Theoretical Computer Science 410 (2009) 467–480

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Hop chains: Secure routing and the establishment of distinct identities

Rida A. Bazzi [a,*], Young-ri Choi [b,1], Mohamed G. Gouda [b]

[a] School of Computing and Informatics, Arizona State University, Tempe, AZ 85287, United States

[b] Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, United States

ARTICLE INFO

Keywords:
Computer networks
Sybil attack
Routing
Security
Byzantine

ABSTRACT

We present a secure routing protocol that is immune to Sybil attacks and that can tolerate collusion of Byzantine routers. It can tolerate either initial collusion of Byzantine routers or runtime collusion of non-adjacent Byzantine routers, both in the absence of runtime collusion between adjacent routers. For these settings, the calculated distance from a destination to a node is not smaller than the actual shortest distance from the destination to the node. The protocol can also simultaneously tolerate initial collusion of Byzantine routers and runtime collusion of adjacent Byzantine routers but in the absence of runtime collusion between non-adjacent routers. For this setting, it guarantees a bound on the difference between the calculated distance and the actual shortest distance. The bound depends on the number of Byzantine routers on a path. The protocol makes very weak timing assumptions and requires synchronization only between neighbors or second neighbors. We propose to use this protocol for secure localization of routers using hop-count distances, which can be then used as a proof of identity of nodes.

1. Introduction

In peer-to-peer networks, physical entities (or hosts) communicate with each other using pseudonyms or logical identities. Logical identities are assumed by software processes that execute on the hosts to provide or request services from other hosts. To the outside world, a host is identified with the software process that provides the logical functionality. For example, a host that provides storage space would appear to other hosts as a software process that handles and replies to requests for reading and writing data. In the absence of direct physical knowledge of a remote host, or certification by a central authority, it is possible for one physical entity to appear in the system under different names or *counterfeit* identities. Douceur [1] was the first to thoroughly consider this problem. In his terminology, an entity launches a *Sybil attack* when it appears under different identities. He claims that in the absence of a central certifying authority, the Sybil attack cannot be solved in practice.

Bazzi and Konjevod [2] proposed the use of geometric techniques to determine how many identities amongst a group of identities belong to distinct entities and thereby reducing the harm due to Sybil attack. Their work is based on existing evidence that roundtrip delays in the Internet exhibit geometric properties [3]. They provided solutions under a variety of adversarial assumptions including colluding entities and beacons (used for localizing entities) without assuming a central certifying authority with direct knowledge of entities in the system. While the work of Bazzi and Konjevod is a significant step forward, it makes some restricting assumptions. For instance, the results about the geometry of roundtrip delays apply to systems in which routers are honest and they are not necessarily applicable in systems in which routers are corrupt. Also,

* Corresponding author.
E-mail addresses: bazzi@asu.edu (R.A. Bazzi), yrchoi@cs.utexas.edu, ychoi@vmware.com (Y.-r. Choi), gouda@cs.utexas.edu (M.G. Gouda).
[1] Present address: VMware, Inc., Palo Alto, CA, United States.

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.10.004

their solutions require accurate measurements of roundtrip delays and clock synchronizations between routers that can be far apart physically. This is not always possible given the variability of network load and delays.[2]

In this work, our goal is to present a solution to the Sybil attack problem under the weakest possible system assumptions and in the absence of a central authority with direct knowledge of entities in the system. In particular, we are interested in a solution that does not require strong timing assumptions and that does not rely on the geometry of roundtrip delays as such a geometry, if it exists, can be disrupted by dishonest routers. Our solution can tolerate stronger adversarial settings while making weaker system assumptions. In particular, we assume that routers can be dishonest and we allow for more collusion between the routers. Also, we require very rough synchronization between non-adjacent routers (in a sense that we will define precisely later). For some settings, we require synchronization, but only between adjacent (or almost adjacent) routers, which means that the synchronization we require is local. Relaxing the synchrony and synchronization assumptions is a major improvement over the results of [2] and we believe that it is an improvement that brings the results closer to a practical setting.

At the heart of our approach is a secure distance vector routing protocol that can tolerate Byzantine routers, Sybil attack by routers and collusion between routers. The protocol does not require the use of shared private keys between nodes; it only requires, as far as keys are concerned, that the destination's public key is known a priori by nodes in the network. Under the assumption of no collusion between corrupt nodes, a first version of the protocol guarantees that no node can have a calculated hop-count distance, or simply distance, to destination that is shorter than its real or actual shortest hop-count distance to destination. In the presence of initial collusion, in which corrupt nodes can share information initially but not afterward, the second version of protocol guarantees that no honest router can have a calculated hop-count distance to destination that is shorter than its real or actual shortest hop-count distance to destination. In the presence of initial collusion between any two nodes and runtime collusion between adjacent corrupt nodes, in which corrupt nodes can communicate with each other at any time, the second protocol guarantees the following: for any path P from destination to an honest node u, the calculated hop-count distance of node u is not less than the number of honest nodes on P plus the number of corrupt components of P (a corrupt component is a maximal subpath that contains corrupt nodes). In other words, every sequence of adjacent corrupt nodes on a path can appear to be one node. In the presence of remote collusion between nodes and if there are no colluding adjacent nodes, then a further modification guarantees that the hop-count distance calculated by an honest node is not shorter than the shortest distance from destination to the node. The protocol has two basic components. The first component, which is based on [4], is a practical and simple protocol that enables a node to determine if another node is its physical neighbor. The second component is a novel use of key chains, which we call *hop-chains*, that enable the destination to certify *remotely* its distance to nodes in the network. To tolerate initial collusion, we introduce *mistrust hop-chains* to prevent nodes from cheating by initially agreeing on keys. The idea is to associate two keys for each node; one key is generated by the node and one key is generated by a neighbor of the node. This secure routing protocol we present is a significant contribution on its own. The protocol is more secure and requires fewer assumptions than other secure routing protocols in the literature (see Section 10).

Our solution to the Sybil attack problem proposes to use the secure routing protocol in order to come up with a secure localization protocol for networks in which hop-count distances from a number of beacons (or anchor points) can be used to localize nodes. This is along the lines of the approach of [2], but replacing roundtrip delays with hop-count distance.

2. Identities and public keys

In our model, only the destination has a public key that needs to be known by all other nodes. Other nodes need to have identities that cannot be forged by faulty nodes. A faulty node should not be able to create messages that appear to originate from *the identity of* a correct node. This can be achieved by having each node randomly choose its own public key and corresponding private key. We assume that the key space is large enough so that corrupt nodes can with negligible probability guess or generate keys identical to those of honest nodes. Also, correct honest nodes generate different keys with high probability. This guarantees that with high probability nodes cannot forge messages, but does not rule out that corrupt nodes can replay messages.

In our framework, the identity of a node is its public key. For an honest node, this identity is unique and does not change over time. For a corrupt node, there can be multiple identities, one for each public key that the node chooses. Note that this use of public keys does not require a central authority and it is fundamentally different from their use in authentication. We spell out our assumptions about keys in Section 6.

3. Neighbor computation

The ability of a node to determine whether another node is its neighbor is an important ingredient for our secure routing protocol. Before we explain how that determination can be done, we need to precisely define what we mean by "determining if a node is the neighbor of another node".

[2] In their work, they propose approaches to handle inaccuracies, but these approaches are incomplete.

We say that a node is the neighbor of another node if the two nodes can communicate directly and not through an intermediate node. In wireless networks, this requires that the nodes are in each other's range. In wired networks, this requires that the nodes either have access to a shared communication link or share a private link.

In our model, nodes are known to other nodes through their public keys. So, determining whether a node is the neighbor of another node reduces to determining if the owner of the private key corresponding to a given public key is in the neighborhood of the node. What we determine is something subtly different from the foregoing. We determine if a node with *access to* the private key corresponding to a given public key is in the neighborhood of the node. The distinction is subtle, but important. A node has *access to* the private key if it has the private key or it is in collusion with a node that has the private key. If there is no collusion other than initial collusion between nodes, then having access to a key and having a key are the same thing.

