Rating Certificates

Eunjin (EJ) Jung and Mohamed G. Gouda
Department of Computer Sciences
The University of Texas at Austin
Email:{ejung,gouda}@cs.utexas.edu

***Abstract*— We consider a system where each user has a public key and a private key. In this system, a certificate is a data item that is issued by one user u and contains the public key of another user v. A third user w that knows the public key of u can verify that this certificate has not been corrupted (by an adversary) since it was issued by u, and so can accept the public key in the certificate as the correct public key of v. User w can use this accepted public key of v in two ways. First, w can securely communicate with v. Second, w can obtain more public keys of other users, as it used the public key of u to obtain the public key of v. However, the safety of the second use is questionable if u, the issuer of the certificate, has concluded that it cannot trust v enough to accept a public key merely because v accepts it. To solve this problem, we propose that each certificate should have a "rating". The rating of a certificate describes how much trust the issuer puts on the subject concerning key acceptance. We present two algorithms for computing the set of all users that can accept the given public key where all certificates have ratings. The first algorithm is simple, but its time complexity is $O(n^3)$, where n is the number of users in the system. The second algorithm is more sophisticated, but its time complexity is $O(e)$, where e is the number of certificates in the system. This algorithm meets the lower bound of the worst case complexity. We also discuss how to find an input to these two algorithms, and present two algorithms that compute an optimal set of certificates that are necessary for a user to accept the public key of another users.**

I. Introduction

Many of the security problems in the Internet can be solved by a collection of basic security building blocks such as authentication, privacy, integrity, non-repudiation, and authorization. For example, if we can authenticate the source of packets, denial of service attacks can be diminished. Also, if we can guarantee privacy and authentication in transactions on the Internet, identity theft can be reduced. In the Internet, certificate systems are extensively used to provide some of those basic security building blocks.

There are two types of certificate systems: one with trusted certificate authorities, and the other without trusted certificate authorities. An example of certificate systems with trusted certificate authorities is Secure Socket Layers (SSL) [1], and an example of certificate systems without trusted certificate authorities is Pretty Good Privacy (PGP) [2].

In a certificate system with trusted certificate authorities, users can use any certificate issued by one of the trusted certificate authorities to securely communicate with other users. For example, in the current Internet, users trust any certificate issued by VeriSign and use the public key in the certificate for secure communication over SSL. The certificate system in SSL can be represented as an hourglass certificate graph. An example of an SSL certificate system is shown in Fig. 1. Each node represents a user and each edge represents a certificate.

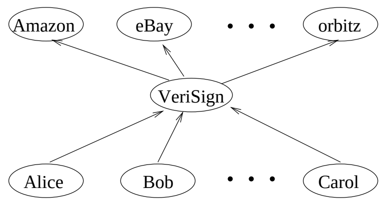

Fig. 1. An example of an SSL certificate system

This type of certificate systems has limitations. In most certificate systems, there are much fewer trusted certificate authorities than users. The overhead of issuing certificates is on the few trusted certificate authorities, and also if a private key of a certificate authority is revealed to an adversary, the scope of the damage can be catastrophic. We can use a hierarchical structure of certificate authorities to distribute the load and the risk of a few certificate authorities, as in Lotus Notes [3]. However, the construction and planning of hierarchical structure of certificate authorities requires a single point of control for the whole system.

A certificate system without trusted certificate authorities is called self-organized. In a self-organized certificate system, users issue certificates for other users. There may be a key server which stores public keys of users and certificates issued by users, but the key server does not guarantee that these certificates are correct, i.e. that they contain the correct public keys. Therefore, in such a system, it is not clear whether users can trust the certificates issued by other users or not. An example of a certificate system without trusted certificate authorities is PGP.

PGP certificate systems are self-organized: any user can issue a certificate for another user. Imagine a user Alice creates her own public and private key pair and advertises the public key to her friends, including Bob. In turn, Bob issues a certificate for Alice's public key, so that Carol, who does not know Alice's public key, can use the public key of Bob and his certificate to obtain Alice's public key. Later, David issues a certificate for Carol. The certificate system between four users, Alice, Bob, Carol, and David is shown in Fig. 2.

In the example PGP system in Fig. 2, Carol already issued a certificate for Bob, which indicates that Carol knows the public key of Bob. However, Carol does not trust the certificates

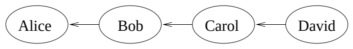

Fig. 2. An example of a PGP certificate system

issued by Bob. Hence, she does not accept the key in the certificate issued by Bob for Alice as the public key of Alice. The problem is, when another user David needs to securely communicate with Alice, he needs to know whether the certificate issued by Bob is trustworthy or not, before using the key in the certificate as the public key of Alice. To solve this problem, Carol needs to share her trust information about Bob with David.

This is not a problem in a certificate system with trusted certificate authorities, such as SSL, since any certificate issued by trusted certificate authorities should contain correct public keys. However, in a self-organized certificate system such as PGP, each user needs additional information to decide whether to accept the public key in each certificate as the correct public key or not.

In the following sections, we define our system model more formally, and discuss the problems in using certificates to obtain public keys of other users. We show what information can be added to certificates so that users can share their trust information about other users. We discuss the semantics of the added information. We present two algorithms that compute, for a given public key and a given set of certificates, the set of all users that can accept this public key as the correct one. We continue to discuss how to obtain the input set of certificates for these algorithm, and present two algorithms that compute an optimal set of certificates that are necessary for a user to accept the public key of another user.

II. Certificate Systems

We consider a system where each user u has a private key $R.u$ and a public key $B.u$. In this system, in order for a user u to securely send a message m to another user v, user u needs to encrypt the message m using the public key $B.v$, before sending the encrypted message, denoted $B.v\{m\}$, to user v. (This message may be a session encryption/decryption key for further secure communication.) This necessitates that user u know the public key $B.v$ of user v.

