Appeared in the Proceedings of the 7th Nordic Workshop on Secure IT Systems---NORDSEC 2002.

Virtual Hierarchies - An Architecture for Building and Maintaining Efficient and Resilient Trust Chains

John Marchesini and Sean Smith
Department of Computer Science *
Dartmouth College

{carlo,sws}@cs.dartmouth.edu

DRAFT of May 17, 2002

Abstract

In *Public Key Infrastructure (PKI)*, the simple, monopolistic organizational model of certificate issuing entities works fine until we consider real-world issues. Then, issues such as scalability and mutually suspicious organizations create the need for a multiplicity of certificate issuing entities, which introduces the problem of how to organize them to balance resilience to compromise against efficiency of path discovery. Many solutions involve organizing the infrastructure to follow a natural organizational hierarchy, but in some cases, such a natural organizational hierarchy may not exist.

However, systems research has given us *secure coprocessing* for securely carrying out computations among multiple trust domains. Cryptography has produced a number of methods for distributing cryptographic computations, such as *secret splitting* and *threshold cryptography*. Last, distributed computing has given us *peer-to-peer networking*, for creating self-organizing distributed systems.

In this paper, we use these latter tools to address the former problem by overlaying a *virtual hierarchy* on a mesh architecture of peer certificate issuing entities, and achieving both resilience and efficiency.

*This work was supported in part by Internet2/AT&T, by IBM Research, and by the U.S. Department of Justice, contract 2000-DT-CX-K001. However, the views and conclusions do not necessarily represent those of the sponsors. A preliminary version appears as TR2002-416.

1 Introduction

1.1 The Problem

Background By separating the privilege to decrypt or sign a message from the privilege to encrypt or verify, *public-key cryptography* enables forms of trusted communication between parties who do not share secrets *a priori*. Eliminating the need for shared secrets has multiple advantages. On a global level, it potentially enables extending trusted communication across organizational boundaries, between parties who have never met. But it can also reduce overhead in managing communication between parties even on a local level, within one organization: the number of needed keys goes from $\Omega(n^2)$ to $O(n)$.

PKI has many definitions; the most commonly accepted definition refers to how one participating party learns what the public key is for another party. Typically, approaches to PKI begin by condensing trust: rather than *a priori* knowing the public key of each party in the population, the relying party instead knows the public key of a designated special party, who in turn issues signed statements (e.g., certificates and CRLs) about members of the population.

This designated party is typically called the *certificate authority (CA)*. Some approaches separate the process of issuing certificates from the process of

identifying and authorizing keyholders to receive the certificates; in these approaches, the latter tasks become the responsibility of the *registration authority (RA)*. Since the CA must hold and wield a private key of considerable value, implementations apply various protections to that private key, such as housing it in a hardened cryptographic module that is kept offline. Some ambiguity thus results regarding what the term "CA" refers to: the entity (typically online) that issues and manages certificates; this entity, minus the RA duties; or this entity's specific machine that houses the private key. In this paper, we use the former implication.

More than One This simple PKI model of one CA servicing a user population suffers from some inherent limitations. For one thing, for certification to be meaningful, the CA must be in some position to certify the identity of keyholders according to some uniform policy. This can limit those whom a CA certifies to sets where such a social or business relationship is plausible (such as employees of a small enterprise). For another thing, the relying party must be able to identify and trust the CA for the keyholder—which again limits the a CA's set of relying parties to sets where such a social or business relationship exists.

These limitations create the need for multiple CA/keyholder sets, which in turn creates the need to organize these PKIs so that public key operations can take place across these sets. PKIs within an organization are becoming a common occurrence. Such systems have been well studied, and are often built from commercially available components. Within an organization which has a PKI, the certificates generated by the organization's CA are meaningful. Outside of such an organization, however, those same certificates are meaningless unless some agreement between a number of organizations is in place.

The question thus arises of how to organize multiple CAs. The basic literature (e.g., [10]) gives a serious examination of this question. Readers unfamiliar with this literature may be tempted to assert that the only natural solution here is to use a *name-constraint hierarchy*: group CAs into sets that have some natural social peer relationship; for each group, establish a new CA that certifies the CAs in that group; and continue this process upward, so that the we result in a tree with a single trust root. For example, one might follow the DNS hierarchy, and assume that a global root certifies a `edu` CA, which certifies a `dartmouth.edu` CA, which certifies a `cs.dartmouth.edu` CA, which certifies each member of our department.

Although apparently natural, this approach has many drawbacks.

- Hierarchies are not always *scalable*, in that they cannot permit the participating fraction of the population to grow gradually. Suppose the natural social hierarchy has four levels, and two unrelated leaves want to establish a trust relationship. They can only do this if all the interior CAs—from the first leaf, up to the root, then down to the second leaf—are already participating in the PKI.

- Hierarchies are not always *usable*. The globally unique names determined by the natural hierarchy may not necessarily be usable by the humans who need to use them to make trust judgments. A colleague reports that his `foo.com` domain has 100K machines whose names are of the form `bar.foo.com`. Not only is this namespace crowded—a typo will likely give the user the wrong machine—but it also changes dynamically: a sysadmin the user never meets changes machine names without notification.

- Hierarchies do not always *exist.* The relationship between users and their immediate CAs is usually (but not always) natural. However, the upper regions get murky. With mobile devices [11, 21] or collections of universities or government departments, one typically encounters federations of peers with no clear natural organization. Indeed, except for DNS and perhaps the Roman Catholic church, it is hard

to find a naturally hierarchy whose upper regions are well-defined. Which U.S. military service[1] should be the root? Which DOE laboratory?[2] Why should CREN or USPS or NIH be the root over academia or citizens at large?

PKI as a system Consequently, we're going to take the unorthodox view of looking at "PKI" as a *system* that has various properties, instead of an automatic mirror of a social arrangement that may not necessarily be appropriate, even if it exists.

We might start by thinking of conventional CAs (from the simple model above) as nodes, and trying to decide how to link them together—perhaps by creating new CAs—in a directed graph, where edges go from a CA to each entity that it certifies.

Two desirable properties of any PKI are *resilience* and *efficiency*.

- By *resilience*, we mean the ability of the system to tolerate the discovery that any given key pair has been compromised. What trust judgments become impossible? How many key pairs must be revoked and reissued?
- To verify a certificate, a relying party needs to find a path from a *trust root* (a node it *a priori* trusts) to the certificate in question. By *efficiency*, we mean the running time of the standard algorithm to discover this path.

Resilience and efficiency are typically competing goals.

Structured Centralization. Many current architectures impose a rigid structure on the organization of CAs, which means that path construction and validation can be deterministic and efficient. Although this structure permits path algorithms to traverse the topology within some efficient time constraints, it also results in a large amount of authority residing in a single place (e.g. the root CA). This centralization of authority directly decreases resilience in that if the root CA is compromised, the entire PKI is unusable until it can recover.

