Implementing EFECT

by

Ivan Nestlerode

B.S., Computer Science and Engineering
Massachusetts Institute of Technology, 2000

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2001

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by . . . .
Hari Balakrishnan
Assistant Professor of Computer Science, MIT
Thesis Supervisor

Certified by . . . .
Dan Heer
Technical Manager, Lucent Technologies
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Implementing EFECT

by

Ivan Nestlerode

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the design, implementation, and benchmarking of a software prototype of EFECT [EFECT], a new certificate scheme that handles revocation more gracefully than do current schemes. This prototype includes a client browser, a certificate verification tree library, and a directory server.

The thesis includes analysis, both mathematical and empirical, to determine the optimal values of EFECT's parameters in terms of both speed and space. Finally, the thesis includes a benchmark comparison of the optimized EFECT and a comparable X.509 [X509] system. This comparison serves as proof that EFECT does indeed outperform the X.509 scheme in some common scenarios.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor of Computer Science, MIT

Thesis Supervisor: Dan Heer
Title: Technical Manager, Lucent Technologies

Acknowledgments

This work is dedicated to my family. Their support in these last five years (and the many years before) has been invaluable.

I would like to thank Hari Balakrishnan (my MIT thesis advisor) and Dan Heer (my Lucent technical manager) for their guidance. I would also like to thank my co-worker Irene Gassko for coming up with EFECT, and for discussing its details with me many times over the course of this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The lack of a viable public key infrastructure has been a major barrier to the widespread deployment of encryption and authentication technology. Algorithms have already been developed that provide the strong encryption and authentication desired by those who wish to purchase goods or communicate securely over the Internet. The problem lies in distributing the keys for these algorithms. The current X.509 public key infrastructure is cumbersome and fails to adequately address the problem of certificate revocation.

Recent events highlight this revocation problem and the inability of the current infrastructure to cope with it. In October 2000, it was announced that the private key of Sun Microsystems[1] had probably been compromised. In late January 2001, imposters posing as Microsoft Corporation managed to get VeriSign to sign a certificate proclaiming that they were Microsoft. In these cases attackers had the ability to pose as Sun and Microsoft respectively, gaining all of the software access privileges accorded to these two large corporations. Technically, these two certificates were revoked, but since current Web browsers do not perform any sort of checks for revocation,[2] these revocations were effectively meaningless.

[1]Corporate names mentioned herein are trademarks and/or registered trademarks of their respective companies (Sun Microsystems, Microsoft Corporation, VeriSign, and Netscape).

[2]In theory, a revocation mechanism exists in the X.509 scheme. In practice, the mechanism is awkward enough that browser manufacturers avoided implementing it after a few unsuccessful tries at getting it to work.

Such high-profile debacles threaten to undermine the credibility of cryptographic technology in general. Certificate revocation must be addressed if cryptography is to gain and maintain acceptance.

A new certificate scheme called EFECT has been developed that solves the revocation problem more elegantly than do current certificate schemes like X.509. By providing a more elegant revocation solution, EFECT will hopefully make manufacturers of Web browsers and other security-related products more likely to deal with revocation properly.

This thesis describes the design and implementation of a software prototype of the EFECT scheme. It also contributes an analysis of how a parameter of the EFECT scheme affects the scheme's performance. Finally, the thesis contributes an experiment comparing the relative performances of X.509 and EFECT. The experimental results suggest a scenario in which EFECT is a much more appropriate certificate scheme than X.509.

The structure of the thesis is as follows. The next chapter provides the reader with enough background in cryptography and key infrastructure to understand the significance of and motivation for the EFECT scheme. Chapter 3 describes the details of both the X.509 and EFECT schemes, highlighting the differences between the two and the advantages gained by using EFECT. Chapter 4 lists the requirements for the EFECT prototype and describes the design decisions made to meet those requirements. Chapter 5 provides specific implementation details about each part of the prototype. Chapter 6 describes results from two experiments: one involving the optimization of an EFECT parameter, and one comparing the relative performances of EFECT and X.509 in an off-line, bulk verification scenario. Chapter 7 lists ideas for future improvement of the EFECT prototype. Chapter 8 concludes the thesis.

Chapter 2

Background

This chapter gives the reader enough background in cryptography and key infrastructure to understand the significance of the EFECT scheme. It describes the major cryptographic paradigms in the order they occurred historically, explaining the problems at each stage and how these problems were solved by the next stage. It begins with the problem of insecure communication and the solution of secret key cryptography. It ends with the problem of certificate revocation, which leads us into our discussion of EFECT in chapter 3.