3.1. Immediate neighbors

A first step in neighbor determination is to broadcast a message requesting for neighbors to provide their public keys. The goal of neighbor determination is to determine if a neighbor of the node has access to the private key corresponding to the provided public key.

A naive approach for determining whether a node is the neighbor of another node is to send a request message and wait for a reply within a short period of time. The reply should allow for the transmission time, roundtrip delay and any local computation at the node to encrypt and decrypt messages exchanged to prevent third parties for interfering with the communication. Unfortunately, the time for computations can be substantial especially if public key encryption is involved which makes the approach vulnerable to a man-in-the-middle attack.

A better approach is similar to the one taken by [4] in which communication is done in the clear to eliminate a long processing time. In a first phase, a node sends a random bit in a message encrypted with the destination's key. The destination decrypts the message and recovers the bit. In a second phase, the node broadcasts a random one bit message in the clear to all its neighbors. Upon receipt of the message, the destination XOR the received message with the random bit and sends the resulting bit in the clear to the sender. The extra processing time is minimal. The probability that the destination sends the correct answer without knowing the random bit is 1/2. This probability can be made arbitrarily small by repeating the two phases multiple times. A corrupt node B cannot compromise this scheme by launching a man-in-the-middle attack in a timely manner. But, a corrupt node B can execute the first phase by colluding with another node C, which decrypts the first phase message and provides the value of the bit to B. This way, B can execute the second phase. Thus, this two-phase approach guarantees that *B has access* to the private key corresponding to the public key it sends to A.

3.2. Neighbors of neighbors

To tolerate runtime collusion between non-adjacent nodes and if there are no colluding adjacent nodes, our routing protocol requires the ability for a node to determine if another node is the neighbor of its neighbor. To determine neighbors of neighbors, we propose to use the same approach we use for determining neighbors. The difference is that we should allow more time for the message to be forwarded to a neighbor of a neighbor and for the reply to arrive to the sender. This reduces the accuracy of the measurements.

3.3. Effects of congestion

If two nodes have a dedicated link between them, then adjacency determination (neighbors) can be done without interference by other nodes. In a wireless medium, other nodes can interfere in the communication by launching denial of service attacks. We do not address denial of service attacks in this paper. Our assumption about congestion is fundamentally different from the assumptions in [2]. In our work, we make the realistic and practical assumption that two *adjacent* (immediate neighbors) nodes can communicate with no congestion for some periods of time, whereas in [2], a similar assumption is made for nodes that are many hops apart.

3.4. Timing considerations

In our neighbor computations, we assume that the dominant factor in the delay is due to transmission and processing, but not propagation delay. The transmission rate between two adjacent nodes is determined by their hardware and it is reasonable to assume that nodes can measure time to an accuracy of 1 bit. In wireless networks, speeds of 100 Mbps can be considered high. At this speed, a 4 kByte frame takes around 0.32 ms. During that time, a signal can propagate up to 96 km which is way beyond the range of node to node transmission in ad-hoc networks. In wired networks, propagation delay can be substantial for transatlantic communication, but such communication has to go through known entities that charge for their services and cannot be part of any ad-hoc network.

It should be clear that while our neighbors determination algorithm assumes 1-bit messages, these messages can be appropriately padded to obtain messages of a more appropriate length.

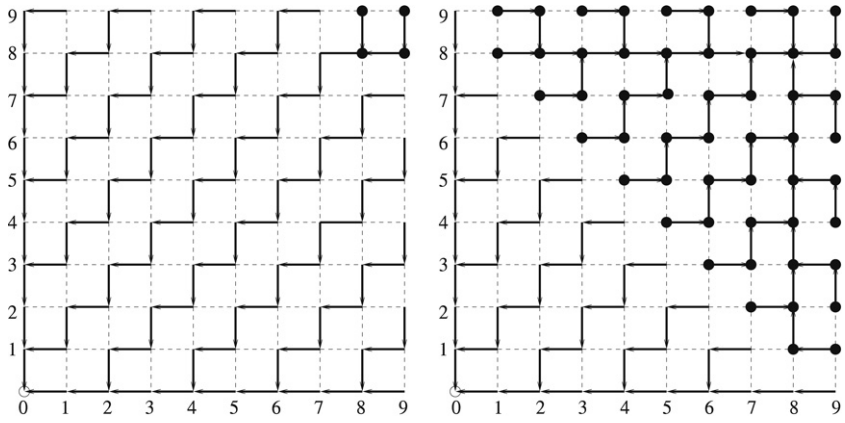

Fig. 1. Routing tree when (8, 8) does not lie (left) and when (8, 8) lies (right).

4. Distance vector routing and its vulnerabilities

In a traditional distance vector routing protocol, nodes in a network collaborate to build a spanning tree whose root is the ultimate destination node d. Initially, no node u other than d has a parent in the routing tree, and the distance of node u is infinite. Only the ultimate destination node d is in the routing tree, and it periodically broadcasts an advertisement message of the form $adv(d, 0)$ to its neighbors.

When a node u whose current distance is greater than $s + 1$ receives an $adv(v, s)$ message, node u makes node v its parent in the tree, and sets its distance to be $s + 1$. Once node u has a parent in the tree, node u becomes connected to the tree and starts sending an $adv(u, s')$ message periodically, where s' is the current distance of u. When u stops receiving advertisement messages from its parent for a certain time period, it stops sending advertisement messages. A node that is disconnected from the tree can become reconnected at a later time if it receives an advertisement message from one of its neighbors.

4.1. Effects of misbehavior

A node can cause harm if it drops packets it is supposed to forward[3] or if it reports a false (shorter) distance to destination.

The first type of misbehavior is illustrated on the left side of Fig. 1 for the case of a 10*10 grid whose ultimate destination is node (0, 0) and in which node (8, 8) does not forward messages it receives. The black circles illustrate the 3 affected nodes and node (8, 8).

A corrupt node can cause more harm by falsely reporting its distance to destination and dropping messages. This is illustrated on the right side of Fig. 1 in which node (8, 8) reports a distance of 2 to destination. The black circles illustrate the 52 affected nodes and node (8, 8) (in general, the number of affected nodes depends on the difference between the reported distance and the real distance).

Note that it does not help a corrupt node to advertise a distance longer than its actual distance because in that case the neighbors of the corrupt nodes will choose alternative nodes for routing. Also, there is no way to force a corrupt node, save for re-initializing it, to forward messages that it receives. Our goal for a secure distance vector routing protocol is to prevent corrupt nodes from reporting distances that are smaller than their actual distances to destination. When this is not possible, we aim at reducing the difference between the actual distance and the reported distance.

4.2. Adversary models

The proposed secure routing protocol tolerates strong adversaries. We consider Byzantine failures with three models on how they can collude.

4.2.1. Byzantine failures

In the Byzantine failure model, faulty, or corrupt, nodes can behave arbitrarily. In particular, they can advertise multiple public keys (attempt Sybil attacks) and they can replay or resend messages received from others.

[3] There are techniques to detect nodes that do not forward messages [5], but in this paper we do not consider the problem of detecting such nodes.

4.2.2. Initial collusion of nodes

Corrupt nodes that initially collude can share information before the execution of the protocol, but they cannot communicate information that they learn during the execution of the protocol. To our knowledge, this model has not been considered by others. It makes sense to consider it in our setting because nodes are known to others through their public keys. It is not clear how a node can find or *trust* another node to collude with. If a node is corrupted with a virus, for example, then two corrupted nodes share the common information that the virus carries and therefore they initially collude. Non-colluding nodes cannot share their public and private keys, but colluding nodes can share initially any number of private and public keys, but not keys they receive from other nodes.

