Verifying Security Property of Peer-to-Peer Systems Using CSP

Tien Tuan Anh Dinh and Mark Ryan

School of Computer Science,
University of Birmingham, Birmingham
United Kingdom, B15 2TT
{ttd,mdr}@cs.bham.ac.uk

Abstract. Due to their nature, Peer-to-Peer (P2P) systems are subject to a wide range of security issues. In this paper, we focus on a specific security property, called the *root authenticity* (or RA) property of the so-called structured P2P overlays. We propose a P2P architecture that uses Trusted Computing as the security mechanism. We formalize that system using a process algebra (CSP), then verify that it indeed meets the RA property.

Keywords: Peer-to-Peer, Trusted Computing, formal verification, CSP.

1 Introduction

Peer-to-Peer (P2P) overlays are *heterogeneous* systems that consists of *autonomous* peers (or nodes), in which most traffic is among peers. The most well-known P2P applications, file-sharings [1,2], are built on top of *unstructured overlays*. In these overlays, peers form a random topology, i.e. a peer can connect to any other peer. Searching is done by broadcasting the query to neighboring nodes (which is not scalable). Unstructured overlays support approximate search, i.e. searching for items *close* to the search key k. They are suitable for dynamic environments (where peers leave and join frequently).

In this paper, we focus on *structured P2P overlays*, in which peers form rigid topologies, i.e. a node only connects to a certain set of neighbors. Examples are Chord [3], Pastry [4], etc. It is necessary to update the topology when nodes join or leave the network, which is an expensive operation. Exact-match searching is done deterministically and is very efficient. In many overlays, it takes $O(logN)$ hops, where N is the number of peers. On the one hand, structured overlays are restricted to relatively stable environments where joining and leaving are infrequent. On the other hand, they are much more scalable than the unstructured counterpart, and therefore can support applications with very large numbers of participants. Existing applications of structured overlays range from global storage systems [5], P2P-based communications [6], application-level multicast [7], P2P-based marketplaces [8] to botnets [9].

Due to the decentralization nature, P2P systems are subject to a wide range of security attacks. They are normally caused by the lack of an identity management mechanism or of a central authority. In this work, we investigate an

D. Gritzalis, B. Preneel, and M. Theoharidou (Eds.): ESORICS 2010, LNCS 6345, pp. 319–339, 2010.

attack, called the *false-destination* attack, in which the adversary falsely claims that it is the destination of a search key k. We say that a P2P system satisfies the *root authenticity* (or RA) property if it is secure from this attack. In structured overlays, where the data key k is uniquely assigned to a *root node*, there are various reasons behind this attack. For example, in the P2P storage system, the adversary wishes to censor a piece of data identified by the key k. It will (falsely) claim to be the root node of k. As the consequence, all traffic regarding k (queries or deposit of the data) will be forwarded to the adversary, which now has total control of the data it wanted to censor.

Despite the early recognition of security problems in P2P systems, not much progress has been made in addressing them. Most related work propose solutions that are probabilistic. They normally do not scale well and incur great overhead when churn (nodes joining and leaving the network) is frequent. More importantly, there is a serious lack of formal studies of P2P security. For P2P overlays to be used in applications that demand a certain level of security, such formal studies are vital.

Contribution. Our main contributions from this paper are as follows:

1. We discuss security issues of P2P systems, categorizing those issues as inherently belonging to different levels of abstraction. This enables us to clarify how the RA property relates to the various sets of security problems.
2. We propose a P2P architecture aiming to satisfy the RA property. This system makes use of Trusted Computing as the underlying security mechanism.
3. We explain our formalization in CSP of the proposed system, and describe our approach to verifying that the system does indeed satisfy the RA property.

The detailed CSP model can be found in the Appendix. For the complete proof, see [10].

Related Work. Sit et al. [11] present a taxonomy of security attacks on structured P2P overlays. Castro et al [12] propose a secure routing mechanism based on a number of components: secure ID assignment, secure neighbor maintenance and secure message forwarding. Their solution is probabilistic and incurs large overhead.

Regarding the false destination attack, the closest to our work is that by Wang et al. [13]. Their solution assumes the existence of a certificate authority (or CA) that when a node joins issues certificates to $2.l + 1$ neighbors, where l is the number of closest neighbor to a peer in one direction (i.e. its *leafset*). This approach is probabilistic and its effectiveness increases with l. Ganesh et al. [14] propose another solution that assumes peers regular publishing their ID certificates. For verification, it relies on name-space density estimation, which is probabilistic.

Work on formalizing and verifying P2P systems are limited in numbers. Borgstrom et al. [15] model Distributed K-ary Search (DKS), a structured overlay in Calculus of Communicating Systems (CCS). They verify that the routing

protocol in DKS, under the static case (when no node joining or leaving), is correct. Bakhshi et al. [16] model Chord in π-calculus and verify that the stabilization protocol in Chord is correct.

Paper's Organization. In the next section, we discuss the range of security issues in P2P systems, focusing on the RA property. We show that RA is implied by another property, called the *neighbor authenticity* (NA) property. In Section 3, we introduce our P2P architecture built on top of Trusted Computing. We show details of the routing and churn protocols that make use of the Trusted Platform Modules (TPMs) running at each peer. In Section 4, we describe the CSP model of the above architecture and define (in CSP) the NA property. We then explain our approach for verifying that the current model satisfies the NA property in Section 5. Finally, we conclude and discuss future work in Section 6.

2 Security Issues in P2P Systems

2.1 Overview

Example of a Structured P2P System. There are a number of structured P2P systems, each differs from another in its topology, routing or maintenance protocols. In this paper, we consider Chord, one of the earliest structured P2P systems. Chord is more popular than the other systems, due to its simplicity and efficient routing protocol.

Let $\mathcal{P}$ and $\mathcal{D}$ be the set of peers and data objects. In Chord, members of both sets are *hashed* using a non-collision hash function into the same circular ID space $\mathcal{ID}$. Fig. 1 shows an example in which $\mathcal{ID} = [0, 2^{10})$ and there are 6 peers with IDs of $144, 296, 498, 609, 775$ and 1000. The two data objects are hashed to the value 550 and 744. Each peer in Chord connects to a peer immediate on

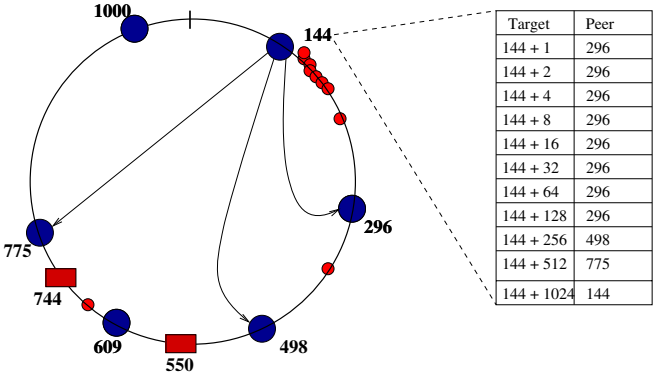


Fig. 1. Example of a Chord overlay. The big circles represent peers, the rectangles represent data objects, and the small circles represent the IDs that are used to construct the finger table of peer 144.

its left (*predecessor*) and on its right (*successor*). In Fig.1, the predecessor and successor of peer 144 are 1000 and 296 respectively.

