Building Certification Paths: Forward vs. Reverse

Yassir Elley, Anne Anderson, Steve Hanna, Sean Mullan, Radia Perlman, Seth Proctor
{yassir.elley, anne.anderson, steve.hanna, sean.mullan, radia.perlman, seth.proctor}@sun.com
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803

Abstract

In general, building and validating a certification path connecting a trust anchor to a target can be a very time-consuming process. As such, any optimizations are valuable. Certification paths are commonly built from the target to the trust anchor ("building in the forward direction") or from the trust anchor to the target ("building in the reverse direction"). This paper presents a comparison of these two approaches, analyzes the advantages and disadvantages of each approach, and concludes that building in the reverse direction is often more effective than building in the forward direction.

1. Introduction

Public key cryptography can be used for many purposes, including checking someone's signature or encrypting something only they can read. Several public key algorithms exist, including RSA [11], ElGamal [5], and DSA [9]. A variety of protocols use public key techniques, including IPSec [8], S/MIME [4], and TLS [3].

Public key cryptography allows two parties to securely communicate with each other without any prior contact. However, public key systems depend on securely establishing the other party's public key. For example, if Alice wants to verify the signature on a document she has received from Bob, she needs to know Bob's public key. If Bob's public key successfully verifies the signature, then Alice can be sure that the document must have been signed using Bob's private key, which only Bob knows. If Alice is somehow tricked into believing that an impostor's public key is Bob's public key, then the impostor can sign the document and Alice will verify the signature using the impostor's public key and will believe that the document was signed by Bob. Therefore, Alice needs to be sure that she has the correct public key for Bob.

A public key certificate is a signed statement that is used to establish an association between an identity and a public key. The entity that vouches for this association and signs the certificate is the *issuer* of the certificate and the identity whose public key is being vouched for is the *subject* of the certificate. If Alice has a certificate that vouches for Bob's public key and Alice trusts the issuer of that certificate and knows the issuer's public key, then Alice can trust that the public key in the certificate is truly Bob's public key, assuming that the signature on the certificate verifies successfully.

Typically, Alice uses a chain of certificates, also known as a *certification path*, which starts with a public key she trusts (one of her trust anchors) and ends with Bob's public key. In order to validate the path, Alice uses her trusted public key to verify the first certificate in the path. She extracts the public key and other information contained in the first certificate, uses it to verify the second certificate, and continues this process until she reaches the target certificate vouching for Bob's public key. There are many steps involved in validating the certification path, including verifying the signatures, ensuring that the subject and issuer names chain correctly, verifying that none of the certificates have expired or been revoked, and processing the various other extensions found in certificates. The PKIX Certificate and CRL Profile [6] describes a typical validation procedure.

2. Building Certification Paths

If Alice doesn't already have a valid certification path from one of her trust anchors to Bob, she must build one. Building a certification path (also known as discovering or developing a certification path) is an important topic with little published research. Silva and Stanton discuss various techniques for building certification paths in their paper [12], but do not cover several important issues that are presented in this paper.

Certificates are often stored in directories. The subject and issuer names in X.509 [10] certificates are hierarchical X.500 names, which map directly to directory

entries. A particular certificate can be stored at the issuer's directory entry, at the subject's directory entry, or at both places. Using these directory entries, we can attempt to build a certification path connecting a trust anchor to the target. There are two basic ways of building certification paths, depending on where the certificates are stored in the directory.

The PKIX LDAPv2 Schema [1] defines the schema that is to be used when certificates are stored with LDAP servers in an LDAPv2 environment [2]. An entity that is authorized to issue certificates is referred to as a *certification authority* (CA), while an end-user that is not authorized to issue certificates is referred to as an *end-entity* (EE). A certificate issued to another CA is referred to as a *CA certificate*, while a certificate issued to an end-entity is referred to as an *EE certificate*. This schema requires EE certificates to be stored in the userCertificate attribute of the subject's directory entry ("stored with the subject," for short). CA certificates must be stored with the subject in the "forward" element of the crossCertificatePair attribute and can optionally be stored with the issuer in the "reverse" element of the crossCertificatePair attribute. When a certification path is built from the target to a trust anchor, this is called building in the *forward* direction. When a certification path is built from a trust anchor to the target, this is called building in the *reverse* direction.