4.2.3. Runtime collusion of adjacent nodes

Even though it is not clear how nodes can find other nodes to collude with, it makes sense to consider adjacent nodes that are colluding at runtime. In fact, if two adjacent nodes are initially colluding, they could discover that they have the same keys by communicating with each other and then decide to collude. Our protocol tolerates run-time collusion between adjacent nodes (or connected component consisting of corrupt colluding nodes).

4.2.4. Runtime collusion of non-adjacent nodes

Finally, we consider runtime collusion between non-adjacent nodes, but in the absence of collusion between adjacent nodes. We assume that non-adjacent corrupt nodes can communicate information with each other at any time.

The first protocol we present in Section 6 tolerates Byzantine failures, but not initial collusion or runtime collusion. Another versions of the protocol in Section 7 tolerates (1) initial collusion only, and (2) initial collusion plus runtime collusion of adjacent nodes (i.e. a connected component of colluding nodes), respectively. A final version of the protocol in Section 8 tolerates runtime collusion of non-adjacent nodes, but only in the absence of collusion between adjacent nodes. Following the presentation of the protocols, we present our approach to dealing with Sybil attacks in Section 9.

5. Assumptions

5.1. Public and private keys

Every node u in the network creates its own public key(s) (BK_u), and corresponding private key(s) (RK_u). The public key of the ultimate destination node d, BK_d, is known to all nodes. The encryption of a message m with a key BK_u is denoted $BK_u\langle m\rangle$ and we have $RK_u\langle BK_u\langle m\rangle\rangle = m$. Similarly, $BK_u\langle m\rangle$ is the decryption of m using the public key of u, and $BK_u\langle RK_u\langle m\rangle\rangle = m$. It should be clear from context if a public key is used for encryption or decryption.

If a node receives a message (m, m') such that $BK_u\langle m'\rangle = BK_u\langle RK_u\langle m\rangle\rangle$, then m' must have been encrypted by u, which has RK_u. To reduce the size of messages, we assume the existence of a message digest, or one way hash, function MD. The use of MD is not needed for the correctness of the protocol. If a node receives a message (m, m') such that $BK_u\langle m'\rangle = BK_u\langle RK_u\langle MD(m)\rangle\rangle$, then m' must have been encrypted by u.

5.2. Timing assumptions

The timing assumptions we make in this paper are needed for neighbor calculation and for ensuring progress.

5.2.1. Neighbor calculation

The first is an assumption about accurately measuring time between neighbors. We assume that local synchrony allows the computation of neighbors (or neighbors of neighbors) as described in Section 3. Except for this computation, this assumption does not figure directly in the proofs of protocols we present; we simply assume that the neighbor computations are accurate.

5.2.2. Progress

We assume that nodes can send messages at positive intervals of time to maintain progress. This is needed to send advertisement messages from time to time. We also assume that a node can timeout a parent that does not send messages for a long period of time. This is needed to allow a node to drop an unresponsive parent and choose a new parent.

6. Tolerating non-colluding Byzantine failures

This section presents a basic protocol for the case in which nodes are subject to Byzantine failures, but they do not collude. We also assume that there are no adjacent Byzantine nodes.

In the protocol, each node u that is connected to the routing tree maintains a hop-chain that verifies its hop-count distance to destination. The hop-chain contains a sequence of public keys and a sequence of certificates. The public keys are supposed to be keys of a sequence of nodes $d = u_0, u_1, \ldots, u_k = u$ that form a path from d to u. The certificates vouch that every node in the sequence is the neighbor of the next node in the sequence. Finally, a hop-chain has a date dt that specifies the period of validity of the chain. The date changes infrequently relative to the communication delays in the network, and it does not require any tight synchronization between the nodes. A hop-chain has the format: $\langle dt, BK_{u_0}, BK_{u_1}, \ldots, BK_{u_{k-1}}, BK_{u_k}, C_{u_0}, C_{u_1}, \ldots C_{u_{k-1}}, C_{u_k}\rangle$, where $C_{u_0} = RK_{u_0}\langle MD(dt, BK_{u_0})\rangle$, and $C_{u_i} = RK_{u_{i-1}}\langle MD(dt, BK_{u_i})\rangle$, $0 < i \leq k$.

Definition 1. The *length* of a hop-chain H_u of a node u, denoted $len(H_u)$, is the number of certificates in H_u.[4]

It is straightforward to see that only a node that has access to RK_{u_0} can generate C_{u_0}, and only a node that has access to $RK_{u_{i-1}}$ can generate C_{u_i}. We say that a hop chain is valid if it is of the form $\langle dt, K_0, K_1, \ldots, K_k, C_0, C_1, \ldots, C_k\rangle$ and

- $K_0 = BK_d$ and $BK_d\langle C_0\rangle = MD(dt, BK_d)$
- $BK_{i-1}\langle C_i\rangle = MD(dt, K_i)$, $0 < i \leq k$.

A node u that receives a hop-chain can locally check its validity. The protocol ensures that the owners of successive keys in the sequence are neighbors or are identical (creating successive bogus identities). It will follow that the hop-chain length of node u minus one cannot be less than the shortest distance from u to d.

6.1. Protocol description

Initially, no node u other than d has a parent in the routing tree, and the distance of node u is infinite. Only node d is initially connected to the routing tree.

Each hop-chain contains a date field dt that indicates the date (time) at which the chain is generated. The root of the routing tree, node d, periodically updates dt every P seconds. Thus, node d periodically recomputes its chain $\langle dt, BK_d, RK_d\langle MD(dt, BK_d)\rangle\rangle$ every P seconds. The period of time P is chosen to be larger than the transmission delay between nodes in the network.

A node u that is connected to the tree periodically broadcasts an advertisement message to its neighbors every p seconds where $p \ll P$. The advertisement message of node u has the form $adv(BK_u, dt, H_u)$, where dt is the latest date that u is aware of, and H_u is the latest chain that u has. To keep the code simple, we do not explicitly maintain a list of neighbors of u in the code and we do not explicitly write in the code the recipients of the advertisement message (the neighbors of u).

When a node u receives an advertisement message, $adv(bk, t, h)$, where bk is a public key, t is a date, and h is a hop-chain, u ignores the message if t is smaller than the latest date u is aware of — the message is too old. If the message has a date that is more recent than the latest date at node u, u verifies that the message is valid. If the message is valid, u tentatively decides to use the received hop-chain to calculate its distance to destination. An adv message is valid, if bk is the *public key of a neighbor* of u, the date of the message is the same as the date of the hop-chain h, the hop-chain h is valid, and the last key in the chain is equal to bk. Note that in this case it is possible that the distance to destination might increase due to the newly received hop-chain. This can happen because the older information that the node has might be obsolete. If it is the case that: the date of the message is the same as the most recent date u is aware of; the message is valid; and, the length of h is smaller than the current distance of u to destination, u tentatively decides to use the received hop-chain to calculate its distance to destination. The decision is tentative because the node needs to first add itself to the chain so that it can advertise the newly adopted distance to other nodes. This is achieved by interacting with the sender of the advertisement message — its potential parent.

When a node u tentatively decides to use the received chain to calculate its distance to destination, u makes bk its potential parent, assigns h to the potential chain ph, and assigns t to the potential date pdt. Finally, u sends a reply message to node bk and waits to receive a certificate from bk — its potential parent. The reply message is of the form $rpl(bk, BK_u, t)$. While node u is waiting to receive a certificate, u ignores any advertisement messages u receives until u receives a certificate or u times out. This is only to keep our code easy to follow. In fact, there is no loss in ignoring advertisement messages, since advertisement messages will be sent periodically, and so node u can receive them later.

When a node u that is connected to the routing tree receives a reply message $rpl(bk, bk', t)$, where $bk = BK_u$, and t equals dt in its own hop-chain, node u first computes a certificate $c = RK_u\langle MD(dt, bk')\rangle$, and then u sends an acknowledgment message $ack(bk', t, c)$ to node bk'.