Let *successor(k)* be the peer immediate on the right of the key k in the ID ring. In Chord, $successor(k)$ is the **destination node** of k, which is responsible for storing the data identified by the key k. For instance, $successor(300) = successor(400) = 498$ and $successor(250) = 296$. The data 744 is stored at peer 775. For efficient routing, each node p in Chord also maintains a *finger* table of size m, where 2^m is the size of $\mathcal{ID}$ and $finger[i] = successor(p + 2^{i-1})$ for $1 \leqslant i \leqslant m$. In the above example, the 4^{th} and 9^{th} finger of peer 144 points to node 296 and 498 respectively.

To search for $successor(k)$, the searching peer forwards its query to the neighbor, which is one of its successor, predecessor or finger pointers and is furthest from it but still on the left of k. The query is then executed at the new node, until the current node is the closest on the left of k. The search then stops and the current node's successor is returned. In Fig. 1, the routing path from node 144 for $successor(744)$ is $144 \rightarrow 498 \rightarrow 609$. Finally, 775 is returned as the destination of k.

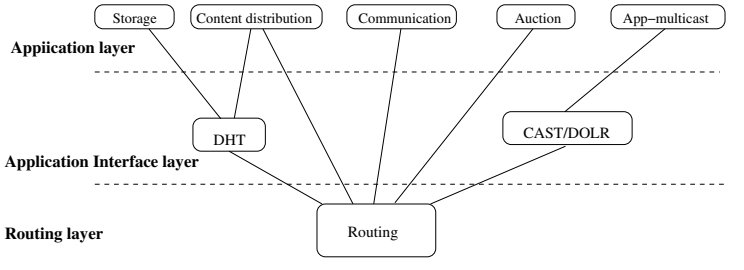


Fig. 2. Different levels of abstraction in P2P systems. Adapted from [17].

P2P Abstraction Layers. A P2P system can be studied at different levels of abstractions, as illustrated in Fig. 2:

+ *Routing layer*: implements the **route(k)** protocol, which, if successful, returns $successor(k)$. In *unstructured* P2P systems, this protocol is implemented by broadcasting the query to all neighbors. In *structured* P2P systems, this is implemented more efficiently by deterministically forwarding the query to neighbors closer to k.
+ *Application interface layer*: implements the **store(k,data)**, which places the tuple (k,data) at a node. In a Distributed Hash Table (DHT) system, for example, this tuple is stored at the destination node of k. In other systems, it is also stored at nodes along the routing path.
+ *Application layer*: consists of application-specific protocols that utilize the lower levels. For instance, P2P communication systems make use of the routing layer, whereas P2P storage systems use the application interface layer to store data in a scalable way.

Security Issues. Given the abstraction in Fig. 2, an adversary can perform attacks against a P2P system at more than one level. In the following, we discuss inherent security concerns at each level. The hierarchy in Fig. 2 suggests that to achieve security at one level, one must at first address the security issues at levels below it.

1. Routing layer: the adversary can corrupt the routing protocol. For example:
   + No routing: queries are dropped. As the consequence, the network is partitioned into parts that can not reach each other.
   + Redirection: queries are forwarded to malicious nodes. They could also be forwarded to innocent nodes, in attempt at a DDoS attack.
   + Impersonate the final node in the routing path
2. Application interface layer: the adversary can corrupt the store(k,data) protocol in the following ways:
   + Dropping the tuple (k,data) that is destined to stored at its node.
   + Tampering with the data.
3. Application layer: the adversary can compromise application-specific properties. For example:
   + Corrupting data: malware, etc.
   + Attacking other mechanisms such as replication, access control, etc.

2.2 Root Authenticity Property of a P2P System

As previously described, the adversary may attempt to impersonate the destination of a search key, called the false-destination attack. For example, in Fig.1, an adversary controlling peer 498 and 775 could convince peer 144 that node 775 is the destination for the data 550 (the correct destination, given the current configuration of the network, is node 609). The RA property implies that such an attack is not possible. In the following, we present a more formal definition of this property.

Let $\mathcal{ID} = [0, 2^m)$ for a reasonable large value of m. For any $x, y, z \in \mathcal{ID}$, denote $inBetween(z, x, y)$ as the predicate that indicates that going clock-wise from x one gets to z before y. More precisely:

$$inBetween(z, x, y) = \big(|z \ominus x| + |y \ominus z| = |y \ominus x| \big)$$

where $\oplus$ and $\ominus$ are addition and subtraction in *modulo* 2^m, and $|x \ominus y|$ is the function returning the clock-wise distance between y and x[1].

Let $\mathcal{P}$ be the set of peers in the network. We can refer to them by their unique IDs. Let V be an honest peer that searches for the destination of a key k. V can perform the following operations:

1. $V.route(k)$: the P2P routing protocol. We will not consider the details of the function *route*. In our analysis, we allow it to be any function returning a value in $\mathcal{P}$.

[1] $|x \ominus y| = t \;\Leftrightarrow\; 0 \leqslant t < 2^m \;\wedge\; y \oplus t = x.$

```
1. L <- V.getPredecessor(D);
2. if (V.neighborVerification(L,D))
      if (inBetween(k,L,D))
         return true;
3. return false;
```

Fig. 3. Details of the `V.destVerification(k,D)` protocol

2. $V.getPredecessor(D)$: V contacts D and asks for its predecessor. Similar to *route*, we allow $V.getPredecessor$ to be any function returning a value in $\mathcal{P}$.
3. $V.neighborVerification(L, R)$: for any $L, R \in \mathcal{P}$, V checks if L is the predecessor of R in the current network. The details of this protocol is the main focus of Section 3.
4. $V.destVerification(k, D)$: for any $k \in \mathcal{ID}$, V checks if D is the destination of k, given the current configuration of the network. The details of this protocol are shown in Fig.3.

Definition 1 (Root Authenticity (RA) Property). *Let $\mathcal{P}^t$ be the set of current nodes in the P2P system, at a given time t. Assume that the system evolves from t to $(t+1)$ as a new peer joins or an existing peer leaves the system. The RA property is defined as:*

$$\forall D, k \in \mathcal{ID}, t\,.\;\; V.destVerification(k, D) \\ \Rightarrow\;\; D \in \mathcal{P}^t \;\wedge\; \left(\forall D' \in \mathcal{P}^t \setminus \{D\}\,.\, |D' \ominus k| > |D \ominus k|\right)$$

Definition 2 (Neighbor Authenticity (NA) Property). *Let $\mathcal{P}^t$ be the set of current nodes in the P2P system, at a given time t. Assume that the system evolves from t to $(t+1)$ as a new peer joins or an existing peer leaves the system. The NA property is defined as:*

$$\forall L, D, t\,.\;\; V.neighborVerification(L, D) \\ \Rightarrow\;\; \{L, D\} \subseteq \mathcal{P}^t \;\wedge\; \left(\forall D' \in \mathcal{P}^t \setminus \{L\}\,.\, |D' \ominus L| \geqslant |D \ominus L|\right)$$

Informally speaking, the RA property requires that for any key k and a peer D at time t, if $destVerification(k, D)$ returns true then D is the closest peer on the right of k at time t. The NA property requires that at time t for any peer L and D in the network, if $neighborVerification(L, D)$ returns true then L is in fact the immediate left neighbor of D at time t. From these definitions, we have the following proposition:

Proposition 1. $NA \;\Rightarrow\; RA$

This theorem means that if the neighbor verification protocol is correct, then the system satisfies the RA property.

