An Attack-resistant, Scalable Name Service

Raph Levien and Alexander Aiken
EECS Department
UC Berkeley

Abstract

Deployment of transparent cryptography on the Internet requires a large-scale, secure name service for mapping hostnames and e-mail addresses to public keys. Existing proposals suffer from numerous single points of vulnerability as well as high certification cost and scalability problems. We present a distributed name service designed to operate correctly even when a significant fraction of servers are compromised, and designed to be efficient in latency and use of CPU, bandwidth and storage as it scales to Internet size.

Keywords: Name Service, Attack Resistance, Certificates, PKI.

1 Introduction

A long-standing goal of security research has been deployment of *transparent cryptography:* the automatic encryption and decryption of messages traveling over communication channels. Encryption technology has been successfully deployed at the application (PGP, S/MIME), connection (SSL), and transport (IPSEC) levels. More elusive is a satisfactory Public Key Infrastructure (PKI), which solves the problem of finding the correct public key to use for a given message[Bra97].

A PKI provides a secure database with the following services:

- *registering* a public key for a new name
- *updating* the public key of an existing registered name
- *lookup* of the public key registered for a given name

A difficulty facing any PKI is that the problem of determining who owns the rights to a given name is not well-defined; rather, it is the subject of social, economic, legal, and political issues (as amply demonstrated by the controversy around the management of the root namespace of DNS[Post99]). The approach taken by existing PKI designs is to assign authority over the namespace to one or more *certification authorities* (CA's), generally composed of *root CA's* that delegate authority over some or all of the namespace to various subsidiary CA's. A consequence of this design decision is the *root vulnerability*—the compromise of the public key of any of the root CA's leads to catastrophic security failure.

Root vulnerability is a serious concern; even though individual CA keys tend to be fairly well protected, Netscape 4.07 ships with 45 root CA keys enabled. Further, the growing popularity of CA software for off-the-shelf platforms (such as Entrust/PKI and Entegrity's Notary) implies that CA's will be deployed in contexts where it is difficult or impossible to protect the CA key with a high degree of assurance.

Often, the CA given root authority is implementing a fairly simple policy. For example, VeriSign's Class 1 certificates are issued on a strictly first-come, first-served basis. The InterNIC issues second level domains on an effectively first-come, first-served basis, with the possibility of revoking the name (and issuing it to someone else) for non-payment of the bill. Actual implementation of these policies is delegated to a CA. We have only VeriSign's word that they conform to the policy. If the CA key were to be compromised, then an attacker could rearrange names and keys completely at will, with no regard to the policy.

It is possible to implement these policies and that avoid the risk of root compromise. The naming service presented in this paper factors the PKI problem into two sub-problems: a *policy language* for expressing policies, and a distributed *trusted third party* (TTP) for implementing policies. In cases where the complex blend of factors controlling ownership of a piece of namespace can be distilled into a well-defined policy, our naming service can provide much higher assurance with lower certification cost than existing PKIs.

A design factored in this way can only be a successful PKI if two goals are met: the policy language should be rich enough to express a range of policies that are useful in the real world and the distributed TTP should implement these policies with a high degree of assurance. Most of the paper is devoted to presenting a policy language and distributed TTP meeting these goals.

In a traditional PKI, the authority granted to a CA extends symmetrically to registration and updating—the CA has equal power to both register and update name/key bindings. However, in practice there are many interesting *asymmetrical* policies where the power to update is more restricted than the power to register. The first-come, first-served policy is an extreme example of this asymmetry; authority to register new names is trivially granted, but authority to update existing names is never granted. The key contribution of the policy language is the *policy constraint* mechanism, which allows expression of policies ranging from the symmetrical case to first-come, first-served, with many interesting points in between. This mechanism is discussed in detail in Section 2, along with the rest of the policy language.

The distributed TTP is implemented as a federation of *servers* in a network. The results of queries are determined by a majority voting system. A major problem with voting is that an attacker can overwhelm the system by introducing a large number of phony servers (which could exist on the same physical machine, reducing the cost of the attack). We designate this the *server spam attack.* To overcome the server spam problem, we develop a *group trust metric,* which chooses a subset of servers considered trustworthy. The design of the TTP is presented in Section 3 and the group trust metric is presented in Section 3.1.

Section 4 fills in a number of engineering details regarding transport and revocation, and presents experimental results showing that the system scales to Internet size. Optimum scaling is achieved when the number of servers scales as the square root of the number of users.

2 The Policy Language

One key component of the system is a policy language for controlling registrations and updates.

The policy language is designed to meet these goals:

- Different policies can be specified for different areas of the namespace.
- Certification expense is low if security against unauthorized registrations need not be high.
- There is high security against unauthorized registrations if higher certification expense is tolerated.
- The system has good security against unauthorized updates.
- The system has low vulnerability to root compromise.

Before defining the policy language itself, we must define the space of *names* and *requests* over which the policies are defined. Names are familiar Internet hostnames and e-mail addresses, such as "raph@cs.berkeley.edu." Names are hierarchically structured. The parent of "raph@cs.berkeley.edu" is "cs.berkeley.edu". Similarly, the parent of "cs.berkeley.edu" is "berkeley.edu". Names with children are *domains.*

A request is either a *registration* or an *update* request, containing a name, a value to be bound to that name, and policies to be associated with that name. A value is generally a public key bound, but can also be a policy fragment or group, bound to specialized policy and group names. Additionally, requests are accompanied by signatures from zero or more public keys. Lookups are not considered in the policy language, as this design considers the database to be public, thus lookups always authorized.