2.1 The Need for Cryptography

The need for cryptography is a very old one. Militaries have always wanted secure communication channels for delivering messages. With the current popularity of communicating over computer networks, this need for secure channels has spread to the population at large.

Communications over computer networks are inherently insecure: they can be both eavesdropped and manipulated without either communicating party's knowledge. In order to secure these communications, we turn to cryptography for encryption and authentication.

Encryption gives us privacy. It obscures the messages being sent over the network, so that only the authorized parties can read them.

Authentication gives us the ability to trust messages. When messages are authenticated, the parties can be assured that the messages are correct (have not been tampered with) and that they are from the proper party (and not an imposter).

A cryptographic key infrastructure gives us both encryption and authentication capabilities.

2.2 Secret Key Infrastructure

In the early days of cryptography, encryption and authentication were achieved using secret key cryptography. Secret key cryptography requires that the two parties doing the encryption and authentication share a common secret (the secret key).

This is a cumbersome requirement. Probably the worst part of the requirement is that in order to have secure communications, the two parties must have already agreed on a shared secret. If they have a way to agree on a shared secret, then it would seem they already have a secure communication channel. It may be possible for two parties to share a key in person, and then go on to communicate over a network, but this approach does not scale well to something like the Internet where people frequently do not have the option of meeting in person before communicating.

2.3 Public Key Infrastructure

Since the publication of [DH], focus has shifted to public key cryptography. Public key cryptography's key infrastructure requirements are more reasonable than those for secret key cryptography. Each party has a pair of keys: one public and one private. Private keys are not revealed. Public keys are published for everyone in the group to see. All parties know the algorithms D, E, S, and V for decryption, encryption, signing, and verification respectively. These algorithms are all keyed.

These requirements are more easily met than those for secret key cryptography. Two parties need not share a secret before communicating. Each party just needs knowledge of the other's public key.

2.3.1 Encryption

Public key infrastructure allows anyone with a given public key K, to send encrypted messages to the party associated with K. As previously mentioned, the two parties need not share a secret before the communication.

To encrypt a message m to Bob, Alice (or anyone else) would use Bob's public key B as follows to produce a ciphertext c: $E_B(m) = c$. To decrypt this ciphertext c, Bob would use his private key b as follows to obtain the message m: $D_b(c) = m$.

2.3.2 Digital Signatures

Public key infrastructure also allows parties to create *digital signatures*, cryptographic analogs of traditional paper and ink signatures. Digital signatures are different on each document that is signed, unlike those of the paper and ink variety. A signature by the party associated with public key K can be verified by anyone who knows K (anyone involved in the key infrastructure). This verification means two things: that the signed message has not been tampered with since the signature, and that the message was signed by the key holder and not someone else.

To sign a message m, Bob would use his private key b as follows to produce signature σ: $S_b(m) = \sigma$. To verify this signature σ, Alice (or anyone else) would use Bob's public key B as follows: $V_B(m, \sigma) = \mathit{true}$ if and only if $\sigma = S_b(m)$.

2.3.3 The Problem of Public Key Spoofing

At first glance, public key infrastructure seems to have solved all of the problems of the secret key infrastructure. Upon closer inspection though, we see that it assumes that everyone knows which public key belongs to which party.

Let us see what would happen if this were not the case. A malicious party who wished to read messages sent to Bob could advertise his own public key M as Bob's public key (which would really be B, not M). If Alice were to come across this key M when trying to send messages to Bob, she might encrypt using M. In this case, the malicious party could read these messages since he would have the private key corresponding to the public key used in the encryption.

This "spoofing" of public keys is a significant problem with public key infrastructure. In order for the system to work, everyone involved must know without a doubt which key belongs to which party.

2.4 Certificates and Certification Authorities

The usual solution to the public key spoofing problem is to have a trusted authority who issues digitally signed statements that bind identities to public keys. These signed statements are called *certificates*. The trusted authority issuing the certificates is called a *certification authority* or a *CA* for short.

It is assumed that all users obtain the CA's public key in a secure manner. This can be done in person, but in practice it is often done by hard-coding the key into software that all parties run. For example, a commercial CA called VeriSign distributes their public key with Netscape's Web browser.

