An Efficient Distributed PKI for Structured P2P Networks

François Lesueur Ludovic Mé Valérie Viet Triem Tong
SUPELEC, SSIR Group (EA 4039)
Avenue de la Boulaie - CS 47601 - 35576 Cesson-Sévigné cedex - France
E-mail: firstname.lastname@supelec.fr

Abstract

In decentralized P2P networks, many security mechanisms still rely on a central authority. This centralization creates a single point of failure and does not comply with the P2P principles. We previously proposed a distributed PKI for P2P networks which allows to push security mechanisms to the edges of the network but relies on unaffordable maintenance operations using byzantine agreements. In this paper, we address this shortcoming and propose efficient maintenance operations without any agreements. Our improvements allow a real deployment of this P2P PKI.

1 Introduction

P2P networks have been increasingly used for the last few years to design low cost and highly available systems. These large networks are based on a fully distributed architecture, in which every peer have an equivalent role. There are mainly two types of P2P networks: unstructured ones and structured ones. Unstructured P2P networks (Gnutella [6]), in which requests are broadcasted or routed through random walks, allow complex queries. Structured ones (Chord [19], Pastry [14], Kademlia [13]), in which requests are routed using generated routing tables, are limited to simpler queries but allow better performance. We consider here structured P2P networks security and we more precisely use the Kademlia overlay, although our scheme could be adapted to other overlays with some extra cost.

To provide secure peer identifier assignment, secure routing or membership control, [5, 18, 20] rely on Certification Authorities. In [4], Cao *et al.* propose to authenticate VoIP users using certificates. Skype also relies on such a central point for authentication purposes. In [1], Adya *et al.* present FARSITE, a P2P file system which uses a Certification Authority to enforce security properties usually proposed by networked file systems, for instance access control to files based on the intelligible names of the users. In [2], Aiello *et al.* propose Likir, a global security infrastructure for Kademlia which also relies on such a central point. All these central points are points of failure and trust and thus do not fulfill the P2P requirements.

It is thus crucial to provide fully distributed security mechanisms for P2P networks and, for instance, coupling [2] with a distributed Public Key Infrastructure would yield fully distributed security schemes for P2P networks. In [11], we propose such a distributed PKI which distributes a Certification Authority: signing a certificate needs the collaboration of a fixed ratio of the peers. Then, considering such a certificate valid is similar to trusting that this ratio of the peers would not collude to fake a certificate. A certificate contains the public key of its owner and any rights that may be needed by this owner (access rights, name ownership, ...). Such a distributed PKI can also be used for mitigating the sybil attack [12], excluding misbehaving peers [10] or providing intelligible names for P2P Voice over IP rather than cryptographic key hashes.

Nevertheless, the maintenance of the PKI proposed in [11] relies on byzantine agreements in groups composed of a few tens of members: we argue in this paper that agreements are inefficient in this context. Our contribution here is thus a set of efficient equivalent operations without any byzantine agreement, providing a lightweight P2P PKI.

In Section 2, we present the related work. In Section 3, we precisely describe the original distributed certification scheme we propose to extend in this paper. Then, in Sections 4 to 7, we present our contribution to the different operations, namely *split, refresh, merge* and *join*, used to maintain the sharing scheme without byzantine agreement. In Section 8, we evaluate our proposition. Finally, we conclude and suggest some future work.

2. Related Work

In this section, we present previously proposed distributed PKIs. Distributing a PKI can be accomplished using *threshold cryptography*. Threshold cryptography, based on Shamir secret sharing [17], consists in splitting a secret key and distributing the resulting shares on different enti-

ties. Then, ciphering a message needs the collaboration of a given number of entities. Since nobody knows the secret key, this key is better protected from misbehaving people. We present here threshold cryptography based on [7].

(t, n)-threshold cryptography allows to cipher a message with any t shares chosen among those issued to n entities, each entity usually owning one distinct share. t shares are needed to encipher a message, but $t-1$ shares hold *no* information on the secret key. An attacker must thus obtain t shares of the secret key to be able to recover the full key.

When some entity wants to obtain the signature of some data d, it asks t other entities to sign d with their own share. The signature of d by the secret key is then a combination of the t partial signatures. For instance, if the signature function f is homomorphic, i.e., $f(x+y) = f(x) \times f(y)$ (the RSA function is homomorphic), the combination of the partial signatures can be their multiplication. Only the partial signatures are published and not the key shares. Such threshold cryptography schemes have been previously used to provide distributed certification.

In [9], Kong *et al.* propose a distributed certification scheme based on the cooperation of t peers, t being a fixed number of peers during the whole life of the system. The choice of t is a problem since $t = 3$ might for instance be a correct value for a network composed of 10 peers but is clearly too small when the same network has grown to 1000 peers: t should be a ratio of the number of peers.

In [16], Saxena *et al.* propose to adapt the threshold t *dynamically* using algorithms proposed by Frankel *et al.* in [8]. A server manages a counter of the network size and detects the need for changing the threshold. Besides scaling and robustness problems in threshold changing algorithms, the reliance on a server is opposed to P2P basics and lowers the availability of the network. None of these schemes is thus adequate to distribute certification in a P2P network.

In [11], we propose a distributed certification scheme in which the threshold is dynamically adjusted to a constant ratio t of the number of peers. Moreover, this ratio is enforced using a fully distributed scheme and allows to tolerate misbehaving peers in the network. This scheme is thus adequate for P2P networks and, to our best knowledge, is the the only one satisfying all these constraints. However, this proposition relies on byzantine agreements which consume high bandwidth and are hard to compute and implement: in this paper, we propose an equivalent scheme with cheaper maintenance operations. We detail the original scheme in the following section.

3. Background: Original Scheme

In this section, we detail the original scheme of [11] we extend in this paper. We first provide a general overview and we then focus on the maintenance operations.

