Abridging Certification Authorities

DRAFT - Please do not distribute

Matthias Jacob
Princeton University
mjacob@cs.princeton.edu

Guido Appenzeller
Stanford University
appenz@cs.stanford.edu

Abstract

Certificates in the Internet map online identities like DNS names to cryptographic keys. Since they are highly trusted entities that can only be issued by a few trusted certification authorities (CAs), they are a costly security component in today's Internet. These CAs have to do extensive checks to ensure that a server is trusted. However, a large part of the Internet's servers do not require ultimate trust levels. We explore alternatives to the old-fashioned CA structure and come to the conclusion that these approaches provide completely sufficient security guarantees for low-volume servers. Many of these low-volume servers do not even use certificates because to them they are either too expensive or do not provide reasonable trust levels. These alternative techniques, when deployed, will be an inexpensive way for issuing certificates and hence increasing trust and security in networks.

1 Introduction

In secure communication such as an SSL connection over a network it is a common problem to match an online identity such as a DNS name with the public key used for sending encrypted messages. The current solution to this problem is the deployment of certificates. Globally-trusted Certification Authorities (CA) verify whether the matching between a host's DNS name and public key is correct and issue a certificate that every node in the network can use to check the matching. During the initiation of the secure communication, for example using SSL, hosts that participate in the communication exchange their public keys. If no certificate exists the genuineness of a host's identity solely relies on its DNS name. DNS servers in the network are responsible for resolving a host's DNS name to its IP address, but it is a well-known fact that DNS servers are prone to attacks in which an adversary spoofs the DNS name and matches it to a different IP address or redirects an IP address to a different one.

However, certificates are expensive. First, certificates are highly trusted, so the CA needs to check external documentation when it issues a certificate. Second, it is crucial that the private signing key of the CA does not leak, and protecting the key is costly. Third, CAs have to distribute the public key for checking the certificates to the clients, and these are usually pre-installed in the web browser. Hence, it is not possible to easily add new CAs which limits the competition. The result is that for many hosts such as low-volume web servers or nodes in peer-to-peer networks obtaining certificates is too costly. In the community there exists a strong desire for cheap certificates [5, 1, 4]. However, so far, a reasonable solution has been elusive. The most popular currently available alternative to highly trusted certificates are low-cost certificates [3, 18]. In the verification process for these certificates the CA sends a single verification email to a domain administrator. While this process is quick and convenient, an attacker can intercept this verification email and forge the certificate. It is doubtful that this is a viable way to certify trust in a server.

In this paper we provide a novel approach to cheap certificates. We observe that a state-of-the-art CA decides whether to certify a host solely based on information it obtains at the time the host requests the certificate. After the CA has issued the certificate it is valid over a long time period (eg. a few years). In order to invalidate the certificate the CA has to manually publish it on its certificate revocation list (CRL) [16]. The process of manually verifying trust in a host and publishing the CRLs is expensive, tedious, and unreliable.

1.1 Time-based trusted authentication

We approach the problem of issuing cheap certificates by using "trust over time": When a host consistently responds with the same public key in the initial public key exchange over some given time interval it reaches a higher trust level. We make the assumption that an attacker is unable to spoof its DNS name over a long time period.

This is a common principle in the physical world, as for example in credit reports. In order to determine credit worthiness a creditor requests a credit report. This credit report shows whether a potential client has had any irregularities on paying back credits in the past. If the client has been in good standing over some time period, the creditor trusts her and grants the loan.

In this paper we develop CAs that initiate SSL connections periodically in order to find out whether the host consistently publishes its public key. The CA automatically adjusts the host's trust level and issues a new certificate. Since the certification process works automatically these certificates can have finer-grained expiration dates than the X.509 certificates and CRLs become less important.

The remainder of the paper consists of four parts. The section 2 part defines important terms of the certification system. In section 3 we investigate the design for the CA using trust over time. We show how to make the CA accountable and what the security guarantees are in section 4, and finally explain in section 5 how a fully distributed CA works.

2 X.509 Certification System

In this paper we are looking at computer networks that typically have a large number of hosts. Hosts can take different roles. For example, in a client-server system server hosts provide the service that the client hosts use. On the other hand in a peer-to-peer network all hosts are clients and servers.

In our model we separate between clients and servers. Clients need to make sure that they are communicating with the right server, and hence, they request the certificate that verifies the matching between DNS name and public key. Summarized, the typical network consists of the following components:

- **Client:** A client initiates a connection to a particular server in order to request services from this server. In order to exchange data securely the client needs to ascertain that it possesses the public key that matches the server's private key. The client uses certificates to verify whether the public key from the server is the correct one,
- **Server:** A server provides network services to clients and publishes to the client the public key that corresponds to its private key for secure data exchange. To show that the public key belongs to the server the CA issues a certificate that contains this public key and the server's DNS name.
- **Certification Authority (CA):** The certification authority issues certificates to servers. These certificates contain information about the server's DNS name and the public key it uses to exchange data with clients. A certificate in the X.509 specification also contains information about the physical identity of the server and access policies, but this is not the focus of this paper.

The CA system consists of two operations. Initially, a server asks the CA to *issue* a certificate. For this operation the server passes along to the CA its DNS name *server*, its public key *pk*, and some external identification *id*. After checking external identification the CA returns the certificate containing the DNS name and the

Figure 1: The role of the certification authority in the Internet. A server uses a private key for signing or encrypting data and publishes the corresponding public key. The client contacts the server by using its DNS name and gets the correct public key in the server's certificate. Finally, the client trusts the CA, and the CA trusts the server. Under the assumption that trust is transitive, hence, the client can also trust the server.

public key of the server. The CA signs the certificate with its signing key. Any client in the system can now *request* a certificate from a server. The client verifies the signature of the certificate and, if valid, obtains the public key *pk*.