If a user u knows the public key $B.v$ of another user v in this system, then user u can issue a certificate, called a certificate from u to v, that identifies the public key $B.v$ of user v. This certificate can be used by any user in the system that knows the public key of user u to further acquire the public key of user v. Note that this model encompasses both certificate systems with and without trusted certificate authorities. A certificate system with trusted certificate authorities will have only certificates issued by those certificate authorities, whereas a certificate system without trusted certificate authorities may have certificates issued by any user in the system.

A certificate from user u to user v is of the following form:

$$\langle u, v, B.v, \texttt{info}, \texttt{sig}\rangle$$

This certificate is signed using the private key $R.u$ of user u, and it includes five items:

u	is the identity of the certificate issuer,
v	is the identity of the certificate subject,
$B.v$	is the public key of the subject v,
`info`	is other relevant information regarding this certificate such as expiration date, and
`sig`	is an encrypted message digest of this certificate. It is constructed by computing a message digest of all other four items in this certificate and encrypting the message digest with the private key $R.u$ of issuer u.

For simplicity, a certificate $\langle u, v, B.v, \texttt{info}, \texttt{sig}\rangle$ is denoted (u, v). Any user x that knows the public key $B.u$ of user u can use $B.u$ to decrypt `sig` in (u, v). User x continues to compute the message digest of all other four items in the certificate. If the decrypted message matches the message digest computed by user x, then user x can accept the key $B.v$ in certificate (u, v) as the public key of user v.

The certificates issued by different users in a system can be represented by a directed graph, called the *certificate graph* of the system. Each node u in the certificate graph represents a user u and its corresponding public and private key pair $B.u$ and $R.u$. Each directed edge (u, v) from node u to node v in the certificate graph represents a certificate $\langle u, v, B.v, \texttt{info}, \texttt{sig}\rangle$.

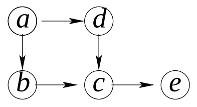

Fig. 3. A certificate graph example

Fig. 3 shows a certificate graph for a system with five users: a, b, c, d, and e. According to this graph,

user a issued two certificates (a, b) and (a, d)
user b issued one certificate (b, c)
user c issued one certificate (c, e)
user d issued one certificate (d, c)
user e issued no certificates.

A simple path (v_0, v_1), (v_1, v_2), $\cdots$, (v_{k-1}, v_k) in a certificate graph G, where the users $v_0, v_1, \cdots, v_k$ are all distinct, is called a *certificate chain* from v_0 to v_k in G of length k. v_0 is the *source* of the chain and v_k is the *destination* of the chain. Source v_0 of the certificate chain from v_0 to v_k can accept all the keys $B.v_1 \cdots B.v_k$ in these certificates as public keys of the users $v_1 \cdots v_k$ in this chain, respectively. For example, user a in Fig. 3 may use the certificate chain $(a, b)(b, d)$ to accept the public keys $B.b$ and $B.d$ of user b and user d.

III. The problem of trust in certificate systems

The use of certificate chains to accept public keys of other users is based on the assumption that all the certificates in the chain are correct, i.e. all the keys in the certificates are indeed the correct public keys of the subjects. However, this may not be the case all the time. Some users may not take enough precautions before issuing certificates and as a result issue a certificate with a wrong public key for the subject. Some user may intentionally issue incorrect certificates to impersonate other users. When a user knows that some user is not trustworthy or is suspected of issuing incorrect certificates, this trust information needs to be shared with other users.

PGP manages this trust information for each user, but does not support sharing. In PGP, a user puts the public keys of other users that it believes to be correct in its local *public key ring*. Each entry in the key ring contains an ID of another user, its public key and other information.

For each public key in the key ring, users can choose from four levels of trust, `completely trusted`, `marginally trusted`, `untrusted`, and `unknown` to assign to. (The default level of trust is `unknown`.) Note that this level of trust is *not* trust on whether the public key is the correct one for that user or not. Rather, the level of trust is trust on the public key as a signing key. In the default setting of PGP, for a user to accept a key $B.x$ as the public key of a user x, it needs to find either one `completely trusted` public key or two `marginally trusted` public keys in its own key ring, that sign certificates which has the same key $B.x$ as the public key of user x. Fig. 4 shows the example system with level of trust in each user's key ring. For example, David in the system in Fig. 4 `completely trusted` Carol's public key in his key ring, so David accepts Bob's public key in the certificate $(Carol, Bob)$.

PGP users can tune their own security settings by changing COMPLETES_NEEDED, MARGINALS_NEEDED, and CERT_DEPTH. The first two parameters indicate how many trusted keys are needed to accept a public key. CERT_DEPTH indicates how long a certificate chain can be to accept the public key in the last certificate in the chain.

However, there are two problems with the current level of trust scheme in PGP. First, the trust information is not shared with other users. In the example in Fig. 4, David `completely trusted` Carol's public key, and Carol has `untrusted` Bob's public key. There is a certificate chain from Carol to Alice, $(Carol, Bob)(Bob, Alice)$. However, Carol will not accept this public key of Alice because Carol has no trust on Bob's public key to introduce a new key. When David wants to obtain Alice's public key, David may want to use the certificate chain $(David, Carol)(Carol, Bob)(Bob, Alice)$. Even though David is relying on Carol's public key in the certificate chain $(David, Carol)(Carol, Bob)(Bob, Alice)$, David may make a different, in fact more vulnerable decision from Carol's and accept the public key in this chain as the public key of Alice.

Second, it is not clear how users can decide which level of trust they should assign to public keys. When a user u assigns `completely trusted` to a public key of user v, $B.v$, does it mean that u will accept any public keys in certificates signed by the matching $R.v$, or u will accept any public keys in certificate chains starting with (u, v)? The current specification of PGP does not specify the semantics clearly.

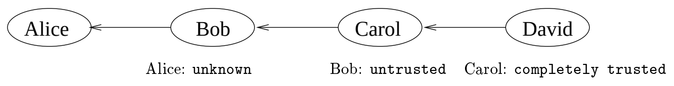

Fig. 4. An example of a PGP certificate system with level of trust

This example shows that users need to share their trust information with other users regarding which certificates are trustworthy. Moreover, this trust information needs to be signed so that other users can verify that this information is not tampered by an adversary. One way to share this trust information in a tamper-resistant manner is to add this trust information to certificates. In the following section, we propose a solution to this problem by adding "ratings" (trust information) to certificates.