3.1. General Overview

The network is characterized by an RSA public/secret key pair (P, S), $P = (d, m)$ being publicly known and $S = (e, m)$ being shared among the peers (no peer knows S entirely). This key pair is originally generated by founding members using a distributed algorithm as Boneh and Franklin proposed in [3]. The sharing of the network secret key is based on the homomorphic property of the RSA enciphering function. Generally, if $S = (e, m)$ denotes the RSA network secret key, we can pick s *shares* $e_1, \ldots, e_s$ such that $e = \sum_{i=1}^{s} e_i$ and then for any data d:

$$d^e[m] = d^{\sum_{i=1}^{s} e_i}[m] = \Big(\prod_{i=1}^{s} d^{e_i}[m]\Big)[m]$$

In other words, the RSA signature of d with S, which is $d^e[m]$, is equal to the product of the signatures with each share modulo m, m being publicly known since it appears in the network public key $P = (d, m)$.

The network is thus decomposed in s *sharing groups*, each group being formed by g_{min} to g_{max} members. Each share of the network secret key is assigned to a specific sharing group and replicated on all its members, the sum of all the shares being constant and equal to e. Signing a certificate requires then every share and thus the collaboration of one peer of each sharing group. Given g_{min} and g_{max}, the ratio t of peers needed to sign a certificate verifies $\frac{1}{g_{max}} < t < \frac{1}{g_{min}}$ and thus, sharing groups can split (resp. merge) when peers join (resp. leave) with only local knowledge to enforce this ratio t.

Each share is uniquely identified by a binary $shareId$ and is known by peers which identifiers are such that $peerId = shareId*$ in binary form (i.e., $shareId$ is a binary prefix of $peerId$). Each peer knows only one share and nobody knows S entirely.

In Figure 1, there are three shares e_0, e_{10} and e_{11}. Nodes which identifiers begin with 0 know e_0, those with 10 know e_{10} and those with 11 know e_{11}. To get some data signed with S, one must obtain and multiply this data signed with e_0, e_{10} and e_{11}, and thus need the cooperation of three peers which identifiers begin respectively with 0, 10 and 11. In this paper, we use e_i to denote the numerical share e_i and $\breve{e}_i$ to denote the group of peers knowing e_i.

To obtain a certificate, a peer deploys a covering binary tree among the sharing groups. One peer of each sharing group is involved and a certificate is obtained only if one peer of each sharing group agrees with the request. Then, partial certificates are aggregated along the return path of the tree and the root obtains the fully signed certificate. This tree-based algorithm, combined with some redundancy to tolerate attackers, allows scalable certification process.

This system is vulnerable to the sybil attack but we propose in [12] a protection against this attack by coupling the

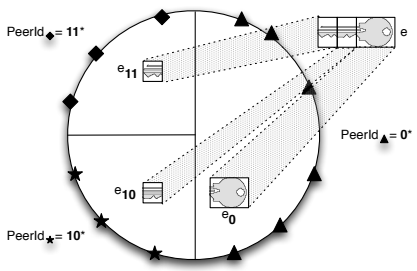

Figure 1. Distribution of three shares e_0,e_{10} and e_{11} with $e = e_0 + e_{10} + e_{11}$.

certification process itself with *SybilGuard* [21]. Using this protection, each peer is provided with a public/private key pair distributedly signed with the network secret key. In the following of this paper, we suppose that peers are provided with similar certificates and persistent identifiers. We also note that only online peers are part of the sharing scheme.

3.2. Maintenance Operations

Four maintenance operations -*split*, *refresh*, *merge* and *join*- allow to enforce the ratio of peers needed to obtain a valid certificate. These original operations are indeterministic, atomic and synchronized among peers of a sharing group: they use byzantine agreements in sharing groups.

When a group knowing the share e_i becomes composed of more than g_{max} members, after some peers join, this group must *split* into two groups $\check{e_{i0}}$ and $\check{e_{i1}}$ such that $e_i = e_{i0} + e_{i1}$. A peer in the group initiates a byzantine agreement to (1) decide whether to split and (2) decide a random value Δ. Then, peers of the group $\check{e_{i0}}$ know $e_{i0} = \Delta$ and peers of the group $\check{e_{i1}}$ know $e_{i1} = e_i - \Delta$. After this operation, the sum of all the shares remains e and the certification process is thus still available.

After a split, each group *refreshes* its share in order to render old shares useless. In fact, after a split, the two new groups know the old share and the randomly chosen Δ: although each peer belongs to only one of the two new groups, it also knows the other share. In the refresh operation, two shares are mixed together such that these two shares change yet the sum of all the shares remains constant. A peer in the group initiates a byzantine agreement to (1) decide whether to refresh, (2) discover another randomly chosen group and (3) globally choose a random value Δ. The group launching the refresh adds Δ to its share and the other group subtracts it: shares change yet the sum remains constant.

When a group $\check{e_{i0}}$ becomes composed of less than g_{min} members, after some peers leave, this group must *merge* with its neighbor group $\check{e_{i1}}$ to guarantee the availability of its share. A peer in $\check{e_{i0}}$ initiates a byzantine agreement to (1) decide whether to merge, (2) discover the neighbor group $\check{e_{i1}}$ and (3) obtain the sum of the shares to form $\check{e_i}$. After this operation, the sum of all the shares is still e.

Finally, when a new peer joins the network, it asks its g_{min} closest peers to obtain the list of members of its sharing group and its share.

However, the proposed maintenance operations rely on byzantine agreements which pose the following problems. First, sharing groups are composed of 20 to 40 members and executing byzantine agreements in such large groups is itself a challenging problem. Then, this problem is even harder in a P2P setup where peers can join and leave at any time, since there is already a problem in defining the set of peers participating to the agreement. Finally, implementing byzantine agreements is a difficult task.