Once users have this one key, they can securely find anyone else's public key either in an on-line certificate directory or from the CA directly. This solves both the problem of shared secrets and the problem of public key spoofing. Now to communicate securely with Bob, Alice would simply get Bob's certificate (from a directory or the CA) before using it to encrypt the messages she wanted to send. By verifying the CA's signature on the certificate, Alice would know positively that the key on the certificate was indeed Bob's public key. Alice could also sign her messages to Bob with her private key. Bob would decrypt the messages with his private key. He could also verify Alice's signatures by obtaining Alice's certificate, verifying that certificate with the CA's public key, and then verifying Alice's signatures with the public key in Alice's certificate.

2.5 The Problem of Certificate Revocation

With a CA in place signing certificates, it is easy to find the proper key needed to communicate with someone. The problem is that private keys can be stolen or compromised. It is worth noting that certificates usually have a natural expiration date written on them. Certificates are only valid if the signature is good and they have not expired. This is meant to coincide with the lifetime of a cryptographic key, but key compromise is not a predictable phenomenon.

The problem arises when a key is compromised before its certificate expires naturally. If Alice's private key a is stolen, she should stop using the key pair (a, A), and create a new one: (a', A'). Unfortunately, there will still be a signed statement from the CA claiming that Alice's public key is A. Digital signatures cannot be undone, so what is the CA to do at this point?

The usual solution is to have the CA issue a statement saying that the key A has been revoked. The CA must also issue a new certificate for A'. The problem then becomes how to distribute this revocation notice to everyone in the system. Without proper distribution of this revocation notice, people using the system would continue using A to encrypt messages to Alice, resulting in insecure communication. This is known as the certificate revocation problem.

The current solutions to this problem are cumbersome. The solution used by today's most common system involves the CA issuing very large lists of revoked certificates that everyone must check before using a certificate for communication. Downloading and scanning through these lists can take significant time and bandwidth, especially as the lists grow large. This solution is far from satisfactory, and the EFECT solution is better in many cases.

Chapter 3 will discuss in more detail how current systems deal with the problem of revocation, and will then explain EFECT's more elegant solution to the problem.

Chapter 3

EFECT

For more information on any of the schemes mentioned in this chapter, the reader is referred either to [EFECT], which describes each scheme in some detail, or to the original papers listed in the bibliography. When we speak of determining the *freshness* of a certificate in this chapter, we are referring to determining that the certificate has been issued and has not been revoked.

3.1 X.509 Certificates and CRL's

The "Internet X.509 Public Key Infrastructure"[X509] is currently the most commonly deployed certificate scheme. The scheme uses X.509 certificates and certificate revocation lists (CRL's). An X.509 certificate contains a serial number, the names of the CA and the user, the public key of the user, dates of issuance and expiration, and a signature from the CA over all of these fields. A CRL is a CA-signed list of serial numbers of revoked certificates.

In the X.509 certificate scheme, determining freshness of a certificate involves downloading the latest CRL from a repository, verifying the CA signature on that CRL, and searching the CRL for the serial number of the certificate in question. It is worth noting that one must download a new CRL as often as certificates can be revoked: weekly revocation means downloading one CRL per week, daily revocation means one CRL per day, etc. Given Micali's estimate of a 10% certificate revocation rate, these CRL downloads can take a significant amount of time and bandwidth given a CA that revokes things quickly and serves millions of customers.

3.2 Revocation: A New Perspective

X.509 and other current certificate schemes handle revocation using different strategies, but there is a similarity shared by all of these strategies: determining freshness is a separate process from obtaining the certificate. This means that communicating securely involves two initial steps: obtaining a person's certificate, and then obtaining information about the freshness of that certificate. Basically, a certificate is not useful without more current knowledge about its freshness.[1]

Let us assume that revocation information is updated daily (current schemes operate under a similar time frame). If we have to recheck the freshness of a certificate daily, having the certificate itself seems of little use. If we must recheck the certificate on-line each day, why not just get the certificate itself at the same time? Why not issue certificates daily with an expiration time of a day? It would provide the same freshness guarantee while simplifying the two-step process down to a single step. This is the central idea behind EFECT. Certificates are reissued daily, eliminating the need for revocation without sacrificing any freshness.

3.3 The EFECT Scheme

As previously mentioned, a CA in the EFECT scheme reissues every certificate on a daily basis (certificates expire daily). To update the certificate of a compromised key, there is no longer a need to issue a revocation statement and a new certificate. We simply alter the certificate the next day to reflect the key change. When a user gets a certificate, she knows it has not been revoked within a day. This freshness guarantee is exactly the same as the one provided by current two-step schemes.