Why RA Property. In a system not satisfying the RA property, an adversary A can falsely convince an honest peer that it is the destination of a key k. There are various reasons behind such attacks. Since the data identified by k is stored at the destination of k, the attacker may launch this attack to gain control or censor a particular piece of data. In P2P storage systems, for example, the attacker having the data can modify or removing them from the system. In other applications where controlling more data could imply economic gains (such as P2P-based marketplaces), there are more tangible incentives for A to initiate the attack. In P2P-based communication systems (such as Voice Over IP (VOIP) or instant messaging), the data generally contains the connection details of the communicating hosts. Having control over such data means that A might be able to eavesdrop the communication, or even prevent it from happening. The impact cause by these attacks can be worsened by the adversary launching *Sybil* attacks, in which the adversary has many identities and therefore controlling multiple nodes at different locations in the network.

Recently, there are much research on reputation systems for P2P. One fundamental element of a reputation system is the feedback mechanism, by which a peer can rate the behavior of another. In many cases, this requires a peer being able to verify if another is telling the truth or not. The RA property implies that a peer can check if the result of route(k) is the correct destination, therefore it can confidently give good or bad feedback to the nodes involved in the routing path.

3 A Secure P2P System Using Trusted Computing

3.1 Trusted Computing and Trusted Platform Modules

Trusted Computing is a collection of current and future initiatives to root security in hardware that have been under development since about 2003. It is set to transform the computing security landscape over the next decade. Currently, the most noticeable manifestations are the *Trusted Platform Module* (TPM), Intel's *Trusted eXecution Technology* (TXT) and *Virtualisation Technology* (VT-d).

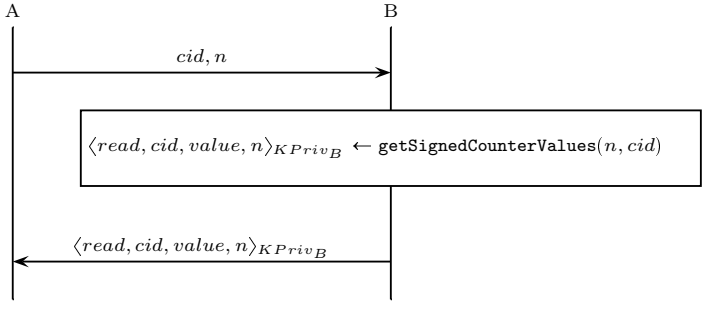
The TPM is a hardware chip currently shipped in high-end laptops, desktops and servers made by all the major manufacturers and destined to be in all devices within the next few years. It is specified by an industry consortium [18], and the specification is now an ISO standard [19]. There are currently 100M TPMs in existence as of 2008, and this figure is expected to be 250M by 2010 [20,21]. The TPM provides hardware-secured storage, secure platform integrity measurement and reporting, and platform authentication. Software that uses this functionality will be rolled out over the coming years. The TPM is commonly used for:

1. Secure storage. User processes can store content that is encrypted by keys only available to the TPM.
2. Platform measurement and reporting. A platform can create reports of its integrity and configuration state that can be relied on by a remote verifier.
3. Platform authentication. A platform can obtain keys by which it can authenticate itself reliably.

In the P2P context, we make use of the following set of the TPM features:

1. A TPM can create a public/private key pair $\langle KPriv, KPub \rangle$, called an *Attestation Identity Key* (AIK). The TPM can identify itself using the AIK, which can be certified using a certificate authority.
2. *Monotonic counters:* each TPM maintains a set of monotonic counters. On a counter with ID *cid*, one can perform the following operations:
   (a) `TPM_ReadCounter`(*cid*): returns the current value of the counter *cid*.
   (b) `TPM_IncrementCounter`(*cid*): increments and returns the new value of the counter *cid*.
3. *Transport sessions:* the TPM commands can be grouped and executed together within a transport session. The session can be *exclusive*, meaning that no other commands can be executed outside of the session when it is active. Furthermore, the session's log can be signed by the TPM.
   (a) `TPM_EstablishTransport`(*exc*): sets up a transport session. The flag *exc* determines if the session is exclusive. A session handle, *sHandle* is returned.
   (b) `TPM_ExecuteTransport`($comm, sHandle$): executes *comm*, which contains a *wrapped* TPMs command, inside the session *sHandle*.
   (c) `TPM_ReleaseTransportSigned`($n, sHandle$): closes the transport session and signs its log containing input, output of all the commands executed in $sHandle$. n is used as the non-replay nonce in the signature.

Example. As an example, suppose the TPM of an agent B has a counter *cid*, whose value is of interest to an agent A. Protocol 1 and Fig. 4 illustrates how A finds out the latest value of *cid*.



Protocol 1: A queries B for the latest value of its counter *cid*

First, A sends *cid* and a freshly generated nonce n to B, which then executes `getSignedCounterValues`(n, cid) on its local TPM. The `getSignedCounterValues`(n, cid) procedure, detailed in Fig.4, first establishes a transport session with the TPM, then executes `TPM_ReadCounter`(*cid*) within that session. Finally, the session is closed and the TPM's signature on the session's log is returned, in which n is used as the non-replay nonce. Notice that it is not possible for B to generate such a signature without having its TPM executing the `TPM_ReadCounter`(*cid*) command inside a transport session.

```
1. sHandle <- TPM_EstablishTransport(true)
2. wc <- wrap command TPM_ReadCounter(cid)
3. TPM_ExecuteTransport(wc, sHandle)
4. sig <- TPM_ReleaseTransportSigned(sHandle,n)
5. return sig
```

Fig. 4. `getSignedCounterValues`(n, cid) is executed by the local TPM at B

3.2 A Secure P2P System Using Trusted Computing

Assumptions. In our proposed P2P system, we assume that all peers have support for the trusted computing infrastructure. In particular, peers are equipped with TPMs. For a large-scale P2P system, one may question if this assumption is reasonable. We wish to stress that firstly, more computers are being shipped with trusted computing support. Secondly, our proposal could work with any other infrastructure that supports the features listed in Section 3.1. It could be in the form of smart-cards or online services. These alternatives could be better choices than TPMs due to their flexibility and wider range of trusted functionalities.

Regarding the *churn* model, we assume that peers leave the network gracefully. This means peers notify their neighbors (or other relevant entities) before exiting.