For all these reasons, we think that the maintenance of this scheme cannot rely on byzantine agreements. We propose thus in this paper a novel contribution to maintenance operations enforcing the same key sharing scheme yet tackling the shortcomings of previously proposed ones. Our novel maintenance operations are lightweight, do not depend on cooperation in sharing groups nor on synchronization among peers. These operations are easy to implement and we already run a basic implementation on PlanetLab.

Although the original scheme was based on Chord, we base this new one on Kademlia [13]. Indeed, the XOR metric used in Kademlia decomposes the identifier space in a way similar to sharing groups and so, Kademlia provides a tree structure very similar to the one used for sharing the key. Kademlia offers thus efficient primitives for maintenance operations of the distributed PKI. Please notice nevertheless that our new scheme should still be adaptable to other structured overlays at some extra communication cost.

In the four following sections, we present our novel split, refresh, merge and join operations. As in the original scheme, these operations only involve communications among one or two groups of constant size and their communication overhead is thus independent of the network size. These operations use a data structure denoted *sharing tree* which we describe throughout these sections. Sharing trees allow to maintain all the possible shares in the system using a compact representation.

4. Split Operation

When a group $\check{e_i}$ is composed of more than g_{max} members, this group must split into two groups knowing respectively e_{i0} and e_{i1}, such that $e_i = e_{i0} + e_{i1}$. In the original scheme, the maintenance operations are synchronized and agreed among the peers of a sharing group. When a group splits, all the peers of this group split at the same time and use the same agreed random value to calculate the new shares. In the scheme we propose, peers take decisions lo-

cally without byzantine agreements, and then all the peers of a given group must be able to split at slightly different moments yet generate the same new shares.

To solve this problem, all the possible shares are predefined in a *sharing tree* at the initialization of the system; then, during the life of the network, peers do not know the whole sharing tree but the sum of their knowledge implicitly defines this whole sharing tree. We thus explain how to create storable sharing trees and we describe the split protocol. In Section I, we describe the refresh operation which alters these sharing trees to enforce the confidentiality of shares.

4.1. Initial Sharing Trees

The global sharing tree is a binary tree which root is e, the network secret exponent shared among peers, and which contains all the possible shares. Each node is labeled by the share identifier represented. This tree is recursively constructed at the initialization of the system such that for every share e_i, $e_{i0} = RNG_{h(e_i)}$ and $e_{i1} = e_i - RNG_{h(e_i)}$, where $RNG_{h(e_i)}$ denotes the integer randomly generated from the seed $h(e_i)$. This construction complies with the sharing scheme previously presented. The subtree of root e_i can be generated knowing only e_i and can thus be easily stored and transferred. Moreover, since h is a one-way hash function, it is not possible to calculate the parent of a given node. Since share identifiers have a bounded length, this tree is finite and can be represented as in Figure 2. Leafs are labeled with the longest possible share identifiers. Throughout this paper, we use the terms *node* to refer to nodes of this tree and *peer* to denote nodes of the P2P overlay.

No peer knows this global sharing tree. Instead, every peer belonging to the sharing group $\check{e_i}$ only know the subtree which root is e_i. This subtree is denoted the sharing tree of e_i and contains all the shares e_{i*}. Knowing only their local share e_i, peers can calculate any needed subshare. At every moment, each sharing group knows a different subtree and the sum of the roots of all these subtrees is e. Sharing trees are illustrated in Figure 2.

4.2. Split Protocol

Each peer monitors other members of its sharing group $\check{e_i}$ as well as newcomers in order to detect when its group becomes composed of more than g_{max} members. More precisely, since each created group must itself be composed of g_{min} members, the split is only done if $\check{e_{i0}}$ and $\check{e_{i1}}$ would both be composed of more than g_{min} peers. If a peer decides to split, it moves from the group $\check{e_i}$ to $\check{e_{i0}}$ or $\check{e_{i1}}$, depending on its peer identifier. Then, it generates the convenient sharing tree e_{i0} or e_{i1} using its current sharing tree.

For instance, a peer which binary identifier starts with 00 and which belongs to the group $\check{e_0}$ knows the sharing tree of

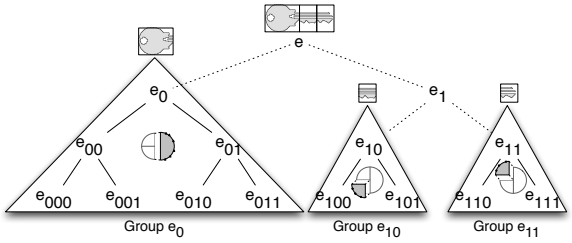

Figure 2. Sharing tree with share identifiers on 3 bits. There are three sharing groups $\check{e_0}$, $\check{e_{10}}$ and $\check{e_{11}}$ knowing their respective sharing trees. The sum of the roots $e_0 + e_{10} + e_{11}$ is e and no peer knows e nor e_1.

e_0. When this peer decides to split, it goes into the group $\check{e_{00}}$ and chooses the left subtree as its sharing tree: the root of this subtree is e_{00}. Knowing the current sharing tree, every peer can split on its own yet every peer of a sharing group make coherent split operations.

However, as in the original scheme, peers of $\check{e_i}$ know both e_{i0} and e_{i1}; even worse, peers of $\check{e_i}$ know every share e_{i*} derived from e_i. The refresh operation alters sharing trees and addresses this precise security issue.

Since this operation is not synchronized among peers of a sharing group, some peers may see later that there is a new neighbor and thus split later. In that case, some peers are in $\check{e_i}$, some others in $\check{e_{i0}}$ and others in $\check{e_{i1}}$. According to the certification mechanism used on top of this sharing, each of these peers may be queried for the partial signature with e_i. If a peer in $\check{e_i}$ is queried, it returns this partial signature. If a peer in $\check{e_{i0}}$ is queried, the algorithm states that the peer must recursively query itself for the signature with e_{i0} and must query another peer for e_{i1}. This other peer is either in $\check{e_{i1}}$ or in $\check{e_i}$ and, in both cases, knows e_{i1} and can proceed with the partial signature. This transient inconsistency is thus compatible with the certification process.