Hierarchies are the canonical example of this structured approach. Traditionally, hierarchies achieve $O(logV)$ (where V is the number of CAs) verification time, because paths in a tree are well-defined and easy to find (Figure 1). We noted many drawbacks above; another drawback is that hierarchies place increasing amounts of value on the private keys of interior nodes. If the adversary were to compromise an upper-level CA or even the root CA, the entire PKI must suspend operation until a recovery can occur (i.e. all certificates issued by that CA are revoked, and new ones are reissued with the CA's new private key) (Figure 2).

Thus, hierarchies obtain efficiency at the cost of resilience.

Unstructured Decentralization. In opposition to this view is the method of organizing CAs in a more decentralized way, in an effort to increase resilience by not placing so much authority in one centralized place. However, decentralization implies that path validation algorithms must now do more work and must often use non-determinism to decide if a received trust chain is valid. These properties translate into a decrease in efficiency and an increase in complexity on the part of the verifier.

Meshes are the canonical example of the unstructured approaches. Mesh PKI architectures have been developed in part to avoid this single point of failure (Figure 3). However, the non-deterministic nature of peer-to-peer organization increases the path verification algorithm significantly (Figure 4). Due to the fact that not all possible choices lead to a trusted CA, coupled with trial-and-error construction of the trust path (a path to a trusted CA),

[1] The first author would assert that it's the branch to which he belonged.

[2] The second author would assert that it's the DOE laboratory for which he used to work.

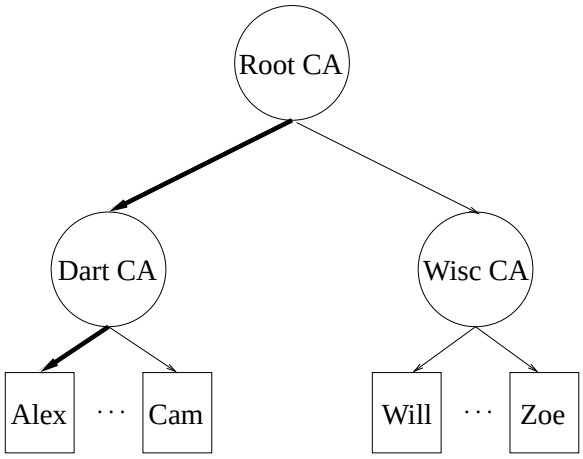

Figure 1 When Zoe receives a certificate chain from Alex, she verifies that the Dartmouth CA certificate is signed by the Root CA. She then verifies that Alex's certificate is signed by the Dartmouth CA. As the number of CAs grows, it takes $O(logV)$ time to verify the path from the Root CA to the user which needs to be verified.

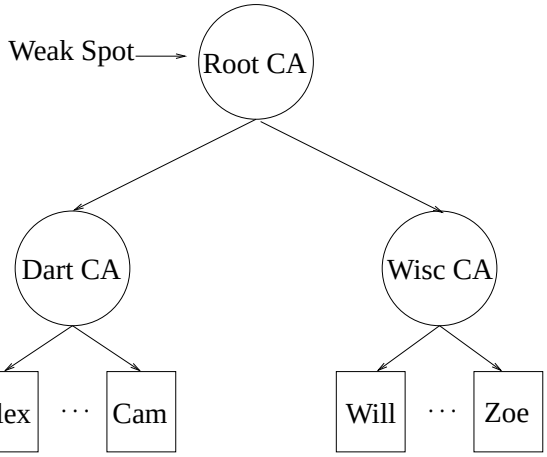

Figure 2 If the Root CA is compromised, the system goes down until all certificates are revoked and new certificates are issued.

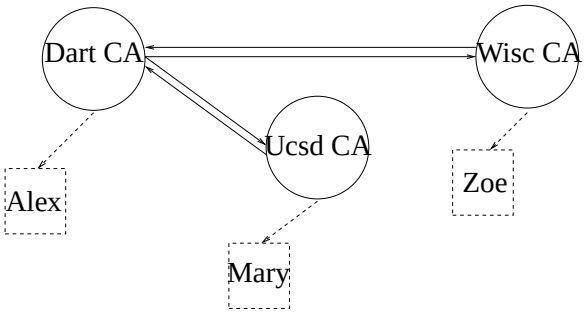

Figure 3 Meshes offer increased resilience. If the Wisconsin CA's private key is disclosed, the other CAs can continue to operate. Only Wisconsin is affected. When Wisconsin gets back online, it may rejoin the CA network.

verification time in these schemes is usually high. Further, mesh architectures make no guarantee to avoid cycles, leading to choices in the path construction algorithm which may never terminate.

Thus, meshes obtain resilience at the cost of efficiency.

Other Approaches Finding algorithms which increase the efficiency of path construction in decentralized organizations is an emerging area of research. Algorithms which use *certificate extensions* (such as name constraints and policy extensions), as well as loop elimination techniques have been developed to enhance efficiency [8]. Our concern however, is the underlying organization of CAs, and how they may be arranged to achieve efficiency and resilience.

Other common architecture schemes are more hybrid.

Extended Trust Lists are used to allow users the ability to maintain lists of CAs which they choose to trust. Each entry in this list may represent a single CA or an entire PKI, which itself could be a Hierarchy or a Mesh. This scheme poses new challenges for validation algorithms, as the starting point for these algorithms could be any node in the

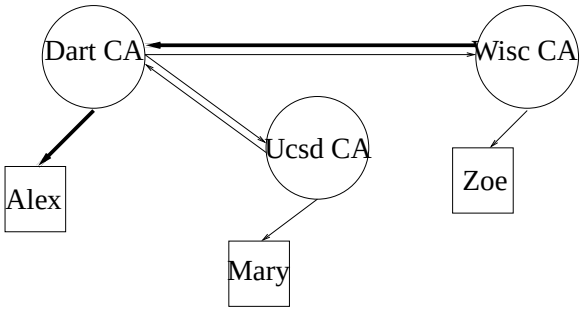

Figure 4 Meshes offer decreased efficiency. When Zoe receives a certificate chain from Alex, she verifies that that a trust path exists from Alex's CA to a point which she trusts. In this example, the Dartmouth CA and Wisconsin CA are cross-certified, so she believes Alex's certificate is good. As the number of CAs grows, it takes $O(V)$ time to verify a certificate chain.

list. The implication is that a path may have be to be constructed using every entry in the list as a starting point.

Bridge CAs provide another alternative to the common approach of cross-certifying enterprise PKIs through peer-to-peer relationships. Cross-certification without a Bridge CA results in $(n^2 - n)/2$ relationships for n enterprises (in graph theory, this graph is known as a complete graph on n vertices, and is named K_n [20]).