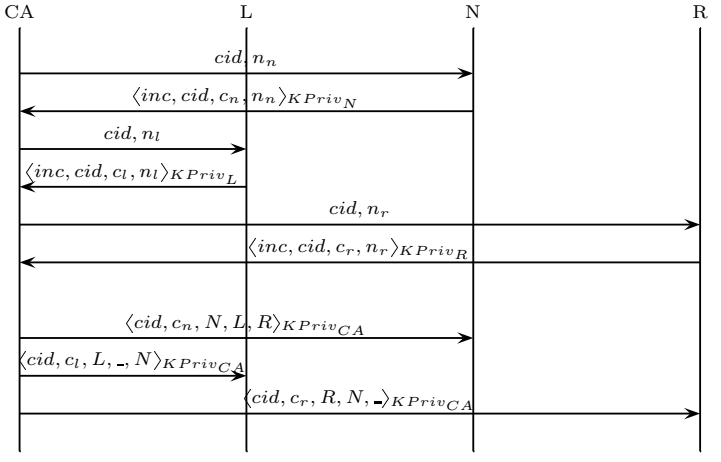
Certificate Authority. In our system, there exists a certificate authority (CA) which is trusted to issue *neighbor certificates* as peers join and leave the network. The CA does not have to run on trusted hardware. It acts as a single point of trust, but as discussed later, is unlikely to be a performance bottleneck.

The CA has an asymmetric key pair $\langle KPriv_{CA}, KPub_{CA}\rangle$. In the joining process, for example, a new peer N contacts CA to be issued a neighbor certificate, which is of the form:

$$\langle cid, v, N, L, R\rangle_{PrivK_{CA}}$$

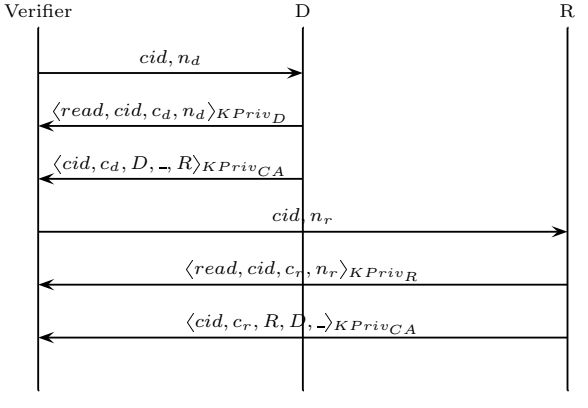
where v is the current value of the counter cid. L, R are the immediate left and right neighbors of N, at the moment the certificate is issued. These nodes also receive new certificates from the CA. It is important that the CA *knows* the correct immediate left and right neighbors of N at any given time in order to issue such certificates. There are several ways for the CA to acquire this knowledge. For simplicity, we assume that it maintains a list of peers currently in the network. When N joins, it checks that N is not already in the list, then issues the relevant certificates and adds N to the list. It performs the opposite when N leaves the network.

Joining/Leaving Protocol. Protocol 2 illustrates the protocol between the CA and other nodes when N joins the network. The CA knows that L, R are the immediate left and right neighbor of N in the current network. First, it asks N, L and R to increment their specific counters (cid). Once received the signatures on the new counter values, the CA adds N to its list of existing peers, then issues new certificates for N, L and R containing information of the new neighbors.



Protocol 2: Peer N joins in between L and R in the network

When a peer E leaves the network, the protocol is similar, except that the CA only issues certificates for E's current neighbors.



Protocol 3: Peer V verifies if R is the current right neighbor of peer D

Routing Protocol. Consider a peer V searching for the destination node of a key k. First, the normal P2P routing protocol (Chord or Pastry routing, for example) is used, which returns a peer D. Before *accepting* D as the destination of k, V performs the verification protocol with D, as illustrated in Protocol 3. The Verifier queries the latest value of D's counter *cid*, namely c_d. It then asks D for the certificate of C_d that contains c_d. By doing this, the Verifier is confident that C_d is the latest certificate issued by the CA to D.

C_d contains information of D's right neighbor, namely R. The Verifier then asks for R's latest certificate, C_r in the same way it did for D. The verification returns true if C_d and C_r match, meaning that in C_d, R is the right neighbor of D and in C_r, D is the left neighbor of R.

The reason for requiring certificates from both D and R is to avoid the following scenario. D is an adversary, it executed the joining protocol properly and has already left the network (gracefully). However, the routing protocol returns D, and since it is still online, D returns its out-of-date certificate, which would be accepted by the Verifier. In other words, D is accepted as the destination of k. This violates the RA property, which requires the destination node to be a node currently in the network.

Discussion. It can be seen from the description of the routing and joining protocols that the CA is a relatively *off-line* entity, since its only involvement is during churn events. The CA is not consulted during the routing process. In a typical P2P system, the rate of query-routing is considerably more frequent than the rate of churn. We therefore argue that the CA is unlikely to be a performance bottleneck.

More specifically, let M be the number of nodes in the network. Churn events can be modeled by a Poisson distribution. This means that a peer's *session time* (duration during which the peer stays in the network) follows an exponential distribution. Let c be the *churn rate*, so that the session times are exponentially distributed with the expected value of $\frac{1}{c}$. It then follows that the expected number of churn events that the CA has to deal with per time unit is $c.M$. For a large (but relatively stable) network, i.e. M is in order of millions and the average session time is in the order of days, $c.M$ is small enough so that the CA would not become a bottleneck.

In our current design, the CA maintains a list of peers currently in the network, which could be a concern. A typical computer nowadays can deal with M in the size of millions. For scalability, however, it will be better to relieve the CA from maintaining such a list. Instead, during the joining or leaving process, the CA also asks for the certificates of the joining or leaving peer as well as of its immediate neighbors. If the certificates match, the CA then issues new certificates as described early. We conjecture that if all the certificates before a churn event were issued correctly, then so are the new ones after the event is completed. We plan to investigate this system in future work.

Finally, the current churn model is quite strict, as we consider peers leaving gracefully, i.e. they notify the CA before leaving. We could make it more realistic by taking the *fail-stop* and *Byzantine* failure models into account. To deal with these failures, a *time-out* mechanism is needed that indicates when a certificate will expire. More specifically, peers need to contact the CA regularly to have their certificates renewed, or else they will be considered having left the network. This would imply more overhead for the CA, as it needs to issues more certificates and keeps track of which peers have left the network. Detailed investigation of the *time-out* mechanism is left for future work.

4 Formal Model in CSP

A brief introduction to CSP syntax and its semantic can be found in the Appendix. CSP has three denotational semantic models: *traces*, *stable failures* and *failures/divergences*. In this work, we only use the traces model, especially the *refinement* relation on traces. In particular, let $traces(P)$ and $traces(Q)$ be set of traces of the process P and Q, then Q is said to refine P, written as $P \sqsubseteq_T Q$ if:

$$traces(Q) \subseteq traces(P)$$

4.1 The System Model in CSP

The model consists of several agents, as shown in Fig.5

1. *Nonce Manager*: supplies fresh and unique nonces for other agents. These values are used during joining, leaving and verification to avoid replaying of old counter values. They are communicated by the NonceManager process to others via the *send* channel.
2. *TPM*: models the trusted hardware used in the system. Each TPM has a counter *cid* for the P2P operations. The counter ID is known to all peers. In addition, each TPM is identified by an unique ID, which can be used as the peer ID. Balfe et al. [22] propose a mechanism for enforcing *stable* IDs based on the Direct Anonymous Attestation (DAA) protocol. In our context, however, we could just assign the ID to be the public part of an AIK. During P2P operations, such ID shall be unique, even though more than one AIKs can be generated for a TPM. It is because the counter *cid* is unique in each TPM. If multiple IDs are used by a TPM, then updating the counter value of one ID will effectively invalidate the states of the other IDs.