A request is processed as follows:

- The signatures on the request are verified, resulting in a list of public keys that signed the request.
- The body of the request is used to retrieve an appropriate *policy* from the database.
- The policy is evaluated over the list of public keys signing the request, resulting in a boolean value.
- If the boolean value is *true,* then the request is applied to the database.

This workflow is similar to that of PolicyMaker[BFL96] and other trust management systems.

For registration requests, the appropriate policy is the *registration policy* of the name's parent. For update requests, the appropriate policy is the *update policy* of the name. The update policy for a name is set on registration of the name (i.e. it cannot itself be directly updated). The registration policy for a domain is set on registration and can be updated.

The policy language is defined as follows:

$$\begin{array}{rcl} \textit{policy} & ::= & \textit{int}\ '\textit{of}'\ \textit{group} \\ & | & \textit{policy}\ '\wedge'\ \textit{policy} \\ & | & \textit{policy}\ '\vee'\ \textit{policy} \\ & | & 0 \\ & | & 1 \\ & | & '*'\ \textit{policy-name} \end{array}$$

$$\begin{array}{rcl} \textit{group} & ::= & '\{'\ \textit{key}\ '\}' \\ & | & \textit{group}\ '\cup'\ \textit{group} \\ & | & \textit{group}\ '\cap'\ \textit{group} \\ & | & '*'\ \textit{group-name} \end{array}$$

$$\begin{array}{rcl} \textit{key} & ::= & \textit{key-hash} \\ & | & '*' \ \textit{key-name} \end{array}$$

$$\textit{policy-name} \ ::= \ \textit{id} \ '?' \ \textit{name}$$

$$\textit{group-name} \ ::= \ \textit{id} \ '@@' \ \textit{name}$$

$$\textit{key-name} ::= \textit{name}$$

The semantics of the policy language are given in Figure 1. If P represents a policy, then $[\![P]\!]$ represents the interpretation of this policy, represented as a predicate over sets of keys. A request signed by the set of keys T is accepted by the policy if and only if $[\![P]\!](T)$ is true. $[\![P]\!]$ is a monotone predicate.

An example policy is "$2\ \textit{of}\ \{a, e, i, o, u\}$". A request signed by a and e meets the policy, but one signed by a, b, and c does not. Logical connectives have their obvious meanings. For example, "$2\ \textit{of}\ \{a, e, i, o, u\} \vee 2\ \textit{of}\ \{b, c, d, f, g\}$" is met by a request signed by b and c, but not one signed by a and b. The policy 0 rejects all requests, while the policy 1 accepts all requests. Thus, a registration policy of 1 implements first-come, first-served, while an update policy of 0 ensures that the name cannot be updated.

The keys can either be specified directly (as the hash of the public key itself) or indirectly through the *late binding* mechanism. Late binding allows keys to be specified by name as well, which is useful when the key for a given role may change. A server evaluating a policy simply looks up the name in the database, replacing the name reference with the retrieved value. For example, the policy "*1 of {*raph@cs.berkeley.edu, *root@cs.berkeley.edu}*" is resolved into "*1 of {0x12345, 0x6789a}*" (assuming those are the public keys for the two names), then evaluated.

The language provides additional syntax for groups. For example, one might choose to register "`{*root@cs.berkeley.edu, *postmaster@cs.berkeley.edu}`" as the value for the name `admins@@cs.berkeley.edu`. Any reference to "`*admins@@cs.berkeley.edu`" would get replaced with "`{*root@cs.berkeley.edu, *postmaster@cs.berkeley.edu}`", and finally resolved to the actual keys bound to the names.

Finally, the language allows for late bindings of fragments of policy language. As mentioned above, update policies may not themselves be directly updated. However, late binding of policies provides a mechanism to do this indirectly. For example, suppose that we wanted to register a new key 0x98765 for `foo@bar.org`, so that initially it can be updated by any one of `admins@@bar.org`, but that this update policy can be changed by `root@bar.org`. The requests are shown in Figure 2.

The policies presented above provide protection against unauthorized registrations and updates, but no recourse if an unauthorized registration slips through.

2.1 Policy Constraints

Suppose that an attacker manages to defeat a registration policy, either through compromising some keys or through fraud. In general, the attacker will register an update policy for the new registration under his own control, and specifically *not* under control of the domain owner. The asymmetry between registration and updates allows our design to avoid root vulnerabilities, but this scenario shows that the restrictions on power to update may be too limiting.

Our solution is to add policy constraints to the policy language. Briefly, the domain owner chooses a policy constraint that ensures that update policies for names registered within the domain allow the name to be changed if needed. New registrations are not accepted unless the update policy for the newly registered name meet the parent's policy constraint.

Our policy constraint language is a small extension to the policy language. A policy P *satisfies* the policy constraint P' if it is weaker than P', i.e. every set of keys accepted by P' is also accepted by P.

A policy constraint of 0 is satisfied by all policies (i.e. all policies are weaker than 0). In conjunction with a registration policy of 1, which is satisfied by all requests, this policy constraint implements a strict first-come, first-serve policy, in which the first person to claim the name gets all rights to it, with all rights regarding changes given to the registrant and none given to the domain owner.

By contrast, setting the policy constraint equal to the registration policy (for example, "$2\ \textit{of}\ *\textit{registrars}$") implements a symmetrical policy, in which authority applies identically to registrations and updates.