When a node u that is waiting to receive a certificate from its potential parent receives an acknowledgment message $ack(bk', t, c)$, where $bk' = BK_u$, and $t = pdt$, node u checks the validity of the certificate in the message. The certificate c is valid if it is encrypted with the corresponding private key of the last key pbk in the potential hop-chain ph (the key of its

[4] Note that $len(H_u)$ minus one is equal to the hop-distance from destination to u.

```
1:  var   dt          : integer,     // date
2:        H_u, ph     : integer,     // current/potential hop-chain
3:        ds          : 0..dmax+1,   // distance, initially dmax+1
4:        BK_p        : integer,     // parent's permanent public key
5:        trc         : 0..tmax,     // time to remain connected
6:        wait        : boolean,     // wait for ack msg or not, init. false
7:        pdt, t      : integer,     // potential and received date
8:        h, c        : integer,     // received hop-chain/certificate
9:        bk, bk'     : integer      // received keys
```

Fig. 2. Variables of a node u.

potential parent) and so $\langle c\rangle = pbk\langle MD(dt, BK_u)\rangle$. If the certificate in the message is valid, u makes node pbk its parent in the routing tree and updates its distance to destination. Finally, u computes its (new) hop-chain by adding BK_u and c to the hop-chain of its parent, ph.

When a node u has a parent and does not receive any valid advertisement message from its parent for a time period of $tmax \times p$ seconds, u concludes that it is no longer connected to its parent. Thus, u disconnects from the tree by making its distance infinite, and stops sending advertisement messages.

After node u sends a reply message to its potential parent v, u starts a timer and times out after w seconds, at which time u no longer considers v as its potential parent. The value of w is chosen to be large enough to accommodate a roundtrip delay to a neighbor including time for public key encryption and decryption. The protocol variables and specification of a node u are given in Figs. 2 and 3.

6.2. Correctness proofs

It should be clear that if the topology of the network is changing at a rate that is larger than the rate at which updates are executed, there can be in general no guarantees on the relationship between the calculated distance to destination and the actual distance to destination. Therefore, in establishing the properties of the protocol, we will assume that the topology of the network is fixed. The results would still apply during periods in which the topology is stable and only changes infrequently. In that case, the lemmas might be violated at the times of change, but such a discrepancy is unavoidable for any routing protocol in a dynamic network due to the fact that information about topology changes take time to propagate and before it reaches a given node, the routing information at that node might be obsolete (it is straightforward to construct scenarios for which this is the case).

We show that no node u can advertise a path to destination that is shorter than the hop count distance from u to destination. Also, we show, that each node u will eventually calculate a path that is equal to the shortest path consisting solely of honest nodes (a good path) from u to destination.

Definition 2 (*Path Length*). The *length* of a path P from the destination to a node u, denoted $len(P)$, is the number of nodes in path P.

Definition 3 (*Good Path*). A *good path* from a node u to the destination node d is a path from u to d that contains only correct nodes.

Lemma 1 (*Routing Information is Eventually up to Date*). *If there is a* good path *from a node u to the destination node d, then there is a time after which all hop-chains validated by correct nodes have a date after the date of the last topology change in the network.*

Proof. The proof is by induction on the length of the good path. For the base case, $d = u$. Node d updates its time and hop-chain at regular intervals of time (lines 2–5). For the induction step, consider a path of length $l > 1$ and consider the node u' that is the neighbor of u on the path. The length of the path from u' to d is $l - 1$. It follows, by the induction hypothesis, that eventually, u' will only validate and advertise hop-chains that have a date larger than the date of the last topology change in the network. Node u will receive advertisement of such hop chains and by line 21 of the code will accept a valid advertisement from u' whose date is larger than the date of the last topology change. From that time on, all hop-chains validated by u will have a date larger than the date of the last topology change. □

We note here that this lemma applies to all the protocols we present in this paper and we will not prove the same lemma for the other protocols as the proof is identical.

Lemma 2 (*Len(Hop-Chain) ≥ Len(Shortest Path)*). *For every honest node u, $len(H_u) \geq len(S)$, where S is the shortest path from node d to node u and $len(S)$ is the number of nodes in path S.*

Proof. We assume that all hop-chains have a date that is larger than the date of the last topology change in the network. This is guaranteed by the previous lemma.

Consider the shortest valid hop-chain that an honest node can (potentially) have. In such a chain the owners of the keys are all distinct. In fact, if two keys have the same owner, one can obtain a shorter chain as follows. Let K_i and K_j be two keys with the same owner and let $K_{i+1}, \ldots, K_{j-1}$ be the keys that appear between K_i and K_j in the key list. If the two keys are identical, it is easy to check that the chain obtained by deleting the keys $K_i, K_{i+1}, \ldots, K_{j-1}$ and the corresponding certificates

```
1:    timeout DATE expires → // d periodically updates date
2:                           if (u = d) then
3:                               dt := UPDATE_DT;
4:                               H_u := ⟨dt, BK_u, RK_u⟨MD(dt, BK_u)⟩⟩;
5:                               timeout DATE after P

6:  [] timeout ADV expires → // u periodically sends advertisement
7:                           if (u = d) then
8:                               send adv(BK_u, dt, H_u);
9:                               timeout ADV after p
10:                          elseif (u ≠ d) then
11:                              trc := MAX(trc − 1, 0);
12:                              if (trc > 0) then
13:                                  send adv(BK_u, dt, H_u);
14:                                  timeout ADV after p
15:                              elseif (trc = 0) then
16:                                  ds := dmax+1

17: [] timeout RPL expires→  // no longer wait for ack from potential parent
18:                          wait := false

19: [] rcv adv(bk, t, h)  →  // if valid more recent adv received from a neighbor or
                             // a current adv received from node closer to d
20:                          // update potential parent and reply to sender
21:                          if ¬wait ∧ (t > dt ∨ (t = dt ∧ len(h) < ds))
                             ∧ valid(adv(bk, t, h)) then
22:                              pdt := t; ph := h;
23:                              wait := true;
24:                              send rpl(bk, BK_u, t) to bk;
25:                              timeout RPL after w
26:                          // if valid advertisement received from parent
27:                          // stay connected for a longer period
28:                          if ((trc > 0) ∧ (bk = BK_p) ∧ (len(h) = ds)
                             ∧ valid(adv(bk, t, h))) then
29:                              trc := tmax

30: [] rcv rpl(bk, bk′, t) →  // if valid reply received from a node
31:                           // compute a certificate and send it to sender
32:                           if (BK_u = bk) ∧ (t = dt) ∧ (trc > 0) then
33:                               send ack(bk′, dt, RK_u⟨MD(dt, bk′)⟩) to bk′

34: [] rcv ack(bk, t, c)  →  // if valid ack received from potential parent
35:                          // update its parent, distance, chain, and send adv
36:                          if wait ∧ (BK_u = bk) ∧ (pdt = t)
                             ∧ valid(ack(bk, t, c)) then
37:                              dt := pdt;
38:                              H_u := COMP_CERT(ph, c);
39:                              ds := len(ph);
40:                              BK_p := GET_BKP(ph);
41:                              wait := false;
42:                              trc := tmax;
43:                              send adv(BK_u, dt, H_u);
44:                              timeout ADV after p
```

Fig. 3. A specification of a node u.

is also a valid hop chain which is shorter than the original. If the two keys are different, then the Byzantine owner of K_i and K_j could have affected a shorter chain by replacing K_j with K_i, changing the certificate generated by K_j to be generated by K_i, and deleting the keys $K_i, K_{i+1}, \ldots, K_{j-1}$ and their corresponding certificates to provide a shorter chain to its neighbors.