- **issue_certificate(server,pk,id) (server → CA):**
  ({server, pk, id}: *server → CA*) ⇒
  ((id valid) ⇒
  ({cert_{CA}(server, pk)}: *CA → client*))

- **request_certificate(server) (client → server):**
  (server: *client → server*) ⇒
  ({cert_{CA}(server, pk)}: *server → client*)

Once the CA has issued a certificate it is valid until its expiration date given on the certificate. A certificate becomes invalid before its expiration date when the CA publishes it on a CRL that contains a list of all invalid certificates.

2.1 Security Analysis

We guarantee the security of the system when any client gets the correct public key for a server. There exist three kind of attacks: First, an adversary can manipulate any contents a server, CA or client transmits over the network. Second, an adversary can compromise a server and obtain its private key. Third, an adversary can compromise the CA and get the private signing key:

- **Attacks from the network:**
  Since the certification authority signs all certificates it is not possible for an adversary to modify or forge certificates during transmission. Since the certification process does not take place online due to external identification, modifying the server's key during transmission is not a problem. An adversary can spoof the DNS resolution between client and server in which case the client gets redirected to a different physical host, but in order to verify a public key this host needs to have a valid certificate.

- **Compromising servers:**
  When an adversary compromises a server and its private key leaks the adversary can decrypt all messages the server receives, but the adversary cannot change the key without getting a new certificate. In this case the CA needs to put the certificate on the CRL such that clients stop communicating with the compromised server. The certificate cannot protect against passive attacks when the adversary compromises the server.

- **Compromising the CA:**
  The CA is the critical component of the system. If an adversary compromises the CA and obtains the private signing key the adversary can issue false certificates and listen to data from any client in the system that connects to a server.

The CA is the crucial component of the system, and it has to spend a lot of effort in protecting the private signing key from leaking which is one of the reasons certificates are expensive, and we will address this problem later in the paper.

In the next section we first look into how we can automate the process of certifying a server such that costly verification of external identification is not necessary anymore.

3 CA using trust over time

The weakness in the standard X.509 certification authority system is to establish trust between the CA and

the server. The CA needs to verify offline that the DNS name matches the public key of the server. Usually, the CA verifies this mapping by examining external identification such as a driver's license. This is an expensive process that costs money and time.

In order to get around this expensive verification method we apply the "trust over time" principle to the certification of a server. If a server has a certain trust level the CA issues the certificate, or even simpler, it just issues a certificate containing the trust level. The CA establishes the trust level by checking periodically whether the public key of the server remains the same over a given time interval. If the server's public key changes someone has potentially spoofed the DNS server.

3.1 System overview

Figure 2 shows the system with the CA using trust over time. When the server wants to obtain a certificate it submits its DNS hostname and public key to the CA and receives a certificate of trust level $t' = 0$. After the CA sends this first certificate to the server it periodically checks the server's public key and adjusts trust level t'.

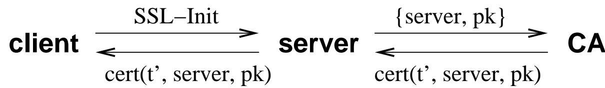

Figure 2: The CA using trust over time.

3.2 Design Issues

We now investigate design issues of the system. Especially, we examine how a client can decide whether the server is trusted or untrusted based on the trust level.

3.2.1 Trust level update

The CA increases the trust level when the server returns the public key from the original certificate in the initial handshake and decreases it when the server does not return the right public key. When the check for the server's public key fails, the CA can either set the trust level to some lower value which means that we do not need to revoke the certificate. When an adversary compromises the DNS server and redirects traffic the trust level needs to drop immediately such that a client can decide whether or not to trust the server. However, this process of decreasing the trust level usually takes time. In the traditional X.509 system this latency also exists when the CA publishes the certificate on the revocation list that notifies the client about the server's compromise.

3.2.2 Operations

The CA using trust over time has three operations: **request_certificate**, **issue_certificate**, and **update_certificate**. Unlike the X.509 CA the trust over time CA can only *issue* a new certificate of trust level 0 or *update* a certificate to a certain trust level. There is no "strong" certificate anymore that the CA issues. After a server reaches a certain trust level t the client can consider the server to be trusted. During the update operation CA updates the trust level according to the update function $u(t)$.

When a client *requests* a certificate the server sends the client the certificate as in X.509, but in addition the certificate now contains the trust level t of the server.

- **request_certificate(server) (client $\rightarrow$ server):**
  (server: *client* $\rightarrow$ *server*) $\Rightarrow$
  (cert$_{CA}$(server, t, pk): *server* $\rightarrow$ *client*)
- **issue_certificate(server,pk) (server $\rightarrow$ CA):**
  ({server, pk}: *server* $\rightarrow$ *CA*) $\Rightarrow$
  (SSL-init: *CA* $\rightarrow$ *server*) $\Rightarrow$
  ({server, pk'}: *server* $\rightarrow$ *CA*) $\Rightarrow$
  ((pk' = pk) $\Rightarrow$
  (cert$_{CA}$(server, 0, pk): *CA* $\rightarrow$ *client*))
- **update_certificate(server,pk) (CA $\rightarrow$ server):**
  ((SSL-init: *CA* $\rightarrow$ *server*),t) $\Rightarrow$
  (({server, pk'}: *server* $\rightarrow$ *CA*),t) $\Rightarrow$
  ((pk' = pk) $\Rightarrow$
  (t'= u(t),

(cert$_{CA}$(server, t', pk): $CA \rightarrow client$)))

3.3 Security Analysis

The analyzes for attacks from networks, compromising servers, and compromising the CA are similar to the X.509 certification system. The remaining question is how the CA needs to update the trust level and how the client decides when it can trust a server. A client needs to distinguish whether it can or cannot trust a server. In the trust over time CA this depends on the trust level, and there needs to be a threshold value below which the server is untrusted and above which it is not.

