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

John Marchesini and Sean Smith
Department of Computer Science *
Dartmouth College
Technical Report TR2002-416

{carlo,sws}@cs.dartmouth.edu

February 4, 2002

Abstract

In Public Key Infrastructure (PKI), the simple, monopolistic CA model works fine until we consider the real world. Then, issues such as scalability and mutually suspicious organizations create the need for a multiplicity of CAs, which immediately introduces the problem of how to organize them to balance resilience to compromise against efficiency of path discovery.

However, security has given us tools such as *secure coprocessing*, *secret splitting*, *secret sharing*, and *threshold cryptography* for securely carrying out computations among multiple trust domains; 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 CAs, 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.

1 Introduction

1.1 The Problem

Current architectures for PKI attempt to provide *resilience* and *efficiency*. In this context we take resilience to mean the ability of a certificate issuing entity, a *Certificate Authority* (CA), to protect its private key from disclosure, and efficiency to mean the running time of some algorithm which attempts to verify a *certificate* (typically, something binding a name to an key), or chain of certificates (i.e. *trust chain*) issued by CAs in the architecture. These are most often competing goals.

Practically, current architectures manipulate the arrangement of the CAs in order to achieve different goals.

Many current architectures impose a rigid structure on the CAs so 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).

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 author-

ity in one centralized place. The implication is 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.

We believe that both properties—efficiency and resilience—are important to any PKI, and 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 current architectures fail to provide both.

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 [5]. Our concern however, is the underlying organization of CAs, and how they may be arranged to achieve efficiency and resilience.

Common Architectures.

Hierarchies and *meshes* are canonical examples of the structured and unstructured approaches, respectively.

Traditionally, *Hierarchies* are used to achieve $O(logV)$ (where V is the number of CAs) verification time. The drawback of this approach is that it forces the CA's private key to be stored in a central location. If an intruder were to compromise the root CA, the entire PKI must go offline 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).

Mesh PKI architectures have been developed in an effort to avoid this single point of failure. However, the non-deterministic nature of peer-to-peer organization increases the path verification algorithm significantly. 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), 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.

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 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. One common approach is to cross certify enterprise PKIs through peer-to-peer relationships, which 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 [14]).

The Bridge allows each of the enterprise PKIs to cross certify to it, resulting in a star topology and reducing the number of relationships to n. While a seemingly attractive solution, the Bridge architecture does not solve path validation issues in that each of the enterprises themselves may be Hierarchies, Meshes, etc [7] [9].

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

1. Trust chains produced by any of the entities may be verified in an efficient manner.

2. The secrets (private keys) can not be found in any one place, and the fragments are fairly randomly distributed throughout the topology.

1.2 Our Approach

Terms The mechanism we propose which accomplishes this task is a *virtual hierarchy*, a logical hierarchy formed in a peer-to-peer network. A virtual hierarchy is a tree in which nodes represent certificate issuing/message authenticating entities, and edges represent trust relationships between them. While each node represents one logical entity in the virtual hierarchy, it is comprised of more than one entity in the physical layer, none of which may perform the certificate issuing/message authentication task alone as none of the entity holds the entire private key.

We use the term *collective* for the physical group of entities which acts as a single node in the virtual hierarchy.

Physical Layer The physical layer is a peer-to-peer network of secure coprocessors. 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.

2 Overall Structure

We approached the problem in two stages, the first being to implement a peer access layer which allows secure coprocessors [11] (we use IBM 4758s [1]) 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 1.

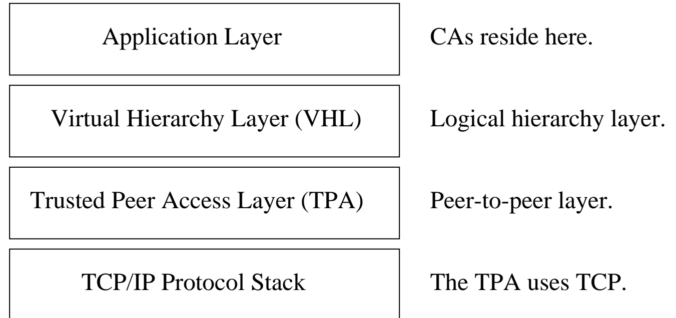


Figure 1: The protocol stack.

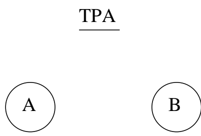
Our prototype implements the VHL and the TPA. The prototype version of the VHL contains a command line interface so that we don't need to integrate with a CA at this stage in development. The peer access layer running inside of the IBM 4758s is depicted as the TPA layer, and the algorithms which construct 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 following example, 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.

3 Virtual Hierarchy Layer

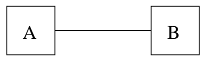
From the highest level, the virtual hierarchy (i.e. the logical hierarchy in the peer-to-peer network)

[1] Recently, a security vulnerability 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.



VHL

Step 1: Neither of the two machines hold any secrets, as denoted by the circle.