If certificates are stored with the subject, it is easy to build in the forward direction (from the target to the trust anchor). We start by retrieving all the certificates in the target's directory entry. Each of these is a possible candidate certificate. We select one of these certificates, and then retrieve all the certificates from the directory entry of this certificate's issuer. We continue with this procedure working our way to the trust anchor. If certificates are only stored with the subject, it is extremely difficult to build in the reverse direction from the trust anchor to the target because we have no idea where in the directory to search for certificates issued by the trust anchor.

Similarly, if certificates are stored with the issuer, it is easy to build in the reverse direction (from the trust anchor to the target). This time, we start by retrieving all the certificates in the trust anchor's directory entry and work our way toward the target. If certificates are only stored with the issuer, it is extremely difficult to build in the forward direction (starting with the target). Storing certificates with the issuer and the subject provides the greatest flexibility.

For certain trust models, building a certification path is fairly straightforward because we can take advantage of constraints inherent to that particular trust model. An example of this is a top-down hierarchical trust model, where the trust anchor is typically a root CA at the top of a hierarchical namespace which issues certificates for each CA at the next level of the namespace and so on until the end entities are reached. Since each entity below the root has exactly one certificate issued for it by its superior, building in the forward direction from the target to the trust anchor is more effective than building in the reverse direction in this trust model. If we start with the target and build in the forward direction, there is only one possible candidate certificate at each step of the process, which greatly simplifies the process of building the certification path. Building in the reverse direction is more problematic with this trust model. The fan-out from the root CA may be substantial with the root CA having issued many certificates to its subordinates. If we start with the trust anchor and build in the reverse direction, there will be many candidate certificates to choose from at each step of the process, thus making it more difficult to build the certification path.

For more general trust models, however, we can not take advantage of these particular constraints. An example is a cross-certified trust model, where every entity can issue certificates for every other entity. In this case, building a certification path can be a very time-consuming process. We have to follow every certificate issued by the trust anchor (if building reverse) or issued for the target (if building forward). There may be many certificates issued by each entity and there may be many certificates issued for each entity. Any technique that can be used to filter out some of those certificates is very valuable.

To this end, it is useful to validate the certification path as it is being built. This allows us to quickly reject paths that fail to validate, and then we can backtrack and try other paths. The PKIX certification path validation procedure [6] is defined in the reverse direction, starting with the trust anchor and proceeding to the target. As a result, it naturally lends itself to building in the reverse direction. Even when building forward, we can validate certain parts of the certificates as we are building. However, certain steps of the validation algorithm are much less effective when building forward.

3. Name Constraints

One of the certificate extensions that is particularly useful in filtering out certificates as we are building is the name constraints extension. "The name constraints extension … indicates a name space within which all subject names in subsequent certificates in a certification path shall be located." [6] For the purposes of validation, the certification path is processed in the reverse direction, starting with the trust anchor and proceeding to the target.

Suppose Alice works at Company A and is trying to build a certification path to Bob, who works in the eastern division of Company B. Figure 1 illustrates the hierarchy of certificates that Alice is dealing with. We have used DNS names instead of X.500 names to simplify the

diagram and have used NC as an abbreviation for the name constraints extension.