5. Refresh Operation

When a group $\check{e_i}$ splits into two groups $\check{e_{i0}}$ and $\check{e_{i1}}$, each peer knows the share of the sibling group, since it knows the sharing tree of e_i. Moreover, an attacker knowing $e_i = e_{i0} + e_{i1}$ can still use this share on behalf of the two new groups. Both $\check{e_{i0}}$ and $\check{e_{i1}}$ must execute a refresh operation to make old versions of these shares unusable.

To execute a refresh operation, the group $\check{e_{i0}}$ randomly chooses another group $\check{e_x}$. Then, $\check{e_{i0}}$ and $\check{e_x}$ exchange some random value Δ which is added to e_{i0} and subtracted from e_x. After the refresh operation, the new versions of e_{i0} and e_x are only known in their respective group and thus a peer

of $\check{e_{i1}}$ does not know the new version of e_{i0}. Moreover, e_{i0} has changed and thus, $e_{i0} + e_{i1}$ has changed too: the old e_i is out-of-sync with the global sharing and cannot be used anymore to obtain distributed signatures with e.

The refresh operation must tolerate inconsistent groups, where some peers have split and some others have not, since maintenance operations are not synchronized and can be done concurrently. Consequently, the refresh operation cannot rely on the current state of sharing groups, which may be different among peers. This operation is thus rather based on alterations of sharing trees which are always coherent among peers (if a peer has not split yet, its sharing tree contains the sharing trees of peers having split). First, we present how sharing trees are altered to materialize a refresh. Then, we explain the refresh protocol. Finally, we describe the usage of this operation.

5.1. Altered Sharing Trees

As we saw in the preceding section, share identifiers have a bounded length and thus sharing trees are finite. A refresh consists in adding a random value Δ to a leaf and subtracting it from another leaf. When the value of a leaf changes, all the values on the path from this leaf to the root of the sharing tree are updated to maintain the property that for all nodes e_i, $e_i = e_{i0} + e_{i1}$. If a peer knows the sharing tree of e_0 and if Δ is added to the leaf e_{001}, then Δ is also added to e_{00} and e_0, as shown on Figure 3.

Finally, old values must be discarded to render old shares useless, else an attacker could progressively obtain shares at different moments and combine them to attack the system. Imagine that at some time, attackers know the share e_1. Then, after the group e_1 has split, attackers obtain every share e_{0*}. If initial values were not discarded, attackers would be able to use them to reconstitue the network secret key e; if old values are discarded progressively, attackers obtain values which have been refreshed and cannot recover e. Discarding old values enforces that attackers need to belong to every group at the same time to obtain e.

However, when the value of e_i becomes $e'_i = e_i + \Delta$, e_{i0} is not equal to $RNG_{h(e'_i)}$ and cannot be calculated anymore from e_i which has been discarded. Some nodes of the sharing tree must thus be explicitly defined when a leaf changes. When a leaf is updated, this leaf as well as all its ancestor nodes are updated and explicitly defined. Then, if e_i and e_{i1} are updated and defined to e'_i and e'_{i1}, e_{i0} remains implicit and can be calculated as $e_{i0} = e'_i - e'_{i1}$. Defining the paths from the updated leafs to the root allows to calculate every node of the sharing tree. This scheme is illustrated in Figure 3 where updated nodes in bold are explicitly defined and stored, whereas other nodes are implicit and can be calculated. We show in Section I that storing a sharing tree in our setup needs less than 1 Mbyte.

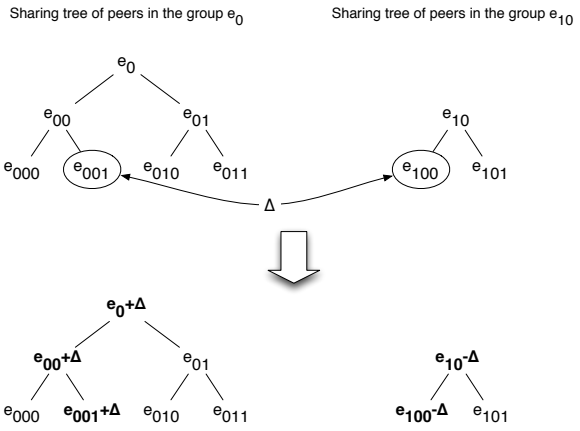

Figure 3. Refresh operation between the leafs e_{001} and e_{100} in sharing groups $\check{e_0}$ and $\check{e_{10}}$.

5.2. Refresh Protocol

To enforce this behavior, the protocol must ensure that if any honest peer does the refresh, then all honest peers do. The initiator peer P creates a refresh request between its local leaf, equal to the longest prefix of its peer identifier, and a randomly chosen remote leaf l using a random Δ value. In Figure 3, the identifier of P is 001101, the local leaf is thus e_{001} and the remote leaf is e_{100}. P sends the refresh request to every other peer of its group and to peers of the remote group. To discover the remote sharing group responsible for the leaf identifier l, in Kademlia which uses the XOR metric, P lookups the resource of identifier $l00\ldots00$ and obtains the x peers sharing the longest prefix with this resource. With $x = g_{min}$, all these peers belong to same sharing group responsible for l and the probability for P to obtain only attackers is minimized. Every peer receiving this refresh request for the first time broadcasts it once to both involved sharing groups. Finally, the hash of the refresh request is stored along the two leafs in order to prevent another identical refresh. If one honest peer receives the refresh request, then all honest peers of both groups apply it, which ensures the consistency of sharing trees.

This operation tolerates transient inconsistencies in the sharing groups, when some peers have split whereas some others have not yet. Nodes having already split know the members which should belong to their sharing group, even if they have not split, and thus forward the refresh to them. To the opposite, peers which have not yet split consider that peers having split are still in their group and thus also forward the refresh operation to them. In all cases, sharing trees of every peer remain consistent.