[1]We assume that users of these systems check for revocation. Not bothering to check for revocation is not considered an acceptable solution in this thesis or in [EFECT]. Think of the financial problems of a store that did not bother to verify its customers' credit cards.

Daily reissuing is not feasible using other certificate schemes. Signing millions of individual certificates at 1 second per signature takes weeks if done as a serial computation. The reason EFECT can do daily reissuing is that EFECT CA's do not sign individual certificates. An EFECT CA essentially signs all of a day's certificates in a single digital signature, making the computational overhead much more reasonable.

The EFECT CA does this by arranging a day's certificates as the leaves of a Merkle tree [Merkle]. A Merkle tree employs a collision-free hash: a cryptographic primitive for "fingerprinting" data.[2] Each certificate is hashed in the leaf nodes. Sibling node hashes are concatenated and then hashed to compute the hash of the parent node. This proceeds all the way up to the root of the tree. Due to these recursive applications of the hash function, the hash value at the root then depends on every bit of information in the leaves (certificates). By signing this root hash, the EFECT CA is essentially signing all of the information in all of the certificates for that day.

Verifying an X.509 certificate involves verifying the digital signature on that certificate. Then that certificate is checked against a CRL to check for revocation. An EFECT certificate has no signature on it. Instead, an EFECT certificate contains a body (the certified information) and a path (the authentication). This path is a list of hashes up the tree from the certificate leaf to the root, including the sibling hashes along the way. By verifying the EFECT CA's signature on the root hash of the tree, and hashing the certificate body upwards with the provided hashes, the verifier knows that if the signed root matches her computed root, the certificate is valid. There is no revocation check since EFECT certificates are issued for one-day periods and are never revoked.

This path verification works because of the collision-free property of the hash. The logic is that if someone could give a path to the root that was not in the EFECT CA's tree, that person could find collisions in the hash function. Since the hash function is

[2]A collision-free hash is a function that maps input of an arbitrary length to an output of fixed length. It has two important and related properties. First, it is hard to invert (to compute the input given only the output). Also, it is hard to find a collision: two inputs that hash to the same output.

collision-free, this forgery is infeasible. The curious reader is referred to [Merkle] for a more rigorous proof of this assertion.

Note that the verification of the EFECT CA's root signature only needs to happen once a day for a given verifier. Further verifications only require hashing of the path from the certificate to the root, plus an equality check against the root hash signed by the CA.

The CA's building and hashing of the tree are guaranteed to be efficient. The underlying tree is a B+tree, so standard tree operations (insertion, deletion, and search) are guaranteed to be logarithmic in the number of certificates. Each day involves incremental changes to the tree (to add new certificates and delete old ones), which may make slight changes to the overall structure of the tree. Only the changed parts of the tree need to be rehashed. Even the initial hashing of the entire tree does not take very long since, according to Rivest, hashing is approximately 10,000 times faster than computing a digital signature. Exact times for this initial building and hashing can be found in section 6.2.1.

3.4 EFECT Advantages

As will be demonstrated in section 6.2.2, certificate verification is much more efficient in EFECT than in X.509. Additionally, EFECT has other advantages that are even more important than improved verification speed. This section discusses these advantages.

3.4.1 Off-line Certificates

Certificates were originally proposed as a mechanism for distributing public keys securely in an off-line manner. First, Diffie and Hellman [DH] proposed a trusted directory that would list names and corresponding public keys. They mentioned specifically that this directory could reside on-line. Later, Kohnfelder [Kohnfelder] noted that a trusted on-line directory is a performance bottleneck. He proposed splitting the directory into individually signed entries called certificates. By splitting the directory into certificates, users of the system would no longer need to contact a trusted authority in an on-line manner. With the authority's public key, certificate verification could happen off-line.

Given that the original purpose of certificates was to move key distribution off-line, the EFECT scheme attempts to deal with the revocation problem in as off-line a manner as possible. For a given verifier in the EFECT scheme, there is only one on-line communication per day: acquiring the signed root of the day from the CA. By verifying this one signature in the morning, the verifier can do the rest of the day's certificate verifications off-line by comparing the computed root hashes on the certificates to the signed one.

Note that this attempt to be as off-line as possible is in stark contrast to schemes such as SDSI/SPKI [SDSI], which require that all freshness checks be made on-line. It would seem that given the original motivation for certificates, this on-line approach is hardly an improvement over the original Diffie-Hellman idea of a central, trusted, on-line directory.