So, we consider the shortest possible chain H_u that an honest node can have. The owners of all keys in the key list of H_u are different. We show that any two keys that are adjacent on the list must be from adjacent nodes in the network. Since there are no adjacent Byzantine nodes, for any two adjacent keys on the list, the owner of one of them must be correct, for, otherwise, either the owner is the same Byzantine node or two non-adjacent Byzantine nodes that are colluding or two adjacent Byzantine nodes. For a pair of adjacent keys K_i and K_{i+1}, if the owner of K_i is correct, it follows that the owners of the two keys are different because the owner of K_i only validates keys from adjacent nodes. If the owner of K_{i+1} is correct, it follows that the owners of the two keys are adjacent because a correct node only accepts hop-chains from adjacent nodes. Since we assume that there are no adjacent Byzantine nodes and no colluding Byzantine nodes, it follows that the owners of two keys that are adjacent in the list must be distinct adjacent nodes in the network.

From the above, it follows that the owners of keys that appear in the hop-chain key list for a path from d to u must be distinct. Also, the hop-chain has a date that is larger than the date of the last topology change. It follows that the path defined by the hop-chain is an actual path in the network, hence the result. □

Lemma 3 (*Len(Hop-Chain) ≤ Len(Good Path)*). *For every honest node u, if there exists a good path G from node d to node u, then eventually* $len(H_u) \leq len(G)$ *holds.*

Proof. We assume that all hop-chains have a date that is larger than the date of the last topology change in the network. This is guaranteed by the Lemma 1.

Consider a path $d = u_0, u_1, \ldots, u_n = u$ consisting of only honest nodes from destination to u. The proof is by induction on the length of the path. For the base case, $d = u$ and the path is of length 1 and the chain length for d is of length 1. Assuming the statement is true for paths of length $n-1$, it follows that eventually u_{n-1} will have a hop-chain whose length is $\leq n-1$ and that hop-chain will be considered valid by every honest neighbor of u_{n-1} . When u_{n-1} sends that chain to u, either u already has a chain of length $\leq n$, in which case the statement is satisfied, or u will use the chain and create a certificate for u_{n-1} to obtain a chain of length $\leq (n-1)+1 = n$. □

7. Tolerating initial collusion

The protocol of the previous section is vulnerable to initial collusion. Consider two nodes u and v that share the public and private keys BK_u and RK_u. Assume that v is the node that is farther from the destination. Since v has u's keys, the neighbors of v will consider the owner of the private key RK_u to be their neighbor. If at some point, node v receives an advertisement message with a chain that contains the public key of u, v can *cut* the chain, only keep the portion of the chain that is identical to u's chain, and present that portion to its neighbors. Node v can then advertise u's chain to its neighbor, and the neighbors of v will find the received chain to be valid, in effect v manages to claim a distance to destination that is shorter than its actual distance to destination.

The reason for the success of this attack is that a node is certified based only on the initial information of the node, and initially colluding nodes share all their information. To get around this difficulty, we need to certify nodes based on information that they do not have initially. This can be achieved by having a parent in the routing tree create public/private key pairs for its children. These *temporary* keys will be used alongside the *permanent* keys of a node. We say that they are temporary because their values depend on the identity of the parent of a node at a given time. For a node u, we denote these keys with TBK_u and TRK_u. For the destination node d, we really need no temporary keys, but we introduce them to make the protocol more uniform.

7.1. Protocol overview

In the modified protocol, nodes use temporary keys and permanent keys to check the validity of a certificate and therefore of a chain. The *mistrust* hop-chain, or hop-chain, of a node has the format: $\langle dt, (BK_{u_0}, TBK_{u_0}), \ldots, (BK_{u_k}, TBK_{u_k}), C_{u_0}, C_{u_1}, \ldots, C_{u_{k-1}}, C_{u_k}\rangle$, where $C_{u_0} = RK_{u_0}\langle TRK_{u_0}\langle MD(dt, BK_{u_0}, TBK_{u_0})\rangle\rangle$, and $C_{u_i} = RK_{u_{i-1}}\langle TRK_{u_{i-1}}\langle MD(dt, BK_{u_i}, TBK_{u_i})\rangle\rangle$, $0 < i \leq k$.

It is straightforward to see that only a node that has RK_{u_0} and TRK_{u_0} can generate C_{u_0}, and only a node that has $RK_{u_{i-1}}$ and $TRK_{u_{i-1}}$ can generate C_{u_i}. We say that a mistrust hop-chain is valid if the chain is of the form $\langle dt, (K_0, T_0), (K_1, T_1), \ldots, (K_k, T_k), C_0, C_1, \ldots, C_k\rangle$ and

- $K_0 = BK_d$ and $TBK_d\langle BK_d\langle C_0\rangle\rangle = MD(dt, K_0, T_0)$
- $TBK_{i-1}\langle BK_{i-1}\langle C_i\rangle\rangle = MD(dt, K_i, T_i), 0 < i \leq k$.

Using permanent and temporary keys, the protocol ensures that the owner of every pair of permanent and temporary public keys in the sequence encrypts the certificate of the owner of the next pair of public keys in the sequence. Thus, a corrupt node u that initially colludes with another node v closer to destination cannot use the hop-chain of v, since u has no access to the temporary private key of v.

In the modified protocol, *rpl* and *ack* processing is modified as in Fig. 4. When a node u that is connected to the routing tree receives a reply message $rpl(bk, bk', t)$, where $bk = BK_u$, and t equals dt in its own hop-chain, node u randomly chooses temporary public/private key pair $(tcbk, tcrk)$ for node bk'. Node u then computes a certificate $c = RK_u\langle TRK_u\langle MD(dt, bk', tcbk)\rangle$, and also computes an encrypted temporary private key $r = bk'\langle tcrk\rangle$. Finally, node u sends an acknowledgment message $ack((bk', tcbk), t, r, c)$ to node bk'.

When a node u that is waiting to receive a certificate from its potential parent receives an acknowledgment message $ack((bk', bk''), t, r, c)$, where $bk' = BK_u$, and $t = pdt$, node u first checks the validity of the certificate in the message. The certificate c is valid if it is encrypted with the corresponding permanent and temporary private keys of the last key pair $(pbk, tpbk)$ in the potential chain ph (the keys of its potential parent) and so $tpbk\langle pbk\langle c\rangle\rangle = MD(t, BK_u, bk'')$. If the *ack* message is valid, u makes node pbk its parent in the routing tree and updates its distance to destination. Finally, u computes its temporary public and private keys, TBK_u and TRK_u by assigning bk'' to TBK_u, and $RK_u\langle r\rangle$ to TRK_u, and then computes its (new) hop-chain by adding (BK_u, TBK_u) and c to the hop chain of its parent, ph.

```
1:  [] rcv rpl(bk, bk', t)  →  // if valid reply received from a node
2:                             // choose temp. keys and send a cert. to sender
3:                             if (BK_u = bk) ∧ (t = dt) ∧ (trc > 0) then
4:                                 (tcbk, tcrk) := GEN_KEYS;
5:                                 send ack((bk', tcbk), dt, bk'⟨tcrk⟩,
                                   RK_u⟨TRK_u⟨MD(dt, bk', tcbk)⟩⟩) to bk'

6:  [] rcv ack(bk, bk', t, r, c) → // if valid ack received from potential parent
7:                                 // update its parent, distance, chain, and send adv
8:                                 if wait ∧ (BK_u = bk) ∧ (pdt = t)∧
                                   valid(ack(bk, bk', t, r, c)) then
9:                                     dt := pdt;
10:                                    TBK_u := bk'; TRK_u := RK_u⟨r⟩;
11:                                    H_u := COMP_CERT(ph, c);
12:                                    ds := len(ph);
13:                                    BK_p := GET_BKP(ph);
14:                                    wait := false;
15:                                    trc := tmax;
16:                                    send adv(BK_u, dt, H_u);
17:                                    timeout ADV after p
```

Fig. 4. *rpl* and *ack* processing to handle initial collusion.

7.2. Correctness proofs

For the correctness proofs, we assume that the network topology is stable for the period of time of interest.