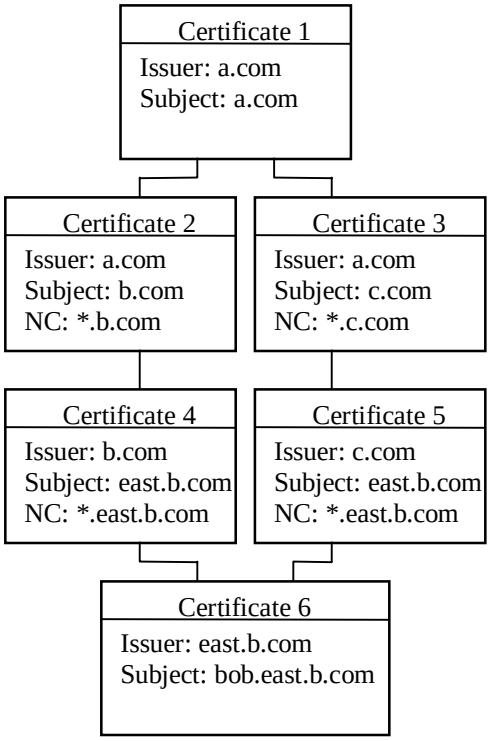

Figure 1: Name Constraints Scenario

When building reverse, Alice starts with her trust anchor (Certificate 1). She then has to choose between two candidate certificates for the next leg of the path (Certificates 2 and 3). She can examine the name constraints in each candidate certificate and reject any candidate whose name constraints do not allow the target subject to be reached. In this case, Alice rejects Certificate 3 because the name constraints in Certificate 3 do not let her reach the target subject (i.e. bob.east.b.com). After accepting Certificate 2, she examines Certificate 4 as a candidate for the next leg of the path. At this point, there are two checks she can make to determine whether to accept Certificate 4. First, she can verify whether Certificate 4's subject is consistent with the name constraints in Certificate 2. Additionally, she can make sure that the name constraints in Certificate 4 would allow her to reach the target subject. In this case, both checks succeed and Alice can accept Certificate 4 and continue building the certification path to Bob.

When building forward, Alice starts with a target certificate (Certificate 6) and then has to choose between two candidate certificates for the next leg of the path (Certificates 4 and 5). If the name constraints in either candidate do not allow the target subject to be reached, she will be able to reject that candidate immediately. In this case, neither candidate will be rejected. Alice may decide to accept Certificate 5, only to find out in the next step, when she is examining Certificate 3, that the name constraints in Certificate 3 do not allow the target subject to be reached. More generally, when building forward, Alice may follow some other certificate which will never even connect to the trust anchor. In this way, she may keep accepting certificates aimlessly without getting closer to the trust anchor.

In general, building reverse is more effective than building forward with regard to processing name constraints. When building reverse, we are always "homing in" on the target subject and are rejecting any candidate certificates that could not possibly lead us to the target. When building forward, we are not able to "home in" on the trust anchor, because name constraints only apply to SUBSEQUENT certificates in the path, not to preceding certificates in the path.

4. Policy Processing

Another set of certificate extensions that allows us to filter out certificates as we are building is the set of policy extensions. The algorithm used to process these policies is defined by X.509 [10]. A revised policy processing algorithm is currently being developed by the PKIX working group [7]. Since this revised algorithm is still a work in progress, we present our analysis based on the X.509 standard. Although the details of the algorithms are different, our conclusions remain the same even with the revised algorithm.

For an EE certificate, the certificate policies extension is used by the issuer to specify what policies were followed when issuing the certificate and for which purposes the certificate may be used. For a CA certificate, the certificate policies extension is used by the issuer to place limits on which policies are considered acceptable in a certification path which includes this certificate. One example of a set of certificate policies is a group of trust ratings which indicate the level of care the issuer has taken in authenticating the subject before issuing a certificate for that subject. A HIGH trust rating indicates that a great deal of care has been taken to properly authenticate the subject. Perhaps the issuer met the subject in person and verified his identity using a driver's license. MEDIUM or LOW trust ratings correspond to lower levels of care. Different domains may have different names for equivalent policies, and the policy mappings extension can be used to indicate that a specified policy in the issuer's domain is equivalent to another specified policy in the subject's domain. Thus, a HIGH trust rating in Company A's domain may map to a CONFIDENTIAL trust rating in Company B's domain. Users with particular policy requirements can specify a

set of initial policies that they find acceptable (or the special value “any-policy”). They can require that every certificate must contain an acceptable policy. They can also prohibit the use of policy mappings in every certificate. Certificates themselves may also have a policy constraints extension that requires explicit policies or inhibits policy mappings after a specified number of subsequent certificates.

When processing policies, two state variables are used: user-constrained-policy-set and authority-constrained-policy-set. The user-constrained-policy-set consists of the policies that the user currently considers to be acceptable, while the authority-constrained-policy-set consists of the policies that the CAs in the certification path currently consider to be acceptable. The user-constrained-policy-set is initialized with the user's initial policies and the authority-constrained-policy-set is initialized with “any-policy”. There are three reasons that policy processing could reject a certificate during validation (failure modes):

1. If explicit policies are required (whether specified by the user or by a preceding certificate), we must make sure that at least one member of the user-constrained-policy-set appears in the certificate's certificate policies.
2. We must replace the authority-constrained-policy-set with the intersection of itself and the certificate's certificate policies and we must make sure that the result is not null.
3. After performing all intersection steps, we must make sure that the intersection of the authority-constrained-policy-set and the user-constrained-policy-set is not null.