The Bridge allows each of the enterprise PKIs to cross-certify to it, resulting in a star topology between enterprises and reducing the number of relationships to n. Bridges are also used for bridging different policy domains. In these situations, the PKIs themselves are usually complex.

While this is an attractive solution if all of the enterprises are hierarchies, the Bridge architecture does not solve path validation issues in general as each of the enterprises themselves may be Meshes [10, 14].

The *Bottom Up With Name Constraints* model [12] is one which shares our goal of allowing organizations to construct their own PKI and then connect it to other organizations' PKIs. The model assumes a hierarchical namespace and that CAs are certified in both directions, down (from parent to child) and up (from child to parent). The model also allows for CAs to cross certify directly.

Path validation in this Bottom Up model is quite efficient due to the presence of certification in both directions. The validation algorithm begins by starting at a trust anchor and looking first for a cross-certified CA which is either an ancestor of the target or the target itself. If this fails, the algorithm proceeds up to the parent CA and searches through its cross-certified CAs. This terminates when a cross certificate is found or when a common ancestor is found.

This model also has impressive resilience properties. If a key is compromised, the compromised CA can issue new certificates to all of the CAs which are certified (up, down, and cross). The communities belonging to each certified CA will automatically be bound to the new key, without having to make any changes.

The major difference between this model and ours is that we relax the assumption of a hierarchical namespace. As mentioned earlier, hierarchies are not always usable, scalable, or present.

1.2 Our Solution

We believe that both properties—efficiency and resilience—are important to most PKI systems. We thus propose an architecture and are developing a prototype which aims to bridge the gap between these seemingly competing goals. We feel this is novel as most current architectures fail to provide both.

Our objective is to devise an architecture which allows for CAs to organize themselves in such a way as to maintain the following two invariants:

1. **Efficiency.** Trust chains produced by any of the entities may be verified in an efficient manner, meaning that trust chains are loop-free. This is common in hierarchy schemes.

2. **Resilience.** The secrets (private keys) do not exist in any one place or can not carry out cryptographic computations without collaboration. The fragments/parties are fairly randomly distributed throughout the topology.

This would be useful when the authority of the CAs in the real world is not easily represented by a strict hierarchy or when certificates need to be used frequently outside of the issuing namespace.

For example, a visiting professor from the University of Wisconsin who comes to Dartmouth should be able to use his or her private key even though the institutions' PKIs are not cross-certified. If both institutions were participating in a *virtual hierarchy*, no cross-certification would be required to verify statements signed by the professor's private key, which was issued by the University of Wisconsin CA, and is being used at Dartmouth.

Overview The mechanism we propose which accomplishes this task is a *virtual hierarchy*, a logical hierarchy formed in a peer-to-peer network. As with a standard hierarchy, we can model a virtual hierarchy as a tree with nodes and directed edges. Leaves can represent bottom-level users; their parents represent their natural CA.

However, the remaining nodes are *virtual* CAs. Although each such node is a logical entity in the virtual hierarchy, it is represents the *collective* action of a set of conventional CAs. Consequently, we use the term *collective* for this set. (See Figure 5.)

Figure 5 and Figure 6 sketches this arrangement.

We obtain this collective action via cryptography. There are a number of cryptographic schemes which require collaboration in order to perform cryptographic computations. The broad category for these methods is known as *threshold cryptography*. Some examples of threshold schemes are: *secret sharing* [15] and *multi-party signature schemes* [7]. One particularly attractive scheme is *Multi-Party RSA* [4], which we will discuss in 3.4.2.

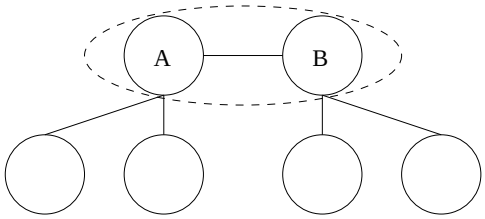

Figure 5 A single collective. All of the nodes are CAs. The nodes inside the oval are maintaining a portion of the private key privilege (e.g. via threshold cryptography), which is acting as the Root CA key. The other nodes are directly connected to one of the nodes which collaborate in cryptographic computations for the collective and may "use" the key (i.e. make a broadcast to have a message signed by the nodes in the oval). It should be noted that it is possible to put all of the nodes inside the oval, meaning that each member of the collective would maintain a potion of the private key privilege.

For ease of implementation only, in our initial prototype we chose secret splitting, where each party holds a fragment of the private key. A subset of entities in the collective actually possess the key fragments which, when assembled, acts as the parent CA for all of the members of the collective. Section 3.1 discusses these issues further.

We have developed (and prototyped) algorithms that allow natural CAs to form collectives in and ad hoc manner, and then form collectives into a hierarchy (where a virtual node can itself become certified by another collective) in order to maintain a good tree structure. (See Figure 6.)

This approach thus obtains the goals we desired:

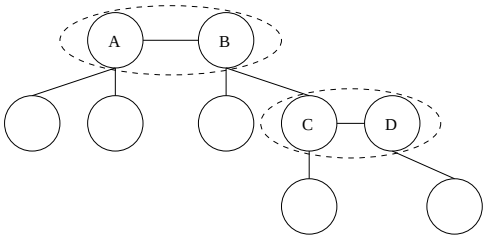

Figure 6 A hierarchy is emerging. Here there are two collectives, linked by the CA denoted as "C". C is a member of the original collective shown in Figure 5, and is a member collaborating in cryptographic computations (along with D) of the second level collective.

1. **Efficiency.** By maintaining the structure of a hierarchy, we retain an expected $O(logV)$ trust chain verification cost, with no loops.

2. **Resilience.** By distributing the higher-level CA private key privileges among multiple parties, we retain the resilience of decentralized approaches.

Physical Layer The physical layer is a peer-to-peer network of secure coprocessors [17] (we use IBM 4758s [3]). The secure coprocessor is not strictly necessary to make the virtual hierarchy layer work. However, since nodes in this layer are CAs, they must all have a cryptographic module, and using trusted hardware adds to the security of the scheme in that if the machine which houses the module is compromised, the module itself is still secure. Practically speaking, part of our decision to use secure coprocessors came from the fact that we already had some devices, we had some familiarity with the programming environment, and the modules we had are validated to FIPS 140-1 Level 4.

[3]Recently, a security vulnerability [3] has been demonstrated in an application (IBM's CCA) which runs on the 4758. It should be noted that this vulnerability belongs to the application, and not the 4758 platform. At the time of writing, the 4758 has no known vulnerabilities.

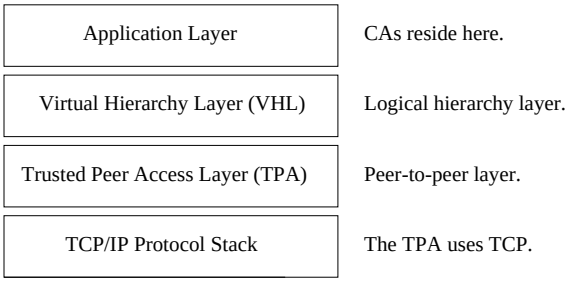

Figure 7 The protocol stack.