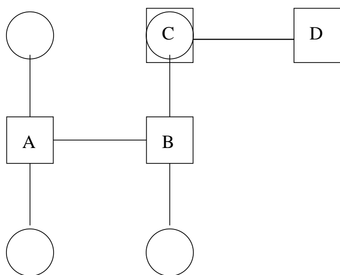


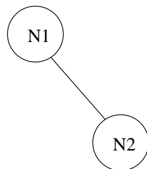
Step 2: A connects to B and they negotiate a secret. The two parties now each hold half of the secret, as denoted by the square. A collective is formed and a node is established in the virtual hierarchy.

A

B

Step 3: 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.



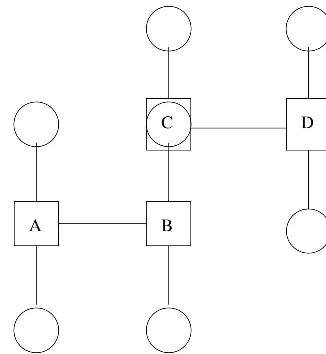


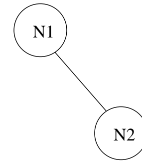
Step 4: Machine D makes a connection to machine C, causing them to negotiate a new secret and each store half of it. This operation forms a new collective and a new node N2 in the virtual hierarchy.

Machine C is now in both collectives.

TPA

VHL





Step 5: 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.

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. This trust root (negotiated secret) is then stored in fractional pieces 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 role 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. The result of spreading pieces of the secret among a group of peers alleviates the single point of failure problem.

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 we make no claims that

we are achieving a more dynamic solution than the status quo. If two CAs join in any of the traditional architectures, the communities will require some level of re-keying. Our approach requires some re-keying as well.

3.1 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 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 [12]). 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 hold a key for one collective may not hold keys for another. This is done by placing some restrictions on the caller. Allowing nodes to hold keys for two collectives simultaneously forces that node to hold two separate trust chains and it 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

Lastly, for ease of implementation, and for the sake of reducing network traffic, we take the approach that pieces of the negotiated secret are scattered throughout the collective. Many cryptographic techniques can enable this behavior. For simplicity, we consider simple secret splitting [6]. 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 an *ephemeral* 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 threshold cryptography schemes [10]. For purposes of this analysis and prototype, we assume that the secret splitting is sufficiently representative. We discuss implications of these other techniques in Section 3.4.2, and plan to address these topics in future work.

3.2 The Algorithms

The pseudocode procedures in the Pseudocode section at the end of this paper maintain the invariants put forth in our objectives. The client and server actions (Figure 4 and Figure 5) 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 prototype.

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 is a feature, 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 for 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, by noting that the original two parties to connect form the root collective. This is the code path on lines 28-35 in Figure 4, and lines 27-33 in Figure 5. 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 4 and 28-35 in Figure 5 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 4 and 36-42 in Figure 5 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.

Secondly, 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 4 and 36-42 in Figure 5 are executed when the caller is a member of a root collective and Lines 48-58 in Figure 4 and 43-50 in Figure 5 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 5).

3.4.2 Secret Distribution

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* [6], 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) [4].

The problem with this scheme is that we don't 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.

A better solution would be to allow multiple nodes to key fragments, in an effort to produce a fair amount of key redundancy in the collective. We flag this for future work.

In opposition to ephemerally reassembling the key and letting the result sign some statement, is to send the statement around to each node holding a key fragment. There are schemes such as secret sharing and threshold cryptography which employ this technique[10]. This requires a fair amount of extra traffic, as fragment holders are unknown, and thus require a broadcast for each stage of operation. This scheme would require broadcasting the statement to be signed a number of times, so that the node holding the first stage key fragment may perform the appropriate operation, then the node holding the second may operate on the result, and so on.

4 The Trusted Peer Access Layer

The TPA implements a protocol for *trusted third parties* 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 trusted third party in such a way that an intruder could not discover the plaintext of the message — *not even if the intruder is host* (that is, the computer which houses the card). The protocol need only provide a decentralized means to locate items stored among those participating in the network (e.g. Gnutella) [8].

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 cipher-text along with a socket number to the host so that it may place the cipher-text into a TCP payload and fire it to the intended recipient. When a packet arrives, the host program pulls the cipher-text 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) [13, 3].

Secondly, we consider encryption. Once nodes have authenticated, the initiator sends four DES 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 extending the prototype to use a threshold cryptography scheme where the message would travel around the collective to be operated on instead of the key being reassembled at one machine. It may also be useful to merge this with an actual CA and set up a PKI to further prove the concept.

References

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

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

[3] 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.

[4] 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.

[5] 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.

[6] H. Feistel. Cryptographic coding for databank privacy. Technical Report RC 2827, IBM Research, Mar 1970.

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

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

Pseudocode

Each entity maintains the global variables listed in Figure 2.

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 verifiy 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 estab-

lished 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 2: 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 3: 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 4: 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 5: Client Action pseudocode.