   The CSP process representing a TPM receives nonces on the channel *receive*. It then sends back a signature on the latest counter value, after a

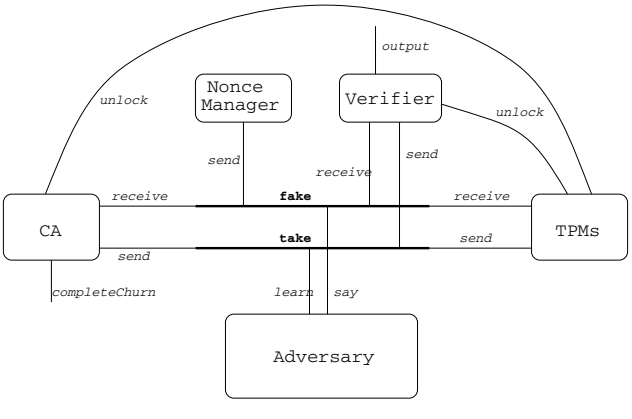


Fig. 5. Channels used by the processes

read or an increment operation. The signature is of the form of the event $SqR.\langle n, i, c\rangle$ (after a read operation) or $SqI.\langle n, i, c\rangle$ (after an increment operation) where n, i, c are the received nonce, the TPM's ID and the counter value.

3. *Certificate Authority*: issues certificates during churn events, as explained in the previous section. The CA process uses the *send* and *receive* channels for sending nonces and receiving other messages. Once it has issued all relevant certificates for a churn event, it outputs on the channel *completeChurn* to signal that the churn event has completed. For example, the event $completeChurn.Churn.\langle join, i\rangle$ where i is a peer indicates that i has successfully joined the network.
4. *Verifier*: picks a random peer D and asks it to returns its immediate right neighbor R. The Verifier performs the verification and then outputs whether it *accepts* that R is the immediate right neighbor of D. This basically models the `neighborVerification` protocol described in section 2. It follows from Proposition 1 that if the Verifier cannot be *fooled* then the RA property is met.

   The Verifier process sends nonces and receives other messages on the *send* and *receive* channel respectively. Events on the channel *output* indicates that the Verifier accepts that one peer is the immediate right neighbor of another. For instance, $output.L.R$ means it is convinced that R is the immediate right neighbor of L in the current network.
5. *Adversary*: models the attacker trying to break the RA property of the system. We give the adversary total control of all the peers in the network, i.e. it controls all the TPMs, even though it cannot fake signatures generated by the TPM.

   The Adversary process uses the *learn* and *say* channel to eavesdrop and send messages from and to the other agents. We model the Adversary being able to remember all messages it has seen, i.e. having infinite memory, so that it could replay old messages. It can be seen from Fig.5 that all the *send* and *receive* channels are renamed to *take* and $fake$, which are in turned mapped to the *learn* and *say* channels (by a many-to-one mapping function). This renaming scheme introduces non-determinism and as an effect increases the adversary's power.

The detailed CSP models for these agents can be found in the Appendix.

4.2 Checking the RA Property

As Proposition 1 indicates, the RA property is implied by the correctness of the neighbor verification protocol `neighborVerification`, which can be formalized by the following process:

$$
\begin{aligned}
Spec(ps, pn) = \quad & \Box_{i \in pn}\ completeChurn.Churn.\langle join, i\rangle \rightarrow Spec(ps \cup \{i\}, pn \backslash \{i\}) \\
\Box\ & \Box_{i \in ps}\ completeChurn.Churn.\langle leave, i\rangle \rightarrow Spec(ps \backslash \{i\}, pn \cup \{i\}) \\
\Box\ & \Box_{i \in ps}\ output.i.right(i,ps) \rightarrow Spec(ps, pn)
\end{aligned}
$$

ps, pn are the sets of peers currently in and not in the network respectively. These sets change after events in the *completeChurn* channels occur, i.e. after a churn event completed. The function $succ(i, ps)$ returns the immediate right neighbor of i in the set ps, which is defined as follows:

$$right(p, ps) = r \quad \text{if } r \in ps \ \wedge \ \forall p' \in ps \backslash \{r\} \,.\, |r \ominus p| \leqslant |r \ominus p'|$$

Let *System* be the CSP model of the entire system (the details can be found in the Appendix). To prove that the system satisfies the RA property is equivalent to showing the following:

$$Spec(\{\}, \mathcal{P}) \ \sqsubseteq_T \ System \tag{1}$$

5 Verification

The current CSP model *System* is very large and complex. Even if one only considers a network with a small number of peers, this model still contains too many states and transitions to be checked automatically by a model checker. Our approach for the verification is to firstly find an abstraction of the original model, called *Abstraction*, whose state-space is smaller. In particular, the abstraction satisfies: $Abstraction \sqsubseteq_T System$. Next, we show that $Spec(\{\}, \mathcal{P}) \sqsubseteq_T Abstraction$, which then implies that Eq.1 is correct.

Due to the space constraint, we describe our approach only briefly here. More details can be found at [10]. First, we arrive at *Abstraction* through the following steps:

1. Weakening the adversary. The original adversary has infinite memory that helps it remember and replay old messages. We weaken it by removing the memory, i.e. allowing the adversary to only *relay* messages. It turns out that the new model using this weakened adversary has the same traces as the original.
2. Reducing the nonce set. The NonceManager process supplies unique and fresh nonces from a potentially *infinite* set. We make use of the *data independence* techniques, developed by [23,24], to derive an abstraction of the original model that uses only 2 nonces (one used by the CA and another by the Verifier). In *System* (in all processes except for NonceManager), nonces are used only for equality check and polymorphic operations (tupling, listing). In other words, these processes are independent of the nonce type.
3. Reducing the counter set. We use the same technique as above to reduce the counter set (which is potentially infinite) to only one value. In the current model, the TPM process uses counter values with the '>' operators, which makes it dependent of the counter type. Therefore, before applying the data independence techniques, we transform the model (without reducing its trace set) to make it independent of the counter type.

After these reduction steps, we arrive at *Abstraction*, the model less complex but refined by *System*. We have implemented a small instance of *Abstraction*

for a system with 3 peers in FDR [25], the model-checker tool for CSP. The check for $Spec(\{\}, \mathcal{P}) \sqsubseteq_T Abstraction$ returns **true** after **13,501,797** states and **73,831,002** transitions. Next, we generalize this result to an arbitrary number of peers by proving that $traces(Abstraction) \subseteq traces(Spec(\{\}, \mathcal{P}))$ for any $\mathcal{P}$. The proof is constructed via induction, as follows:

1. (Base case). Let tr be a trace of *Abstraction* such that $tr \upharpoonright \{|completeChurn|\} = \langle\rangle$ where $\upharpoonright$ is the restriction operator (for example, $sq \upharpoonright X$ removes non-X elements from sq). Then $tr \in traces(Spec(\{\}, \mathcal{P}))$.
2. (Inductive case). For any $\theta \neq \langle\rangle$, let tr be a trace of *Abstraction* such that:

$$tr \upharpoonright \{|completeChurn|\} = \theta \ \wedge \ tr \in traces(Spec(\{\}, \mathcal{P}))$$

Let tr' be another trace of *Abstraction*, then:

$$\forall e \,.\, tr' \upharpoonright \{|completeChurn|\} = \theta \hat{} \langle e \rangle \ \Rightarrow \ tr' \in traces(Spec(\{\}), \mathcal{P})$$

6 Conclusion

In this paper, we have discussed various security problems in structured P2P systems. We focus on the false-destination attacks, in which an adversary can falsely claim to be the destination of a search key. Due to the nature of structured P2P overlays (keys are stored at unique root nodes), there are a number of reasons for such attacks. A P2P system is secure from these attacks if it satisfies the root authenticity (or RA) property, which is implied by the neighbor authenticity (NA) property. We propose a P2P architecture aiming to meet this property, using Trusted Computing as the security mechanism. We then describe our formalization of the proposed architecture in CSP, then our verification that the RA property is indeed met.