IV. Ratings in Certificates

To solve the two problems discussed in the previous section, one could imagine adding the trust level assigned by each user to a certificate. For example, Carol in Fig. 4 could add the level `untrusted` to the certificate $(Carol, Bob)$ so that David could decide not to use the certificate $(Bob, Alice)$. However, this only solves the first problem of sharing trust information, not even completely. The level `marginally trusted` needs a parameter `MARGINALS_NEEDED` to be effective. One can add this parameter as part of the certificate as well. Still, the second problem of how to choose the appropriate level is not solved. This is because PGP treats a public key as a "signing key". For David to accept a key in a certificate chain, David needs to trust each public key in this chain as a signing key. David could use the trust information in the certificate, if added, but this trust information is assigned by a stranger, so David may not feel comfortable trusting that information.

We add a new field called "rating" to a certificate: a rating of a certificate is trust information that the issuer has on the key acceptance decision made by the subject. Now a certificate from user u to user v with a rating contains six items as follows:

$$\langle u, v, B.v, \texttt{rating}, \texttt{info}, \texttt{sig} \rangle$$

. u, v, $B.v$, `info`, and `sig` are the same as explained in Section II. `rating` is a rating which can be any one of the following that will be explained shortly:

```
accepted
independent
k-accepted
```

For simplicity, a certificate of a form

$$\langle u, v, B.v, \texttt{rating}, \texttt{info}, \texttt{sig} \rangle$$

is denoted $(u, v, \texttt{rating})$ where `rating` is

A	for	`accepted`
I	for	`independent`
$A(k)$	for	k-`accepted`

A certificate $(u, v, \texttt{rating})$ with rating provides two pieces of important information: the public key of user v and the rating of user v by user u. Any user w that knows the public key of user u can accept the public key of user v as long as they could verify the certificate. Moreover, user w can use the `rating` information in this certificate to decide whether to accept the public keys accepted by user v or not.

1) `accepted`: A certificate of the form

$$\langle u, v, B.v, \texttt{accepted}, \texttt{info}, \texttt{sig}\rangle$$

indicates that u accepts $B.v$ as the public key of user v and that u also accepts any public key $B.x$ of a user x, if v accepts the same key $B.x$ as the public key of user x.
In Fig. 5, user u issues a certificate (u, v, A) and user v issues two certificates $(v, w, \texttt{rating})$ and $(v, x, \texttt{rating})$. Clearly v accepts both keys $B.w$ and $B.x$ as the public keys of users w and x, and the rating of the certificate (u, v, A) is `accepted`, so user u accepts the same keys $B.w$ and $B.x$ in certificates $(v, w, \texttt{rating})$ and $(v, x, \texttt{rating})$ as the public keys of user w and user x, respectively.

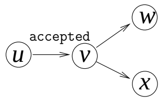

Fig. 5. An example of `accepted` certificates

Note that u will accept the public keys of users w and x in the certificate chains $(u, v, A)(v, w, \texttt{rating})$ and $(u, v, A)(v, x, \texttt{rating})$ regardless of the ratings in the certificates $(v, w, \texttt{rating})$ and $(v, x, \texttt{rating})$.

2) `independent`: A certificate of the form

$$\langle u, v, B.v, \texttt{independent}, \texttt{info}, \texttt{sig}\rangle$$

indicates that u accepts $B.v$ as the public key of user v. It also indicates that whether u accepts a key $B.x$ as the public key of a user x or not is independent of whether v accepts the same key $B.x$ as the public key of user x or not. In other words, user u does not accept any key $B.x$ as the public key of a user x just because v accepts the same key $B.x$ as the public key of user x.
In Fig. 6, user u issues two certificates (u, v, I) and (u, w, A), user v issues a certificate $(v, x, \texttt{rating})$, and user w issues a certificate $(w, x, \texttt{rating})$. Since user u issues a certificate to user v, u clearly accepts the public key $B.v$ of user v. However, user u would not accept $B.x$ as the public key of user x if there were only one certificate chain from u to x through v $(u, v, I)(v, x, \texttt{rating})$ in Fig. 6, because the rating of certificate (u, v, I) is `independent`. In the example certificate system in Fig. 6, there is another certificate chain from u to x $(u, w, A)(w, x, \texttt{rating})$, and the rating of (u, w, A) is `accepted`. Since w accepts $B.x$ as the public key of user x, u accepts $B.x$ as well.

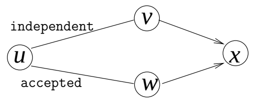

Fig. 6. An example of `independent` certificates

3) k-`accepted`: A certificate of the form

$$\langle u, v, B.v, k\texttt{-accepted}, \texttt{info}, \texttt{sig}\rangle$$

indicates that u accepts $B.v$ as the public key of user v. It also indicates that u accepts a key $B.x$ as the public key of user x, if u has also issued certificates for users $v_1 \cdots v_{k-1}$,

$$(u, v, A(k)), (u, v_1, A(j_1)), \cdots, (u, v_{k-1}, A(j_{k-1})),$$

where each $j_i \le k$, and each v_i accepts the same $B.x$ as the public key of user x. (Intuitively, the issuer has $\frac{1}{k}$ of the full trust on the subject.)
In Fig. 7, user u issues three certificates $(u, v, A(2))$, $(u, w, A(3))$, and $(u, y, A(3))$. User v issues a certificate $(v, x, \texttt{rating})$, user w issues a certificate $(w, x, \texttt{rating})$, and user y issues a certificate $(y, x, \texttt{rating})$. Clearly users v, w, and y accept the key $B.x$ in $(v, x, \texttt{rating})$, $(w, x, \texttt{rating})$, and $(y, x, \texttt{rating})$ as the public key of user x. User u issued three certificates $(u, v, A(2))$, $(u, w, A(3))$, and $(u, y, A(3))$, so it also accepts the key $B.x$ in the certificates $(v, x, \texttt{rating})$, $(w, x, \texttt{rating})$, and $(y, x, \texttt{rating})$ as the public key of a user x. Note that the certificates (v, x), (w, x), (y, x) must include the same key $B.x$ as the public key of user x for user u to accept it.