Perhaps most interesting are policy constraints intermediate between these two extremes. For example, a registration policy of

$$1\ \textit{of}\ \{*\textit{registrar@icann.org}, *\textit{ecommerce@internic.net}\}$$

but an update policy of

$$3\ \textit{of}\ *\textit{arbitrators@@uspto.gov}$$

$$
\begin{array}{rcl}
[\![k \textit{ of } S]\!](T) & = & |S \cap T| \geq k \\
[\![P \vee P']\!](T) & = & [\![P]\!](T) \vee [\![P']\!](T) \\
[\![P \wedge P']\!](T) & = & [\![P]\!](T) \wedge [\![P']\!](T) \\
[\![0]\!](T) & = & false \\
[\![1]\!](T) & = & true
\end{array}
$$

Figure 1: Semantics of the policy language.

```
register updatepolicy?foo@bar.org = 1 of *admins@@bar.org,
  update-policy = 1 of {*root@bar.org}

register foo@cs.bar.org = 0x98765,
  update-policy = *updatepolicy?foo@bar.org
```

Figure 2: Requests for late-bound update policy.

would allow names to be cheaply and easily registered by more-or-less secure e-commerce servers, but require three trusted arbitrators to undo registrations that are found to be fraudulent.

A small extension to the policy constraint language provides for some negotiation of a set of keys mutually trusted by the domain owner and the registrant. The additional syntax "*int 'choose' group*" can be specified for groups appearing in policy constraints. For each such k *choose* S appearing in the policy constraint, the registrant can supply an S' such that $|S'| = k \wedge S' \subseteq S$.

An example formulated in terms of trust may be helpful. Let us imagine that the owner of the domain trusts that both vowels and consonants can be counted on to undo an unauthorized registration. However, some registrants may fear that consonants are corrupt and may undo the registration. Others may fear the same from vowels.

An appropriate policy constraint is:

$$3 \textit{ of } (5 \textit{ choose } \{a, b, c, d, e, f, g, i, o, u\})$$

Thus, a consonant-fearing registrant may register with an update policy of "3 *of* $\{a, e, i, o, u\}$". In this case, the registrant provides the instantiation $\{a, e, i, o, u\}$. It is syntactically obvious that the policy then meets the policy constraint.

Given a policy constraint of the form k *of* $(k'$ *choose* $S)$, the registrant is protected against unwanted updates if he can trust $1 + k' - k$ of the keys in S. The domain owner is guaranteed the ability to force updates if he can trust $|S| + k - k'$ keys.

In summary, policy constraints provide a mechanism for specifying policies that are acceptable to *both* domain owner and registrant.

3 Distributed TTP for implementing policies

The basic concept of the distributed trusted third party is that transactions (registration, update, lookup) are distributed to a subset of servers, each of which independently performs the transaction. The result of lookup is determined by a majority response. All computations are performed starting with the complete list of servers in the system. This list is distributed and kept up to date using an NNTP-like [RFC977] broadcast protocol (see Section 3.5). From this list, two subsets of servers are computed: First, the choice of "trustworthy" servers relied upon to contribute to the majority. Second, the choice of "responsible" servers for each name. Careless choices of either of these subsets may leave the system as a whole open to attack.

This section proposes specific algorithms for these choices, and concludes with a security proof that they guarantee high assurance even in the face of massive attack.

3.1 Choice of trustworthy servers

The choice of trustworthy servers is one of the main technical contributions of this paper. The goal is to maximize the number of servers chosen that are actually legitimate, and to minimize the number of bad servers chosen, even in the face of massive attack. Trust is always computed relative to a *seed*, which is a server (or, more generally, set of servers) locally known and trusted by the client. As in all PKI designs, selection of the trust root is critical for security. In particular, if all servers in the seed are under control of the attacker, then no security is guaranteed.

The attack model is as follows. The set of servers is partitioned in to the following three classes:

- *Good* servers always behave as expected, and all peer certificates are for other good or confused servers.
- *Confused* servers behave as expected but may have issued peer certificates for bad servers.
- *Bad* servers are considered to be under the control of an attacker.

There are no certificates directly from good servers to bad ones (see Figure 3). The servers that behave as expected are partitioned into "good" and "confused" on the basis of whether or not certificates exist to bad servers.

The next step in evaluating the trust metric is to assign a *capacity* c_x to each node x based on the shortest-path distance from the seed to x. These capacities are also a tunable parameter. The capacity of the seed should be equal to the number of good servers in the network, and the capacity of each successive level should be the previous level's capacity divided by the average outdegree. In the example shown in Figure 4, the total number of servers is fixed at 20, and the outdegree at 3.

In cases when the seed consists of more than one server, then it is represented in the graph as a "virtual" server with edges to all the servers in the seed. Clients have the best assurance of security when all servers in the seed are known to be good, although the system can tolerate a fraction of even seed servers being bad.

From the certificate graph (in which every server corresponds to a node and every peer certificate to an edge), a graph with capacity-constrained edges is constructed, according to the general rule shown graphically in Figure 5. Briefly, each original node A is split into $A-$ and $A+$, and an edge from $A-$ to $A+$ is added, constrained by the capacity assigned in the previous stage to A, minus one. An edge in the original graph from A to B is represented by an edge from $A+$ to $B-$. Finally, a unit capacity edge is added from each $A-$ to a "supersink" node. The sum of the capacity from $A-$ to $A+$ and the capacity from $A-$ to the supersink is equal to the capacity assigned to A in the previous stage.