If none of these failure modes is encountered, the current certificate’s policy mappings are then processed. If policy mappings are inhibited (whether specified by the user or by a preceding certificate), the policy mappings are not processed at all. However, if policy mappings are allowed, the user-constrained-policy-set and authority-constrained-policy-set are augmented as follows. For each policy mapping in the current certificate, if the user-constrained-policy-set contains the issuer domain policy for that policy mapping, then the corresponding subject domain policy is added to the user-constrained-policy-set. Similarly, if the authority-constrained-policy-set contains the issuer domain policy for that policy mapping, then the corresponding subject domain policy is added to the authority-constrained-policy-set.

Using our example, let us assume that Alice requires a certificate policy corresponding to a HIGH trust rating in every certificate. Figure 2 illustrates the hierarchy of certificates we are dealing with. We have used CP as an abbreviation for the certificate policies extension and PM as an abbreviation for the policy mappings extension.

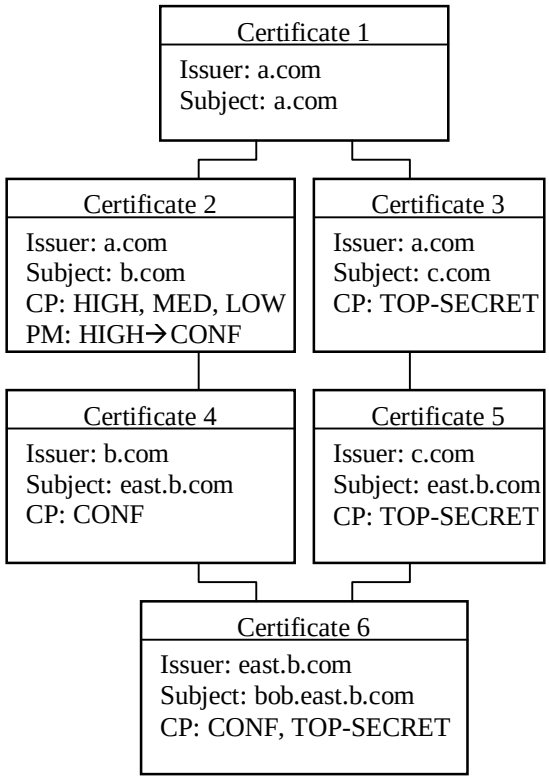

Figure 2: Policy Processing Scenario

When building in the reverse direction, we can quickly reject certificates that do not contain acceptable policies. In this example, the user-constrained-policy-set is initialized with {HIGH} and the authority-constrained-policy-set is initialized with “any”. We start with the trust anchor (Certificate 1) and must decide between two candidate certificates for the next leg of the path (Certificates 2 and 3). We can immediately reject Certificate 3, because TOP-SECRET has no intersection with the user-constrained-policy-set. We accept Certificate 2 because its certificate policies intersect with both of the state variables (i.e. with the user-constrained-policy-set and the authority-constrained-policy-set). After processing Certificate 2, the user-constrained-policy-set becomes {HIGH,CONF} and the authority-constrained-policy-set becomes {HIGH,MED,LOW,CONF}. Thus, we are able to accept Certificates 4 and 6 because CONF intersects with both of the state variables.

When building in the forward direction, it is much more difficult to filter out certificates based on their policy extensions. The main culprit here is policy mappings. If we examine the failure modes dealing with the user-constrained-policy-set (failure modes 1 and 3), we observe that the user-constrained-policy-set can keep changing and increasing because of policy mappings. As

a result, when building forward, we must first build the entire certification path before we can check whether either of these failure modes has been encountered. To illustrate this, let us take an example where initial-policies = HIGH. We start by examining a target certificate (Certificate 6), which has been issued with certificate policies of CONF and TOP-SECRET. At this point, we may think that Certificate 6 can be rejected because neither of these policies intersect with the initial policy of HIGH. However, this is only true if policy mappings are inhibited from the very first certificate. In the general case where policy mappings are not inhibited, a previous certificate closer to the trust anchor may have a policy mapping of the form HIGH→CONF or HIGH→TOP-SECRET, which would augment the user-constrained-policy-set to be {HIGH,CONF} or {HIGH,TOP-SECRET}, in which case the candidate certificate is clearly valid. For this reason, we have to accept Certificate 6. Similarly, when choosing between Certificates 4 and 5 for the next leg of the path, we will not be able to reject either of them. In fact, when policy mappings are allowed, we can not filter out any certificates based on these failure modes. This is a major drawback when building in the forward direction.