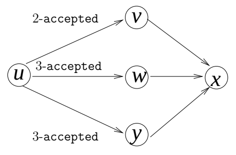

Fig. 7. An example of k-`accepted` certificates

Note that the semantics of these ratings are for key acceptance, not for key signing. This enables simpler use of ratings. An issuer of a certificate can assign the rating for the subject depending on whether to accept a key just because this subject accepts the key or not. Even though this trust information will be shared with all other users, the decision itself is a local decision of the issuer.

V. Key Acceptance Examples

In the previous section, we defined three ratings for certificates, and showed their semantics. Consider the example certificate graph G in Fig. 8. Each edge (u, v) in G is labeled with the rating of the corresponding certificate $(u, v, \texttt{rating})$. Based on these ratings, we can compute the set C of all users that can accept $B.dst$ as the public key of user dst.

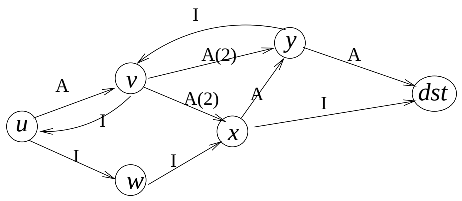

Fig. 8. An example of a certificate graph with ratings

The set C can be computed in three steps.

1) First, observe that each of the users x and y can accept $B.dst$ as the public key of user dst, since each of them has already issued a certificate declaring that the public key of dst is $B.dst$. Note that the rating in the certificate (x, dst, I) is `independent`. No matter what the rating is in the certificate (x, dst), x declared that the public key of dst is $B.dst$ in this certificate, so x accepts $B.dst$ as the public key of user dst.
2) Second, user v can accept $B.dst$ as the public key of user dst since v has issued two certificates with a rating $A(2)$ each, for users x and y (and x and y can accept $B.dst$ as the public key of user dst from the first step of our discussion). Note that user w cannot accept $B.dst$ as the public key of user dst even though w has issued a certificate for user x (and x can accept $B.dst$ as the public key of user dst, as discussed above), since the rating of the issued certificate (w, x, I) is `independent`.
3) Third, user u can accept $B.dst$ as the public key of dst since u has issued a certificate to v with rating A (and v can accept $B.dst$ as the public key of user dst from the second step of our discussion).

The set C is therefore:

$$\{dst, x, y, v, u\}$$

In the following section, we present two algorithms that user u can use to decide whether to accept a public key or not based on a certificate graph. Note that this input certificate graph does not have to be the full certificate graph of the system. For example, user u does not need the certificate chain $(u, w)(w, x)(x, dst)$, since the ratings in (u, w) and (w, x) are `independent`. In Section VII, we discuss how to compute the input certificate graphs for the following two algorithms.

VI. Key Acceptance Algorithms

The example discussed in the previous section is in fact how Algorithm 1 below, given a certificate graph G and a user dst in G, computes a set C of all users that can accept $B.dst$ as the public key of user dst.

ALGORITHM 1 computes the set of accepting users

```
INPUT: a certificate graph $G$ and a user $dst$ in $G$
OUTPUT: a set $C$ of users that can accept $B.dst$
        as the public key of user $dst$

STEPS:
1:  $C := \{dst\}$;
2:  for each certificate $(u, dst, \texttt{rating})$ in $G$ do
3:      $C := C \cup \{u\}$
4:  endfor;
5:  while users can be added to $C$ do
6:      for each user $u$ not in $C$ do
7:          if there is a user $x$ in $C$ such that $(u, x, A) \in G$
8:          then $C := C \cup \{u\}$
9:          else if there are $k$ users $v_0 \cdots v_{k-1}$ in $C$ such
            that for each $v_i$, $(u, v_i, A(k_i)) \in G$ and $k_i \leq k$
10:         then $C := C \cup \{u\}$
11:     endfor;
12: endwhile;
13: return $C$
```

If we apply Algorithm 1 to the example certificate graph in Fig. 8, we compute the following set C of users that can accept $B.dst$ as the public key of user dst.

$$C = \{dst, x, y, v, u\}$$

The complexity of this algorithm is $O(n^3)$, where n is the number of users in the certificate graph. In each execution of the **while** loop in lines 5-13, at least one user has to be added to C, so the **while** loop can be executed at most n times. Also, for each **for** loop inside the **while** loop, the number of users that may not be in C is at most n, so the **for** loop can be executed at most n times. Also, to find the users that match the **if** condition inside the **for** loop, Algorithm 1 may consider up to n users. In total, the complexity of Algorithm 1 is $O(n^3)$. The correctness of Algorithm 1 is straightforward from the definition of ratings.

Algorithm 2 also takes the same input, a certificate graph G and a user dst in G and computes the same output, a set of users that accept the key $B.dst$ in G as the public key of user dst. However, Algorithm 2 computes the set in much less complexity than Algorithm 1. The intuition of Algorithm 2 comes from constructing a minimal spanning tree. Starting from user dst in G, Algorithm 2 considers each certificate (u, v) that is issued by a user u, $u \notin C$, for a user v, $v \in C$. If the certificate has the rating of `accepted`, then user u is added to C and the certificates issued for u will be considered later. Algorithm 2 continues adding users to C until there is no more certificate issued by a user that is not in C for a user in C.