In this graph, a simple integer maximum network flow from the seed to the supersink is computed, with one additional constraint: for any node x, if there is flow from $x-$ to $x+$, then there is flow from $x-$ to the supersink. Fortunately, the standard Ford-Fulkerson maxflow algorithm with the usual heuristic of augmenting along shortest augmenting paths automatically satisfies this constraint[FF62].

After computation of a network flow, the set of servers chosen is that with flow from $x-$ to the supersink. An example of such a maximum flow and assignment is shown in Figure 6. All nodes except for one (drawn with a dotted circle), have been chosen. For reference, the actual flows into each $x-$ node is shown.

3.2 Security proof

In this section, we prove the following theorem:

Theorem 1 *The number of bad servers chosen by the above metric is bounded by* $\sum_{x \in S}(c_x - 1)$*, where* S *is the set of confused servers chosen.*

Proof outline. Consider the cut that includes all edges from good or confused nodes to the supersink, and all edges from confused nodes to bad nodes. This is clearly a cut because there are no direct edges from good to bad nodes.

The total flow across this cut is equal to the total number of nodes chosen. Thus, the number of bad nodes chosen is equal to the total flow minus the number of good and confused nodes chosen. It follows that the flow from confused to bad nodes is equal to the number of bad nodes chosen.

The flow from a confused node x to bad nodes cannot exceed $c_x - 1$, as the total flow into x is bounded by c_x and the above-mentioned constraint requires that there be unit flow from $x-$ to the supersink if there is any flow at all. The constraint also ensures that the flow from non-chosen confused nodes is zero. Thus, the total number of bad nodes chosen is bounded by $\sum_{x \in S}(c_x - 1)$. □

Note that the number of bad servers accepted depends only on the number of confused servers, not on the number of bad servers. Thus, server spam attacks are frustrated.

Also, the above analysis assumes that a confused node can issue certificates for any number of bad nodes. Sharper results are possible when the number of certificates issued by confused servers for bad nodes bad nodes is limited, using the techniques of [LA98].

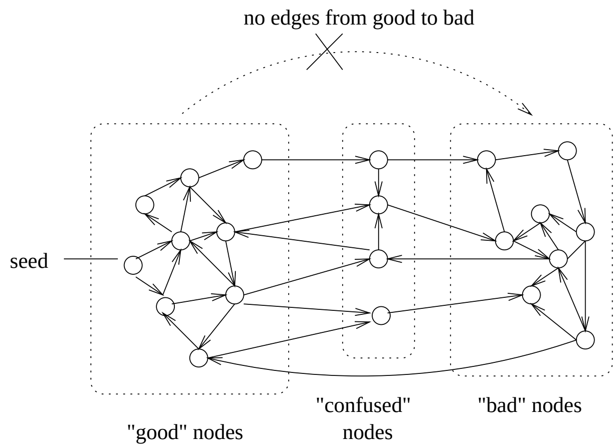

Figure 3: Attack model.

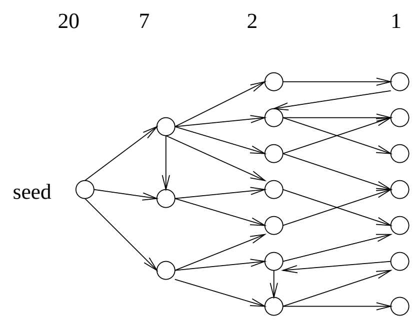

Figure 4: Assignment of capacities.

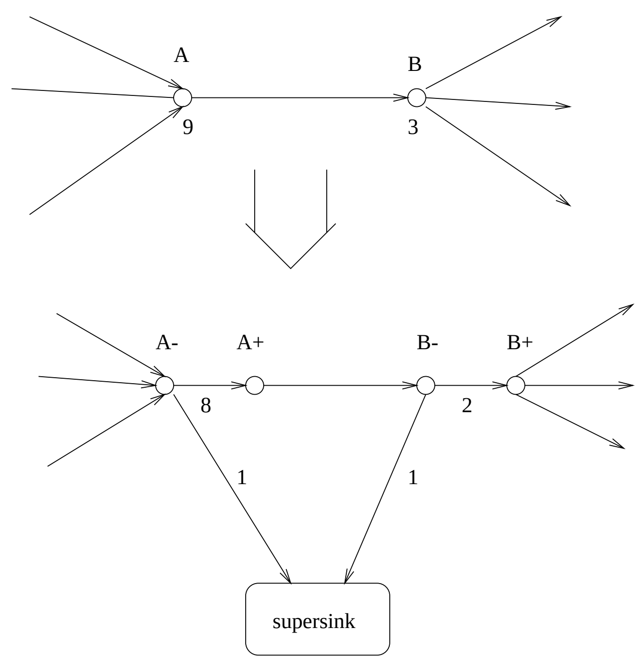

Figure 5: Construction of edge-constrained network.

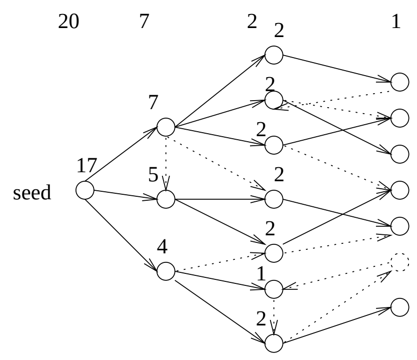

Figure 6: Example of flow.