A refresh operation may also create a transient inconsistency in the sharing where some peers have already added Δ

to their share whereas some others have not yet subtracted Δ: in that case, the sum of the shares does not equal e for an instant. However, in the usual case of an honest refresh, this instant can be made very short: the initiator first lookups every needed IP address and then launches the refresh, in which case this inconsistency is bounded by the few hundred milliseconds for the IP message to attain every other peer. Even in the case of a refresh launched by an attacker, this inconsistency is very brief because the first peer receiving the request forwards it to every other concerned peer. To totally avoid this problem, a peer which has just refreshed can also send the two possible versions of a partial signature and the requester peer chooses the right one by testing the global signature with the public key.

We next discuss the usage of the refresh operation.

5.3. Usage

First, we explain how to prevent attackers from abusing the system. Then, we analyze the possibility for shares to become negative after a refresh operation.

5.3.1. Preventing Abuses

Without further precautions, an attacker may issue many refresh operations to update many leafs and then, sharing trees would need too much space to be stored or transferred. The number of refresh operations must thus be limited to prevent abuses. Although, if attackers know e_i and if the new group e_{i1} refreshes with another group also containing an attacker, attackers know the old version of e_{i1} as well as the random Δ used to refresh and can thus deduce the new version of e_{i1}. The number of refresh operations must thus be sufficient to ensure that at least one of them has been done with a group containing no attacker. We first calculate that throwing 10 refresh operations after a split enforces the confidentiality of shares. We then argue that this limitation can be obtained by providing a single refresh operation to each peer and show how to enforce this limit.

Based on [11], the probability that there is no attacker in a sharing group is $P_{HonestGroup} = (1-k)^g$, where k is the ratio of attackers and g the size of sharing groups. In an ideal system where shares are only known in their respective group, attackers know a given share with probability $1 - P_{HonestGroup}$. In our proposition, an attacker may know a share if it is in its sharing group or if it knows all the refreshed values, *i.e.*, if every group used to refresh contains at least one attacker. Note that this an upper bound since attackers also need to know a parent sharing tree and this required condition is thus not sufficient. With n refresh operations, the probability that at least one refresh is exchanged with a group containing no attacker is $P_{HonestRefresh} = 1 - (1 - P_{HonestGroup})^n$. In our system, the probability that attackers know a share is thus $1 - (P_{HonestGroup} \times P_{HonestRefresh})$. This probability does not depend on the global size of the network. If we consider the ratio between the probabilities that an attacker knows a share in our system and in an ideal system, 20 refresh operations maintain this ratio between 1 and 1.02 for groups composed of 20 to 40 members, as proposed in [11]: 20 refresh operations allow to ensure the confidentiality of shares. Each group thus needs to throw 10 refresh operations after a split since it also receives on average 10 refresh operations in the meantime.

Consider now that every peer are limited to a single refresh operation. We first consider a network where no peer leaves. In that case, every group starts with $g_{min} = 20$ members and exists until it is composed of $g_{max} = 40$ members: 20 new members join the group and if each peer can launch a single refresh, then the group can launch 20 refresh operations. In a real setup, peers join and leave, which changes two points. First, joining members are either new peers or rejoining peers: a rejoining peer may have already done its refresh operation. Then, much more than 20 members join a group during its lifetime, since some other peers leave: this ensures with high probability that some of these peers are new members.

However, a critical case with no possible refresh is attained when a group $e_{\check{i}0}$ goes from 20 to 40 members with only rejoining peers and when these rejoining peers have already issued their refresh operations. This case may only be triggered if this group used to be split in $e_{\check{i}00}$ and $e_{\check{i}01}$ but merged to $e_{\check{i}0}$ and then to $e_{\check{i}}$, which correspond to more than halving the number of peers in this zone of the identifier space. This situation is rare in the lifetime of a P2P network and can be managed if peers economize their refresh operations. When a group splits, members wait a random delay before launching a refresh and, as soon as 10 refresh operations have been sent by other peers of the group, they keep their refresh possibilities for later use. There is then high probability that rejoining peers still have their refresh possibilities.

We propose thus to limit each peer to a single refresh where the remote leaf identifier and the random value Δ are generated using a specific random number generator. This RNG is seeded by the unique signature of some predefined data with the secret key of the peer throwing the refresh. The refresh request contains then this signature and each peer can check the proposed values.

5.3.2. Negative Shares

In the refresh operation, a random value Δ is added to a local leaf and subtracted from a remote leaf. Since there are no more agreements nor synchronization, peers receiving the refresh request cannot ensure that Δ is lower than their

share. If Δ is higher than this share, this share becomes negative. We analyze here the consequences of negative shares. We also note that this problem would also exist if Δ was subtracted from the local share since, even if an honest peer could check whether Δ is lower than its share, an attacker would still be able to throw this refresh request to the remote group. Without a byzantine agreement, this remote group cannot check whether the refresh is legal or not and every refresh request must thus be accepted.

When some shares are negative, signing some data d with e is still $d^e[m] = d^{\sum_{i=1}^{s} e_i}[m] = \left(\prod_{i=1}^{s} d^{e_i}[m]\right)[m]$, but we must ensure that all the $d^{e_i}[m]$ exist. If e_i is negative, $d^{e_i}[m]$ exists if and only if d is invertible modulo m.

d is invertible modulo m if and only if $gcd(d, m) = 1$. Given the RSA key generation procedure, m is the RSA modulus and is equal to $p \times q$ with p and q prime numbers. Then, $gcd(d, m) = 1 \Leftrightarrow (gcd(d, p) = 1 \wedge gcd(d, q) = 1)$. We thus know that d is invertible modulo m if d is not a multiple of p nor of q. We show in the following that the structure of d used for RSA signature implies that at most 2 over the 2^{160} possible values for d are not invertible.