The complexity of Algorithm 2 is $O(e)$, where e is the

ALGORITHM 2 optimally computes the set of accepting users

```
INPUT: a certificate graph $G$ and a user $dst$ in $G$
OUTPUT: a set $C$ of users that accept $B.dst$
        as the public key of user $dst$

STEPS:
1: $C := \{dst\}$; $Z := \{\}$;
2: for each certificate $(u, dst, \texttt{rating})$ in $G$ do
3:     $C := C \cup \{u\}$;
4:     add all the certificates $(x, u, \texttt{rating})$ issued for $u$
       in $G$ to $Z$
5: endfor;
6: for each user $u$ in $G$, c[u] := 0;
7: for each certificate $(u, v, \texttt{rating})$ in $Z$ do
8:     remove $(u, v, \texttt{rating})$ from $Z$;
9:     if $(u, v, \texttt{rating})$=$(u, v, A)$ and $u \notin C$
10:    then $C := C \cup \{u\}$;
11:        add all the certificates $(x, u, \texttt{rating})$ issued
           for $u$ in $G$ to $Z$
12:    else if $(u, v, \texttt{rating})$=$(u, v, A(k))$ and $u \notin C$
13:        then c[u] := c[u]+$\frac{1}{k}$;
14:            if c[u]$\geq$1
15:            then $C := C \cup \{u\}$;
16:                add all the certificates $(x, u, \texttt{rating})$
                   issued for $u$ in $G$ to $Z$
17: endfor;
18: return $C$
```

number of certificates in the certificate graph. Each certificate (u, v) in G is added to Z only when user v is added to C, and each user v is added to C only once. (Initially the users that have issued certificates for dst are added to C, and after that, the conditions in line 9 and 12 assure that v is not in C before it is added.) Therefore, each certificate in G can be added to Z at most once. Once it is added, each certificate (u, v) will be considered exactly one time. The first step in the **for** loop in lines 7-17 is to remove the certificate, and then the same certificate (u, v) cannot be added to Z again since v is already in C. So the **for** loops in line 2 and the **for** loop in lines 7-17 can be executed at most e times in total. **for** loops in line 6 takes n steps, as it initialize n elements in the array. In total, the complexity of Algorithm 2 is $O(n + e)$. Since the input graph is a certificate graph, it is safe to assume that $n \leq e$. It is because that any user in the certificate graph should have at least one certificate that is either issued by this user or for this user. Therefore, $O(n + e)$=$O(e)$.

We prove Algorithm 2 correct by induction on the length i of certificate chains. For $i = 1$, any user who has issued a certificate for dst will be added to C (line 3). Assume any user that were to accept $B.dst$ as the public key of dst based on the ratings in certificate chains of length at most i is added to C. Any user u that has certificate chains of length at most $i+1$ to dst will be added to C if it has issued a certificate with `accepted` rating for any node in C (line 9). Similarly, if it has issued k certificates for users in C with k_i-`accepted` ratings, where $k_i \leq k$, it will have $c[u] \geq 1$ and be added to C (line 14). There are no other ways users could accept $B.dst$. Therefore, the set computed by Algorithm 2 includes all users that accept $B.dst$ as the public key of user dst in G.

This algorithm is optimal in the sense that it meets the lower bound of the worst case complexities of any algorithms that compute the set of users that accept $B.dst$ as the public key of user dst. The example certificate graph in Fig. 9 shows when any algorithm has to consider all the three certificates to compute the set. In fact, any certificate graph that has only one certificate chain from any user to user dst, where all the certificates have `accepted` ratings, needs exactly e steps to find all the users that can accept the public key of user dst. Therefore, Algorithm 2 meets the lower bound of the worst case complexity.

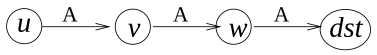

Fig. 9. A certificate graph with the worst case complexity

VII. Certificate Distribution

In the previous section, we showed two algorithms that compute a set of users that can accept the public key of user dst in a given certificate graph. These algorithms can be used by two types of entities: a public key server with a large database of certificates or users with a limited database of certificates.

In PGP, users can find certificate chains from src to dst from search services based on public key servers [4], [5]. A public key server, for example keyserver.net [6], stores a collection of public keys and certificates issued by users. User src can use this collection to find a certificate chain from src to dst by itself, or use the search services such as [4], [5]. However, it is proven that finding all the certificate chains from one user to another is NP-hard [5], so these services provide an approximate set of certificate chains. Still the time complexity of computing a set of chains from one user to another is $O(n^{O(l)})$, where n is the number of users in the certificate graph and l is the length of the longest acceptable chain.

If the algorithms in the previous section are run by a public key server then we can assume that the input certificate graph is the full certificate graph of the system and user dst can simply rely on this service to learn more public keys. When user src needs to learn the public key of user dst, src can ask the key servers to run Algorithm 2, and the server can decide whether a user src can accept the public key of user dst or not on behalf of user src. However, if this kind of services are not available, users need to make a decision on a partial certificate graph.

Making decisions based on partial certificate graphs is rather tricky. Consider the certificate graph in Fig. 8. Assume that user u wants to find the public key of user dst. If user u only had the certificate chain $(u, w, I)(w, x, I)(x, dst, I)$ shown in Fig. 10, then user u would not have accepted the public key of dst. If user u wants to accept the public key of user dst, then user u needs to find more certificates.

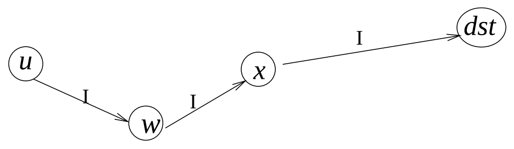

Fig. 10. A certificate chain with `independent` ratings in Fig. 8

One way for user u to find the public key of user dst is to send queries to user v. (Note that user u has the rating of `independent` for user w, so user u cannot accept any key based on a certificate chain through w. Therefore, there is no point in asking user w for more certificates.) However, it is hard to guarantee that user v will be online when u needs it.

Instead, users can store a subset of certificates locally [7], [8]. The scheme in [7] does not require the knowledge of the full certificate graph but may need to flood the network with query/response messages to find the subset of certificates to store. On the other hand, the scheme in [8] requires the knowledge of the full certificate graph but assigns a subset of certificates to each user such that users src and dst are guaranteed to be able to retrieve all certificate chains from src to dst in the union of their local subsets, if there were such chains in the original certificate graph. In both schemes, users src and dst can use the union of their local subsets as the input to Algorithm 1 or Algorithm 2 to decide whether to accept the public key of dst or not.