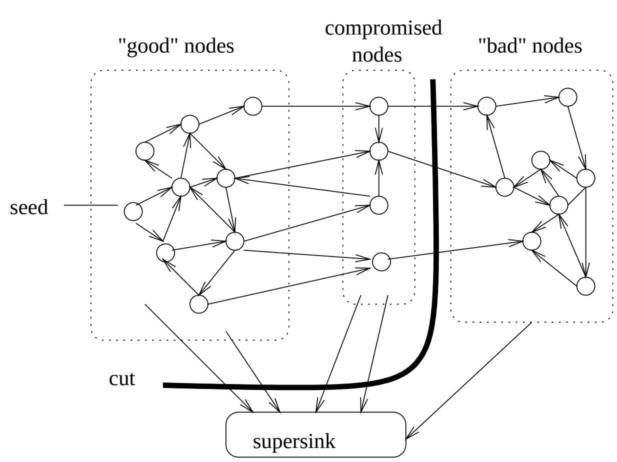

Figure 7: Cut used in security proof.

3.3 Choice of responsible servers

The choice of responsible servers is designed to meet the following goals:

- It is difficult for an attacker to influence the set of responsible servers for a name.
- It is easy to determine, given a name, which servers are responsible.
- It is easy to determine, given a server, which names it is responsible for.

The first goal is important because an attacker that can influence the choice of responsible servers for a name can bias this choice to servers under his control, thus increasing the chances of successful attack.

The second goal is important for efficient lookups (the most common operation).

The third goal is important when new servers join the system—they must be initialized with all names already registered for which they are responsible.

The choice of responsible servers is as follows: each server's public key is hashed, placing it on a ring (i.e. the integer value of the hash mod 2^{160}, assuming SHA-1 or some other 160 bit hash function). The name is also hashed twice, with either "0" or "1" appended. All servers within distance Δd on the ring of these hash results are responsible for the name. Δd is a tunable global parameter of the system. A too small Δd raises the probability of there being *no* responsible servers for a name to a significant level (this probability is less than $e^{-0.5 n_{avg}}/N$, where n_{avg} is the average number of servers selected and N is the total number of servers[1]). Conversely, the number of public key operations and the communications bandwidth for both registration and lookup scales linearly with Δd, so it should not be too large. A compromise value of 30 for n_{avg} ensures an extremely high probability of at least one responsible server (less than 10^{-9} chance of failure) while keeping communications bandwidth reasonable.

The reason for appending 0 or 1 is to decrease the correlation between sets of responsible servers. If this step were omitted, then an attacker that compromised $(1+x)2\Delta d$ contiguous servers would gain control of $n_{avg}x$ names on average.

The process is shown graphically in Figure 8. The hash function (typically SHA-1 in practice) is denoted by 'h', and the dark strokes represent the servers selected.

3.4 Request Processing

Update and registration requests are processed as follows: start with the list of servers, compute the subset of responsible servers, send the request to each of those servers, and each server processes the request.

Lookup requests are processed as follows: start with the list of servers, compute the subset of responsible servers, limit those still further to only include trustworthy servers (using the trust metric described in Section 3.1), do the query on this subset, determine a majority response.

Update and register requests are handled by all responsible servers because different clients may have different trusted server sets. Nonetheless, the system resists server spam attacks because the number of bad nodes accepted in majority voting does not depend on the number of bad servers in the system, only the number of confused servers.

3.5 New Server Joining Protocol

When a new server joins the system, three steps ensure that the system's global invariants are maintanined:

- All other servers are notified of the new server.
- The new server gets a complete server list.
- The new server gets copies of all name/key bindings for which it is responsible.

We assume that on initial configuration, the new server has the Internet address and public key of at least one existing good server.

The problem of notifying all other servers reduces to a broadcast of the server information. Here we present a simple, efficient, and robust broadcast protocol.

As in Section 3.3, the protocol is based on distances in a hash space. Here we define two similar but uncorrelated hash distance functions, d_0 and d_1. When a server receives a broadcast message for the first time, it sends it to both its d_0- and d_1-neighbors (i.e., all servers within Δd in both d_0 and d_1 hash space). If any of these neighbors is not functioning at the time of the broadcast, the message is stored in a queue and retried until the remote server becomes functional.

A detailed analysis of this protocol is beyond the scope of this paper, but it is worth noting that the set of neighbors in one hash space is pseudorandom in the

[1]It is possible to reduce this fraction even further, but these alternate designs have enough added complexity to be beyond the scope of this paper.

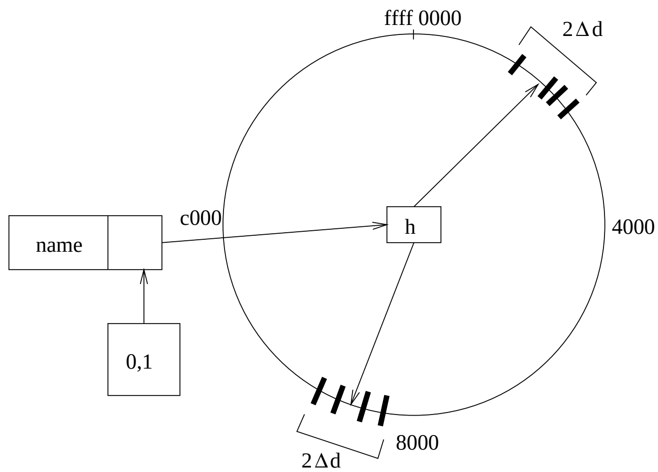

Figure 8: Choice of responsible servers.

other, so the graph of message transmissions contains an embedded pseudorandom graph. Random graphs have small diameter, so total propagation time is small. In addition, the existence of many redundant paths of propagation makes it difficult for an attacker to block the message.

