would have removed its ACK timeout and thus it would not have generated an FA^{1st}. So u has generated an FA^{1st} because of **not** accepting **this** message before the expiration of its ACK timeout. Next we prove that since u has not accepted this ACK or this FA^{1st} there must be a faulty node near by.

In case of an ACK, the HMACs are generated in an onion like style by ACK.*gen* so

the HMAC u has received has been invalid. By Lemma 7.10 we can conclude that the faulty node must be among the next-nodes of u and thus $d(u, w) \leq f + 1$. In case of an FA^{1st}, this FA^{1st} has caused v to delete the DATA message(s) and its ACK timeouts. If u had accepted this FA^{1st} then it would have removed its DATA message and its ACK timeouts and thus $\text{FA}^{1st}.gen$ must be one of the active nodes of u. The ACK timeout at u has still expired and thus, by Lemma 7.15 and since $\text{FA}^{1st}.gen$ is one of the active nodes of u, there is a faulty node w among the prev-nodes or next-nodes of u and $d(w, u) \leq f + 1$.

Next we consider the case where v has not accepted the DATA message sent by u to v. In order to verify a DATA message, v must verify the HMACs generated by DATA.*listOfNodes.pastNodes*, i.e. v's prev-nodes, and it must agree with DATA.*listOfNodes.futureNodes*. Again there are two cases depending whether there is a faulty node among v's prev-nodes. If there is a faulty node w among v's prev-nodes that has caused the verification at v to fail we are done, since $d(u, w) \leq f + 1$. If all the prev-nodes are non-faulty and still v has not accepted the DATA message, then, by Lemma 7.20, the only explanation is that v and some of its prev-nodes did not agree on DATA.*listOfNodes.futureNodes* and we can conclude that $d(u, w) \leq f + 1$. □

Chapter 8

Neighborhood Discovery

In this chapter we present an algorithm for neighborhood discovery. We prove that by the end of the neighborhood discovery each node x knows its σ-hop neighborhood unless there are faulty nodes. For each neighbor y in its σ-hop neighborhood x will know all the information needed for the routing and for verifying y's authenticated messages. The neighborhood information is gathered by the HELLO message mechanism.

The algorithm is comprised of rounds and it is based on synchronizer alpha [3] with some modifications to synchronous systems. At round i, each node x generates keys with its i-hop neighbors and gathers the information of its i-hop neighbors. At the beginning of each round i, a timeout is set and upon its expiration, round i ends and round $i+1$ begins. Of course a faulty node can prevent messages from being received or accepted and the timeout mechanism is used to overcome such faults.

8.1 Messages and Timeouts

Two types of messages are used in the algorithm, HELLO and FINISH messages.

8.1.1 HELLO message

HELLO messages are used to inform the local neighborhood of the generator of the HELLO message what is the size of the neighborhood known to it. The format of the HELLO message is depicted in Table 8.1.

Observe that HELLO messages do not include HMACs and are not authenticated.

HELLO message Format

gen	*round*

Field	Description
ID *gen*	The generator of the HELLO message
NUMBER *round*	The current round number of the HELLO.*gen*, i.e. HELLO.*gen* knows its *round*-1 local neighborhood

Table 8.1: HELLO message format

We denote by HELLO(i), an HELLO message with round i.

8.1.2 FINISH message

FINISH messages are only sent at round 1.5 of the algorithm and are used to inform the 1HNs of the generator of the FINISH message who are its 1HNs. The format of the FINISH message is depicted in Table 8.2.

FINISH message Format

gen	*1*HN*s*	$\mu TESLAInterval$	$\mu TESLAHmac$

Field	Description
ID *gen*	The generator of the FINISH message
SET *1*HN*s*	The 1HNs known to the FINISH.*gen*
NUMBER $\mu TESLAInterval$	The interval at which the key for this FINISH message will be disclosed
NUMBER $\mu TESLAHmac$	A μTESLA HMAC generated by FINISH.*gen*

Table 8.2: FINISH message format

The FINISH message can only be verified by the nodes that can verify the μTESLA HMAC, and thus it can only be verified by the 1HNs of the FINISH.*gen* that have

completed the handshake with FINISH.*gen*. We denote by FINISH(*1*HN*s*), a FINISH message with these *1*HN*s*.