We identify a few avenues to be explored in the future work. Our current churn model is relatively strict, since peers are only allowed to leave gracefully. To be more realistic, it must allow for fail-stop and Byzantine failure. The CA may not have to keep information of which nodes currently in the network. It means that before issuing certificate to a peer, the CA would ask for its neighbor certificates and only continue if they match. A further extension would be to remove the CA altogether. In this case, the certificate and verification mechanism might become much more complex and the system might not be able to cope with a complex churn model. In another direction, because the TPM was not designed with P2P applications in mind, its operation set may be too restricted for such systems. Therefore, it would be interesting to find an abstraction of general-purposed trusted hardware that is more powerful, flexible and suitable for P2P. Finally, we plan to carry out performance analysis (via simulation) of our proposed P2P architecture as well as its extensions in order to evaluate the overhead incurred from having the RA property.

References

1. Gnutella Project: Gnutella specification. WorldWide Web (July 2007), http://rfc-gnutella.sourceforge.net/developer/testing/
2. Emule Project: emule homepage. World Wide Web (May 2002), http://www.emule-project.net/
3. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: 2001 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 149–160 (2001)
4. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, pp. 329–350 (2001)
5. Rowstron, A., Druschel, P.: Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. SIGOPS Operating Systems Review 35(5), 188–201 (2001)
6. Bryan, D.A., Lowekamp, B.B., Jennings, C.: Sosimple: A serverless, standards-based, p2p sip communication system. In: International Workshop on Advanced Archtectures and Algorithms for Internet Delivery and Applications, pp. 42–49 (2005)
7. Castro, M., Duschel, P., Kermarrec, A.M., Rowstron, A.: Scribe: a large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in Communication 20(8) (2002)
8. Dinh, T.T.A., Chothia, T., Ryan, M.: A trusted infrastructure for p2p-based marketplaces. In: 9th IEEE International Conference on P2P Computing, pp. 151–154 (2009)
9. Holz, T., Steiner, M., Dahl, F., Biersack, E., Freiling, F.: Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In: 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats, pp. 1–9 (2008)
10. Dinh, T.T.A., Ryan, M.: Checking security property of P2P systems in CSP. Technical Report CSR-10-07, School of Computer Science, University of Birmingham (2010), http://www.cs.bham.ac.uk/~ttd/files/technicalReport.pdf
11. Sit, E., Morris, R.: Security considerations for peer-to-peer distributed hash tables. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 261–269. Springer, Heidelberg (2002)
12. Castro, M., Druschel, P., Ganesh, A., Rowstron, A., Wallach, D.S.: Secure routing for structured peer-to-peer overlay networks. ACM SIGOPS Operating Systems Review 36(SI), 299–314 (2002)
13. Wang, P., Hopper, N., Osipkov, I., Kim, Y.: Myrmic: Secure and robust DHT routing. Technical report, University of Minnesota (2006)
14. Ganesh, L., Zhao, B.Y.: Identity theft protection in structured overlays. In: EEE Workshop on Secure Network Protocols, pp. 49–54 (2005)
15. Borgström, J., Nestmann, U., Alima, L.O., Gurov, D.: Verifying a structured peerto- peer overlay network: The static case. In: Global Computing, pp. 250–265 (2004)
16. Bakhshi, R., Gurov, D.: Verification of peer-to-peer algorithms: A case study. Electronic Notes in Theoretical Computer Science 181, 35–47 (2007)
17. Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., Stoica, I.: Towards a common api for structured peer-to-peer overlays. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 33–44. Springer, Heidelberg (2003)

18. Trusted Computing Group: TPM Specification version 1.2. Parts 1–3 (2007), http://www.trustedcomputinggroup.org/specs/TPM/
19. ISO/IEC PAS DIS 11889: Information technology – Security techniques – Trusted platform module
20. Trusted Computing Group: Press release (2008), http://www.trustedcomputinggroup.org/news/press/member_releases/WAVETCGPROMOTI%ONMW5_31_FINAL_.pdf
21. Trusted Computing Group: TCG timeline (2008), http://www.trustedcomputinggroup.org/about/corporate_documents/
22. Balfe, S., Lakhani, A.D., Paterson, K.G.: Trusted computing: Providing security for peer-to-peer networks. In: International Conference on Peer-to-Peer Computing, pp. 117–124. IEEE Computer Society, Los Alamitos (2005)
23. Lazic, R.S.: A Semantic Study of Data-Independence with Applications to the Mechanical Verification of Concurrent Systems. PhD thesis, Oxford University (1997)
24. Broadfoot, P.J.: Data Independence in the Model Checking of Security Protocols. PhD thesis, Oxford University (2001)
25. Formal System Europe Ltd: FDR2 model checker tool. World Wide Web, http://www.fsel.com/software.html

Appendix

CSP

Events.

a, b	event communicated by CSP processes
Σ	universal sets of events; $a, b \in \Sigma$
$c.v$	communication of event v on channel c
$\|c\|$	set of all events on channel c

Processes.

P, Q	CSP process
$STOP$	do nothing
$a \rightarrow P$	prefix
$c.v \rightarrow P$	communicate value v on channel c, then behave as P
$P \,\Box\, Q,\ P \not< b \not> Q$	external choice and condition (if b then P; else Q)
$P \underset{X}{\parallel} Q$	P and Q executed concurrently, synchronized on X
$P \,\|\|\| \, Q$	interleaving
$P[\![R]\!],\ P \setminus X$	renaming relation and hiding events in X

Trace Models. This is one of the three denotational semantics supported by CSP. The other two are stable failure and failure-divergence semantics.