2 Overall Structure

We approached the problem in two stages, the first being to implement a peer access layer which allows secure coprocessor to communicate securely, and the second was to implement the virtual hierarchy algorithms on top of that layer. The resulting protocol stack is depicted in Figure 7.

Our prototype implements the VHL and the TPA. The prototype version of the VHL contains a command line interface so that we don't need to integrate with a CA at this stage in development. The peer access layer running inside of the IBM 4758s is depicted as the TPA layer, and the algorithms which construct and maintain the logical hierarchy are shown as the VHL. The two layers are implemented as separate processes, with the output of the VHL being piped into the TPAL using standard UNIX pipes.

Before we discuss the layers in detail, a simple example will be useful in understanding the high level operation and what we are trying to achieve. In the example shown in Figures 8 through 12, two machines A and B will connect, negotiate a secret, and store on half of the secret. This operation forms the root collective. Four more machines will join the collective, and are able to use the secret maintained by A and B. Then a new collective will be formed by C and D, and the hierarchy will grow. This is a simple example, but will serve to familiarize the reader with the basic concept.

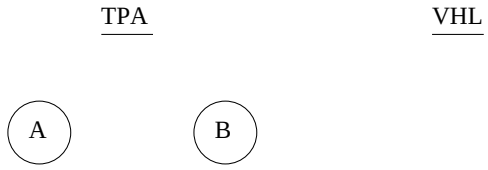

Figure 8 Neither of the two machines share the private key privilege for a virtual CA.

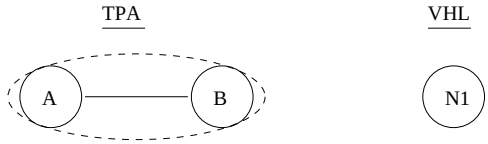

Figure 9 A connects to B and they establish a virtual CA, and the two parties now share a portion of the private key privilege (as denoted by the oval). A collective is formed and a node is established in the virtual hierarchy.

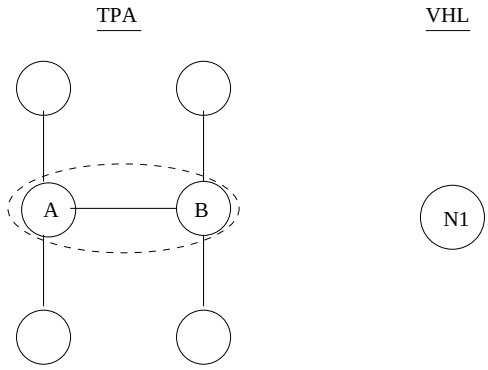

Figure 10 More nodes have joined the collective, although none are required to negotiate keys, as the maximum size of a collective for this example is six. Since the four new CAs are just using the key held by A and B, the virtual hierarchy remains unchanged.

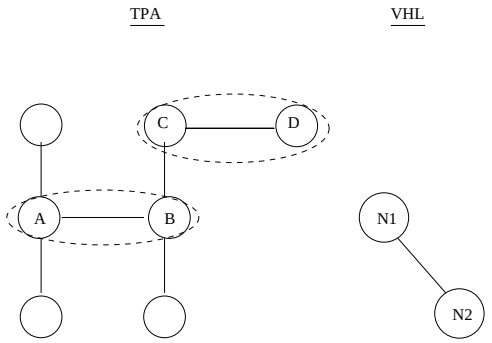

Figure 11 Machine D makes a connection to machine C, causing them to establish a new virtual CA and share a portion of the private key privilege. This operation forms a new collective and a new node $N2$ in the virtual hierarchy. Machine C is now in both collectives.

3 Virtual Hierarchy Layer

From the highest level, the virtual hierarchy (i.e. the logical hierarchy in the peer-to-peer network) is constructed by an algorithm that allows peer CAs to establish a secure connection and negotiate a secret which each of their communities may use as an end-point in their trust chain. Pieces of the trust root (negotiated secret) are then stored among the peers who negotiate it.

This leads us to make the following two claims:

Increased Resilience The result of this negotiation produces a root "entity" whose privilege is distributed among the n parties who are at a distance 1 (i.e. directly connected) to one of the actors in the negotiation. This group of n parties is a collective and all act as though the "entity" is their root CA. The result of spreading pieces of the secret (in our scheme) or requiring collaboration to perform cryptographic computation (in threshold schemes) among a group of peers alleviates the single point of failure problem.

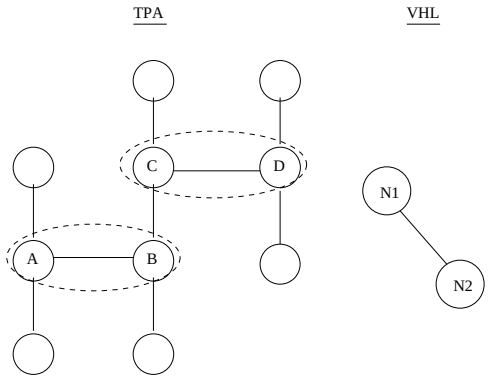

Figure 12 Three CAs make connections to C and D, joining the new collective. Since a private key has already been established for that collective (held by C and D), the new members do not need to negotiate one, and the virtual hierarchy remains unchanged.

Increased Efficiency The result of this negotiation produces a root "entity" whose role is to act as a trust point for the n parties who are at a distance 1 to one of the actors in the negotiation. We will show that this maintains a hierarchical trust structure similar to one which would be found in a physical hierarchy of CAs. Maintaining this hierarchy allows trust calculations to be performed at an average of $O(logV)$ time (again where V is the number of CAs participating in the network).

It is important to realize that our solution, like other current schemes, would require each community to perform some amount of work to reflect the new topology.

3.1 Simplifying Constraints

Our algorithm follows several rules that constrain (and simplify) the problem.

In order to maintain the property that verification may be done in $O(logV)$, we design our algorithm to maintain the invariant that there are no cycles in the connection graph produced by the connection network of CAs. These connections are accomplished using the protocol TPA Layer.

We maintain this invariant because if we were to allow cycles at this layer, we would break the hierarchical structure by transforming it from a tree into a less-structured graph. Breaking the hierarchical structure would have the following two implications:

First, to perform an efficient path verification algorithm in this graph, the algorithm would need to locate the shortest correct (i.e. matching the certificate chain) path. This would take longer than $O(logV)$ in the average case.