With regard to failure mode 2, which deals with the authority-constrained-policy-set, we can filter out some certificates when building forward, but still not as effectively as when building reverse. We observe that the authority-constrained-policy-set is typically reduced in the reverse direction and is only augmented by policy mappings. We also observe that because of this intersecting nature, the policy mappings of one certificate only apply to the immediately following certificate ("following" in the reverse sense). When building forward, we will be maintaining a forward-policy-set state variable, which is initialized with the certificate policies in the target certificate. Let us take an example. Assume we have already found a target certificate that has been issued with certificate policies of CONF and TOP-SECRET. Therefore, our forward-policy-set is currently {CONF, TOP-SECRET}. We are now trying to choose between several candidates for the next certificate to prepend to this path. The situation is illustrated in Figure 3.

When deciding whether a candidate certificate should be accepted or rejected, we need to run the policy processing algorithm in the reverse direction on the new path to see if it is feasible. If we accepted Certificate 3, then after processing Certificate 3 in the reverse direction, the authority-constrained-policy-set would be {HIGH,MED,CONF}, which would clearly allow us to accept the target certificate (Certificate 5). Therefore, we would accept Certificate 3 when building forward. If we accepted Certificate 4, then after processing Certificate 4 in the reverse direction, the authority-constrained-policy-set would be {HIGH,CLASSIFIED}, which would not allow us to reach the target certificate. Therefore, we would reject Certificate 4. Similarly, we would accept Certificate 1 and reject Certificate 2. In general, we will only accept candidates under either of two conditions:

- if the candidate has a certificate policy which intersects with the forward-policy-set
- if policy mappings are not inhibited AND the candidate's subject domain policy intersects with the forward-policy-set AND the candidate has a certificate policy which intersects with the corresponding issuer domain policy

We refer to the specific certificate policies of the candidate that meet either of these conditions as the candidate's useful certificate policies. After accepting a candidate certificate, the forward-policy-set is replaced with the candidate's useful certificate policies and these are used to determine which certificates to accept next. Using our example, after accepting Certificate 3, our new forward-policy-set would simply be {HIGH}. We would not include MED in the forward-policy-set because that

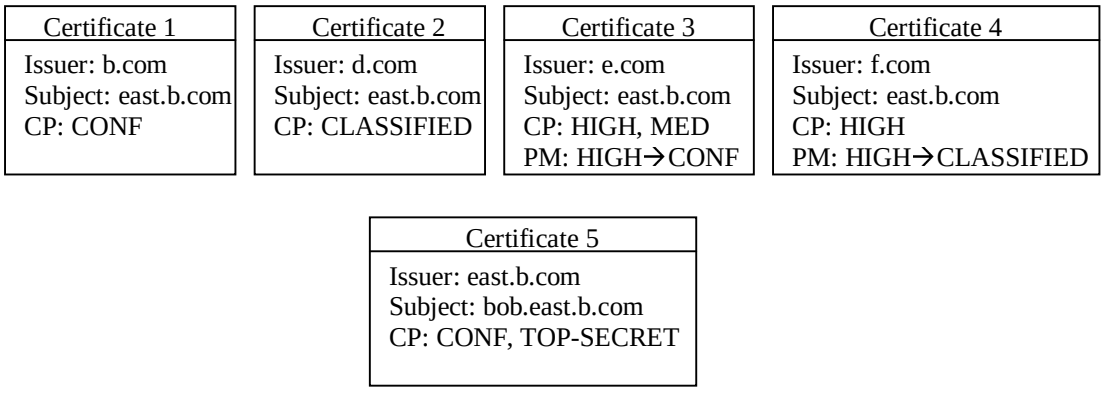

Figure 3: Policy Processing Scenario (Forward Direction)

certificate policy did not meet either of the two conditions and was therefore not useful in allowing us to reach the target certificate.