For example, in the certificate system in Fig. 8, users u and dst may store the following certificates locally.

$$\begin{array}{rl} u \text{ stores} & \{(u, v, A), (u, w, I), (v, y, A(2)), (v, x, A(2))\} \\ dst \text{ stores} & \{(y, dst, A), (x, dst, I)\} \end{array}$$

The union of these two sets is the certificate graph shown in Fig. 11. When Algorithm 2 is applied on this certificate graph, it computes the same set C=$\{dst, x, y, v, u\}$.

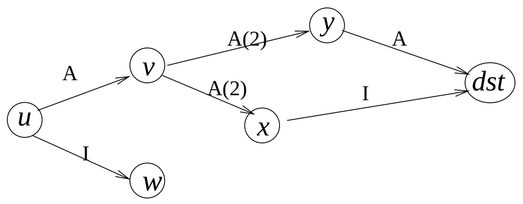

Fig. 11. A certificate graph from local database of certificates

Note that in a certificate system with ratings, not all of these certificate chains are necessarily useful. For example, the certificate chain $(u, w, I)(w, x, I)(x, dst, I)$ in Fig. 10 is not useful for user u to learn the public keys of users x and dst. In fact, any certificate chain that contains a certificate with a rating `independent` before the last certificate in the chain will not be used in key acceptance decision. We present an efficient algorithm, Algorithm 3, to compute a subgraph for each user dst that contains only useful certificate chains. Algorithm 3 is similar to Algorithm 2, and the pseudo code is below. The time complexity of Algorithm 3 is $O(min(e, ln))$, where n is the number of users in G, e is the number of certificates in G, and l is the length of the longest acceptable chain.

Algorithm 3 computes a subgraph $G.dst$ of the certificate graph G for each user dst in G, given a parameter l. l is the length of the longest acceptable chain. If l is small, then users can accept fewer public keys, but users will have less risk of using incorrect certificates. If l is large, then users can accept more public keys, but users may face more risk of using incorrect certificates. $G.dst$ contains all the certificates that are needed by any user u in G to decide whether to accept $B.dst$ as the public key of user dst or not. If a user src is in $G.dst$, then user dst can compute a subgraph $G.dst(src)$ that can be used by user src to verify that user src can accept the public key of user dst. On the other hand, if user src is not in $G.dst$, then src cannot accept the public key of user dst in a given certificate graph G. Given a subgraph $G.dst$, Algorithm 4 below computes a subgraph $G.dst(src)$ optimally.

In the certificate system in Fig. 8, Algorithm 3 computes $G.dst$ as shown in Fig. 12. Note that the certificate (x, y, A) is also in the original certificate graph in Fig. 8, but not included in $G.dst$ here. Since user x knows the public key of user dst and has already issued the certificate $(x, dst,$ `rating`$)$, the certificate (x, y, A) is not necessary for user x to accept the public key of user dst. Also all the certificates with `independent` rating that are not issued for dst are excluded from $G.dst$.

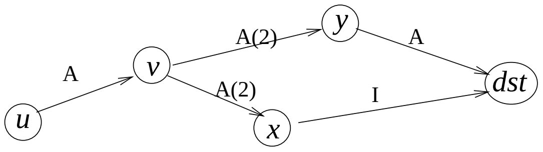

Fig. 12. $G.dst$ of the certificate graph in Fig. 8

The time complexity of Algorithm 3 is $O(min(e, ln))$, where n is the number of users in G, e is the number of certificates in G, and l is the length of the longest acceptable chain. It is easy to see that the time complexity of Algorithm 3 cannot go over $O(e)$, as it considers each certificate in G at most once. Also the while loop in lines 8-21 runs at most $l - 1$ times, and each loop can add at most n nodes to $G.dst$. Therefore Algorithm 3 executes at most ln steps. Therefore the time complexity of Algorithm 3 is $O(min(e, ln))$.

This algorithm can be run by the key server for a user dst, or user dst can flood the network with query/response messages for the same result. (In particular, each addition of a certificate in Algorithm 3 would require a pair of query/response messages.) Note that if the certificate system is dynamic, i.e. a new certificate may be issued and/or a certificate may be revoked, then this subgraph $G.dst$ should be refreshed. The key server

ALGORITHM 3 computes the subgraph $G.dst$ with length limit l

```
INPUT: a certificate graph $G$ and a user $dst$ in $G$,
        and the length limit $l$
OUTPUT: a subgraph $G.dst$ of $G$

STEPS:
1: $G.dst := \{\}$;
2: $Z := \{\}$;
3: for each certificate $(u, dst, \texttt{rating})$ in $G$ do
4:      $G.dst := G.dst \cup \{(u, dst, \texttt{Rating})\}$;
5:      add all the certificates $(x, u, \texttt{rating})$ issued for $u$
            in $G$ to $Z$
6: endfor;
7: $level$ := 2;
8: for each user $u$ in $G$, c[u] := 0; T.u := {};
9: while $Z \neq \{\}$ and $level \leq l$
10:     for each certificate $(u, v, \texttt{rating})$ in $Z$ do
11:          remove $(u, v, \texttt{rating})$ from $Z$;
12:          if $(u, v, \texttt{rating})$=$(u, v, A)$ and $u \notin G.dst$
13:          then $G.dst$ := $G.dst \cup \{(u, v, A)\}$;
14:               add all the certificates $(x, u, \texttt{rating})$
                      issued for $u$ in $G$ to $Z'$
15:          else if $(u, v, \texttt{rating})$=$(u, v, A(k))$ and $u \notin G.dst$
16:               then c[u] := c[u]+$\frac{1}{k}$;
17:                    T.u := T.u$\cup\{(u, v, A(k))\}$
18:                    if c[u]$\geq$1
19:                    then $G.dst$ := $G.dst$$\cup$T.u;
20:                         add all the certificates $(x, u, \texttt{rating})$
                            issued for $u$ in $G$ to $Z'$
21:     endfor;
22:     $Z := Z'$;
23:     $level := level + 1$
24: endwhile;
25: return $G.dst$
```

may recompute this periodically or on request by user dst to keep it up to date. User dst can periodically initiate flooding to refresh $G.dst$ as well.