We use here the RSASSA-PKCS1-v1_5 [15] signature algorithm. This algorithm is the deterministic RSA signature algorithm officially proposed by the RSA Laboratories. It should be noted that, to the contrary to RSA deterministic encryption schemes which are for instance vulnerable to chosen plaintext attacks, no attack is known against this deterministic signature scheme. The use of a probabilistic scheme for RSA signature is thus not mandatory. In RSASSA-PKCS1-v1_5, the RSA exponentiation is done on an encoded version of the data to sign and we calculate here the probability for this encoded version d to be a multiple of p or q.

According to the specification, d has the length of the modulus m and is of the form `0x00||0x01||PS||0x00||T`. PS is an expendable octet string containing `0xff` values, used to make d the suited length. T is the DER encoding of the `DigestInfo` value, containing the type of hash used and the hash value to sign.

We consider a RSA key of 1024 bits and we use SHA-1 to hash data. d is thus 1024 bit long, T is 280 bit long (120 bits for coding that SHA-1 is used and 160 bits for the SHA-1 hash) and PS is thus expanded to 720 bits. d is of the form `0x00||0x01||0xff||...||0xff||0x00||T`, with `0xff` repeated 90 times (720 bits are 90 bytes). Since the RSA key is 1024 bit long, the key generation procedure generates both p and q 512 bits long, *i.e.*, the first of the 512 bits of p and q equal 1. We show by contradiction that there is at most one encoded message which is a multiple of p.

Imagine that d is a multiple of p, d is of the form `0x00||0x01||0xff||...||0xff||...`, where there are 720 bits of `0xff`. We thus know that d is greater or equal to $\sum_{i=511}^{1008} 2^i$, *i.e.*, to $2^{511}\frac{1-2^{498}}{1-2}$, and that p is greater or equal to 2^{511}. This yields that $d + p$ is greater or equal to $2^{511}\frac{1-2^{498}}{1-2} + 2^{511}$, *i.e.*, to 2^{1009}. Moreover, encoded messages start with `0x00||0x01` and thus, valid encoded messages are strictly lower than 2^{1009}.

Imagine now that there exists more than one multiple of p which is a possible encoded version of some data and that we choose d as the smallest of these multiples. In that case, $d+p$ is already greater than all possible encoded messages. There is at most one encoded message which is a multiple of p. The same property holds for q. Since we use SHA-1 to hash data, there are 2^{160} possible encoded messages, 2 of them being non invertible, which is negligible. The same reasoning and figures apply also to 2048 bit RSA keys.

However, an attacker may exploit the system to find a non invertible d, which is a multiple of p, and then deduce p. It should be noted that this attack is not linked to the sharing of the secret key, but to the knowledge of the modulus m, which belongs to the public key: this attack is in fact against the standard RSA scheme. However, in our system, an attacker may distribute the exhaustive search of a non invertible d by asking random certifications to honest peers.

To prevent that, we must ensure that an attacker spends more time asking a random certificate than inverting a random data. The certificate request contains thus itself the inverse of the encoded message d representing the data to sign, in which case the attacker has no interest in asking random certificates. Moreover, we benchmarked that inverting d in our setup is a fast operation, typically 50 times faster than calculating an exponentiation.

6. Merge Operation

When a group $e_{i0}^{\check{}}$ is composed of less than g_{min} members, this group must merge with $e_{i1}^{\check{}}$ into one group knowing e_i, such that $e_i = e_{i0}+e_{i1}$ (the case of $e_{i1}^{\check{}}$ is symmetric). During a merge operation, peers need to download the sharing tree of their sibling group and we first explain how to do that efficiently. We then present the merge protocol.

6.1. Downloading Sharing Trees

The download operation of a sharing tree must be efficient yet protect the downloader from obtaining fake data from attackers. For this purpose, we first define a hashing scheme w inspired by Merkle Hash Trees to attach an integrity check to each node of a sharing tree. We define that for each defined internal node e_i, $w(e_i) = h(e_i, w(e_{i0}), w(e_{i1}))$ and for each leaf or implicitly defined node e_i, $w(e_i) = h(e_i)$, h being a standard hash function such as SHA-1. The knowledge of $w(e_i)$ allows to check the integrity of the subtree of e_i. The downloader first obtains the integrity check of the wanted sharing tree from different source to verify it. Then, the downloader obtains a

valid tree, even if it has become obsolete by the end of the download. Finally, this sharing tree is updated.

First, the downloader asks all peers of the sibling group $\check{e_{i1}}$ for the value $w(e_{i1})$, which allows to check the integrity of this sharing tree. Since we suppose that most of the peers are honest, the joining peer can use the value returned by more than half of the peers as the true $w(e_{i1})$: this $w(e_{i1})$ allows to uniquely identify the current sharing tree of the group. It is possible that honest peers return different values, if some operation is in progress: in that case, if there is no value returned by more than half of the peers, the downloader retries and eventually obtains a verified $w(e_{i1})$.

Then, the downloader asks different peers for different parts of the sharing tree characterized by $w(e_{i1})$. The asked peer can either know the sharing tree $w(e_{i1})$, in which case it sends the requested part augmented with the hash values w of each node of the tree; the requested peer can also currently know an older version of this tree, if it has not yet received a refresh, in which case it sends an error; finally, the requested peer can know a newer version of this tree, in which case it has cached the last refresh operations and can undo them in order to send the requested version. Using $w(e_{i1})$, the downloader can check the integrity of the tree from the top and possibly re-ask some parts.

Finally, the downloader updates the leafs of its sharing tree in order to know the very last version of this tree. It sends the hash and identifiers of all known defined leafs to every other peer which respond with the updated leafs and newly defined ones. The downloader applies changes proposed by more than half of the other peers. At the end, the downloader knows an up-to-date sharing tree. This download operation is evaluated in Section I.