8.1.3 Timeouts

Two additional timeouts are defined, the HELLO timeout and the FINISH timeout. Their types are T-HELLO and T-FINISH and their values are T-HELLO-TIME and T-FINISH-TIME, respectively. These timeouts ensure, as explained next, that if node x is at round i, it only receives messages of round i or $i+1$.

8.2 The Neighborhood Discovery Algorithm

The algorithm is depicted in Figure 8.1. The trigger for starting the algorithm is internal or when the first HELLO(1) message is received (lines B1, C1-C2). Each node x is required to know its σ hop neighborhood, so if $y \in \text{LN}(x, \sigma)$, there may be a difference of at most σ time units between the time x and y start the neighborhood discovery (of course if there are faults it may be more). Timeouts are used throughout the algorithm and this difference is taken into consideration, i.e. if the timeout for round i should have been t, it would be $t + \sigma$. This calculation of the timeouts guarantee that the ith rounds of nodes that are at most i-hops away have an overlap of t time units. As a result, a message that should have been received at round i, may be received at round $i-1$ and thus it is stored till the beginning of round i (lines D1, E1, H1, M1). At the beginning of round i, each node sends the stored messages to itself as if it has just received them (lines G2-G3, K2).

The timeout of round $i, 1 \leq i \leq \sigma, i \neq 1.5$ takes into consideration the maximal time it takes to perform the handshakes with the neighbors at distance i and the timeout of round 1.5 takes into consideration the time it takes to verify a FINISH message (lines B1, C2, G1, K1). Once the timeout for round i has expired, any messages that are relevant for that round are ignored even if an handshake was started but not completed. As a result, it may happen that x thinks that the handshake with y was completed and y thinks that handshake with x was not completed.

An HELLO or a FINISH message is sent at the beginning of each round (lines B1, C2, G1, K1). FINISH messages are authenticated by μTESLA. The key used for these messages is the key that its μTESLA interval starts at least 1 time unit after sending the message. A FINISH message (line I1) is accepted in the same manner an

```
Init:
(A1)   neighbors = NULL;   i = 1;
Round 1
(B1)   broadcast HELLO(1);   setTimeout(T-HELLO,1);

Upon receiving HELLO(1) from node y
(C1)   if have not sent HELLO message
(C2)     broadcast HELLO(1);   setTimeout(T-HELLO, 1);
(C3)   if handshake with y succeeds
(C4)     neighbors[1] = neighbors[1] ∪ y;
(C5)     addToLn(y, NULL);   store(y's info);

Upon receiving FINISH(1HNs) from node y
(D1)   store(FINISH(1HNs));

Upon receiving μTESLA key for FINISH(1HNs) from node y
(E1)   store(μTESLA key);

Upon expiration of timeout with timeout.type == T-HELLO
(F1)   start round 1.5;
Round 1.5
(G1)   broadcast FINISH(neighbors[1]);   setTimeout(T-FINISH);
(G2)   send all stored FINISH(1HNs) to itself;
(G3)   send all stored μTESLA keys for FINISH(1HNs) to itself;

Upon receiving HELLO(2) from node y
(H1)   store(HELLO(2));

Upon accepting FINISH(1HNs) from node y
(I1)   neighbors[2] = neighbors[2] ∪ calcNeighbors(2, y, 1HNs);
(I2)   addToLn(y, 1HNs);

Upon expiration of timeout with timeout.type == T-FINISH
(J1)   i++;   begin round i;
Round i
(K1)   broadcast HELLO(i);   setTimeout(T-HELLO, i);
(K2)   send all stored HELLO(i) to itself;

Upon receiving HELLO(i) from node y
(L1)   if y ∈ neighbors[i]
(L2)     if handshake with y succeeds
(L3)       neighbors[i+1] = neighbors[i+1] ∪ calcNeighbors(i+1, y, y.1HNs);
(L4)       addToLn(y, y.1HNs);   store(y's info);

Upon receiving HELLO(i+1) from node y
(M1)   store(HELLO(i+1));

Upon expiration of timeout and timeout.type == T-HELLO
(N1)   if i < σ
(N2)     i++;   begin round i;
```