Once user dst has $G.dst$ locally, user dst can run Algorithm 4 on $G.dst$ to compute a subgraph $G.dst(src)$ of $G.dst$ for a user src. The subgraph $G.dst(src)$ serves as a proof to show that user src can accept the public key of dst. When a user src wants to find a public key of user dst, then src can ask dst for a subgraph $G.dst(src)$. When src receives the subgraph $G.dst(src)$, then src executes the following steps:

1) First, src verifies that $G.dst(src)$ contains user src. If not, src cannot accept any public key based on $G.dst(src)$.
2) Second, src verifies that each certificate (src, u) in $G.dst(src)$ is indeed issued by src. User src can either keep the list of all the certificates it issued and compare it with $G.dst(src)$, or simply verify the signatures of (src, u) with its own public key.
3) Third, src verifies the signature of each certificate in $G.dst(src)$ using the public key of the preceding certificate in $G.dst(src)$. For example, user u in Fig. 12 uses the public key of user v to verify the certificates $(v, y, A(2))$ and $(v, x, A(2))$. If any certificate cannot be verified, remove the certificate from $G.dst(src)$.
4) Fourth, src uses the rating information in the remaining certificates in $G.dst(src)$ to decide whether to accept the public key in $G.dst(src)$ as the public key of user dst or not.

Note that dst may not be able to verify all the certificates in $G.dst(src)$, before sending it to src. Unless dst knows the correct public key of user src, dst has to rely on user src to verify the chains in $G.dst(src)$. This is because of the unidirectional nature of certificate verification. Any user can verify the certificate chain if it knows the correct public key of the issuer of the first certificate in the chain. However, if a user does not know the correct public key of the issuer of the first certificate, the user cannot be guaranteed to have correct public keys in any of the certificates in the certificate chain. User src will detect any certificate that has been modified wrongly and reject the incorrect certificate.

Now we show Algorithm 4 used by user dst to compute $G.dst(src)$. Given a subgraph $G.dst$, Algorithm 4 computes a subgraph $G.dst(src)$ that can be used by user src to find the public key of user dst.

ALGORITHM 4 computes an optimal subgraph $G.dst(src)$

```
INPUT: a certificate graph $G.dst$ and a user $src$ in $G$
OUTPUT: a subgraph $G.dst(src)$

STEPS:
1: if $src \notin G.dst$ return {}
2: $G.dst(src) := \{\}$;
3: for each node $u$ in $G.dst$, done[u] := false;
4: for each certificate $(src, u, \texttt{rating})$ in $G.dst$ do
5:     $G.dst(src) := G.dst(src) \cup \{(src, u, \texttt{rating})\}$;
6: endfor;
7: done[src] := true;
8: while $dst \notin G.dst(src)$
9:     if $u \in G.dst(src)$ and done[u]=false
10:    then add all the certificates $(u, v, \texttt{rating})$ issued
             by $u$ in $G.dst$ to $G.dst(src)$;
11:          done[u] := true;
12: endwhile;
13: return $G.dst(src)$
```

For example, assume that user v wants to find the public key of user dst in Fig. 8. User dst has the subgraph $G.dst$ shown in Fig. 12 computed by Algorithm 3. User dst can use Algorithm 4 to compute the subgraph $G.dst(v)$ shown in Fig. 14. Fig. 13 shows $G.dst(u)$ for user u to accept the public key of dst, which is same as $G.dst$. For users x and y, each of them needs only one certificate to accept the public key of dst. Fig. 15 shows $G.dst(y)$.

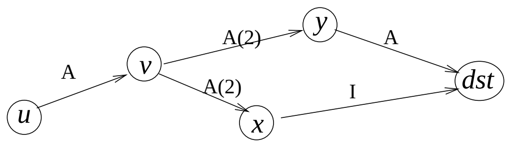

Fig. 13. $G.dst(u)$ computed by Algorithm 4

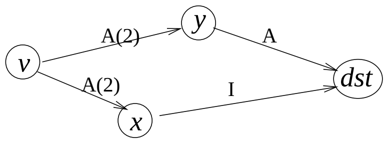

Fig. 14. $G.dst(v)$ computed by Algorithm 4

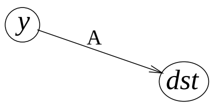

Fig. 15. $G.dst(y)$ computed by Algorithm 4

The time complexity of Algorithm 4 is $O(e)$, where e is the number of certificates in $G.dst$. A certificate in $G.dst$ is added to $G.dst(src)$ at most once, so the complexity is $O(e)$. Also, the resulting $G.dst(src)$ is *optimal* in the sense that all the certificates in $G.dst(src)$ are necessary for user src to accept the public key of dst. In Algorithm 3, for each user u, either one certificate (u, v, A) or k certificates (u, v_i, k_i), where each $k_i \leq k$, are added. Therefore, any certificate issued by u in $G.dst$ is necessary for user u to accept the public key of dst. Algorithm 4 includes the certificates issued by users that are on the chain from src to dst, so all the certificates in $G.dst(src)$ are necessary for user src to accept the public key of dst, i.e. $G.dst(src)$ is optimal.

VIII. Related Work

SSL/TLS [1] certificates do not have any ratings in themselves, but in practice, users accept any certificates issued by trusted certificate authorities such as VeriSign. This is equivalent to having certificates with rating `accepted` from users to trusted certificate authorities.

Lotus Notes [3] offers a hierarchical structure of certificate authorities (CAs) to support a large scale distributed system. Each user name has a "domain" information, and the users with the same domain name have certificates issued by the same CA. The CAs are configured in a tree structure so any user who has the public key of the root CA can verify the certificate chain from the root to any other user and accept the public key in the chain. This is more scalable than having a single CA in the system, but imposes more complexity in the design of CA structure. Every certificate issued by a CA is equivalent to having the rating of `accepted`.