The CA needs to pick the length of the update interval such that an attack on the server has an impact on the trust level. In our model u is the average length of the update interval, f the time interval that is necessary to detect and fix a compromised server (i.e. the time the compromised server is online), and c the average time between to attacks on a server. We assume that c and u are uniformly distributed. The update interval has to be random since an adversary can otherwise find out about the update schedule and interrupt the attack when the update process takes place.

First, we want to analyze the maximum length update interval. This length depends on how reliably the system detects compromised servers. Since u is distributed uniformly and the likelihood of a compromise is $\frac{f}{c+f}$ we get

$$P_{miss} = \frac{u-1}{u} \cdot \frac{f}{c+f} \tag{1}$$

and hence

$$u_{max} < \frac{f}{f - P_{miss}(c+f)}. \tag{2}$$

For example, in a case a server is compromised online for 12h, but it gets compromised once a month. Then the CA needs to update the trust level of the server every 2.5h in order to achieve a miss rate $P_{miss} = 1\%$.

The CA updates the trust level by using a debit/credit scheme. In case an adversary compromises the server the CA deducts D units from the current trust level, and when the server is not compromised it increases the trust level by I units. Hence, we get for the cost function

$$\begin{aligned} C &= \tfrac{1}{u} \cdot \tfrac{c}{c+f} \cdot I - \tfrac{1}{u} \cdot \tfrac{f}{c+f} \cdot D \\ &= \tfrac{1}{u(c+f)}(cI - fD) \end{aligned} \tag{3}$$

Since c is always larger than f the trust level increases on the average, and we need to set a maximum value. In the case where the server is not compromised this trust level is usually this maximum value. The important part, however, is the trust level in the case an adversary has compromised the server. We get

$$C_c = C - \frac{f}{u} \cdot D \tag{4}$$

The result is that the CA needs to pick d according to u, f, and c in the system. For example, in the case from above if our goal is $C_c \approx \frac{C}{2}$ such that a client can distinguish between the two trust levels then $D \approx \frac{C}{5}$ (we can assume that C is the maximum trust level).

This security analysis shows how to adjust the parameters in order to achieve sufficient trust in the server's certificate. We conclude that it is feasible to build a trust over time CA.

3.4 Discussion

When using trust over time in order to verify the validity of a public key the CA needs to initiate checks with the server which can cause scalability problems, but we have shown that even when using large update intervals the trust level can be distinguished sufficiently.

Another advantage of this scenario is that revocation lists basically come for free. The remaining problem, however, is that the CA needs to be trusted to do the checks on the servers regularly. Since there is no offline documentation anymore, the CA needs to be accountable for its actions, and we will discuss this problem in the next section.

4 Accountable CA

In the centralized CA using trust over time any host in the certification system needs to fully trust the CA whether it checks the server's public key at certain times. In the offline case the certification process is verifiable because there exist some physical documents the CA uses. However, in the online case the CA uses evidence that it initiates itself, namely, the periodical checks of the public key. Hence, the CA needs to log the checks for the public key such that any client can verify the correctness of the certificate.

However, we can also decrease the likelihood of false certificates from the CA by introducing the concept of *judges*. Judges are servers selected randomly in the system that decide by majority vote whether the CA issues a certificate or not. Furthermore, different judges use different DNS servers, and hence, if an adversary spoofs one DNS server, only a few judges might not certify the server. Thus, the required number of judges to agree is a security parameter in the system that the CA can use to tolerate some misbehavior of the system.

4.1 System Overview

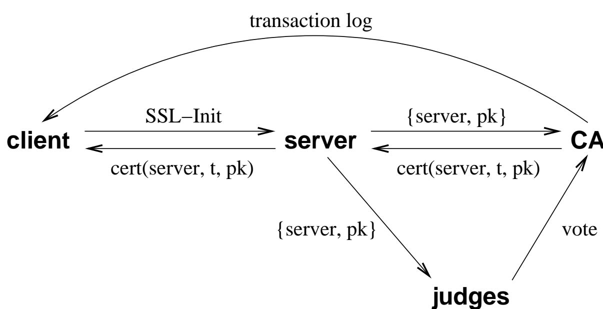

Figure 3: Accountable CA: Judges check the server's public key and vote whether a certificate can be issued.

The system consists of the following entities:

- **Clients:** A client wants to exchange data with a server with a specific name. It requires servers public keys to certified to ensure that the server is indeed who it pretends to be.
- **Servers:** The servers offer services to clients. To demonstrate that the public key it uses is correct, they require the server to provide them with a valid certificate.
- **Certification Authority (CA):** It serves as the coordinator of the certification process and issues site certificates using its private key. The CA keeps a protocol of all transactions in a log, and publishes this log to the clients. This makes the CA accountable.
- **Judges:** Any server that is certified by the CA can serve as a judge. Judges help the CA to verify keys and ensure that the CA does not assert too much power.