Although not explicitly stated in [X509], it is possible to elicit similarly off-line behavior from the X.509 scheme. By downloading the day's CRL in the morning and verifying the signature on it, a verifier can be off-line for the rest of the day. The verifier would verify the signature on an X.509 certificate, and then search the CRL for that certificate's serial number to verify freshness. This is exactly the mode of operation used in section 6.2.2 when comparing the performance of X.509 and EFECT. The advantages of EFECT in this scenario are discussed in detail in that section.

3.4.2 Stronger CA Security

Since an EFECT CA only computes one signature per day, it can afford to use slower signature schemes with stronger security properties.

The security of a digital signature scheme generally increases with the length of the signing key, but unfortunately the amount of time required to sign also increases. A CA must balance the security gained from a longer key against the need to perform a certain number of signatures per day. Since EFECT CA's compute fewer signatures per day than X.509 CA's, an EFECT CA can afford to use a longer key than an X.509 CA for a given user population.

For an EFECT CA, it is also possible to use a threshold scheme where the CA signing key is split into pieces and distributed around the globe. Some majority of the pieces would be needed to use the key, but the compromise of less than the majority of the pieces would not be a problem. The key could be rebuilt and reshared in the event of a partial compromise, so as to make the compromised shares useless. This threshold technique greatly enhances security, but is cumbersome for more than a few signatures. Here as with long keys, the fact that the EFECT CA performs fewer signatures enables the EFECT CA to use a previously infeasible technique to guard against CA key compromise.

Finally, the small number of signatures made by an EFECT CA means that there are less signatures available for cryptanalysis by an adversary than under X.509.

3.4.3 Recovery from CA Key Compromise

The EFECT scheme is more resilient to the compromise of a CA's private key than the X.509 scheme is. In the event that a CA private key is compromised, both schemes require the CA to generate a new key pair, and advertise that new public key over some secure out-of-band channel.

In addition to generating and distributing the new public key, an EFECT CA merely re-signs the day's root hash in the new private key, and broadcasts this re-signed root hash the same way that it broadcasts the daily root hash each day.[3]

Unlike the EFECT CA which only has to compute one signature, an X.509 CA would have to re-sign and redistribute each outstanding certificate in addition to issuing a CRL revoking each outstanding certificate. Also, this CA key compromise is more likely to happen to an X.509 CA than an EFECT CA given that the EFECT

[3]Actually, the CA must re-sign all old roots as well in order for the long term signatures (see section 3.4.4) to work properly. This is still an improvement over re-signing each certificate under X.509 when the number of certificates is larger than the number of days for which we archive signature keys.

CA can take better security precautions for a similarly sized user population (as mentioned in section 3.4.2).

3.4.4 Long Term Signatures

While the majority of digital signatures are used to authenticate ephemeral transactions (network logins, session key negotiations, etc.), some signatures must be verifiable for long periods of time. Things like loans, mortgages, and business contracts must be verifiable for many years after they are signed. The problem then becomes how to verify a digital signature many years after its creation. To securely verify the signatures on a signed document, one must have the certified public keys corresponding to the private keys that signed the document.

One strategy to make this work would be for the signers to attach their certificates to the document being signed (in addition to signing the document with their respective private keys). The problem then becomes CA compromise. In 30 years, a CA key is likely to change. One can't just verify the old certificates using the old CA key as that key may have been compromised, in which case arbitrary contracts could be forged. The certificates for the 30 year old public keys must somehow be signed in the most current CA key to prevent forgery.

In a scheme like X.509 where certificates are individually signed, it is very hard to make sure that a CA changing its key re-signs all old certificates. The CA would have to keep track of each certificate issued over the entire contract time span. Over time, this ends up being a very large storage requirement.

In the EFECT scheme, it is not necessary for the CA to archive entire old certificate trees. It is enough for the EFECT CA to archive just the daily signed root hashes of old trees. If the EFECT CA changes keys, it just re-signs the old root hashes using its new key. This effectively re-certifies all of the old certificates[4] without the need to keep track of each old certificate individually. It is then up to contract signers to

[4]It makes sense to question whether we should be blindly recertifying old certificates in old trees. With proper semantics, the answer is yes. By re-signing old roots, the CA is just certifying each old certificate tree as being the specific certificate tree it had on that given date in the past. The CA is not certifying that certificates from the old tree are valid in the most current timeframe.

store their certificates and hash paths of the day along with the signed contract.

A verifier of an old contract would verify (using the current CA public key) the root hash for the day listed in the contract, verify that the hashes for the attached certificates hash up to the signed old root hash, and finally verify the signatures on the document using the public keys specified in the attached certificates.