Figure 8.1: Neighborhood discovery algorithm

FA is accepted as described in Section 5.2.3. These messages can be verified only if they were generated by a node that has completed the handshake with the receiving node. Observe that any message that is received is also broadcasted, except an unauthenticated μTESLA key message.

Upon receiving an HELLO message by node x at round i that was generated by a new node y, a symmetric key is generated and an handshake is performed (lines C3, L2) if y is known to be i-hop away (line L1). If this is the first round, then no neighbors are known at all and thus x assumes that y is a 1HN. By the end of the handshake, a symmetric key is generated between x and y and the 1HNs of y, the μTESLA basic key and schedule of y, and any information needed by the routing algorithm, such as the location of y, are known to x and the information of x is known to y (lines C5, L4). Observe that the 1HNs of the 1HNs are only known upon accepting a FINISH message (line I2). Once the 1HNs of y are know at round i, it is possible to calculate the $i+1$ hop neighbors among these neighbors based on y, y's 1HNs and the known i-hop neighborhood (lines I1, L3).

Next we discuss how the symmetric key is generated and how the handshake is performed. A symmetric key may be generated by the use of multi-space pairwise key distribution techniques [7, 14] (See Chaper 2), though these techniques do not guarantee the generation of a symmetric key. If these techniques are used, HELLO messages should include the spaces' IDs of each node. The key can also be generated by the use of a digital signature scheme. Authenticated public keys can be acquired by using only local topology knowledge, as described in Chapter 2. In such a case, a handshake should be performed in order to generate the symmetric key. Observe that using public keys is more expensive in terms of memory, battery and computational power than the multi-space pairwise key distribution techniques.

After the generation of the symmetric key K_{xy} between x and y, a cryptographic handshake is performed between x and y. The messages used in the handshake are sent as DATA messages with a fixed path of length at most σ, where only the source generates HMACs and neither ACKs nor FAs are generated. It should be noted that the source has a symmetric key with each node on the path. The payload of the DATA message y sends to x contains all the information x should know about y.

8.3 Correctness

In this section we prove that if by the end of the neighborhood discovery algorithm u does not know its correct σ-hop neighborhood, i.e. there is a node v for which $d(u,v) \leq \sigma$ but v is not in the σ-hop neighborhood of u, or the correct information (μTESLA key, μTESLA schedule, etc.) of v are not known to u, then there is a faulty node w on the shortest path between u and v.

Lemma 8.1. *Let u and v be non-faulty nodes and $d(u, v) = i \leq \sigma$. If there is a non-faulty path P of length i between u and v, then by the end of round i node u knows the correct information of v.*

Proof. The proof is by induction on the round number i. Base case: $i = 1$. u or v should have sent an HELLO(1) message, since they internally sent it or because of receiving such a message. u and v have not started round 1 at the same time but the timeouts take it into consideration, so non-faulty nodes that are at distance i are able to finish the handshakes by the end of round i. Since they are non-faulty, the handshake must have been completed by the end of round 1 and a FINISH(1HNs) should have been sent and accepted by the end of round 1.5, so by the end of round 1.5 u knows the correct information of v.

We assume that all the nodes in P are non-faulty and that u and v have not started round 1 at the same time so it may happen that a FINISH message and its μTESLA key are received at round 1. In such a case they are stored and used at the beginning of round 1.5. The same applies for HELLO(2) messages that may be received at round 1.5. Though, an HELLO(2) message sent by non-faulty node cannot be received at round 1 because of the timeouts mechanism.