The system as shown in Figure 3 uses judges as an instance to control the power of the CA. Compared to the CA using trust over time, first, the system needs to include the judges into the process of issuing a certificate. When the CA issues a certificate it selects a group of judges randomly (as described below), lets these judges vote and only issues the certificate when the judges agree. Second, bootstrapping the system can lead to vulnerabilities if the CA certifies compromised servers.

4.2 Design Issues

There are several design issues we need to investigate. First, a crucial issue of the system is bootstrapping since it is more likely to be vulnerable in this stage. Second, we explain how the CA issues a certificate.

4.2.1 Picking the judges

Picking the judges is crucial for the security of the system. The CA picks judges as follows: After a specified time period the CA generates a random number from a globally known source of randomness (e.g. the stock market). Using this and its own number it picks k judges. Any web site that is certified by the CA can serve as a judge. The list of potential judges is public. The CA does not publish which judges it picked

yet. All judges need to have different domain names to avoid an attack in which an adversary creates virtual servers on one physical machines in order to compromise the system.

4.2.2 Bootstrapping the system

To bootstrap the system the CA selects and certifies a preferably large number of servers. Now the CA can use this pool of certified servers to select judges. It is important that it picks these servers during the bootstrapping process carefully since they are responsible for any further decisions. If most of the servers during the bootstrapping process receive false certificates (i.e. cannot be trusted), the system is vulnerable to compromise. New sites requesting a certificate follow the process described below.

4.2.3 Issuing a Certificate

When a CA issues a certificate for a certain server it executes the following steps:

1. A server connects to the CA and requests a certificate for a public key. The CA immediately initiates an SSL connection back to the server and verifies that the published public key matches the one used by the host that responds to that identity.
2. The CA publishes the certificate request in the log.
3. The CA picks the judges as described above.
4. The CA notifies the judges that they have been selected for judging.
5. Within a given time interval the judges initiate randomly several SSL connections to test whether the host still responds to this key.
6. If the key has not changed after this time interval the judges report their result back to the CA, signed by their own private keys. The CA aggregates their results and if a majority agrees, it issues a certificate to the host. The certificate states that the key belongs to the host and contains the current trust level based on the length of the time interval the key has been valid. Together with the certificate the CA publishes in the log the identity and verdict of all judges and the random number it used to select the judges.
7. The site can now install the certificate. Users connecting to the server will be aware of the host's trust level and can decide whether or not to trust the server.

The CA occasionally verifies the public key of the server. If the server presents a different key, its current certificate will be revoked immediately. If the host reverts back to its old key within a short, given time period the CA issues a new certificate for the public key. If this is not the case or the CA receives a revocation for the key no new certificate gets issued for the key.

If the key of a host does not change over longer periods of time the trust level will increase accordingly and the next certificate that is issued will reflect this increase.

4.2.4 Operations

Formally, we define the following operations. v_{min} is the minimum number of judges to agree, and pick_judges() lets the CA pick the judges randomly. The CA writes every operation to the log, and any client is able to check this log.

- **request_certificate(server) (client $\rightarrow$ server):**
  (server: *client* $\rightarrow$ *server*) $\Rightarrow$
  (cert$_{ck}$(server, t, pk): *server* $\rightarrow$ *client*)
- **issue_certificate(server,pk) (server $\rightarrow$ CA):**
  ({server, pk}: *server* $\rightarrow$ *CA*) $\Rightarrow$
  (SSL-init: *CA* $\rightarrow$ *server*) $\Rightarrow$
  ({server, pk'}: *server* $\rightarrow$ *CA*) $\Rightarrow$
  ((pk' = pk) $\Rightarrow$
  (v = query_judges(n), v $\geq v_{min}$) $\Rightarrow$
  (cert$_{ck}$(server, 0, pk): *CA* $\rightarrow$ *client*))
- **update_certificate(CA $\rightarrow$ server):**

((SSL-init: $CA \rightarrow server$) $\Rightarrow$
({server, pk'}: $server \rightarrow CA$), t, pk) $\Rightarrow$
((pk' = pk) $\Rightarrow$
((v = query_judges(n), v $\geq v_{min}$) $\Rightarrow$
(t'= u(t),
(cert$_{ck}$(server, t', pk): $CA \rightarrow client$))))

We use the following helper functions:

- **verify_server(server) (CA → judge):**
  (({server, pk}: $CA \rightarrow judge$) $\Rightarrow$
  (SSL-init: $judge \rightarrow server$) $\Rightarrow$
  ({server, pk'}: $server \rightarrow judge$) $\Rightarrow$
  ((pk' = pk) $\Rightarrow$
  (true: $judge \rightarrow CA$)))
- **query_judges(n) (CA local):**
  ((J = pick_judges(n), v= 0,
  $\forall j \in J$:(verify_server($server$): $CA \rightarrow$ j) $\Rightarrow$
  ((vote: j $\rightarrow CA$), v= v + vote))

4.3 Security Analysis

- **Attacks from the network:**

  The communication between the host requesting the certificate and the CA is the same as in the approach using off-line verification for each certificate which means that it is generally vulnerable against man-in-the-middle attacks when the identity is used for authentication on access control instead of capabilities contained in the certificate. The second and more interesting aspect is the backward channel from the CA to the host that has this identity. Because of the initial verification process it is certain that a host belongs to the identity, however, if an attacker intercepts all verification requests and returns the correct key, the original system can be locked out. However as the judges are not known during the judging phase and they use standard SSL requests to verify the key, an attacker needs to intercept and alter all incoming requests over a longer period of time. Such an attack will likely require considerable effort and has a substantial risk of getting detected. It seems unlikely that an attacker would take such risks to attack a low security site.