One final problem remains: compromise of contract signers' keys. This problem is not solved by EFECT, nor is it solved by X.509 or any other current certificate schemes. New forward-secure digital signature schemes such as that of Abdalla and Reyzin [AR] may help solve this difficult problem.

3.4.5 Atomic Certificates

Our previous notion of a certificate is a narrow one: a CA-signed binding of a public key to a name. To be more general, a certificate could be any statement signed by a CA. It could be a statement binding a person's name to a date of birth or to a social security number. It is conceivable that eventually there may be CA's that certify similar information. Note that unlike public key certificates, some of these other kinds of certificates may contain sensitive information. Unfortunately, if all of a user's sensitive information is put in a single certificate, the user loses the ability to disclose only the appropriate fields to someone.

In order to provide users with finer-grained control of the disclosure of their information, Raghu [atomic] proposed the idea of atomic certificates. Atomic certificates contain just one certified field, giving the user the power to disclose as little or as much information as is appropriate. The user would just reveal the atomic certificates for the appropriate fields.

When comparing the ability of X.509 and EFECT to handle atomic certificates, it is useful to think of the additional storage overhead per certificate: given one of a user's certificates, how much space (not including the certificate body itself) is required to store another certificate for that user? In the X.509 scheme, having the first certificate does not help you. Storing another X.509 certificate requires space for another digital signature. In the EFECT scheme, if a user's certificates are near each other in the tree,[5] storing another certificate only requires another couple of hashes. This is because the hash paths for the two certificates overlap in all but a few hashes in the bottom.

Most digital signatures (using reasonably large keys) are at least an order of magnitude larger than the output from commonly used hash functions, so EFECT allows a user to store more certificates in the same amount of space. Therefore, atomic certificates can be implemented much more efficiently (in terms of space) under EFECT than under X.509. This will be especially important in the future, when certificates are stored on smart cards and other devices with small memories.

3.4.6 Untrusted Directories

In the X.509 scheme, certificates and CRL's are distributed in various directories (called "repositories" in [X509]). Having many directories eliminates the bottleneck of users' having to get every certificate and CRL directly from the CA. The authors of [X509] claim that these directories can be untrusted since both certificates and CRL's are signed by the CA. While it is true that signed information can be distributed without any doubts as to its authenticity, it is not true that these directories can be truly untrusted. If an X.509 certificate does not exist, there is no CA signature attesting to its non-existence.

This means that a directory can plausibly deny the existence of a certificate regardless of whether that certificate actually exists. This flaw makes possible at least two unpleasant scenarios. In the first scenario, a company bribes a directory into not giving out the certificates of the company's competitors. In the second scenario, a high-profile directory loses part of its database due to technical problems and tries to deny that the lost certificates exist (in order to save face). In both of these scenarios, the user is none the wiser since there is no CA signature attesting to a certificate's non-existence.

In the EFECT scheme, it is possible for a directory to prove the non-existence of a

[5]This is a very reasonable assumption. If the search key for the B+tree is something like "username-field", then all of a user's atomic certificates would indeed be adjacent.

certificate. The proof uses hash paths from the certificate verification tree similar to the ones used for certifying the certificates. The only difference is that at the bottom, the hashes are taken over the search keys instead of over the certificate bodies.

To prove the non-existence of Bob's certificate, the directory would provide the search keys on either side of where Bob's search key would go (say "Alice" and "Charlie") and the search key hash paths of these two search keys. By verifying that the two hash paths were for adjacent search keys in the tree, and that they each hashed up to the search key root hash of the day signed by the CA[6], the user would be convinced of the non-existence of Bob's certificate.

Note what is proved in each step of the verification. By checking that each search key hashes upwards to the root hash, the user verifies that the two search keys are in the tree. By checking the adjacency of the hash paths, the user verifies that the two search keys are adjacent in the tree. Since the two search keys are adjacent, Bob's search key could not be between them, so Bob's certificate is not in the tree. Both checks are necessary to prevent the directory from creating false non-existence proofs.

By requiring directories to prove non-existence of certificates, the EFECT directories can be truly untrusted unlike the X.509 directories. By requiring users to trust directories, the X.509 scheme sacrifices security to gain efficiency in certificate distribution. The EFECT scheme requires no such security sacrifice for efficient certificate distribution.