Second, any such algorithm would require state to be maintained so that the shortest correct path may be calculated without having to account for the time it takes to discover the topology in real time. This could be accomplished by implementing a "smarter" routing algorithm in the TPA Layer (e.g. reverse path forwarding [18]). Because we maintain our no-cycle invariant, we can instead use simple broadcasting. Alternative schemes that relax this invariant are an area for future work.

In addition, our algorithm maintains simplifying restrictions on the communication that occurs between two collectives. First, at least one collective must be a *root collective* (the root node in some virtual hierarchy). Without this restriction, intra-collective connections would break the tree topology. Second, nodes which collaborate in cryptographic computations for one collective may not collaborate for another. Allowing nodes to collaborate for two collectives simultaneously forces that node to hold two separate trust chains and breaks the hierarchical constraints.

We considered having the algorithm maintain a balance invariant on the hierarchy (e.g., each operation would maintain some balance property in the tree). However, this approach could result in large changes in the topology when a single node joins the network. We do make an assumption, however, that the nodes join and leave the network in a random fashion, resulting in a randomly built tree. It

can be shown that randomly built trees have a height of $O(logV)$, and a worse case height of V. (We flag the possibilities of enforcing balance as future work.)

Cryptography In order to meet our claim of increased resilience, it is necessary for our scheme to require collaboration in order to perform cryptographic computations.

For ease of implementation, we take the approach that pieces of the negotiated secret are scattered throughout the collective (as noted earlier). Many cryptographic techniques can enable this behavior. For simplicity, we considered secret splitting [9]. When a request is made to sign something, the key is discovered by the host which received the request by broadcasting to the collective and ordering the pieces of the key. This is a *transient* operation in that the key is not stored at the host. Once the operation has been performed, the host forgets the key.

There are other cryptographic methods for accomplishing the same functionality, such as secret sharing and cooperative signature schemes [16]. These schemes are better, but more complex to implement, so for purposes of this analysis and prototype, we assume that the secret splitting is sufficiently representative. We discuss implications of some of these other techniques in Section 3.4.2, and discuss Multi-Party RSA, as we plan to eventually implement it.

3.2 The Algorithms

The pseudocode procedures in the Pseudocode appendix at the end of this paper maintain the invariants put forth in Section 1.2. The client and server actions (Figure 15 and Figure 16) guarantee the first invariant by eliminating cycles in the topology. We will show that the elimination of cycles is key to allow for efficient validation. They maintain the second invariant by enforcing that parties which negotiate a secret only store a fraction of it. This implies that the secrets are distributed among members of the collective.

The server action is responsible for accepting connection requests, authenticating them, and deciding whether the two parties need to negotiate a secret. This decision is based on whether one of the parties has a key fragment. The presence of such a fragment in either party implies that at least one of the parties belongs to an collective and the other one is joining. The absence of a fragment implies that a new secret must be negotiated, which in turn, implies that a new collective is being formed.

The client action is called from an outside entity (i.e. user code), and is essentially making the same decision as above. The added burdens of avoiding cycles and enforcing assumptions about communication between collectives belongs to this action.

The validation procedure's sole responsibility is to determine whether some trust chain it receives is valid. This is done by traversing the list from the front (trust point) to the rear, and validating each node. The validation for any node is done by the Verify call. If Verify is successful for every node in the chain, then the validation procedure will return true.

3.3 Explanation

The VHL enforces structural correctness and is responsible for verification. This was implemented as a simple command line server which supports connecting to another node, viewing and exchanging trust chains, and validating them. When the program starts, it calls the *AcceptConnections* procedure and the starts the interface. The output of the commands are piped to the program which implements the TPA layer.

It should be noted that the viewing of trust chains is not a feature of this *layer*, as these operations would normally be located higher in the stack (i.e. in the actual CA). Since we wanted a stand-alone application instead of trying to merge our code with a CA (for now) and having to build an entire PKI for testing, we put this functionality in the prototype.

The following is a brief description of the major sections of the piece of the prototype which implements the VHL.

The Logic. The pseudocode functions *AcceptConnections*, *JoinNetwork*, and *Validate* are in the VHL. Logically, the VHL is responsible for maintaining the tree topology as well as the other restrictions mentioned above. In order to accomplish this, it must facilitate some communication facilities other than those available via the TPA. These facilities are used for sending data such as roots, chains, and other variables, back and forth. We did this with simple sockets for the prototype, although something more secure (such as SSL) could be used. What we wanted to avoid was placing this traffic in the TPA, due to the lack of an intelligent routing protocol (this simplicity is what makes Gnutella attractive, however).

The Interface. The interface is quite simple, supporting only three commands:

1. *Connect ipaddress* attempts to establish a connection with the machine at ipaddress. The TPA layer attempts connection first, ensuring mutual authentication and secure key exchange. If successful, a socket is established to send and receive chains. Again, this socket is implemented for our prototype only.

2. *View* prints the current *chain* variable to stdout.

3. *Validate* walks the chain and attempts to validate it.

The Algorithms. Pseudocode implementations of the algorithms are found in an appendix at the end of this paper.

3.4 Analysis

In order to meet our claims of increased resilience and efficiency, we need to show the following:

Structural Correctness The client and server actions maintain the negotiated secrets in a hierarchical, acyclic fashion. This is necessary to get $O(logV)$ average running times for the Validate procedure. (We discuss this more in Section 3.4.1 below.)

Secret Distribution The functions maintain the property that the secrets or collaborating parties fr each collective are distributed throughout the collective. (We discuss this more in Section 3.4.2 below.)

3.4.1 Structural Correctness

The notion of structural correctness is used to show that the client and server actions maintain the secrets in a hierarchical, acyclic topology.

The hierarchy is maintained in two ways. First, note that the original two parties to connect form the root collective. This is the code path on lines 28-35 in Figure 15, and lines 27-33 in Figure 16. As additional nodes join one of these two nodes, they are integrated into the collective as they are at a distance of one from one of the key-holders for the collective.

As nodes make connections with collective members which are not key holders, new collectives are formed. Lines 34-40 in Figure 15 and 28-35 in Figure 16 represent the case where a caller is requesting to start a new collective with a party which does not belong to an already established collective. Lines 41-47 in Figure 15 and 36-42 in Figure 16 define the case where the caller is requesting to start a new collective. It is worth mentioning that a node which does not belong to an collective is the root of an collective which contains only itself.

Second, if there is a connection established between collectives, at least one of the collectives must be a root collective. If this were not the case, it would be possible for two leaf collectives to join, resulting in every node in both trees to be reachable from two different trust roots. This is exactly what we are trying to avoid, as this is the type of situation which leads to validation algorithms having to try multiple paths from an end point to a trust point.

Lines 41-47 in Figure 15 and 36-42 in Figure 16 are executed when the caller is a member of a root collective and Lines 48-58 in Figure 15 and 43-50 in Figure 16 are executed when the caller is attempting to join a root collective.