The new server obtains its copy of the server list quite simply—it queries the server initially contacted, and then all its d_0 and d_1 neighbors, which insures that it receives notification of other servers that have joined at approximately the same time.

Finally, the new server obtains copies of all name/key bindings it is responsible for by querying all servers within $2\Delta d$ in the hash space defined in Section 3.3. It validates each of the name/key bindings, performing the same trust evaluation as clients. On average, approximately one quarter of the name/key bindings stored on these neighboring servers will be relevant, and only these need be transmitted.

4 Engineering Concerns

The previous sections have presented the naming service in somewhat abstract form. This section fills in a number of the concrete details and discusses some of the engineering issues, especially scalability and revocation.

4.1 Scalability

We are concerned with three main issues of scalability: the total communications bandwidth, the storage required on each server, and the number of transactions per second requested of each server. There are three aspects of the protocol that cause scalability concerns:

- maintenance and distribution of the list of all servers,
- communications bandwidth and server load for performing registrations and updates, and
- communications bandwidth for lookups.

4.2 Storage requirements

Each client (and server) must have a copy of the list of all servers, as well as the peer certificates. This corresponds to N public keys, and dN peer certificates, assuming N is the total number of servers, and d is the average number of peers certified by each server.

On average, n_{avg} servers are responsible for each name (n_{avg} is typically 30, as discussed in Section 3.3). To store a total of M name/key bindings, each server is responsible for storing $n_{avg}M/N$ records, each consisting of a name of size s_n, a public key of size s_k, and a signature of size s_s[2].

Then, the total storage required of each server is:

$$Ns_k + dNs_s + n_{avg}M(s_n + s_k + s_s)/N$$

Plugging in typical values of 30 for n_{avg}, 10 for d, 260 for s_k (i.e. assuming 2048 bit keys), 60 for s_s (i.e. assuming DSA), and 32 for s_n, we get $860N + 10560M/N$ bytes. It should be clear that

[2]actually, the signatures may be treated as a cache and re-generated as needed, but the space saving is of dubious value compared with the extra computational load of the signatures

this quantity is minimized when N is on the order of $\sqrt{M}$, in which case the storage requirement of each server becomes approximately $11.4k\sqrt{M}$.

For a typical Internet-sized naming service holding 10^8 names, a value of $\sqrt{M} = 10^4$ means that each client stores about 9 MB for the server graph (i.e. about the same size as a modern Web browser), and each server stores about 115 MB, which is trivial by today's standards.

4.3 Communications bandwidth

The communications bandwidth for each server to download the server graphs to the clients is (where $t_{turnover}$ is the average turnover of the certificate graph and registration database):

$$(Ns_k + dNs_s)(M/N)/t_{turnover}$$

Plugging in the typical values as above, and a figure of one year for $t_{turnover}$, each server consumes about 2.7 kB/second for this task.

If registrations turn over at the same rate, then the bandwidth for registrations is less than four bytes per second, i.e. trivial.

Estimating the communications bandwidth for lookups is difficult because it depends on the number of client lookups, not just the number of names and servers. There are two cases; cached and uncached.

If the certificates binding a name to a key are stored in a cache, the the response to the request may be represented in $s_k + n_{avg}s_s$ bytes, or about 2kB. This is about 1.5 orders of magnitude larger than existing DNS responses, and certainly competitive with X.509-based PKI designs.

If the certificates are not cached, then the client must query n_{avg} servers, requesting the certificates. Each certificate is $s_n + s_s + s_k$ bytes in length (i.e. typically 10.5kB for all the certificates), but in actuality only one of the servers needs to respond with the name and key—those fields *should* be the same in all responses.

The scaling behavior has been tested with a prototype implementation in Java on a cluster consisting of 50 UltraSparc machines. Preliminary results are shown in Figure 9 (total time to register and lookup each name, plotted against the total number of names), Figure 10 (total number of bytes communicated), and Figure 11 (number of servers). While our implementation suffers from a number of scaling problems specific to our choice of Java runtime environment, these results demonstrate the linear scaling of time and space, as well as the square-root scaling of the number of servers.

4.4 Revocation

One concern of many PKI's is timely revocation of certificates that are no longer valid. X.509-based PKI's use Certificate Revocation Lists (CRL's), which cause a number of problems with performance and scalability. In general, to verify that a certificate is still valid, it is necessary to contact the CA and retrieve an up-to-date CRL. Recent work extending X.509 is addressing these issues, but they remain a challenge[PKIX].

Our approach is that servers issue short-lived certificates on a periodic basis. Since the certificates are issued automatically according to a policy, this approach does not significantly add to the administration cost.

Revocation of peer certificates for use in the group trust metric is not as urgent, as the metric is designed to tolerate a fairly high number of bad certificates anyway.

5 Related Work

There are a number of systems that build secure services using relatively insecure machines, including secure logging on untrusted systems[SK98]. Most of this work assumes that the set of servers is fixed, thus there is no protection against server spam attacks. In particular, large networks of these systems are difficult to set up when they span several administrative domains, as the problem of keeping the set of (partially) trusted servers consistent is comparable in difficulty to the PKI itself.