Lemma 4 (*Initial Collusion: Len(Hop-Chain) ≥ Len(Shortest Path)*). *If there are no adjacent Byzantine nodes and Byzantine nodes collude only initially, then, for every honest node u, len*$(H_u) \geq$ *len*(S)*, where S is the shortest path from node d to node u.*

Proof. The proof is essentially identical to that of Lemma 2. The only difference is in establishing that the owners of two adjacent keys on the chain's list of keys are adjacent. In the modified protocol for handling initial collusion, we have key pairs instead of keys. In the proof of Lemma 2, there were three possibilities regarding the owners of those key pairs: the owner is the same Byzantine node, two non-adjacent Byzantine nodes that are colluding (initially), or the owners are two adjacent Byzantine nodes. We can rule out the first and third possibilities but not the second case. The first possibility is ruled out because we are considering a minimal chain and the third possibility is ruled out because we do not allow adjacent Byzantine nodes in our setting. So, we are left with only the second possibility. It is possible that the two owners are non-adjacent colluding Byzantine nodes. We show that the protocol rules out this possibility.

For a contradiction, assume that there is a pair of adjacent permanent keys whose owners are non adjacent colluding nodes. Consider the first two such adjacent key in the list: K_i and K_{i+1}, and their owners p and q. Since d is assumed correct, it follows that the owner r of the pair of keys (K_{i-1}, K'_{i-1}) must exist and is correct (it has to be correct because p is Byzantine and there are no adjacent Byzantine nodes). The certificate of (K_i, K'_i) requires knowledge of the (randomly generated) temporary key K'_i which is provided by r. Since nodes p and q are only initially colluding and all the neighbors of p are correct, it follows that q cannot obtain the temporary key K'_i and can only guess it with very low probability. Hence the certificate of p in the chain cannot be provided by q and the hop-chain is not valid; a contradiction. □

Lemma 5 (*Initial Collusion: Len(Hop-Chain) ≤ Len(Good Path)*). *For every honest node u, if there exists a good path G from node d to node u, then eventually len*$(H_u) \leq$ *len*(G) *holds.*

Proof. The proof of this lemma is identical to that of Lemma 3. □

The last lemma considers initial collusion of nodes as well as runtime collusion of adjacent nodes. We show that in this case, the path length cannot be arbitrarily shrunk. The path can only be shrunk by a number of hops that is roughly equal to the number of adjacent Byzantine nodes. It is important to note that this model of collusion seems to be the strongest model that makes sense in a network in which nodes choose their own identifiers. Nodes that are adjacent can communicate and collude at runtime and this can be extended to connected components of nodes. Two nodes that are not adjacent and for which there is no path that consists solely of Byzantine nodes and that connects them should not be able to communicate at runtime. The situation is illustrated in Fig. 5. Nodes in the same collusion region can appear to be one node thereby potentially shrinking paths between correct nodes. This is illustrated in Fig. 6.

Definition 4 (*Maximal Corrupt Component*). A *maximal corrupt component* is a maximal connected component of Byzantine nodes.

Definition 5 (*Shrinkage of Corrupt Components*). A network is obtained by shrinkage of corrupt components if for every maximal corrupt component, the nodes are replaced with one node that is connected to every node that is adjacent to the component (Fig. 6).

Lemma 6 (*Initial Collusion + Collusion of Adjacent Nodes*). *For every honest node u, if there exists a path P from node d to node u, then len*$(H_u) \geq$ *len*$(P) - (|cor(P)| - |cor\_comp(P)|)$*, where cor*$(P)$ *is the set of corrupt nodes in P and cor_comp*(P) *is the set of maximal connected components of P whose elements are all corrupt.*

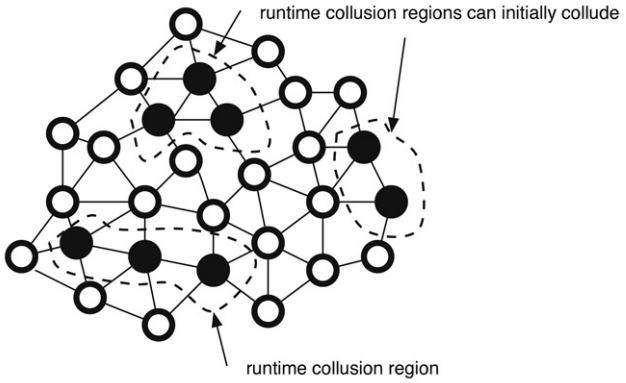

Fig. 5. Runtime and initial collusion in a network.

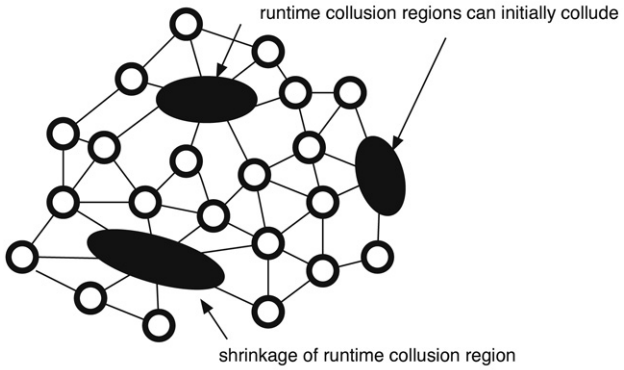

Fig. 6. Runtime and initial collusion in a network: shrunk network.

Proof. We assume that all hop-chains have a date that is larger than the date of the last topology change in the network. This is guaranteed by Lemma 1.

Consider the shortest valid hop-chain that an honest node can have. In such a chain the owners of all key pairs are distinct. In fact, if two key pairs have the same owner, one can obtain a shorter chain as follows. Let (K_i, K_i') and (K_j, K_j') be two key pairs with the same owner and let $(K_{i+1}, K_{i+1}'), \ldots, (K_{j-1}, K_{j-1}')$ be the keys that appear between i and j in the key list. If the two keys pairs are identical, it is easy to check that the chain obtained by removing $K_i, K_{i+1}, \ldots, K_{j-1}$ and the corresponding certificates is also a valid hop chain which is shorter than the original. If the two key pairs are different, then the Byzantine owner of K_i and K_j could have affected a shorter chain by replacing (K_j, K_j') with (K_i, K_i'), changing the certificate generated by (K_j, K_j') with the certificate generated by (K_i, K_i'), and deleting the key pairs $(K_i, K_i'), \ldots, (K_{j-1}, K_{j-1}')$ and their corresponding certificates to provide a shorter chain to its neighbors.

So, we consider the shortest possible chain H_u that an honest node can have. The owners of all keys in the key list of H_u are different. We show that any two keys that are adjacent on the list must be from adjacent nodes in the network obtained by shrinkage of corrupt components. We first note that in the shrunk network, there are no adjacent Byzantine nodes. A maximal corrupt component has access to all keys of all nodes in the component. So, in effect, this node can be considered as one node with many keys.

Since there are no adjacent Byzantine nodes in the shrunk network, for any two adjacent key pairs on the list, the owner of one of them must be correct, for otherwise either the owner is the same Byzantine node or two non-adjacent Byzantine nodes that are colluding or the owners are two adjacent Byzantine nodes. For a pair of adjacent key pairs (K_i, K_i') and (K_{i+1}, K_{i+1}'), if the owner of (K_i, K_i') is correct, it follows that the owners of the two key pairs are different and adjacent because the owner of (K_i, K_i') only validates keys from adjacent nodes (note that due to runtime collusion, a node that is adjacent to one node in a corrupt component is effectively adjacent to all nodes in the component). If the owner of K_{i+1} is correct, it follows that the owners of the two keys are adjacent because a correct node only accepts hop-chains from adjacent nodes (see parenthetical comment above).