The algorithms maintain a topology which avoids cycles. Each node in every collective maintains a *my_root* variable which is set to the root collective. This variable is managed to always contain the node's root collective. As nodes attempt to make connections, they check this so as ensure that they do not attempt to make connections with nodes which already belong to the same tree (Line 7 in Figure 16).

3.4.2 Secret Distribution

Secret distribution is the principle means by which we meet our claim of increased resilience. As noted earlier, threshold cryptography provides many tools. We discuss two.

Secret Splitting The scheme we implemented in our initial prototype is perhaps the simplest. The client and server actions distribute the keys across the collective in such a way that they can be correctly reassembled, and used to sign statements from the collective.

Splitting the private key into x pieces and reassembling them when the collective needs to sign a statement does not invalidate the key. This technique is referred to as *Secret Splitting* [9], and for our prototype, we let $x = 2$. There are formal algorithms for this type of cryptosystem (e.g. Mediated RSA), and emerging architectures which employ it (e.g. Semi-trusted Mediators) [6].

One problem with this scheme is that we do not mandate redundancy of the key fragments. If Alice and Bob each hold a fragment and Alice has a power outage, the collective can no longer sign statements, at least until a new key can be established (which invalidates all the outstanding signed statements), or Alice powers up again.

The second problem with this scheme is that the key must be reassembled to be used. The only justification for this (albeit a weak one) is the fact that we have secure hardware. Without such machinery, this would totally expose the private key for a short time.

Secret sharing [15] would eliminate the first problem but not the second. Multi-Party RSA would eliminate both, which is why we plan to implement it in our next prototype.

Multi-Party RSA In opposition to a transient reassembling of the key and letting the result sign some statement, we envision a scheme which sends the statement around to each node holding a key fragment, and a portion of the signature being applied at that node. We plan to eventually use some instance of Multi-Party RSA to employ this technique in our system [16, 4].

Due to the algebraic properties of RSA, the algorithm lends itself to collaborative signature schemes quite naturally (an idea first proposed by Boyd [7]). Since then, the cryptographic community has generated a number of methods and protocols which utilize these properties. Samples of some such protocols and proofs of their security can be found in Bellare and Sandhu [4] (this is not meant to be a complete list).

Practically, using Multi-Party RSA in our system would allow a subset of members in the collective to possess key fragments, but would relax the assumption that they key is reassembled. The message is

instead broadcast to the collective and comes back signed by the keyholders.

4 The Trusted Peer Access Layer

The TPA implements a protocol for trusted peers which allows them to communicate in a secure fashion. By secure, we mean that all parties mutually authenticate one another, and that all traffic is encrypted by the secure coprocessor in such a way that an intruder could not discover the plain-text of the message — *not even if the intruder is host* (i.e., the computer which houses the coprocessor). The protocol need only provide a decentralized means to locate items stored among those participating in the network (e.g. Gnutella) [13].

Loosely, the TPA Layer is a peer access layer running in secure hardware (the IBM 4758 Secure Coprocessor). The protocol is implemented across two communicating programs, one running on the host and the other residing in the card.

The host code is responsible for 1) implementing a command line interface which allows users (or other programs) to issue commands, 2) connection management between nodes over standard sockets, and 3) handing the TCP payloads to the card for processing and putting response packets from the card onto a socket.

The card code is where the protocol's packet processing logic resides, as well as the routing tables and secrets. The idea is that the card manufactures outgoing packets, encrypts them using secrets negotiated by it and another coprocessor in the network, and sends a chunk of ciphertext along with a socket number to the host so that it may place the ciphertext into a TCP payload and fire it to the intended recipient. When a packet arrives, the host program pulls the ciphertext out of the TCP packet and sends it to the card for processing.

The following is a brief discussion of the four major phases of development that drove our prototype implementation.

Peer-to-Peer. Our first task was to evaluate existing true peer-to-peer protocols that allowed for distributed location without the aid of a central server (like Napster). Gnutella was immediately appealing due to its simplicity, community, and availability of documentation and open source implementations.

It is important to understand what exactly Gnutella is and what it is not. Gnutella is a protocol and nothing more. In v0.4 (the base specification), Gnutella defines five packet types (called descriptors), a format for headers, and six rules for routing. Gnutella is only used to locate files across a network, transfers are done out of band (usually over HTTP).

However, Gnutella is not an implementation of this protocol. There are several implementations in existence, some of which add to the basic protocol, but they implement at least the core functionality described above [2].

We chose to use the core protocol as well as it seemed to fit our needs (actually, the "Push" descriptor type exceeds our needs, so we eliminated it), and could help reduce our time to prototype.

Secure Hardware. The next task was to find a fairly mature code base that implemented an open source Gnutella *servent* (SERVer + cliENT). Our constraints was that it should run on Linux, and be command line driven in order that we may pipe commands to it (something GUI based schemes lack).

We chose Gnut v0.4.25 because it met our constraints, was well documented, and professionally coded [1].

We then undertook the task of finding which pieces of Gnut stayed on the host and which went to the

4758. As stated above, the socket management code remained on the host, and the packet logic and routing tables were ported to CP/Q++ (the native OS of the 4758).

At the end of this phase, we were able to observe 4758-enabled machines store strings and using the command line interface, were able to let other nodes locate them.

Adding Armor. In order to meet our definition of resilience, we had to implement a protocol for authentication and encryption, using the native cryptographic services provided by the 4758.

First, we consider authentication. The first element of our definition of resilience is that nodes must have a way to mutually authenticate one another. Bird et al. explain that nonce based protocols are most secure, and since the 4758 provides a random number generator, we decided to go this way. We ended up implementing FIPS 196, which is essentially the core of most authentication schemes used in practice (e.g. Secure Sockets Layer) [19, 5].

Second, we consider encryption. Once nodes have authenticated, the initiator sends four 3DES keys generated by its 4758 to be used for further encryption of all traffic between the two parties. Two of the keys are for encrypting messages and the other two are used for constructing a keyed Message Authentication Code for each message. We chose DES because it is fast.

The API. Lastly, in order to implement the algorithm above, we made the TPA layer provides the following primitives to higher layers:

1. The ability to place strings into secure storage in the card. For our purposes, these strings will be portions of cryptographic keys.

2. The ability to locate such strings on any machine which is participating in the network.

3. The ability to connect to other machines, authenticate (to) them, and exchange cryptographic secrets which will be used to encrypt all further transmissions.

4. The ability to negotiate a shared secret with another machine.

5 Current Status

We are currently in the process of implementing the prototype. The TPA is lacking encryption support for all traffic. However, we do currently support authentication and are able to locate strings (which would represent cryptographic keys) across machines in the lab.

The design of the VHL is complete, and we are in the process of writing the code. We have a large amount of pseudocode that needs to be implemented and tested. Once these tasks are complete, we plan to make the code available for public download.