6.2. Merge Protocol

Each peer locally decides to merge in two cases. First one is when its own sharing group is composed of less than g_{min} members; second one is when its sibling group is composed of less than g_{min} members. In this second case, the sibling group $\check{e_{i1}}$ needs to merge with $\check{e_{i0}}$, and so $\check{e_{i0}}$ must also decide to merge.

Then, once a peer has decided to merge, it periodically asks peers of the sibling group to obtain their sharing tree. Asked peers finally reply with their sharing tree if they have also decided to merge. Eventually, every peer merge and obtain the same sharing tree which root is the share $e_i = e_{i0} + e_{i1}$.

It is also possible that a group $\check{e_{i0}}$ composed of less than g_{min} members must merge and that the sibling group $\check{e_{i1}}$ does not exist but has split into $\check{e_{i10}}$ and $\check{e_{i11}}$. In that case, $\check{e_{i0}}$ must merge but if $\check{e_{i10}}$ and $\check{e_{i11}}$ only monitored their sibling group, they would not accept to send their share to $\check{e_{i0}}$. Each peer must thus monitor the number of peers of each group it could merge with. In that case, $\check{e_{i0}}$ monitors $\check{e_{i1}}$, even if $\check{e_{i1}}$ does not exist: this is not a problem, since children of $\check{e_{i1}}$ are each composed of more than g_{min} members, and so $\check{e_{i0}}$ detects enough peers in $\check{e_{i1}}$. In the other way, $\check{e_{i10}}$ and $\check{e_{i11}}$ monitor each other and siblings of their ancestor groups. They are thus able to detect that $\check{e_{i0}}$ needs to merge, merge both of them into $\check{e_{i1}}$ and then merge with $\check{e_{i0}}$ into $\check{e_i}$. This possibility must be handled but since its probability is low (peer identifiers are randomly chosen and so uniformly distributed), it does not affect the globally enforced ratio of peers needed to agree to obtain a valid certificate. The number of sharing groups to monitor is a logarithm of the number of peers and is only a few tens for million-peers networks. Moreover, since we use the Kademlia overlay, monitoring the sharing groups is equivalent to managing the *k-buckets* and is thus a free operation as long as g_{min} is lower or equal to the size of *k-buckets* (in the experimental case of [11], g_{min} is 20, which is is the usual size of *k-buckets*).

The merge operation may exhibit the same transient inconsistencies in the sharing groups as the split operation; these inconsistencies are handled in the same way.

7. Join Operation

When a peer joins the network, it joins the sharing group including its peer identifier and needs to obtain the sharing tree of this group. First, the peer needs to obtain its group identifier by asking its neighborhood. Then, it needs to obtain the root of its sharing tree by asking every other peer of its sharing group. From that time, the new peer can participate to distributed certification process. Finally, the remaining of the tree can be downloaded in the background, as previously explained.

When a peer leaves the network, it stores its current sharing tree in order to be able to only download updates when it reconnects.

8. Evaluation

In this section, we evaluate the security and efficiency of our proposition. We simulated our system using *PeerSim*[1] in function of the size of the network. Sharing groups are composed of $g_{min} = 20$ to $g_{max} = 40$ members, which are the values proposed in the original paper allowing to tolerate 20% of attackers in a $10,000$ peer network, after which attackers may be able to forge certificates. The network secret key is 1024 bit long and all the random values used to split or refresh are also 1024 bit long.

We first evaluate the size of generated shares and show that shares remain nearly as long as the secret key when the

[1] http://peersim.sourceforge.net/

network grows. We then analyze the size of sharing trees and show that they can be easily stored and transferred.

8.1. Size of Shares

Initially, all shares are generated from e. Each share e_{i0} is randomly generated on 1024 bits (between 0 and 2^{1023}) and each share e_{i1} is obtained with $e_{i1} = e_i - e_{i0}$. Then, the refresh operations modify these shares by adding or subtracting random values on 1024 bits (between 0 and 2^{1023}). Figure 4 represents the size of shares in function of the number of peers. In Figure 4, we experimentally evaluate the length of resulting shares and we see that (1) the average length is independent from the network size and (2) the minimum size only slightly decreases when the network grows. Even in a large network, an attacker cannot guess a share.

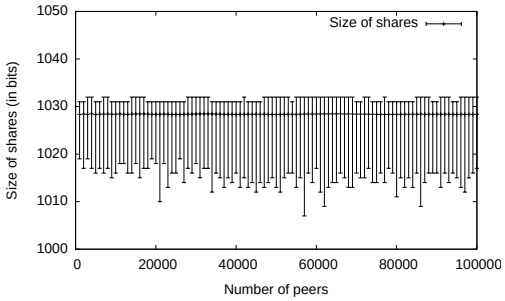

Figure 4. Size of shares in function of the number of peers. For each number of peers, the lowest tic indicates the smallest share, middle tic the average size of shares and highest tic the largest share.

8.2. Size of Sharing Trees

Each peer has to store a sharing tree with some nodes explicitly defined and we evaluate here the size of these sharing trees. We consider that the share identifiers are at most 40 bit long, *i.e.*, the height of sharing trees is bounded by 40 (at most $2^{40} \approx 10^{12}$ sharing groups, hence 20×10^{12} peers). The shares are 1024 bit long.

We then make the assumption that peers are online 10% of the time. On the one hand, this assumption may seem optimistic for file sharing systems where most users disconnect after retrieving one file; on the other hand, this assumption is pessimistic for VoIP services where users stay online most of the time to be reachable. If users are online 10% of the time, then the size of the overlay is ten times smaller than the global number of users. Then, since each user, online or offline, may have issued its refresh operation, sharing trees must contain the refresh of ten times more peers than the number of online peers. If groups are composed of at most 40 members, 800 leafs are updated in each sharing tree, since each refresh affects two groups and since there are ten times more refresh than online peers. This yields an upper bound of $800 \times 40 \times 1024$ bits for the size of sharing tress, which is approximately 4 Mbytes.