3.5 Other Certificate Schemes

There are many other certificate schemes that have been proposed as improvements to X.509. Some of these schemes are Micali's CRS1 and CRS2 [CRS], Kocher's CRT [CRT], Naor-Nissim's 23CRT [23CRT], and SDSI/SPKI [SDSI]. Since EFECT is an attempt to improve upon X.509, it would seem logical to address these others

[6]To enable directories to do proofs of non-existence, the CA must broadcast not just one signed root value of the day, but two. The message would include a timestamp, the root hash value hashing over certificate bodies (for certificate verification), the root hash value hashing over search keys (for non-existence proofs), and the CA's signature over these three things.

attempts at improvement.

This thesis focuses only on X.509 and EFECT for a few reasons. First, it makes sense to compare EFECT to the most commonly used certificate scheme, which is certainly X.509 right now. Also, because X.509 is the scheme in common use today, software implementations of X.509 were easily attainable for the benchmark comparison in section 6.2.2.

Also, the major differences between EFECT and X.509 also exist between EFECT and all of the above-mentioned schemes: X.509 and all of the above-mentioned schemes involve individually signed certificates that may be revoked before their expiration date. EFECT certificates are not individually signed, nor can they be revoked before their expiration. This property that EFECT certificates are never revoked or negated once issued is referred to as *monotonicity* in the literature.

Because of the central difference in individual signing and monotonicity, the arguments about stronger CA security, easier recovery from CA key compromise, and better support for long term signatures and atomic certificates still hold for EFECT versus any of the above-mentioned schemes. The argument about untrusted directories holds against any of the above-mentioned schemes except CRS2 (CRS2 directories can also prove non-existence).

Chapter 4

Requirements and Design Decisions

This chapter describes the various parts of the EFECT prototype, what the requirements were for these parts, and what design decisions were made in order to meet those requirements.

4.1 Client Browser

The client browser allows a user to acquire and manage EFECT certificates. It can be configured to connect to any EFECT directory server on the Internet, and the CA public key is configurable. Using the client browser, a user can download EFECT certificates from a network directory, verify proofs of certificate non-existence from such a directory, verify and examine certificates, and store and load certificates to and from the local disk.

The first requirement for the client browser part of the prototype was that it must have a graphical user interface (GUI). The second requirement for the client browser was that it must be as portable as possible. These requirements were chosen to ensure that the users of the EFECT system would be able to use an easily understandable interface on any computer anywhere, regardless of operating system.

In order to meet both of these requirements, I chose to implement the client browser in Java. The Java Swing package provides a way to write graphical user interfaces that will run on any operating system to which Java has been ported (both Windows and Unix).

4.2 Certificate Verification Tree

The certificate verification tree part of the prototype is an implementation of the tree described in [EFECT]. It is a Merkle tree (hashes at every node) overlayed on top of a B+tree. The order of the tree, k, can be changed during compilation. The certificate verification tree implementation supports insertion, deletion, various searches, and calculation of the hashes.

The first requirement for the certificate verification tree was that it must be fast. The second requirement for the tree was that it must present a clean interface for tree operations (insertion, deletion, hashing, etc.). The speed requirement was meant to ensure that a server performing tree searches would be able to handle heavy query loads. The interface requirement was meant to ensure that there would be a clean, modular separation between the tree code and the directory server code.

In order to meet these requirements, I chose to implement the certificate verification tree in C++. I decided that an object-oriented interface to the tree operations would be the cleanest, which narrowed the language choice to Java or C++. The performance penalty associated with Java was not acceptable for this part of the system, violating the first requirement, so I chose C++.

4.3 Directory Server

The directory server is the replacement for the "repository" of the X.509 system. The directory server maintains a copy of a given day's certificate verification tree. Over a network, the server answers client queries about certificates in its tree. There can be many instances of the directory server running in many different locations. This mirroring of the tree enhances performance of the system and poses no additional problems of trust: the directories can be truly untrusted due to their ability to prove certificate non-existence.

The requirement for the directory server was that it must be fast, and that it must be able to service multiple requests from the network concurrently without blocking on any one of them. Also, the directory server must use the interface provided by the certificate verification tree.

I could have chosen a threaded Java server to handle the multiple concurrent connection requirement, but Java would have violated the speed requirement. Also, the interface to the tree would not have been very clean because it would have involved native (non-Java) function calls within the Java code.