5. Loop Detection

When building a certification path, it is also useful to be able to detect loops as the path is being built. This allows us to quickly reject paths that are just looping around uselessly. At first glance, it may seem that we can detect a loop by keeping track of the distinguished names of the directory entries we have visited. If we encountered the same distinguished name (DN) twice, we would assert that we had encountered a loop and so we would not continue building down that path. Unfortunately, simply comparing DNs is not adequate. This algorithm breaks down if there is more than one public key associated with a DN. It could be that a CA with a given DN has two public keys because it is phasing one of them out. It could also be that there are actually two different CAs with different public keys that just happen to have the same DN. In this case, the path A→B→C→B′→D, where B and B′ are CAs with the same DN but different keys (or the same CA with two different keys), can be a valid path. However, using the algorithm where loops are detected if we encounter the same DN twice, we would have incorrectly rejected this path after encountering B′.

To deal with this situation, we can use the algorithm that a loop is detected if we encounter the same certificate twice. This works well in the absence of policy mappings. However, if policy mappings are present, then it is possible that the policy mappings in a loop of certificates may actually allow certificates to be acceptable after traversing the loop which were not acceptable before traversing the loop. For example, when building reverse, we may be faced with the situation illustrated in Figure 4.

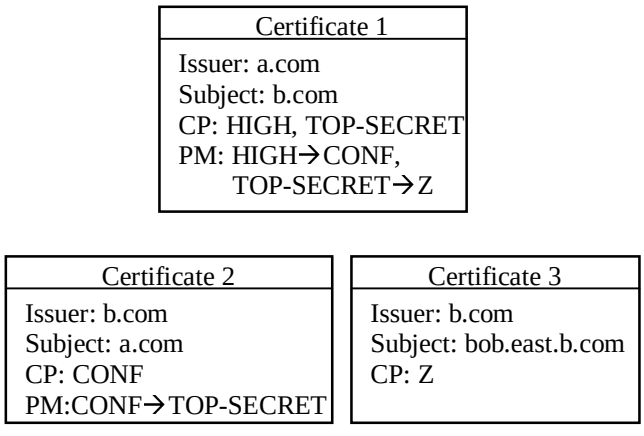

Figure 4: Loop Detection Scenario

Let us assume Alice requires that a HIGH trust rating appear in every certificate. Therefore, the user-constrained-policy-set is initialized to be {HIGH}. After processing Certificate 1, the user-constrained-policy-set becomes {HIGH, CONF} because this set is only augmented when a policy that is already in the set is mapped to another policy. At this point, Certificate 3 is not acceptable for the next leg of the path because its certificate policy of Z does not intersect with the user-constrained-policy-set. Since Certificate 2 is acceptable, we process Certificate 2, and the user-constrained-policy-set becomes {HIGH, CONF, TOP-SECRET}. Now, since the subject of Certificate 2 is a.com, we go back to Certificate 1 for the second time. If we had been using the algorithm where loops are detected when the same certificate is encountered for the second time, then we would have rejected Certificate 1 at this point. However, this would have yielded an incorrect result. After accepting and processing Certificate 1 for the second time, the user-constrained-policy-set becomes {HIGH, CONF, TOP-SECRET, Z}. Now that Z has been added to this set, Certificate 3 has become acceptable for the next leg of the path. Clearly, in this case, Certificate 1 needed to appear twice in the path to allow us to reach Certificate 3. Similarly, it is also possible for the authority-constrained-policy-set to be augmented by policy mappings in such a way that certificates that were not acceptable before traversing the loop become acceptable after traversing the loop. Therefore, we can only assert that a useless loop has been detected if we encounter the same certificate twice AND the two state variables (user-constrained-policy-set and authority-constrained-policy-set) after processing the certificate for the second time do not have any different policies than they had after processing the certificate for the first time. In our example, this was not the case and we did not have a useless loop. As a result, Certificate 3, which was not acceptable after Certificate 1 was processed for the first time, became acceptable after Certificate 1 was processed for the second time. Although this may seem an odd case, we must deal with it properly for the sake of correctness.