However, in practice, many internal nodes of sharing trees are common ancestors of several defined leafs, in which case defining a leaf involves the definition of less than 40 internal nodes. Using simulations presented in Figure 5, we see that sharing trees are less than 1 Mbyte on average. Sharing trees can thus be easily downloaded.

We can also note that the size of sharing trees is nearly independent from the size of the network. In fact, the number of defined leafs is constant and only depends on the size of sharing groups; only the height of the sharing trees depends on the network size and even decreases slowly with the network growth. The chosen representation is scalable. Since the peers are uniformly distributed in the groups, every group tend to split at the same moment which explains the oscillations of the curve.

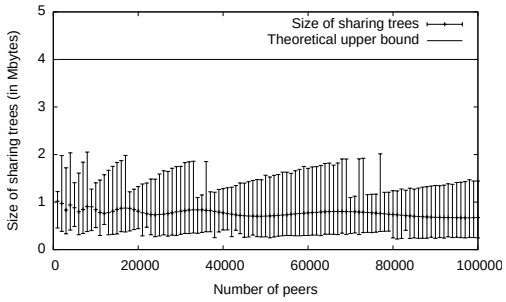

Figure 5. Size of sharing trees.

Finally, to update a sharing tree after downloading it, a peer needs to send identifiers and hash of its defined leafs to every peer of its group. If there are 800 leafs which identifiers are on 40 bits and hash on 160, the downloader only needs to send $(40 + 160) \times 800$ bits, which is 20 kbytes, to every other peer of the group (typically 20 to 40 peers). This process could be optimized by sending the hash of several leafs at once to quickly detect which leafs are out-of-sync.

9. Conclusion and Future Work

We have proposed in this paper lightweight maintenance operations for a previously proposed distributed certification system for structured P2P networks. This previously proposed system provided distributed security primitives but its maintenance did not fulfill the P2P requirements because of the byzantine agreements used.

In this paper, we have proposed scalable and easy to implement maintenance operations providing the same sharing

of the network secret key and thus the same distributed certification functionalities as in [11]. We have then shown the security of our new scheme.

Using this simple maintenance, we have been able to run a preliminary implementation on a few hundred machines on PlanetLab. First results show that the maintenance operations can be handled and that, on a larger network, certification operations should return in a few tens of seconds. We are still working on this implementation and we aim at proposing a VoIP application where users could securely register intelligible names such as "John Smith".

In the current system, the sharing scheme is only managed by online peers. However, attackers would probably stay online and thus the ratio of attackers is higher in the sharing scheme than in the overall peers using the network. An interesting future work would be for online peers to run a mobile agent on the behalf of their offline friends. This would increase the ratio of honest peers in the scheme.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[2] L. M. Aiello, M. Milanesio, G. Ruffo, and R. Schifanella. Tempering kademlia with a robust identity based system. In *Proceedings of the 8th International Conference on Peer-to-Peer Computing (P2P)*, pages 30–39. IEEE Computer Society, 2008.

[3] D. Boneh and M. Franklin. Efficient generation of shared RSA keys. In *Proceedings of the 17th Annual International Cryptology Conference (CRYPTO)*, volume 1294 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[4] F. Cao, D. A. Bryan, and B. Lowekamp. Providing secure services in peer-to-peer communications networks with central security servers. In *AICT/ICIW*, page 105. IEEE Computer Society, 2006.

[5] M. Castro, P. Druschel, A. J. Ganesh, A. I. T. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI)*, pages 299–314. ACM Press, 2002.

[6] Clip2. The gnutella protocol specification v0.4. `http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf`, 2000.

[7] Y. Desmedt. Some recent research aspects of threshold cryptography. In *Proceedings of the 1st International Workshop on Information Security (ISW)*, volume 1396 of *Lecture Notes in Computer Science*, pages 158–173. Springer-Verlag, 1997.

[8] Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung. Optimal-resilience proactive public-key cryptosystems. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 384–393. IEEE Computer Society, 1997.

[9] J. Kong, P. Zerfos, H. Luo, S. Lu, and L. Zhang. Providing robust and ubiquitous security support for mobile ad hoc networks. In *Proceedings of the 9th IEEE International Conference on Network Protocols (ICNP)*, pages 251–260. IEEE Computer Society, 2001.

[10] F. Lesueur, L. Mé, and V. Viet Triem Tong. Detecting and excluding misbehaving nodes in a P2P network. In *Proceedings of the 8th International Conference on Innovative Internet Community Systems (I2CS)*, 2008.

[11] F. Lesueur, L. Mé, and V. Viet Triem Tong. A Distributed certification system for structured P2P networks. In *Proceedings of the 2nd International Conference on Autonomous Infrastructure, Management and Security (AIMS)*, volume 5127 of *Lecture Notes in Computer Science*, pages 40–52. Springer, 2008.

[12] F. Lesueur, L. Mé, and V. Viet Triem Tong. A sybil-resistant admission control coupling SybilGuard with distributed certification. In *Proceedings of the 4th International Workshop on Collaborative Peer-to-Peer Systems (COPS)*. IEEE Computer Society, 2008.

[13] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2002.

[14] A. I. T. Rowstron and P. Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer-Verlag, 2001.

[15] RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard*. RSA Data Security, Inc., June 2002.

[16] N. Saxena, G. Tsudik, and J. H. Yi. Experimenting with admission control in P2P. In *Proceedings of the International Workshop on Advanced Developments in System and Software Security (WADIS)*, 2003.

[17] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

[18] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 261–269. Springer, 2002.

[19] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, Computer Communication Review, pages 149–160. ACM Press, 2001.

[20] D. S. Wallach. A survey of peer-to-peer security issues. In *Proceedings of the International Symposium on Software Security (ISSS)*, volume 2609 of *Lecture Notes in Computer Science*, pages 42–57. Springer-Verlag, 2002.

[21] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: defending against sybil attacks via social networks. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, pages 267–278. ACM Press, 2006.