$$
\begin{aligned}
traces(STOP) &= \{\langle\rangle\} \\
traces(a \rightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle^{\frown} s \mid s \in traces(P)\} \\
traces(P \,\Box\, Q) &= traces(P) \cup traces(Q) \\
traces(P[\![R]\!]) &= \{\, tr \mid \exists s \in traces(P) \bullet s\ R^*\ tr \,\} \\
traces(P \,|||\, Q) &= \bigcup \{s \,|||\, t \mid s \in traces(P), t \in traces(Q)\} \\
traces(P \underset{X}{\parallel} Q) &= \bigcup \{s \underset{X}{\parallel} t\} \mid s \in traces(P), t \in traces(Q)
\end{aligned}
$$

where R^*, $|||$ and $\underset{X}{\parallel}$ are defined as follows:

$$\langle a_1, .., a_n\rangle \; R^* \; \langle b_1, .., b_m\rangle \leftrightarrow n = m \;\wedge\; \forall i \leqslant n \bullet \; a_i \; R \; b_i$$

$$\begin{aligned}
\langle\rangle \;|||\; s &= \{s\} \\
s \;|||\; t &= t \;|||\; s \\
\langle a\rangle^\frown s \;|||\; \langle b\rangle^\frown t &= \{\langle a\rangle^\frown u \parallel u \in s \;|||\; \langle b\rangle^\frown t\} \\
&\quad \cup \{\langle b\rangle^\frown u \parallel u \in t \;|||\; \langle a\rangle^\frown s\}
\end{aligned}$$

$$\begin{aligned}
s \underset{X}{\parallel} t &= t \underset{X}{\parallel} s \\
\langle\rangle \underset{X}{\parallel} \langle\rangle &= \{\langle\rangle\} \\
\langle\rangle \underset{X}{\parallel} \langle x\rangle &= \{\langle\rangle\} \qquad (x \in X) \\
\langle\rangle \underset{X}{\parallel} \langle y\rangle &= \{\langle y\rangle\} \qquad (y \notin X) \\
\langle x\rangle^\frown s \underset{X}{\parallel} \langle y\rangle^\frown t &= \{\langle y\rangle^\frown u \mid u \in \langle x\rangle^\frown s \underset{X}{\parallel} t\} \\
\langle x\rangle^\frown s \underset{X}{\parallel} \langle x\rangle^\frown t &= \{\langle x\rangle^\frown u \mid u \in s \underset{X}{\parallel} t\} \\
\langle x\rangle^\frown s \underset{X}{\parallel} \langle x'\rangle^\frown t &= \{\langle\rangle\} \qquad (x, x' \in X \;\wedge\; x \neq x') \\
\langle y\rangle^\frown s \underset{X}{\parallel} \langle y'\rangle^\frown t &= \{\langle y\rangle^\frown u \mid u \in s \underset{X}{\parallel} \langle y'\rangle^\frown t\} \\
&\quad \cup \{\langle y'\rangle^\frown u \mid u \in t \underset{X}{\parallel} \langle y\rangle^\frown u\}
\end{aligned}$$

System Model in CSP

Events. Let $\mathcal{P}$ and $Nonces = \{Nonce.id \mid id \in NonceIDs\}$ be the set of peers (each having a unique ID) and the set of fresh nonces. Let

$$ChurnEvents = \{Churn.\langle c, i\rangle \mid c \in \{join, leave\}, i \in \mathcal{P}\}$$

be the set of churn events. Let $Counts$ be the set of counter values. We define the set of events representing TPMs' signature on counter values as:

$$SigMessages = \{SqR.\langle n, p, c\rangle, SqI.\langle n, p, c\rangle \mid n \in Nonces, p \in \mathcal{P}, c \in Counts\}$$

For any n, p, c, $SqR.\langle n, p, c\rangle$ and $SqI.\langle n, p, c\rangle$ represent signatures on the results of the `TPM_ReadCounter` and `TPM_IncrementCounter` commands respectively.

Let $NonceMessages = \{SqN.\langle n\rangle \mid n \in Nonces\}$ be a set of events representing a nonce value being passed in the network. Finally, the set of events representing certificates issued by the CA is defined as:

$$CertMessages = \{Cert.\langle p, l, r, c\rangle \mid p, l, r \in \mathcal{P}, c \in Counts\}$$

Then, the set of all events used in the model is defined as:

$$\begin{aligned}
Messages = {} & ChurnMessages \cup SigMessages \\
& \cup NonceMessages \cup CertMessages
\end{aligned}$$

Channels. The following channels are used:

$send, receive$: $Agents.Agents.Messages$
where $Agents = \mathcal{P} \cup \{NM, CA, VF\}$
$take, fake$: $Agents.Agents.Messages$
$learn, say$: $Messages$
$output$: $\mathcal{P}.\mathcal{P}$
$completeChurn$: $ChurnMessages$
$unlock$: $Agents.\mathcal{P}$

NonceManager Process

$$NManager(X) = \Box_{\substack{n \in X \\ j \in Agents}} send.NM.j.SqN.\langle n \rangle \rightarrow NManager(X \backslash \{n\})$$

$$NonceManager = NManager(Nonces)$$

TPM Process

$$TPM(i,c) = \Box_{\substack{n \in Nonces \\ j \in Agents}} receive.j.i.SqN.\langle n \rangle \rightarrow \left(\begin{array}{l} \Box_{\substack{d \geqslant c \\ d \in Counts}} send.i.j.SqR.\langle n, i, d \rangle \rightarrow unlock.VF.i \rightarrow TPM(i,d) \\ \Box \Box_{\substack{d > c \\ d \in Counts}} send.i.j.SqI.\langle n, i, d \rangle \rightarrow unlock.CA.i \rightarrow TPM(i,d) \end{array} \right)$$

$$TPMs = |||_{i \in \mathcal{P}} TPM(i,0)$$

CA Process

$$\begin{array}{l} CAProcess(ps,pn) = \quad |ps| == 0 \ \& \ Join0(ps,pn) \\ \qquad\qquad\qquad\qquad \Box \ |ps| == 1 \ \& \ Join1(ps,pn) \\ \qquad\qquad\qquad\qquad \Box \ |ps| > 1 \ \& \ JoinAndLeaveN(ps,pn) \end{array}$$

$$\begin{array}{l} JoinAndLeaveN(ps,pn) = \Box_{i \in pn} receive.i.CA.Churn.\langle join, i \rangle \rightarrow JoinN(i,ps,pn) \\ \qquad\qquad\qquad\qquad \Box \Box_{i \in ps} receive.i.CA.Churn.\langle leave, i \rangle \rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad \textit{if } |ps| > 2 \textit{ then } LeaveN(i,ps,pn) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textit{else } Leave2(i,ps,pn) \end{array}$$

$$
\begin{array}{l}
\mathit{JoinN(i,ps,pn)} = \\
\quad \Box_{n1,n2,n3 \in Nonces} \; \mathit{receive.NM.CA.SqN}.\langle n1\rangle \rightarrow \mathit{send.CA.i.SqN}.\langle n1\rangle \\
\qquad \rightarrow \left(\begin{array}{ll}
\Box_{c1,c2,c3 \in Counts} & \mathit{receive.i.CA.SqI}.\langle n1, i, c1\rangle \rightarrow \\
& \mathit{let}\ S = ps \cup \{i\} \\
& \quad (l, r) = \mathit{neighbor}(i, S) \\
& \quad (l1, r1) = \mathit{neighbor}(l, S) \\
& \quad (l2, r2) = \mathit{neighbor}(r, S)\ \mathit{within} \\
& \qquad \mathit{send.CA.i.Cert}.\langle i, l, r, c1\rangle \\
& \quad \rightarrow \mathit{receive.NM.CA.SqN}.\langle n2\rangle \\
& \quad \rightarrow \mathit{send.CA.l.SqN}.\langle n2\rangle \\
& \quad \rightarrow \mathit{receive.l.CA.SqI}.\langle n2, l, c2\rangle \\
& \quad \rightarrow \mathit{send.CA.l.Cert}.\langle l, l1, i, c2\rangle \\
& \quad \rightarrow \mathit{receive.NM.CA.SqN}.\langle n3\rangle \\
& \quad \rightarrow \mathit{send.CA.r.SqN}.\langle n3\rangle \\
& \quad \rightarrow \mathit{receive.r.CA.SqI}.\langle n3, r, c3\rangle \\
& \quad \rightarrow \mathit{send.CA.r.Cert}.\langle r, i, r2, c3\rangle \\
& \quad \rightarrow \mathit{completeChurn.Churn}.\langle \mathrm{join}, i\rangle \\
& \quad \rightarrow \mathit{unlock.CA.i} \rightarrow \mathit{unlock.CA.l} \\
& \quad \rightarrow \mathit{unlock.CA.r} \rightarrow \mathit{CAProcess(S,pn\backslash\{i\})}
\end{array} \right)
\end{array}
$$