6 Summary and Future Work

As it turns out, the result of this work has led to many more questions. In its current state, we plan to show proof of concept. As future work on this project progresses, we plan to address some of the questions that have been raised in order to evolve the system past being just a proof of concept.

We are considering many ways to enhance the architecture.

One direction is to examine data structures other than trees. Balanced trees (e.g. AVL or Red-Black trees), and directed acyclic graphs could possibly lead to better solutions.

Another direction is to examine different routing protocols in the TPA. Specifically, reverse path forwarding or some other protocol which is a little smarter than just broadcasting could be interesting.

Our current architecture uses secret splitting, but (as mentioned) cryptography offers more advanced tools. We plan to extend the prototype to use Multi-Party RSA, allowing the message to travel around the collective to be operated on instead of the key being reassembled at one machine.

We plan to make much use of the virtual hierarchy technique in our current Marianas project, which explores using peer-to-peer techniques and secure hardware to build survivable trusted third parties (and which which recently earned an NSF Trusted Computing research grant).

References

[1] Gnut documentation. `www.gnutelliums.com/linux_unix/gnut/doc/gnut.html`.

[2] The gnutella protocol specification v0.4. `www.clip2.com/GnutellaProtocol04.pdf`.

[3] R. Anderson and M. Bond. Api-level attacks on embedded systems. *Computer*, October 2001.

[4] Mihir Bellare and Ravi Sandhu. The security of a family of two-party rsa signature schemes. `citeseer.nj.nec.com/bellare01security.html`.

[5] R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, and M. Yung. Systematic design of a family of attack-resistant authentication protocols, September 1992.

[6] D. Boneh, X. Ding, G. Tsudik, and C. Wong. A method for fast revocation of public key certificates and security capabilities. In *10th USENIX Security Symposium*, pages 297–308. USENIX, 2001.

[7] C. Boyd. Digital multisignatures. In *Cryptography and Coding*, pages 241–246. Oxford University Press, 1989.

[8] Y. Elley, A. Anderson, S. Hanna, S. Mullan, R. Perlman, and S. Proctor. Building certification paths: Forward vs. reverse. In *Network and Distributed System Symposium Conference Proceedings*, 2001.

[9] H. Feistel. Cryptographic coding for data-bank privacy. Technical Report RC 2827, IBM Research, Mar 1970.

[10] R. Housley and T. Polk. *Planning for PKI*. Wiley, 2001.

[11] J. Hubaux, L. Buttyan, and S. Capkun. The quest for security in mobile ad hoc networks. In *Proceedings of the 2nd ACM Symposium on Mobile Ad Hoc Networking and Computing*, October 2001.

[12] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security - Private Communication in a Public World*. Prentice Hall, 2nd edition, 2002.

[13] D. Nicol, S. Smith, C. Hawblitzel, E. Feustel, J. Marchesini, and B. Yee. Survivable trust for critical infrastructure. In *Internet2 Collaborative Computing in Higher Education: Peer-to-Peer and Beyond.*, 2002.

[14] T. Polk and N. Hastings. Bridge certification authorities: Connecting b2b public key infrastructures. In *PKI Forum Meeting Proceedings*, June 2000.

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

[16] G.J. Simmons. An introduction to shared secret and/or shared control schemes and their application. *Contemporary Cryptology: The Science of Information Integrity*, pages 615–630, 1992.

[17] S.W. Smith and S.H. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31:831–860, April 1999. Special Issue on Computer Network Security.

[18] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, third edition, 1996.

[19] U.S. Dept. of Commerce / National Institute of Standards and Technology. *Entity Authentication Using Public Key Cryptography*, February 1997. FIPS PUB 196.

[20] Robin Wilson. *Introduction to Graph Theory*. Addison Wesley, 1997.

[21] L. Zhou and Z.J. Haas. Securing ad hoc networks. *IEEE Network*, pages 24–30, November/December 1999.

Pseudocode

Each entity maintains the global variables listed in Figure 13.

my_root is used to store the root of this node's trust chain, represented as a signed statement issued from the collective owning the "root" key.

my_signature is used to hold the signature of collective. This is useful in constructing and maintaining the chain variable, as well as for determining if parties belong to the same collective.

chain is a list of statements signed by the collectives, which represent certificates in our prototype. Each node in this list is a triple of the form

$$< root, signature\ n, public\ key\ n+1 >$$

such that the public key contained in the nth certificate can be used to verify the next certificate in the list. In the case where the node is the first in the list, the public key may be used to verify the signature directly (i.e. the first node is a self signed certificate), as well as the next one. The important fact to note is that the order of this list maintains the property that Validate can traverse the topology suggested by this list efficiently. This is done by carefully controlling how and where entries are added to the list.

num_connections is used to track the number of current connections, which is vital in keeping the number of nodes n in the collective below some constant maximum. Otherwise, if n would get too large, it will take longer than $O(1)$ to reconstitute the private key, as the broadcast to the collective would get expensive. This implies that issuing a certificate would be quite costly.

This was of little concern in our prototype, as we did not have enough cards to test the boundaries of Gnutella's scalability. Furthermore, our prototype did not issue actual certificates, it only maintained the *chain* variable, which is only a representation of a certificate chain.

have_key is a boolean that determines whether the node owns a key fragment.

Explanation of Auxiliary Functions

The functions described in this section are used by the client and server actions, as well as the validation procedure. The boolean evaluation functions were added in an effort to make the pseudocode as mnemonic as possible.

SendConnectionRequest(ipaddress) sends a request to the TPA to establish a connection with the machine residing at *ipaddress*. The TPA layer sends the string "GNUTELLA CONNECT/0.4" per the protocol specification. The 0.4 is the protocol version number. If the servent is accepting connections, it responds with a random number generated by the 4758, which then begins the FIPS 196 authentication process.

Authenticate() polls the TPA layer to determine whether the connection was completed. Successful connection of two nodes in the TPA layer enforces successful authentication.

SendMyHaveKeyWhenRequested() sets the layer into a loop until 1) it receives a request for the value of the *have_key* boolean value, or 2) timeout occurs.

SendMyRootWhenRequested() sets the layer into a loop until 1) it receives a request for the node's *my_root*, or 2) timeout occurs.

SendMySignatureWhenRequested() sets the layer into a loop until 1) a request for the *my_signature* variable is received, or 2) timeout occurs.

SendMyChainWhenRequested() sets the layer into a loop until 1) a request for the *chain* variable is received, or 2) timeout occurs.

RequestHeHasKey() generates a request for the value of the *have_key* variable and sends it to the

machine on the other side of the connection established in the client or server action.

RequestHisTrustRoot() generates a request for the value of the *my_root* variable and sends it to the connected machine.

RequestHisSignature() generates a request for value of the *my_signature* variable and sends it to the connected machine.

RequestHisChain() generates a request for the *chain* variable and sends it to the connected machine.