From the above, it follows that the owners of keys that appear in the hop-chain key list form a path from d to u (in the shrunk network). Also, the hop-chain has a date that is larger than the date of the last topology change. It follows that the path defined by the hop-chain is an actual path in the shrunk network, hence the result. □

```
1:  [] rcv rpl(bk, bk', t) →  // if valid reply received from a node
2:                            // choose temporary keys randomly, compute a certificate and send it to sender
3:                            if ((BK_u = bk) ∧ (t = dt) ∧ (trc > 0) then
4:                                  (tcbk, tcrk) := GEN_KEYS;
5:                                  send ack((bk', tcbk), dt, RK_u⟨TRK_u⟨MD(dt, bk', tcbk)⟩⟩) to bk'
7:  end
```

Fig. 7. Modification to handle runtime collusion.

8. Tolerating runtime collusion of non-adjacent nodes

The protocol in the previous section will not work if two colluding nodes can share both their permanent as well as their temporary keys provided by their parents. In order to tolerate runtime collusion of non-adjacent nodes, we make the parent of a node u keep the temporary private key of node u. However, implementing this idea is not straightforward. Consider the following modification. The hop-chain format does not change, except that the temporary private key of u is kept at the parent of u. The *rpl* message sent by u contains a nonce that the potential parent v of u should forward to its parent to encrypt with the temporary private key of v. When v receives this *rpl* message, v forwards the nonce to its parent (the potential grandparent of u). Later when v receives the encrypted nonce from its parent, v forwards it to u. Clearly, this will not work because two non-adjacent colluding nodes v and v' can cheat as follows. The farther node v forwards the nonce sent by u to node v' closer to destination, and in turn v' forwards it to its parent. When the parent of v' sends the encrypted nonce to v', v' forwards it to v that forwards it to u. Note that with this attack, two Byzantine nodes that are at an arbitrary distance apart can appear to be adjacent. This situation is worse than what could happen with initial collusion and collusion of adjacent nodes because there we need many Byzantine nodes to shrink the distance considerably.

In the above modification, we need to ensure that the parent of v, the grandparent of u, is a neighbor of v. Thus, we need to run the *neighbor of neighbor* determination protocol described in Section 3 for the owner of the temporary private key of v, which should be a neighbor of a neighbor of u.

The protocol is modified as follows. First, an *ack* message sent from u to v does not contain the temporary private key of v generated by u. Second, when u receives an $adv(bk, t, h)$ message, u needs to check that the received hop-chain is valid and that the owner of the last temporary private key in the chain is a neighbor of a neighbor of u. Fig. 7 illustrates the modification that needs to be done to the second protocol. As in the other protocols, we do not explicitly show the determination of the *neighbor of neighbor* relation.

Lemma 7 (*Collusion: Len(Hop-Chain)* $\geq$ *Len(Shortest Path)*). *For every honest node u, len*$(H_u) \geq$ *len*(S)*, where S is the shortest path from node d to node u.*

Proof. The proof of this lemma is essentially the same as the proof of Lemma 4. The difference in the two settings has to do with how chains are validated. In the protocol that handles initial collusion, when an advertisement message is received, the validity of the last two keys in the chain is done by checking that their owner is a neighbor of u. In the modified protocol, the validity check is done by establishing that the owner of the permanent key is the neighbor of u and the owner of the temporary key is the neighbor of a neighbor of u.

We consider a minimal chain and we want to prove that the owners of adjacent permanent keys on the list are adjacent nodes, one of which is correct. We note first that a minimal chain cannot have more than 2 consecutive keys whose owners are Byzantine. In fact, if we have three or more consecutive keys whose owners are Byzantine, the owners of the middle keys in the sequence could collude with the owners of the first and last key of the sequence and we could obtain a shorter chain. So, for the rest of the proof we assume that no more than two consecutive keys have Byzantine owners. Here, we recall that for this theorem, we assume that Byzantine nodes can only collude remotely and that there are no adjacent Byzantine nodes.[5] As in the proof of Lemma 4, there is a number of possibilities regarding the owners of adjacent keys on the list: the two keys have as owner the same Byzantine node, the owners are two non-adjacent Byzantine nodes that are colluding at runtime or the owners are two adjacent Byzantine nodes. We can rule out the first and third case, but not the second case. It is possible that the two owners are non-adjacent runtime-colluding Byzantine nodes. We show that the protocol rules out this possibility.

For a contradiction, assume that there are two adjacent key pairs whose owners are non adjacent runtime-colluding nodes. Consider the last two such adjacent key pairs in the list: (K_i, K'_i) and (K_{i+1}, K'_{i+1}) and their owners p and q. By the earlier discussion, it follows that the owner r' of (K_{i-1}, K'_{i-1}) must exist and is correct because d and u are both correct by assumption and are on the list. Since p and q are not adjacent and r' verifies that p is a neighbor of a neighbor of q, it follows that either q is also a neighbor of r' (that claims to be a neighbor of a neighbor of p) or q is not a neighbor of r' (in addition to not being a neighbor of p). These two cases are illustrated in Fig. 8. In the first case, it should be clear that one can simply obtain a shorter chain by omitting p and its keys from the chain. In the second case, it is clear that q cannot pass the neighbor of neighbor test. So, in summary p and q cannot be non-adjacent. Also, they cannot be adjacent by assumption. So, there can be no two adjacent keys on the list whose owners are Byzantine. It follows that every Byzantine owner of a key on the list is adjacent to the correct owners of the two keys on the list and the list of keys defines a path from d to u, hence the result. □

[5] As we noted earlier, a model in which nodes collude remotely but not locally does not have a strong justification in practice, but our goal is to study ways to tolerate as many attacks as possible. The protocol for the previous section, in fact, handles what we believe to be the strongest practical adversary.

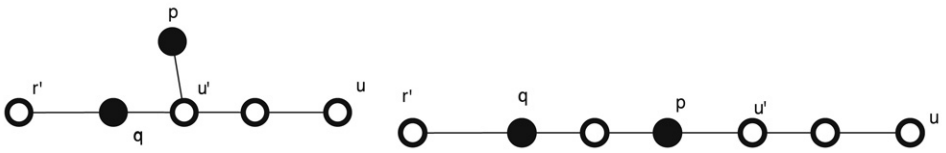

Fig. 8. Possibilities for runtime collusion between non-adjacent nodes.

9. Preventing and mitigating the Sybil attack

We say that a solution prevents Sybil attacks if no entity can successfully pretend to have more than one identity. We say that a solution mitigates Sybil attacks if the solution limits the number of identities that an entity can have. This is done under the assumption that nodes have vast resources, but we assume that corrupt nodes cannot break the public key encryption scheme in use. Bazzi and Konjevod [2] observed that to reduce the harm due to Sybil attacks, it is not necessary to be able to tell whether any two identities reside on different entities, rather it is enough to tell that, amongst a set of identities, a large enough subset of entities reside on different entities. So, our goal is to provide a *distinctness test* that can be used to determine a lower bound on the number of different entities on which a group of identities reside. The test is conservative as the lower bound can be smaller than the actual number of different entities. If the test returns 1 for example, that does not necessarily mean that all the identities reside on the same entity, rather it means that this is the best that can be determined. For a fuller discussion of distinctness tests, we refer the reader to [2].

In the following sections, we show how the routing protocol can be used to mitigate or prevent Sybil attacks under various assumptions about the network. We start by considering restricted settings, and then we consider a general network.

9.1. Sybil attack in an immediate neighborhood

In the simplest setting, we are interested in determining if two identities that are the immediate neighbors of a node reside on distinct nodes. If the nodes are connected with physical links, then it is easy to distinguish nodes because identities that appear on the same link can be considered to be identical. In this case, the number of physical links determines the number of neighbors.