Although our algorithm for detecting loops works well when building reverse, it does not work well when building forward. The difficulty when building forward is that we don't have an accurate representation of the user-constrained-policy-set and the authority-constrained-policy-set until we reach the trust anchor because a certificate near the trust anchor may have policy mappings that we are unaware of when dealing with certificates near the target. Since policy mappings are at the root of the problem, we can come up with an algorithm when policy mappings are not present. We observe that the user-constrained-policy-set and authority-constrained-policy-set are only augmented by policy mappings. Therefore, when building forward, if policy

mappings are inhibited or if there are no policy mapping extensions between occurrences of the same certificate, then we can detect a loop if we encounter the same certificate twice. This algorithm when building forward is not as effective as the algorithm we used when building reverse. Although it guarantees that we will not reject any useful loops, we are not able to reject as many useless loops as we could when building reverse.

6. Signature Processing

Processing signatures is another area in which building forward is less effective than building reverse. Both certificates and CRLs need to have their signatures verified to ensure that they have been signed by the supposed issuers. When building reverse, we start with the public key of the trust anchor, which is explicitly trusted. Therefore, when we are looking for the next candidate certificate, we can very quickly reject any candidate whose signature does not verify with our trusted public key. Similarly, we can quickly reject any candidate certificate that has been revoked, because we can fetch the CRL issued by the candidate's issuer, verify the CRL's signature using our trusted public key, and check to see whether the candidate's serial number is on that CRL.

When building forward, however, we can not immediately reject a candidate if it has an invalid signature, nor can we immediately reject a candidate that has been revoked. Both of these checks have to be delayed for one cycle. The reason for this delay is that we don't yet have the public key of the candidate's issuer, which is needed to verify the candidate's signature and also needed to verify the candidate's revocation status (i.e. needed to verify the signature on the CRL issued by the candidate's issuer). Figure 5 illustrates the situation when we are building forward.

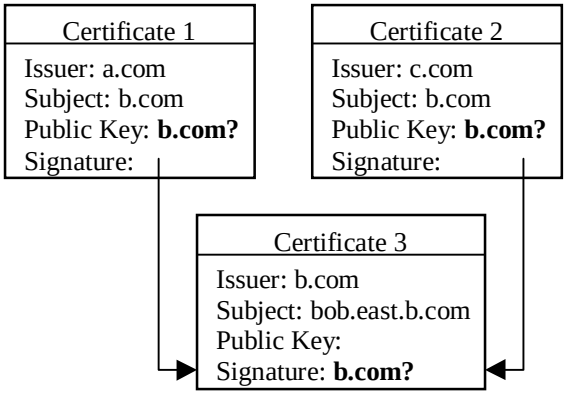

Figure 5: Signature Processing Scenario

When building forward, we begin by finding a target certificate whose subject is bob.east.b.com. There may be many such certificates to choose from (Certificate 3, Certificate 3′, etc). Suppose that we choose to first examine Certificate 3 as a candidate. At this point, we can not check this candidate's signature nor its revocation status, because we don't yet have b.com's public key. So we will have to delay this check and tentatively accept this certificate. This is inefficient because it is quite possible that this certificate has been revoked or that it has an invalid signature.

After tentatively accepting Certificate 3, we have to choose between Certificate 1 and Certificate 2 for the next leg of the path. When examining Certificate 1, we now have a candidate for b.com's public key and we can check whether this public key is able to verify the signature on Certificate 3. Now, if the signature verification fails, we are really not quite sure whether the public key in Certificate 1 is bad, or the signature on Certificate 3 is bad. Neither of those pieces of information are known to be "trusted" a priori. Since we don't want to reject any certificate that could possibly be valid, we are going to have to assume that the public key in Certificate 1 is bad, remove Certificate 1 from consideration (for now), and try all of the other certificates pointing to b.com (in this case, Certificate 2). This is inefficient in the case where the real culprit was Certificate 3 itself (i.e. the signature on Certificate 3 was bad). We won't find that out until we cycle through all the certificates pointing to b.com and notice that none of them are able to verify the signature on Certificate 3. Only then will we backtrack, remove Certificate 3 from consideration, and try another target certificate (perhaps Certificate 3′).

On the other hand, when examining Certificate 1, if the candidate for b.com's public key successfully verifies the signature on Certificate 3, then we can proceed to verify the revocation status of Certificate 3. We would fetch the CRL issued by b.com and verify the CRL's signature using our candidate for b.com's public key. If the CRL signature verification fails, we would try the public key against other CRLs stored at b.com's directory entry (if any). If we were unable to find a CRL whose signature verified, then we would conclude that we are unable to determine the revocation status of Certificate 3. Note that we would not have to examine Certificate 2 in this case. Since the signature on Certificate 3 verified using the public key in Certificate 1, then Certificate 2 would only be useful if it had the same public key as Certificate 1. If that were the case, then Certificate 2's public key would also be unable to verify the signatures on b.com's CRLs.