Assume that the induction hypothesis is correct for round $i, 1 \leq i \leq \sigma - 1$ and we will prove it for round $i + 1$. Let the path P be $u, x, \ldots, y, v$, where $d_P(u, y) = d_P(x, v) = i$. By the induction hypothesis, node u has completed the handshake with y and v has completed the handshake with x by the end of round i. Node y sends its 1HNs during the handshake of round i or by a FINISH message and includes v in its 1HNs and x includes u and thus by the end of round i, $u \in neighbors[i + 1]$ of v and $v \in neighbors[i + 1]$ of u. Since there are no faulty nodes on the path P, u and v complete their handshake by the end of round $i + 1$ and u knows the correct information of v. Observe that an HELLO message of round $i + 1$ may be received at round i and in such a case it is stored and handled at round $i + 1$.

□

The neighborhood discovery algorithm is ran for σ rounds and thus, by Lemma 8.1, we can conclude that if by the end of the neighborhood discovery algorithm a non-faulty node u does not know the correct information of a node v in its σ-hop neighborhood, then there is a faulty node w on the shortest path between u and v.

Chapter 9

Conclusions and Future Work

We have presented the Byzantizer algorithm that transforms ad hoc routing algorithms and makes them resilient to Byzantine faults while maintaining their ad hoc and locality properties. We have shown how the faulty nodes are isolated by deactivating the nodes around them and thus it is guaranteed that once isolated no more faults will occur. Our construction provides the first ad hoc routing scheme with competitive performance resilient to Byzantine failures.

We did not mention enhancement for fully asynchronous models, power management and efficiently handling dynamic systems in which nodes are constantly moving, joining and leaving the system. Power management may be a problem in ad hoc networks and is a major concern in sensor networks. The nodes may have a limited battery power and thus they may stop functioning due to battery depletion and not because of a faulty node that has caused them to generate an FA and deactivate themselves. If a non-faulty node does not send a message it should have because its battery was depleted, then another non-faulty node will generate an FA^{1st} though no faulty node is near by. A simple way to overcome this problem is the following. Each non-faulty node u can generate an FA^{2nd} when its battery power drops below a predefined threshold. By doing this, u informs its $\text{LN}(u, f+1)$ that they should not consider u in the routing anymore and thus it ensures that no FA will be generated because its battery was depleted.

We also did not discuss attacks against the physical and MAC layers, the Sybil attack and the replay attack. Both the physical and MAC layers are vulnerable to DoS attacks. Our proposed scheme can be used to defend against these attacks as well. For example, if a node generates DATA messages at a rate that exceeds a predefined threshold, a non-faulty node can generate an FA^{1st}. In addition, jamming may be

countered by the use of spread spectrum [24]. The information needed by spread spectrum may be acquired during the handshake performed in the neighborhood discovery. The Sybil attack may be countered by the same techniques as in [17] or by assuming that each node has a digital signature which was provided by the manufacturer of the nodes and thus each node can prove that its ID is authentic. Once a non-faulty node is convinced that its 1HN is performing the Sybil attack it can use our proposed scheme and generate an FA^{1st} in order to isolate the faulty node. The replay attack may be countered by using sequence numbers.

Our approach of isolating the faulty nodes requires the deactivation of non-faulty nodes. One direction of research is how to re-incorporate nodes into the network that have been previously declared faulty and removed. Observe that re-incorporating a node into the network does not necessarily justify itself since faulty nodes may be re-incorporated as well and cause additional faults that will result in more FA flooding phases. The tradeoff between the degradation of the routing due to the isolation of the faulty nodes and the cost of re-incorporating nodes into the network should be investigated.

An open question is to improve the cost complexity of local routing schemes resilient to Byzantine faults. Another interesting question is to study lower bounds in this model.

Bibliography

[1] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Highly secure and efficient routing, in Proc. IEEE Infocom 2004.

[2] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Amendment to: Highly secure and efficient routing, 2004.

[3] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

[4] B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens. Mitigating byzantine attacks in ad hoc wireless networks. Technical report, March 2004.