If the nodes are connected through a wireless link, then the situation is similar to the situation described in Douceur's work [1]. Roundtrip delays cannot be used to differentiate two nodes and it seems that the only way to distinguish nodes is through an approach similar to that of Douceur [1], namely requiring the node to prove that it has the resources of multiple nodes. Still, a node cannot have more identities than the number of nodes that can physically fit in the neighborhood of a given node. So, the number of nodes that can fit in the neighborhood of a node is an upper bound on the number of identities that can be successfully claimed by a corrupt node. While in general such a bound can be very large, in some settings this bound can be small if additional knowledge is available about node distribution in space.

9.2. Sybil attack in a line

In this section, we consider a number of nodes connected in a line. This case is the basis for our treatment of Sybil attacks in general networks. We only consider how to detect multiple identities that correspond to the same entity or how to detect that a number of identities above some threshold correspond to the same entity. We are not concerned with a corrupt node that drops messages to disconnect nodes on its sides. This is obviously counterproductive for launching a Sybil attack and, in a general network, if there are alternative paths, (as we will discuss in the next section), the corrupt nodes will not benefit because traffic will go through different paths.

The result of the section will be to determine conditions under which corrupt node can successfully launch Sybil attack in a line. We only describe the approach for the case of initial collusion and collusion between adjacent nodes.

9.2.1. Initial collusion

Consider a sequence of nodes $A = A_0, A_1, \ldots, A_n = B$ such that the actual distance from A to A_i is equal to i and the actual distance from A_i to B is $n - i$. We assume that A and B are honest, the distance between A and B is n, and their public keys are known by all nodes in the line. Nodes A and B are beacons used to locate nodes on the line. Finally, we assume that only initial collusion exists in the network.

Under these assumptions, by Lemma 4, it follows that the length $len_A(H_{A_i})$ of a hop-chain from beacon A to A_i is greater than or equal to i. Similarly, the length of a hop-chain from beacon B to A_i, $len_B(H_{A_i}) \geq n - i$. If there are no corrupt nodes, then $len_A(H_{A_i}) + len_B(H_{A_i}) = n$. It follows that a corrupt node cannot insert any bogus nodes on a hop-chain without being detected because adding bogus nodes will make the sum greater than n.

If the distance between A and B is not known, and only a lower bound n_{low} and an upper bound n_{high} on the distance are known, any node A_i such that $len_A(H_{A_i}) + len_B(H_{A_i}) \leq n_{\text{high}}$ would be accepted. If the actual distance between A and B is n_{low}, then a corrupt node can insert up to $n_{\text{high}} - n_{\text{low}}$ bogus nodes without being detected.

The corrupt nodes as a group cannot insert more than $n_{\text{high}} - n_{\text{low}}$ bogus nodes (identities) without being detected. Since nodes are colluding only initially, this should also create a dilemma as to which nodes should be the ones to insert the bogus identities.

9.2.2. Initial collusion and collusion of adjacent nodes

Consider adjacent colluding nodes. In this case, using the protocol of Section 7, the nodes can shrink the path, but only by the number of corrupt nodes minus the number of corrupt components. This will not affect the above results, because the number of identities that can be created by corrupt nodes would still be $n_{\text{high}} - n_{\text{low}}$. The disappeared corrupt nodes can be replaced by other corrupt nodes that appear elsewhere on the line, but the corrupt nodes cannot add more than $n_{\text{high}} - n_{\text{low}}$ bogus identities.

We summarize the results of this sections with the following lemma.

Lemma 8. *On a line, the number of new identities that can be added is not more than $n_{\text{high}} - n_{\text{low}}$, where n_{high} and n_{low} are upper and lower bounds on the hop-count distance between the end nodes assumed honest.*

9.3. Sybil attack in a network

The results of the previous section form the basis for our approach to handle Sybil attacks in a network. In a general network, under the assumption of no collusion between adjacent nodes, the path from a beacon node to any node cannot be made shorter than it really is. Also, the length of a path from a beacon to any node is not more than the length of the shortest good path. If the network has enough redundant paths, then the distances between nodes are not affected by corrupt routers. In this case, we propose to use hop-count distances of a node from a number of beacons as the identity of the node. Two nodes that appear to be at different locations or a few hop counts from each other could be considered different.

There is already a good amount of work on hop-count based coordinates (see [6] for example), and it is not our goal in this paper to study this topic. We simply propose to use our secure routing protocol in conjunction with hop-count based coordinates in order to assign identities to nodes. Our results show that there is a bound on the amount of shrinkage introduced by corrupt nodes. The effect of that would be to change the location of a node from a point or small neighborhood to a larger neighborhood. Nodes whose locations or the neighborhood in which they are do not overlap can be considered to be different.

10. Secure routing related work

Secure distance vector routing protocols have been proposed by many researchers [7–11]. Existing protocols are based on assumptions that are stronger than the ones we make. The protocol proposed in [7] uses a set of the intrusion detection sensors to detect routing attacks, and requires knowledge of the network topology and sensor positions. SEAD [10] does not prevent corrupt nodes from replying to advertisement messages, and does not consider colluded attackers. In [11], nodes are assigned to unique identifiers (by a central authority), the destination knows all these identifiers, and the clocks of all nodes are tightly synchronized (to use [12]). RIP-RT [8] assumes that corrupt nodes cannot modify the value of the time-to-live field in a probing message and that any two nodes share a key. Moreover, this protocol assumes that each node knows the identifiers of adjacent nodes. The problem of detecting misbehaving nodes was considered in [5], which also proposes an on-demand secure routing protocol. Our routing protocol focuses on reducing harm caused by corrupt nodes that lie their distances, and does not consider detecting such nodes.

References

[1] J.R. Douceur, The Sybil attack, in: Proceedings of the International Workshop on Peer-To-Peer Systems, IPTPS, 2002, pp. 251–260.
[2] R.A. Bazzi, G. Konjevod, On the establishment of distinct identities in overlay networks, Distributed Computing 19 (4) (2007) 267–287.
[3] T. Ng, H. Zhang, Predicting internet network distance with coordinate-based approaches, in: Proceedings of INFOCOM, 2002, pp. 170–179.
[4] S. Brands, D. Chaum, Distance-bounding protocols (extended abstract), in: Proceedings of EUROCRYPT, 1993, pp. 344–359.
[5] B. Awerbuch, D. Holmer, C. Nita-Rotaru, H. Rubens, An on-demand secure routing protocol resilient to Byzantine failures, in: Proceedings of the 3rd ACM Workshop on Wireless Security, WiSE'02, ACM Press, 2002, pp. 21–30.
[6] R. Fonseca, S. Ratnasamy, J. Zhao, C.T. Ee, D. Culler, S. Shenker, I. Stoica, Beacon vector routing: Scalable point-to-point routing in wireless sensornets, in: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation, NSDI'05, USENIX Association, Berkeley, CA, USA, 2005, pp. 329–342.
[7] V. Mittal, G. Vigna, Sensor-based intrusion detection for intra-domain distance-vector routing, in: Proceedings of the 9th ACM conference on Computer and Communications Security, CCS'02, ACM Press, 2002, pp. 127–137.
[8] D. Pei, D. Massey, L. Zhang, Detection of invalid routing announcements in the rip protocol, in: Proceedings of GLOBECOM, 2003.
[9] T. Wan, E. Kranakis, P.C. van Oorschot, S-rip: A secure distance vector routing protocol, in: Proceedings of Applied Cryptography and Network Security, 2004, pp. 103–119.
[10] Y.-C. Hu, D.B. Johnson, A. Perrig, Sead: Secure efficient distance vector routing for mobile wireless ad hoc networks, in: Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications, WMCSA 2002, 2002, pp. 3–13.
[11] Y.-C. Hu, A. Perrig, D.B. Johnson, Efficient security mechanisms for routing protocols, in: Proceedings of the 10th Annual Network and Distributed System Security Symposium, NDSS 2003, 2003.
[12] Y.-C. Hu, A. Perrig, D.B. Johnson, Packet leashes: A defense against wormhole attacks in wireless ad hoc networks, in: Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies, 2003, pp. 1976–1986.