Byzantine fault tolerant systems also address the issue of building a database secure against large-scale attacks [Rei94], [CL99]. These systems also provide strong consistency guarantees. Combining dynamic configuration of servers with the consistency guarantees of Byzantine fault tolerant systems is an interesting topic for further research.

The distributed TTP is similar to that described in [Bla96] for key escrow. However, that system also has no protection against the server spam attack.

The group trust metric extends the existing work on trust metrics, including, [RS97a], [RS97b], and [LA98].

Treating registration and updating separately has been discussed in [CL96]. In their work, the focus is on creating an audit trail in case the CA's are compromised.

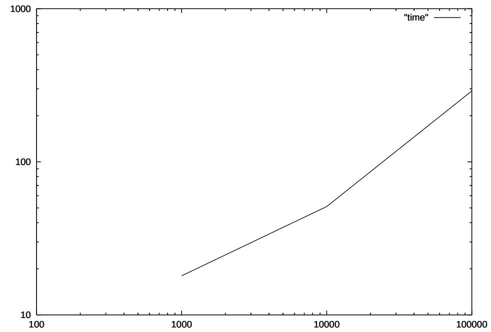

Figure 9: Time vs number of names.

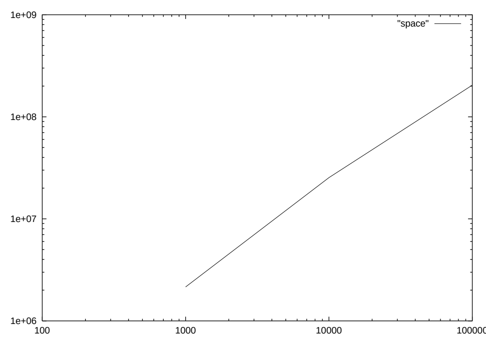

Figure 10: Bytes of communication vs number of names.

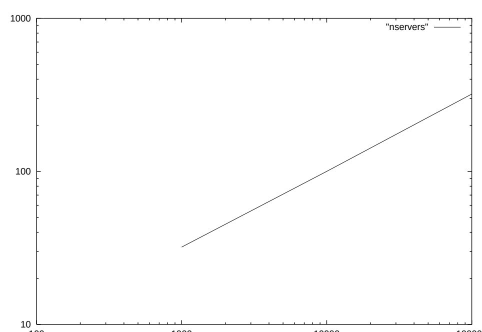

Figure 11: Number of servers vs number of names.

6 Conclusion

The naming service presented in this paper meets the goals of scalability, resistance to attack, and lower certification cost than existing PKI. The classical PKI goal of identifying the "right" key for a name is factored into a policy language and a distributed mechanism for implementing policies expressed in this language. This separation contributes greatly to achieving the goals.

The distributed trusted third party for implementing the policies contains some novel technical contributions, most notably a group trust metric for choosing a subset of trustworthy servers. This trust metric exhibits gradual degradation under attack, in fact the fraction of "bad" servers accepted scales linearly with the magnitude of the attack. All previous known trust metrics intended for automatic evaluation exhibit catastrophic failure under a relatively small attack, in some cases a single key[LA98]. The group trust metric is an essential component of a design which builds a secure service using a federation of relatively insecure servers.

Whether the added security and other benefits are enough to outweigh the cost of replacing existing PKI designs such as X.509 and DNSSEC is an open question. In any case, the techniques presented in this paper should be seriously considered by implementors of naming services for new distributed applications.

References

[Bla96] M. Blaze, "Oblivious key escrow," In *Proc. Workshop on Information Hiding,* LNCS 1174, pp 334–343, Springer-Verlag, 1996.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management." in *Proc. 17th Symposium on Security and Privacy*, IEEE Computer Society Press, Los Alamitos, 1996, pp 164–173.

[Bra97] M. Branchaud, "A survey of public key infrastructures." Master's thesis, Dept. of Computer Science, McGill University, Montreal, March 1997.

[CL96] B. Crispo and M. Lomas, "A certification scheme for electronic commerce." Security Protocols '96, Springer-Verlag LNCS 1189, pp.19–32, 1996.

[CL99] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance." in *3rd Symposium on Operating System Design and Implementation,* New Orleans, February 1999.

[DH76] W. Diffie and M. Hellman, "New directions in cryptography." *IEEE Transactions on Information Theory,* 22:644-654, 1976.

[FF62] L.R. Ford, Jr., and D. R. Fulkerson, "Flows in networks." Princeton University Press, Princeton, NJ, 1962.

[Fro96] A.M. Froomkin, "The essential role of trusted third parties in electronic commerce." Law and Entrepreneurship Program: Innovation and the Information Environment, 75 Oregon L. Rev. 49, 1996.

[LA98] R. Levien and A. Aiken, "Attack resistant trust metrics for public key certification." *7th USENIX Security Symposium*, Jan 26–29, 1998. San Antonio, Texas.

[Post99] D. Post, "Governing Cyberspace, or where is James Madison when you need him", http://www.temple.edu/lawschool/dpost/icann/comment1.html, June 1999.

[Rei94] M. Reiter, "Secure agreement protocols: reliable and atomic group multicast in Rampart," in *Proc. ACM Conf. on Computer and Communications Security 1994*, pp 68-80.

[RFC977] RFC 977, "Network News Transfer Protocol." Internet Engineering Task Force, February 1986.