[5] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens. An on-demand secure routing protocol resilient to byzantine failures. In *Proceedings of the ACM workshop on Wireless security (WiSE '02)*, September 2002.

[6] K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson. Detecting disruptive routers: A distributed network monitoring approach. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy, IEEE Press, Los Alamitos, May 3-6, 1998*, pages 115–124, 1998.

[7] W. Du, J. Deng, Y. S. Han, and P. K. Varshney. A pairwise key pre-distribution scheme for wireless sensor networks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 42–51, New York, NY, USA, 2003. ACM Press.

[8] Y.-C. Hu, D. B. Johnson, and A. Perrig. SEAD: Secure efficient distance vector routing for mobile wireless ad hoc networks. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 3–13, 2002.

[9] Y.-C. Hu, A. Perrig, and D. B. Johnson. Ariadne: a secure on-demand routing protocol for ad hoc networks. In *MOBICOM*, pages 12–23, 2002.

[10] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[11] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. In *First IEEE International Workshop on Sensor Network Protocols and Applications*, pages 113–127, May 2003.

[12] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 243–254, New York, NY, USA, 2000. ACM Press.

[13] F. Kuhn, R. Wattenhofer, and A. Zollinger. Worst-case optimal and average-case efficient geometric ad-hoc routing. In *MobiHoc '03: Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*, pages 267–278, New York, NY, USA, 2003. ACM Press.

[14] D. Liu and P. Ning. Establishing pairwise keys in distributed sensor networks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 52–61, New York, NY, USA, 2003. ACM Press.

[15] H. Luo, P. Zerfos, J. Kong, S. Lu, and L. Zhang. Self-securing ad hoc wireless networks. In *The 7th IEEE Symposium on Computers and Communications*, 2002.

[16] S. Marti, T. J. Giuli, K. Lai, and M. Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Mobile Computing and Networking (MOBICOM)*, pages 255–265, 2000.

[17] J. Newsome, E. Shi, D. Song, and A. Perrig. The sybil attack in sensor networks: analysis & defenses. In *Proceedings of 3rd IEEE/ACM Information Processing in Sensor Networks (IPSN'04)*, pages 259–268, 2004.

[18] P. Papadimitratos and Z. J. Haas. Secure routing for mobile ad hoc networks. In *Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS)*, pages 193–204, 2002.

[19] P. Papadimitratos and Z. J. Haas. Secure message transmission in mobile ad hoc networks. *Ad Hoc Networks*, 1(1):193–209, 2003.

[20] C. Perkins and E. M. Royer. Ad-hoc On-demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 1999.

[21] R. Perlman. *Network Layer Protocols With Byzantine Robustness*. PhD thesis, Massachusetts Institute of Technology, 1988.

[22] A. Perrig, R. Canetti, D. X. Song, and J. D. Tygar. Efficient and secure source authentication for multicast. In *Network and Distributed System Security Symposium, NDSS '01*, 2001.

[23] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. SPINS: security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, 2002.

[24] R. L. Pickholtz, D. L. Schilling, and L. B. Milstein. Theory of spread-spectrum communications - A tutorial. *IEEE Transactions on Communications*, 30(5):855–884, May 1982.

[25] B. R. Smith, S. Murthy, and J. J. Garcia-Luna-Aceves. Securing distance vector routing protocols. In *Proceedings of the Symposium on Network and Distributed System Security*, 1997.

[26] H. Yang, H. Luo, F. Ye, S. Lu, and L. Zhang. Security in mobile ad hoc networks: Challenges and solutions. *IEEE Wireless Communications*, 11:2–11, February 2004.

[27] M. G. Zapata and N. Asokan. Securing Ad hoc Routing Protocols. In *Proceedings of the 2002 ACM Workshop on Wireless Security (WiSe 2002)*, pages 1–10, September 2002.

[28] Lidong Zhou and Zygmunt J. Haas. Securing ad hoc networks. *IEEE Network*, 13(6):24–30, 1999.