NegotiateSecret(new_root, public_key) calls a function in the TPA layer which initiates a key negotiation between the two parties. Once the key is agreed upon, each party stores one half of this key inside of the 4758. In actuality, a pair of the form:

$$< tag, key\ fragment >$$

pair is stored so that the key may be found by knowing only the tag. This function returns a message signed by the negotiated key that may be used as the value *my_root* variable, as well as a public key.

MakeNewChainNode(root, signature, public_key) constructs the triple:

$$< root, signature\ n, public\ key\ n+1 >$$

AppendChain(c) inserts the chain *c* into the back of the local *chain* variable.

PrependChain(c) inserts the chain *c* into the front of the local *chain* variable.

UpdateRootAndPrependChain(r, c) sends the root *r*, and the chain *c* to all the connections except the most recent one. This function is called in the case when collectives are merging and the members (as well as any subtrees) need to be informed of the new root and chain information.

Verify(c) is the core of the validation algorithm. It takes one entry in the *chain* variable (*c*), and attempts to verify the signature using the public key contained in the node *c-1*. In the case where Verifying is working with the first certificate in the list, the public key may be used to directly verify the signature, as well as the signature of the next certificate.

NoOneHasKey() returns

$$(have\_key == false \&\& he\_has\_key == false)$$

DifferentColl() returns

$$(my\_signature! = his\_signature)$$

HeIsRootCollective() returns

$$(his\_root == his\_signature)$$

IAmRootCollective() returns

$$(my\_root == my\_signature)$$

```
1.    my_root         = 0
2.    my_signature    = 0
3.    my_chain        = 0
4.    num_connections = 0
5.    have_key        = false
```

Figure 13 The global variables used in the following procedures. (Note that we use value 0 as NULL: the lack of an instance of this data type.)

```
Procedure Validate( Chain *c )
 1.    current = NULL;
 2.    if (c->first != NULL)
 3.      current = c->first
 4.    while (current != NULL)
 5.    {
 6.      success = Verify( current )
 7.      if (success == false)
 8.        return false
 9.      current = c->next
10.    }
11.    return true
```

Figure 14 Validation pseudocode.

```
Procedure AcceptConnections()
 1.    for(;;)
 2.    {
 3.      if (received_request && num_connections < MAX_CONNECTIONS)
 4.      {
 6.        SendMyRootWhenRequested( my_root )
 7.        SendMyHaveKeyWhenRequested( have_key )
 8.        SendMySignatureWhenRequester( my_signature )
 9.        he_has_key = RequestHeHasKey()
10.        his_signature = RequestHisSignature()
11.        if (have_key == false && he_has_key == true && my_root == 0)
12.        {
13.          his_root = RequestHisTrustRoot()
14.          my_root = his_root
15.          my_signature = his_signature
16.          his_chain = RequestHisChain()
17.          PrependChain( his_chain )
18.          if (num_connections > 1)
19.            UpdateRootAndPrependChain( my_root, my_signature, my_chain )
20.          continue
21.        }
22.        else if (NoOneHasKey() || (DifferentColl() && his_signature != 0))
23.        {
24.          his_root = RequestHisTrustRoot()
25.          NegotiateSecret( new_root, public_key )
26.          have_key = true
27.          if (his_root == 0 && my_root == 0)
28.          {
29.            my_root = new_root
30.            my_signature = new_root
31.            node = MakeNewChainNode( new_root, my_signature, public_key )
32.            AppendChain( node )
33.          }
34.          else if (his_root != 0 && my_root == 0)
35.          {
36.            my_root = his_root
37.            my_signature = new_root
38.            his_chain = RequestHisChain()
39.            AppendChain( his_chain )
40.          }
41.          else if (!IAmRootCollective() || HeIsRootCollective())
42.          {
43.            temp_signature = my_signature
44.            my_signature = new_root
45.            node = MakeNewChainNode( my_root, temp_signature, public_key )
46.            AppendChain( node )
47.          }
48.          else
49.          {
50.            my_root = his_root
51.            my_signature = new_root
52.            his_chain = RequestHisChain()
53.            DeleteChain( my_chain )
54.            AppendChain( his_chain )
55.            if (num_connections > 1)
56.              UpdateRootAndPrependChain( my_root, my_signature, my_chain )
57.            continue
58.          }
59.        }
60.        SendMyChainWhenRequested()
61.      }
62.    }
```

Figure 15 Server Action pseudocode.

```
Procedure JoinNetwork( ipaddress )
 1.    SendConnectionRequest( ipaddress )
 2.    his_root = RequestHisTrustRoot()
 3.    he_has_key = RequestHeHasKey()
 4.    his_signature = RequestHisSignature()
 5.    SendMyHaveKeyWhenRequested( have_key )
 6.    SendMySignatureWhenRequested( my_signature )
 7.    if ((my_root != his_root || his_root == 0) &&
 8.        (!DifferentColl() || ( NoOneHasKey() &&
 9.          (HeIsRootCollective() || IAmRootCollective())))
10.    {
11.      if (have_key == true && he_has_key == false && his_root == 0)
12.      {
13.        SendMyRootWhenRequested( my_root )
14.        SendMyChainWhenRequested()
15.      }
16.      else if (NoOneHasKey() || (DifferenttColl() && my_signature != 0))
17.      {
18.        SendMyTrustRootWhenRequested( my_root )
19.        NegotiateSecret( new_root, public_key )
20.        have_key = true
21.        if (his_root == 0 && my_root == 0)
22.        {
23.          my_root = new_root
24.          my_signature = new_root
25.          his_chain = RequestHisChain()
26.          AppendChain( his_chain )
27.        }
28.        else if (his_root == 0 && my_root != 0)
29.        {
30.          temp_signature = my_signature
31.          my_signature = new_root
32.          node = MakeNewChainNode( my_root, temp_signature, public_key )
33.          AppendChain( node )
34.          SendMyChainWhenRequested()
35.        }
36.        else if (IAmRootCollective())
37.        {
38.          my_signature = new_root
39.          his_chain = RequestHisChain()
40.          DeleteChain( my_chain )
41.          AppendChain( his_chain )
42.        }
43.        else if (HeIsRootCollective())
44.        {
45.          temp_signature = my_signature
46.          my_signature = new_root
47.          node = MakeNewChainNode( my_root, temp_signature, public_key )
48.          AppendChain( node )
49.          SendMyChainWhenRequested()
50.        }
51.      }
52.      else
53.      {
54.        my_root = his_root
55.        my_signature = his_signature
56.        his_chain = RequestHisChain()
57.        PrependChain( his_chain )
58.      }
59.      if (num_connections > 1)
60.        UpdateRootAndPrependChain( my_root, my_signature, my_chain )
61.    }
62.    return
```

Figure 16 Client Action pseudocode.