[RS97a] M. Reiter and S. Stubblebine. Path independence for authentication in large-scale systems. In *Proceedings of the 4th ACM Conference on Computer and Communications Security* (1997).

[RS97b] M. Reiter and S. Stubblebine. Toward acceptable metrics of authentication. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (1997).

[RL96] R. Rivest and B. Lampson. SDSI—a Simple Distributed Security Infrastructure. Presented at USENIX, 1996.
`http://theory.lcs.mit.edu/~cis/sdsi.html`

[SK98] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines." *7th USENIX Security Symposium*, Jan 26–29, 1998. San Antonio, Texas.

A Evaluation of Policy Constraints

This appendix provides a detailed algorithm for evaluating whether $P \subseteq P'$, given two policies P and P' in the policy language described in Section 2.

The algorithm presented here cannot compute this relation over policies that contain late-bound groups or policy fragments. Adding these is a relatively straightforward extension. The algorithm handles late-bound keys without difficulty; a "key" is considered to be an atomic element of a policy, whether it is specified directly or indirectly with a name.

To evaluate whether $P \subseteq P'$, the first step is to convert both P and P' into *normal form.* A policy in normal form is a disjunction of *terms,* where each term is a conjunction of *factors,* each of the form k_i *of* S_i. The S_i contained in a term are all disjoint.

The notation $[P]$ is used to denote the normalized version of P. An arbitrary policy is converted to normal form through repeated application of the rules shown in Figure 12. The variables T and T' stand for terms already in normal form.

Once converted into normal form, the predicate $P \subseteq P'$ is determined by computing the difference $P - P'$, using the algorithm presented in Figure 13. If this difference evaluates to 0, then $P \subseteq P'$. If the difference is nonzero, then there exists some set of keys accepted by P but not by P', thus $P \subseteq P'$ does not hold.

In these rules, the large quantification over $\vec{x}$ can usually be greatly pruned by realizing that k *of* S is 0 when $k > |S|$, and simplifying using the algebraic rules $0 \wedge P = 0$ and $0 \vee P = P$. Thus, an efficient implementation generates the values of $\vec{x}$ by iterating subject to the resulting inequality constraints. Exploiting the identities $(0$ *of* $S) = 1$ and $1 \wedge P = P$ yields additional simplification.

$$
\begin{aligned}
&[P \vee P'] = [P] \vee [P'] \\
&[T \wedge T'] = \bigvee_{\vec{x}} \left(\bigwedge_{i,j} x_{ij} \ \textit{of} \ (S_i \cap S'_j) \right) \wedge \left(\bigwedge_i k_i - \bar{x}_i \ \textit{of} \ (S_i - \overline{S_i}) \right) \wedge \left(\bigwedge_j k'_j - \bar{x}'_j \ \textit{of} \ (S'_j - \overline{S'_j}) \right), \text{ where} \\
&\qquad \overline{S_i} = \bigcup_j S'_j, \quad \overline{S'_j} = \bigcup_i S_i, \\
&\qquad \bar{x}_i = \sum_j x_{ij}, \quad \bar{x}'_j = \sum_i x_{ij}, \\
&\qquad T = (k_1 \ \textit{of} \ S_1) \wedge (k_2 \ \textit{of} \ S_2) \wedge \ldots \wedge (k_n \ \textit{of} \ S_n), \\
&\qquad T' = (k'_1 \ \textit{of} \ S'_1) \wedge (k'_2 \ \textit{of} \ S'_2) \wedge \ldots \wedge (k'_n \ \textit{of} \ S'_{n'}), \text{ and} \\
&\qquad T, T' \text{ are already normal form terms} \\
&[P \wedge P'] = \bigvee_{i,j} [T_i \wedge T'_j], \text{where} \\
&\qquad [P] = T_1 \vee T_2 \vee \ldots \vee T_n, \text{ and} \\
&\qquad [P'] = T'_1 \vee T'_2 \vee \ldots \vee T'_{n'}
\end{aligned}
$$

Figure 12: Algorithm for converting policies to normal form.

$$
\begin{aligned}
&[T - T'] = \bigvee_j \bigvee_{\vec{x}} \left(\bigwedge_i (x_i \ \textit{of} \ (S_i \cap S'_j) \wedge k_i - x_i \ \textit{of} \ (S_i - S'_j)) \right), \text{ where} \\
&\qquad T = (k_1 \ \textit{of} \ S_1) \wedge (k_2 \ \textit{of} \ S_2) \wedge \ldots \wedge (k_n \ \textit{of} \ S_n), \\
&\qquad T' = (k'_1 \ \textit{of} \ S'_1) \wedge (k'_2 \ \textit{of} \ S'_2) \wedge \ldots \wedge (k'_n \ \textit{of} \ S'_{n'}), \text{ and} \\
&\qquad T, T' \text{ are already normal form terms} \\
&[P - T'] = [T_1 - T'] \vee [T_2 - T'] \vee \ldots \vee [T_n - T'], \text{ where} \\
&\qquad [P] = T_1 \vee T_2 \vee \ldots \vee T_n, \text{ and} \\
&\qquad T' \text{ is already a normal form term} \\
&[P - P'] = [(((P - T_1) - T_2) - \ldots - T_n)], \text{ where} \\
&\qquad [P'] = T'_1 \vee T'_2 \vee \ldots \vee T'_{n'}
\end{aligned}
$$

Figure 13: Algorithm for computing differences of policies.