Of course, if the signature on b.com's CRL verified using the public key in Certificate 1 and Certificate 3's serial number was on the list of revoked certificates, then we would conclude that Certificate 3 has been revoked. Again, this is inefficient because we are only now finding

out that Certificate 3 has been revoked. At this point, we would remove Certificate 3 from consideration and try another target certificate (perhaps Certificate 3′).

Even if Certificate 3's signature and revocation status successfully verify using the public key in Certificate 1, we are still not sure if this is really b.com's public key. Since the only public key we really trust is the trust anchor's public key and since we won't be using that trusted key until the entire path is built, it is possible that some malicious user could launch a denial of service attack by putting many bogus certificates and CRLs in the directory to slow us down. We would continue accepting these bogus certificates until we reached our trust anchor, when we would notice that our trusted public key is unable to successfully verify the signature on one of these bogus certificates. When building reverse, we would not have this particular problem. Even if there were bogus certificates in the directory, we would never accept any of them since their signatures would not verify with our trusted public key.

7. Requesting Revocation

Storing certificates only in the directory entry of the subject essentially disables building in the reverse direction, which is a major disadvantage. On the other hand, one advantage of storing certificates with the subject has to do with requesting revocation. If Bob believes that his private key has been compromised, he needs to inform everyone that has signed a certificate for the corresponding public key, so that they can revoke their certificate. If certificates are stored with the subject, then all certificates issued for Bob's public key would be stored in Bob's directory entry. Therefore, Bob would know everyone who has signed certificates for his public key and he would be able to inform them that they should revoke their certificates. If certificates are stored in the directory entry of the issuer, then Bob would not be able to use the directory to figure out who has signed certificates for his public key and would have to use some other mechanism to do this.

8. Conclusion

In this paper, we compared building certification paths in the forward direction (from target to trust anchor) with building certification paths in the reverse direction (from trust anchor to target). For certain trust models, such as a hierarchical trust model, building in the forward direction is more effective because we can take advantage of the fact that every entity has only one certificate issued for it by its superior. For more general trust models, however, we conclude that building in the reverse direction is more effective because it allows us to perform superior validation of the certification path as we are building it, thereby allowing us to more quickly reject certificates that are not useful in constructing a valid certification path. Building in the reverse direction allows us to more effectively process name constraints, policies, signatures, and CRL-based revocation. It also allows us to more effectively detect useless loops of certificates. In order to allow certification paths to be built in either direction, we recommend that certificates stored in a directory be required to be stored with both the issuer and the subject.

9. Bibliography

1. S. Boeyen, T. Howes, P. Richard, "Internet X.509 Public Key Infrastructure LDAPv2 Schema", RFC 2587, June 1999.
2. S. Boeyen, T. Howes, P. Richard, "Internet X.509 Public Key Infrastructure Operational Protocols – LDAPv2", RFC 2559, April 1999.
3. T. Dierks, C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.
4. S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, L. Repka, "S/MIME Version 2 Message Specification", RFC 2311, March 1998.
5. T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory*, v. IT-31, n. 4, 1985, pp. 469-472.
6. R. Housley, W. Ford, W. Polk, D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", RFC 2459, January 1999.
7. R. Housley, W. Ford, W. Polk, D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", Internet Draft (work in progress), <draft-ietf-pkix-new-part1-02.txt>, July 2000.
8. S. Kent, R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.
9. National Institute of Standards and Technology, NIST FIPS PUB 186, "Digital Signature Standard," U.S. Department of Commerce, May 1994.
10. Recommendation X.509. The Directory: Authentication Framework. *Information technology – Open Systems Interconnection*, June 1997.
11. R.L. Rivest, A. Shamir, L.M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, v. 21, n. 2, February 1978, pp. 120-126.
12. A.R. Silva, M.A. Stanton, "Pequi: A PKIX Implementation for Secure Communication," *Proceedings of the 1999 International Networking Conference (INET '99)*, 1999.