- **Compromising judges:**

  We have two possible failures. First, judges can compromise a vote by certifying an untrusted server. Similar, judges can decline a certificate to a trusted server. Both cases can occur when the CA happens to pick at least v_{min} judges from the candidates. Second, judges can sabotage a vote such that no majority exists. The probability that i of the k judges are malicious when n is the number of candidates for judges and m of of n candidates are malicious is:

  $$Q(n,k,m,i) := \frac{\binom{m}{i}\binom{n-m}{k-i}}{\binom{n}{k}} \quad (5)$$

  In total we need at least j judges in order to achieve a majority. Hence, the probability that judges compromise the certification process such that a decision on issuing a certificate gets overturned is:

  $$P(\text{false certificate}) = \sum_{i=j}^{k} Q(n,k,m,i) \quad (6)$$

  However, judges can already sabotage a vote when less than j non-compromised judges vote, i.e. the CA picked more than $k-j$ but less than j malicious judges. The probability that this case occurs is much higher.

  $$P(\text{no certificate}) = \sum_{i=k-j+1}^{j-1} Q(n,k,m,i) \quad (7)$$

  First we investigate the case for $j = \lfloor \frac{k}{2} \rfloor + 1$. Whenever the malicious judges make up more than half of the judge candidates they can compromise the certification process such that the CA issues a false certificate.

  Figure 4 shows the maximum number of malicious judges the candidate list can afford to give the listed security guarantees against a comprise. The larger k gets the closer the thresholds for the probabilities move together. This is plausible since if the CA picks more judges and more than 50% need to agree, absolutely more servers on the candidate list need to be compromised in order to be harmful to the system. When n is large, for a multiple of n the number of malicious servers

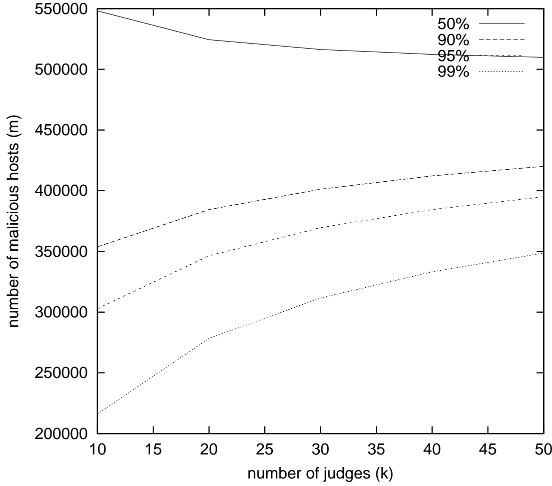

Figure 4: Number of malicious candidates for judges such that the accountable CA for $n = 1000000$ is secure against false certificates with probabilities 50%, 90%, 95%, and 99% in the case $j = \lfloor \frac{k}{2} \rfloor + 1$ depending on the number of judges in the system.

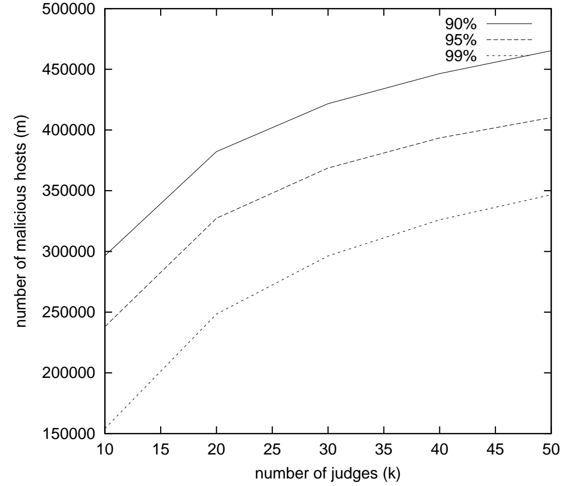

Figure 5: Number of malicious candidates for judges such that the accountable CA for $n = 1000000$ is secure against sabotage attacks with probabilities 90%, 95%, and 99% in the case $j = \lfloor \frac{k}{2} \rfloor + 1$ depending on the number of judges in the system.

required to compromise the system increases almost proportionally. This means in the case of 20 judges an adversary always needs to compromise about 28% of the servers in the system in order to have a 1% chance for overturning a certificate which in our opinion is a reasonable security guarantee given that CAs usually issue certificates for millions of hosts.

Next we look at the security against sabotaging the voting process. The probability for sabotaging is lowest when $j = \lfloor \frac{k}{2} \rfloor + 1$ because half of the judges need to be compromised in order to sabotage a vote. When j is larger fewer judges need to be compromised. A large j means stronger security guarantees against issuing false certificates, but on the other hand it increases the probability for sabotaging.

Figure 5 shows how many malicious candidates for judges are necessary in order to sabotage a vote. For example, if we want to have a 99% security guarantee for $j = 20$ up to 25% of the candidates can be malicious which seems to be a reasonable security assumption.

- **Compromising the CA:**

  One of the goals of the accountable CA is to eliminate the need to fully trust the certification authority. Our goal, however, is not to prevent all attacks from a malicious CA. Instead we want to make the CA accountable such that fraud is easily detectable.

  Incorrect Certificates: The CA can issue certificates for keys that do not reflect the verdict of the judges or for sites that have not been certified at all. This however is very easy to detect as the client can check the public record of all certificates that were issued and verify the signatures of the judges.