The PGP system is discussed in Section III. Fig. 16 shows the same certificate system with rating, instead of the PGP level of trust. Using the ratings, David can decide not to accept the public key in the certificate $(Bob, Alice, I)$ as the public key of Alice.

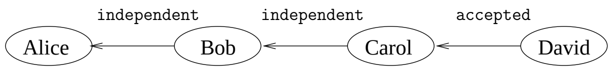

Fig. 16. The example certificate system with rating

SDSI/SPKI [9] supports a boolean delegation flag in their certificates to show that the subject of the certificate can authorize another principal. When this flag is false, no more certificates can be used after this certificate. This is equivalent to having only `accepted` and `independent` ratings in the certificates. Our ratings are more expressive with k-`accepted`.

Two frameworks in [10], [11] are based on certificates with probabilities to express the (un)trustworthiness of subjects. The semantics of these probabilities are defined and algorithms to evaluate certificate chains based on them are presented. However, it is not easy for users to translate their trustworthiness into exact numbers, and the algorithms are complicated. Our ratings offer clear semantics and the algorithms are efficient.

Instead of adding information to certificates, a property of a certificate graph can be used as acceptance criteria. In [5], the min-cut size k of a partial certificate graph from a user u to a user dst is proposed as an acceptance criteria. However, it is proven to be NP-Complete to find such a partial certificate graph from user u to user dst in a certificate graph. Approximation algorithms are provided, but their complexity is still expensive $O(n^{O(l)})$, where n is the number of users in the certificate graph and l is the length of the longest acceptable chain). In [12], the same authors discuss desirable properties of acceptance criteria.

IX. Conclusion

A certificate system provides many useful features such as privacy, integrity, authentication, and authorization. In a certificate system with trusted certificate authorities, a user can learn the public key of another user from a certificate issued by a trusted certificate authority. However, in a certificate system without such trusted certificate authorities, users need to share trust information about other users.

We have discussed the problem of trusting certificates in self-organized certificate systems, such as PGP, and proposed a solution to this problem by adding ratings to certificates. Table IX shows a summary of comparisons among the solutions to the problem of trusting certificates in self-organized certificate systems. The first column shows whether each scheme can share the trust information among users or not. Except for PGP, every scheme supports the trust information by adding it as a field in a certificate. The second column shows the granularity of the trust information of each scheme.

	shared trust info	**partial trust**	**complexity**
PGP	no	per user	not specified
SPKI/SDSI	yes	no	low
probabilities	yes	per certificate	high
ratings	yes	per certificate	low

TABLE I

COMPARISON OF TRUST INFORMATION SCHEMES

The ratings and the two schemes using probabilities support the issuer to specify partial trust for each certificate. However, in PGP a user has one parameter to apply to all the certificates for partial trust. SPKI/SDSI has only one boolean flag to share the trust information, so it does not support partial trust. The third column shows the complexity to use each scheme in evaluating trust information. PGP does not share the trust information, so the complexity is not specified. The complexity of using ratings, and SPKI/SDSI is low since they only require simple arithmetic operations and offer clear semantics. However, the two schemes based on probabilities require more complicated operations and it is hard for an average user to understand the semantics of these operations.

We also showed two algorithms that compute a set of users that can accept a public key. For a given certificate graph G with e edges, the second algorithm has the complexity of $O(e)$, which meets the lower bound of the worst case complexities. These two algorithms need a certificate graph as an input. We discussed how to use the public key servers and local databases of certificates to obtain the input certificate graph for these two algorithms. The resulting certificate graph might contain certificates that are not useful in key acceptance decision. We presented Algorithm 3 that computes a subgraph $G.dst$ for each user dst that contains only the useful certificates for any user to decide whether to accept the public key of dst in G or not. The complexity of Algorithm 3 is $O(min(e, ln))$ where l is the length of the longest acceptable chain and n is the number of users. Based on $G.dst$, Algorithm 4 computes an optimal subgraph $G.dst(src)$ for user src, which contains all the necessary certificates for src to verify and decide whether to accept the public key of dst or not.

REFERENCES

[1] Tim Dierks and Eric Rescorla, “The TLS protocol version 1.1,” Internet Draft (draft-ietf-tls-rfc2246-bis-08.txt), 2004.

[2] P. Zimmerman, *The Official PGP User’s Guide*, MIT Press, 1995.

[3] Søren Peter Nielsen, Frederic Dahm, Marc Lüscher, Hidenobu Yamamoto, Fiona Collins, Brian Denholm, Suresh Kumar, and John Softley, “Lotus notes and domino r5.0 security infrastructure revealed,” 1999.

[4] “Pathfind,” http://the.earth.li/ noodles/pathfind.html.

[5] Michael K Reiter and Stuart G. Stubblebine, “Resilient authentication using path independence,” *IEEE Transactions on Computers*, vol. 47, no. 12, pp. 1351–1362, December 1998.

[6] “Worldwide public key repository,” www.keyserver.net.

[7] Srdjan Capkun, Levente Buttyán, and Jean-Pierre Hubaux, “Self-organized public-key management for mobile ad hoc networks,” *IEEE Transactions on Mobile Computing*, vol. 2, no. 1, pp. 52–64, 2003.

[8] Mohamed G. Gouda and Eunjin Jung, “Certificate dispersal in ad-hoc networks,” in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS ‘04)*. 2004, IEEE.

[9] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, “SPKI certificate theory,” RFC 2693, 1999.

[10] U. Maurer, “Modeling a public-key infrastructure,” in *Proceedings of the European Symposium on Research in Computer Security (ESORICS ’96)*. 1996, Springer-Verlag.

[11] T. Beth, M. Borcherding, and B. Klein, “Valuation of trust in open networks,” in *Proceedings of the European Symposium on Research in Computer Security (ESORICS ’94) LNCS 875*. 1994, pp. 3–18, Springer-Verlag.

[12] Michael K. Reiter and Stuart G. Stubblebine, “Authentication metric analysis and design,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 2, pp. 138–158, 1999.