Other sub-processes, namely *Join0(ps,pn), Join1(ps,pn), Leave2(ps,pn)* and *LeaveN(ps,pn)* are be defined similarly. The function $neighbor(p, ps)$ returns the left and right neighbor of p in ps. More precisely,

$$
\begin{array}{l}
neighbor(p, ps) = (left(p, ps), right(p, ps)) \\
left(p, ps) = l \quad \text{if} \;\; l \in ps \;\wedge\; \forall p' \in ps \backslash \{l\} \,.\, |p \ominus l| \leqslant |p' \ominus l| \\
right(p, ps) = r \quad \text{if} \;\; r \in ps \;\wedge\; \forall p' \in ps \backslash \{r\} \,.\, |r \ominus p| \leqslant |r \ominus p'|
\end{array}
$$

Verifier Process.

$$
\begin{array}{l}
\mathit{VerifierProcess} = \\
\quad \Box_{n \in Nonces} \; \mathit{receive.NM.VF.SqN}.\langle n\rangle \\
\qquad \rightarrow \left(\begin{array}{ll}
\Box_{i,l,r \in \mathcal{P}} & \mathit{send.VF.i.SqN}.\langle n\rangle \\
& \rightarrow \left(\begin{array}{ll}
\Box_{c \in Counts} & \mathit{receive.i.VF.SqR}.\langle n, i, c\rangle \\
& \rightarrow \mathit{receive.i.VF.Cert}.\langle i, l, r, c\rangle \\
& \rightarrow \mathit{if}\ l = r\ \mathit{and}\ l = i\ \mathit{then} \\
& \qquad \mathit{output.i.i} \rightarrow \mathit{unlock.VF.i} \\
& \qquad \rightarrow \mathit{STOP} \\
& \quad \mathit{else}\ \mathit{VerifierProcessN(l,i)}
\end{array} \right)
\end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{VerifierProcessN(l,i)} = \\
\quad \Box_{n \in Nonces} \; \mathit{receive.NM.VF.SqN}.\langle n\rangle \rightarrow \mathit{send.VF.l.SqN}.\langle n\rangle \\
\qquad \rightarrow \left(\begin{array}{ll}
\Box_{cl \in Counts} & \mathit{receive.l.VF.SqR}.\langle n, l, cl\rangle \\
& \rightarrow \left(\begin{array}{ll}
\Box_{ll \in \mathcal{P}} & \mathit{receive.l.VF.Cert}.\langle l, ll, i, cl\rangle \\
& \quad \rightarrow \mathit{output.l.i} \rightarrow \mathit{unlock.VF.i} \\
& \quad \rightarrow \mathit{unlock.VF.l} \rightarrow \mathit{STOP}
\end{array} \right)
\end{array} \right)
\end{array}
$$

Adversary Process

$$MemoryNonce(n) = learn.SqN.\langle n\rangle \rightarrow ReplayNonce(n)$$
$$ReplayNonce(n) = say.SqN.\langle n\rangle \rightarrow ReplayNonce(n)$$

$$MemorySigR(n,i,c) = learn.SqR.\langle n,i,c\rangle \rightarrow ReplaySigR(n,i,c)$$
$$MemorySigI(n,i,c) = learn.SqI.\langle n,i,c\rangle \rightarrow ReplaySigI(n,i,c)$$
$$ReplaySigR(n,i,c) = say.SqR.\langle n,i,c\rangle \rightarrow ReplaySigR(n,i,c)$$
$$ReplaySigI(n,i,c) = say.SqI.\langle n,i,c\rangle \rightarrow ReplaySigI(n,i,c)$$

$$MemoryCert(i,l,r,c) = learn.Cert.\langle i,l,r,c\rangle \rightarrow ReplayCert(i,l,r,c)$$
$$ReplayCert(i,l,r,c) = say.Cert.\langle i,l,r,c\rangle \rightarrow ReplayCert(i,l,r,c)$$

$$Memory = \Vert\!\Vert_{n\in Nonces} MemoryNonce(n) \Vert\!\Vert \Vert\!\Vert_{n\in Nonces, i\in\mathcal{P}, c\in Counts} \begin{pmatrix} MemorySigR(n,i,c) \\ \Vert\!\Vert\ MemorySigI(n,i,c) \end{pmatrix} \Vert\!\Vert \Vert\!\Vert_{i,l,r\in\mathcal{P}, c\in Counts} MemoryCert(i,l,r,c)$$

$$ChurnInitiator = \Box_{i\in\mathcal{P}} \begin{pmatrix} say.Churn.\langle \text{join}, i\rangle \rightarrow ChurnInitiator \\ \Box\ say.Churn.\langle \text{leave}, i\rangle \rightarrow ChurnInitiator \end{pmatrix}$$
$$Adversary = Memory \Vert\!\Vert ChurnInitiator$$

Putting It Together

$$Network = \left(Adversary \underset{\chi_i}{\parallel} TPMs \right) \setminus \chi_i$$
$$CAandVFProcess = CAProcess(\{\}, \mathcal{P}) \Vert\!\Vert VerifierProcess$$
$$OtherAgents = \left(NonceManager \underset{\{|fake.NM|\}}{\parallel} CAandVFProcess \right)$$
$$Impl = \left(OtherAgents \underset{\chi_e}{\parallel} Network\ Big \right) \setminus \{|take, fake, unlock|\}$$

Table 1. Renaming relations and synchronization sets

Name	Details	Applied to
RAd_1	$learn \leftarrow take.i.j \mid i,j \in Agents, \{i,j\} \cup \mathcal{P} \neq \varnothing$	$Adversary$
RAd_2	$say \leftarrow fake.i.j \mid i,j \in Agents, \{i,j\} \cup \mathcal{P} \neq \varnothing$	$Adversary$
$RCom_1$	$send \leftarrow take$	$TPMs$, $CAProcess$ and $VerifierProcess$
$RCom_2$	$receive \leftarrow fake$	$TPMs$, $CAProcess$ and $VerifierProcess$
$RNonce_1$	$send.NM.i \leftarrow take.NM.i \mid i \in \mathcal{P}$	$NonceManager$
$RNonce_2$	$send.NM.j \leftarrow fake.NM.j \mid j \notin \mathcal{P}$	$NonceManager$
χ_i	$\{\|take.i.a, fake.a.i \mid i \in \mathcal{P}, a \in Agents\|\}$	
χ_e	$\{\|take.a.i, fake.i.a \mid i \in \mathcal{P}, a \in Agents\|\}$	