  Incorrect Judge Verdicts: The CA can show incorrect judge verdicts to support the certificate it has issued. Again this is easy to detect. If the judges are real, the CA is not able to produce the correct signature for the judges. The CA could also claim nonexistent Judges however this is easily detectable by looking up the certificates of the judges.

  Collusion with Judges: The CA could collude with a large number of true or artificially created judges. As the CA is accountable for everything it does and all its actions are either deterministic or random, this is not different from the case of colluding judges.

5 A Fully Distributed CA

The CA we described in the above section is accountable, however, it still has some drawbacks of a centralized institution:

- The signing key of the certification authority represents a single vulnerable point in the system, and for a CA that should operate at low cost or even for free it is difficulty to protect its master key. If a malicious party manages to steal the signing key, it can issue bogus certificates. Furthermore, the CA needs to distribute the signing key to the client, e.g. by implementing them in web browsers. The distribution of the signing key is a non-negligible cost factor for a CA.
- Since the certification authority is located in a single place, it is a potential target for denial of service attacks. Again, the low-cost nature of a CA makes it difficult to build up an infrastructure to adequately protect it.
- Finally, in peer-to-peer systems it is desirable, specifically not to have a central institution that the system relies on.

In this section we explain a fully distributed certification mechanism. A fully distributed mechanism has certain disadvantages, notably the absence of a central location that can aggregate results, keep a list of all sites that are involved and provide a comprehensive event log.

5.1 System overview

The distributed CA consists of the following components:

- **Clients:** A client wants to exchange data with a specific server. It requires the server's public keys to be certified to ensure that the server is indeed who it pretends to be.
- **Servers:** A server processes requests from clients. To demonstrate that the server's public key is correct, it requires the server to provide them with a valid certificate. Each server has a unique name s and a public/private key pair (pk_s, sk_s).
- **Judges**: A judge j has a public/private key pair (pk_j, sk_j). There is no central list of judges, any host can be a judge if it desires. A host has an incentive to become a judge if it wants to use certificates. It is not possible to just passively use certificates without being a judge.

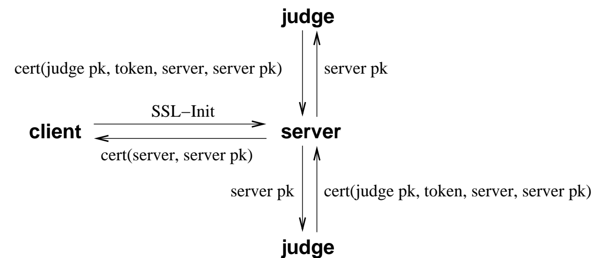

Figure 6: Distributed CA: Clients issue mini certificates to the servers, and the server publishes the list of mini certificates which is basically the certificate.

5.2 Design Issues

In a centralized system trust is mediated through a central authority. It collects testimonials of judges and aggregates them to a verdict on whether a certain key belongs to a certain identity. In a distributed model this is no longer possible. Instead we use the testimonials of other entities directly. The problem with this approach is twofold. First, we have to make sure that an attacker can not create a large number of fake identities and use them to certify a site. We address this with a mechanism called *authentication tokens*. Second, the number of testimonials for a site can become very large. We need a robust but efficient way for a site to prove that it has received a large number of testimonials.

5.2.1 Distributed Certification using Authentication Tokens

In a distributed CA, the capability to issue certificates is distributed among all existing judges. The problem is that in a truly distributed system there is no central instance that records the identity of the judges. An attacker could introduce a large number of fake judges into the system and use their voting capability to issue incorrect certificates. This is similar to an election. Voting is restricted to one vote per person, and verification of the person's identity at the polling station prevents people from casting several votes. However, for online identities it is not possible to verify whether an identity is real or fake.

In order to solve the problem of fake identities we couple certification power to computational power that has a distribution similar to the one we desire for our certification capability. This is a well-known principle from cryptography where computationally hard problems prevent an easy conversion from a given ciphertext to its plaintext. A similar technique of client puzzle we propose has also been used to control junk emails [12, 6]. Computational power has the added advantage that while most CPUs are idle and thus spare CPU power is abundant, there exists no market that allows an entity to buy a large amount of this CPU power.

The system works as follows: Each judge j that wants to certify a random server s has to solve a challenge that requires exclusive control over a certain amount of CPU power. Each judge j has a public/private key pair pk_j, sk_j. The challenge we used for this paper is to find a token T that results in a collision in a cryptographic hash function, e.g.:

$$hash(T|pk_j) = hash(T|pk_j|c) \quad (8)$$

In this equation c is a public constant string in the system. By selecting a suitable hash function or requiring equality only for a certain range of bits we can adjust the hardness of the puzzle. Given the T from equation 8 we now compute

$$T' = hash(T) \quad (9)$$

Every T' is an *authentication token* that certifies exactly one server. We use the last h bytes of T' to address the host the token is determined for. For example this can be an IP address or a DNS host name. The remaining $|T'| - h$ bytes of the token define the quality $Q(T_i')$ that we will use below for determining the trust level of a server.

Using the authentication token T' and its private key sk_j, judge j can issue a certificate $\text{cert}_{j,s}$ for server s that publishes public key pk_s:

$$\text{cert}_{j,s} = \{pk_j, T', s, pk_s\}_{sk_j} \quad (10)$$

When a client wants to verify a certificate from server s it needs to do three things:

1. Given s in $\text{cert}_{j,s}$ check whether the server name matches.
2. Given pk_j in $\text{cert}_{j,s}$ check whether the certificate has been signed by sk_j.
3. Given T and pk_j in $\text{cert}_{j,s}$ check whether equation 8 holds.

If all three checks pass the certificate is valid.

Since there does not exist a trusted instance that issues certificates we need a more elaborate scheme in order to establish trust. The observation is that many judges in the system will send certificates to a server, and therefore we will exploit statistical properties to ensure trust in the system.

5.2.2 Establishing trust

We solve the problem of untrusted certificate issuers by simple statistics. Any authentication token value T' from equation 9 is uniformly distributed. Hence, we know that for all received tokens T_i' the average quality is

$$S_{1,n} := \frac{1}{n}\sum_{i=1}^{n} Q(T_i') = \frac{q+1}{2} \quad (11)$$

where for any token $T_i' : 1 \leq Q(T_i') \leq q$ when q is the maximum quality of a token.

In order to be efficient we want a server to only store the best k out of the n received tokens, but these k tokens are sufficient for telling how many tokens n the server has received in total. W.l.g. we assume that T_i' are ordered such that $Q(T_i') \geq Q(T_{i+1}')$.

From equation 11 we get

$$nS_{1,n} = kS_{1,k} + (n-k)S_{k+1,n} \tag{12}$$

and hence

$$n = \frac{kS_{1,k}}{S_{1,n} + kS_{k+1,n}}. \tag{13}$$

If we assume uniform random distribution of T_i and its subsets, then we can do the following approximation for any arbitrary k:

$$S_{k+1,n} \approx \frac{Q(T_k')+1}{2} \tag{14}$$

Using equation 13 we get

$$n \approx \frac{2kS_{1,k}}{q + k(Q(T_k')+1)+1} \tag{15}$$

If we know the best k tokens we can use this equation to approximate the total number of tokens a server has received. It is not possible for a server to cheat since when it publishes less valuable tokens n and hence the trust level will decrease.

5.2.3 Issuing certificates

Judges continuously spend a certain percentage of their CPU power on searching for authentication tokens and use them to certify sites. This process works as follows:

1. The judges generates a token T' according to equations 8 and 9
2. The judge contacts server s using for which the token is determined using SSL as a regular client.
3. The server responds with a public key pk_s and a list of k certificates C_s.
4. The judge uses the token T' and the public key of the site to issue a certificate $\text{cert}_{j,s} = \{pk_j, T', s, pk_s\}_{sk_j}$ signed with its secret key sk_j.
5. The server accepts the certificate and acknowledges the receipt to the judge
6. The site checks whether the token value T' in $\text{cert}_{j,s}$ is better than the worst value of its currently published list C_s. If this is the case it puts $\text{cert}_{j,s}$ into C_s and removes the certificate having the worst token value.

5.2.4 Verifying certificates

The verification process works as follows:

1. The client connects to server s.
2. The server s sends the client its list of k best certificates C_s.
3. The client verifies all certificates as described in section 5.2.1.
4. The client calculates n using equation 15 to estimate the number of judges that have certified the site. This is the trust-level for server s.

5.2.5 Operations

In the distributed CA there is no physical distinction between client, judge, and server. As in a peer-to-peer network every node incorporates both functions. In the specification of the operations, however, we talk about client and server in order to make clear the role of the nodes in that particular context.

- **request_certificate(server) (client → server):**
  (server: *client → server*) ⇒
  ($\{\text{cert}_{j,server}(pk_j, T, server, pk_{server})\}$:
  server → client)
- **update_certificate(server, pk) (judge → server):**
  (*T' ready* ⇒

(SSL-init: *judge* $\rightarrow$ *server*) $\Rightarrow$
($\{server,\ pk_{server}\}$: *server* $\rightarrow$ *judge*) $\Rightarrow$
($\text{cert}_{judge,server}(pk_{judge}, T', server, pk_{server})$:
judge $\rightarrow$ *server*)))

5.3 Security Analysis

Below we discuss the different types of attacks that can be launched on a distributed CA. We assume communications between judges, sites and clients are encrypted and signed.

- **Denial of Service Attacks:**

  DoS attacks against a site would block certification packets from judges and thus weaken a site's overall certificate set. This can easily be done by a party controlling an upstream router of a site. The judge would detect such an attack as he would never receive an acknowledgement for the certificate. The site can also detect such an attack by comparing the statistics of certificates it receives to another site or posing as a judge itself.

- **Man-in-the-middle Attack:**

  In a man-in-the-middle attack the attacker the attacker can read all traffic going to and from the site. If a judge connects to the site he will receive and issue the certificate with the attackers public key. As judges pose as regular clients selective handling of judges is not possible. With a man-in-the-middle attack it is not possible for an attacker to selectively steal "good" tokens as the certificate the includes the certificate is already sent to the site encrypted with the site's public key.

- **Controlling a large number of judges:**

  An attacker can attempt to generate fake certificates by controlling a large fraction of all the CPU power in the system. Basically, the attacker sets up a malicious server with a trusted domain name and tries to generate a certificate of good quality for this malicious server. At the same time the judges certify the trusted server with the same domain name. Whenever the quality of the tokens for the malicious server is better the attacker wins.

In section 5.2.2 we derive the trust level for a server from the number of tokens it receives since the quality is uniformly distributed. Hence, in the average case the attacker will always loose as long as good judges outnumber malicious ones. In a real case however an attacker has a small probability to find a better set of tokens.

For the analysis we assume that there are n_g good judges and n_m malicious judges. The hardness of the hash function and the quality threshold are set in a way that in a given time interval t a site will receive enough tokens of sufficient quality such the worst out of the best k tokens will be above the quality threshold q_{min}.This is sufficient for a site to authenticate itself to a client. We also assume that mini-certificates expire after the time t. Q denotes the maximum possible quality a token can have.

The probability distribution of the quality of a token T is:

$$p(Q(T) > q_{min}) = \frac{Q - q_{min}}{Q} \tag{16}$$

For the case $k = 1$ in the time t the good judges would generate about one token for a web site. The chance of the malicious judges to find an equally good token in the same time would only be:

$$p(q_m > q_g) = \frac{n_m}{n_g} \tag{17}$$

By using order statistics we can make this more robust. Assuming we base our threshold on the worst of the k best certificates we get approximately:

$$p(q_m > q_g) \approx \left(\frac{n_m}{n_g}\right)^k \tag{18}$$

The largest known network of PC's [2] claims to have used a maximum of 500,000 PCs over it's life time. Overall we estimate that currently no entity controls more than 0.01% of all CPU power worldwide. Hence, the security guarantee of the distributed CA will be sufficiently high for a low-cost CA, even when only a fraction of the world-wide hosts participate.

5.4 Discussion

The lack of a central authority that can aggregate results leads to higher overhead for two principal reasons. First, a representative selection of judges is no longer possible and a much larger number of them has to be active at any time. While the number of judges required to judge a site is independent of the number of sites in the accountable CA, it is a set fraction of all possible hosts in the fully distributed case. Secondly the certificate verification process is more complex. For a central CA a single certificate has to be verified, in the distributed case the number of certificates depends on the ratio of good and bad judges in the system.

We believe that a fully distributed CA is currently mainly of interest for peer-to-peer type networks that change their structure frequently and do not have a centralized institution that could serve as a CA.

6 Related Work

Public key infrastructures are widely studied field. Examples of its use include secure web access [11] or secure DNS for distributing keys is [13].

Reputation Systems [15, 14] are closely related to the work described in this paper. Like our method of "Trust over Time" they attempt to aggregate past experiences of users into a verdict about an entity. Our centralized CA is similar to a reputation system that specializes on the reputation of public keys for communications. The major difference is that certificates requires an automated, binary decision both at the time of certification as well as at the time when a certificate is used. Also most reputation systems assume that an attack using false testimonials is at least detectable. Our certification authority is able to withstand more sophisticated attacks than most reputation systems. However we believe our results on building a fully distributed CA are equally applicable to building a robust and fully distributed reputation system.

The Secure Shell Protocol SSH [19] stores the public key of a remote site locally the first time the site is accessed. This "leap of faith" makes man-in-the-middle attacks in for connections impossible, however it does not prevent them for the initial connection. It has no mechanism to propagate the trust it has in a public key to other clients.

When using identity-based encryption [8] certificates become unnecessary as the identity of a recipient can be directly encoded in the public key. However the private key generator in an identity based system has to decide to whom it should issue the private key that corresponds to an identity.

Some work has taken place designing distributing CAs [20, 17, 9], but most of this work focuses on fault-tolerance of existing CAs whereas this paper tries to establish inexpensive CAs. These approaches use threshold cryptography [10] to split the secret signing key. [7] explains some theoretical groundwork on distributed trust management.

7 Conclusion

In this paper we explain how to design low-cost Certification Authorities (CAs) that have good security guarantees. First, we show that it is possible to distinguish between trusted and untrusted servers in a simple CA using trust over time. In the second part we extend this principle of trust over time and build an accountable CA and conclude that in a typical situation about 28% of the sites that the CA picks for being a judge needs to be malicious in order to compromise the system. In the last part we further reduce the security risk factors of a central CA and create a fully distributed CA. In this distributed CA an attacker needs to compromise a fixed fraction of all sites in order to compromise the system over the long term. Overall, these CAs provide a viable alternative for certifying low-volume servers and nodes in peer-to-peer networks.

References

[1] Cheap SSL Certificates for Small Websites ? Slashdot 02/10/2002.

[2] distributed.net. http://www.distributed.net.

[3] FreeSSL. http://www.freessl.com.

[4] Self-Regulating SSL Certificate Authority ? Slashdot 01/21/2003.

[5] Why are SSL Certificates so Expensive ? Slashdot 03/18/2001.

[6] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, Feb. 2003.

[7] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[8] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *Lecture Notes in Computer Science*, 2139:213–229, 2001.

[9] C. Cachin. Distributing trust on the internet. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 183–192, July 2001.

[10] Y. Desmedt and Y. Frankel. Threshold cryptosystems. *Lecture Notes in Computer Science*, 435:307–315, 1989.

[11] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1, Jan. 1999. Status: PROPOSED STANDARD.

[12] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. *Lecture Notes in Computer Science*, 740:139–147, 16–20 Aug. 1992.

[13] J. M. Galvin. Public key distribution with secure DNS. In *Proceedings of the 6th USENIX Security Symposium*, pages 161–170, 1996.

[14] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. Eigenrep: Reputation management in p2p networks. Technical Report 2002-56, Stanford InfoLab, Department of Computer Science, Stanford University, 2002.

[15] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.

[16] R. L. Rivest. Can we eliminate certificate revocations lists ? *Lecture Notes in Computer Science*, 1465:178–183, 1998.

[17] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, 1996.

[18] J. Rosenberg and D. Remy. GeoTrust quickssl white paper. http://www.geotrust.com/resources/white_papers/, Dec. 2001.

[19] T. Ylönen. SSH - Secure Login Connections Over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, 1996.

[20] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. Technical Report 1828, Department of Computer Science, Cornell University, 2000.