I chose to implement the directory server in C for three reasons. First of all, C is fast and would satisfy the speed requirement. My second reason was that C has easy access to a non-blocking network programming interface. This interface would allow the program to service multiple requests in a single thread of control, eliminating the need to use multithreading. Eliminating multithreading is highly desirable in terms of both program complexity and performance. In a multithreaded program, one must lock all of the tree data structures before using them in order to prevent race conditions. If done improperly, the locking (or lack thereof) can cause bugs that are very hard to find. Even if done correctly, the locking and the thread context switching waste computer cycles that could be better spent servicing requests. My third reason for choosing C was that it would be easy to use the C++ tree implementation from a C program.

Chapter 5

Implementation Details

5.1 Client Browser

I wrote the client browser in Java, using Java's Swing package to implement the graphical user interface. I used Java's security package for the SHA-1 hash function, the DSA signature algorithm, and for parsing of X.509 certificates.

There are two threads of control in the client browser. One thread does all of the network communications (sending and receiving), and the other thread does everything else: responds to GUI input, performs cryptographic computations, and performs file system I/O. I kept the limit at two threads to simplify the programming, as complex multithreaded programs are notoriously difficult to write correctly. I broke the threads up this way because the network operations are the only ones that can block for long periods. By isolating the blocking network calls into a separate thread, I gained the ability to interrupt those calls with a stop button on the GUI.

Java's old thread control methods (`suspend()`, `stop()`, etc.) have been deprecated due to their unsafe nature. Some of them allow race conditions while others allow deadlocks. To do safe thread control, I devised a new method called `safeStop()` for stopping the network thread. The idea was that the user should be able to stop that thread with the push of a GUI button.

The basic idea behind `safeStop()` is that it sets a variable, indicating that the thread should stop. The thread polls this variable periodically to see if it has been asked to stop. The one problem is that the network thread may block during network reads. If the user were to push the stop button during a read, the thread would not stop because it wouldn't be able to poll the variable while blocking. To fix this, `safeStop()` shuts the network socket, interrupting any blocking reads or writes. The network thread gets an exception, polls the variable (which tells it to stop), and stops.

Computed results from the network thread get passed back to the GUI thread via a Vector. Some result passing was necessary, because the network thread cannot draw things on the screen in a thread-safe manner.

Most of the complexity of the client browser comes from the fact that it uses two threads. More threads would have further complicated it, and fewer threads would not have allowed the client to interrupt slow operations.

5.2 Certificate Verification Tree

The certificate verification tree was written in C++, using OpenSSL for the SHA-1 hash function.

First, I wrote the B+tree code. All nodes contain pointers to keys, leaf nodes contain pointers to data, and non-leaf nodes contain pointers to children. I used textbook pseudo-code definitions of the B+tree operations (search, insert, and delete) as the basis of this part of my code. Then, I added the hashes to each node, and wrote methods to hash the tree from the bottom up. Finally, I added methods to extract hash paths and certificates for a given search key.

The certificate verification tree is the part of the prototype that it is farthest from what a real system would need in terms of optimizations. By using pointers explicitly, the existing implementation assumes that all objects are in memory. In a production system, the entire tree would not be in memory all at once, so the representation would need to be changed. Section 7.1 discusses the changes that would be necessary in a real system.

Fortunately, there is a good abstraction layer between the certificate verification tree and the directory server. The changes outlined in section 7.1 could be made with a very small number of changes to the directory server code.

5.3 Directory Server

The directory server was written mainly in C, with the only C++ features used being the object-oriented calls to the certificate verification tree. Network I/O is done in a non-blocking manner using callbacks. This allows the server to handle multiple network connections simultaneously without using threading or multiple processes. This simpler design was much easier to write and debug, and can even outperform a threaded version in cases where the overhead of the threading library is very high (many simultaneous requests).

The server supports a few different modes of operation to allow for different kinds of certificates. The most common mode of operation is to allow anyone to ask for any certificate in the tree, and to allow anyone to ask for non-existence proofs for certificates not in the tree. This is the mode of operation that a directory server for a tree full of public key certificates would use.

There is also a mode of operation where one can only get the hash path for a certificate by presenting the certificate body, and where non-existence proofs are not supported. This mode of operation would be appropriate for a certified tree of sensitive personal data. In this case the server should not give out the data to unauthorized clients, nor should the server reveal which users are in the database. Since the non-existence proofs reveal user names from the database, they are disallowed in this mode of operation.

Chapter 6

Experiments

6.1 Experimental Apparatus

The benchmarks in section 6.2.1 were measured on a Sun Ultra 30 running Solaris 2.6 with 512 MB of physical memory and 1.5 GB of swap. All code was compiled using GCC 2.8.1. Routines for crypto (SHA-1 and DSA) and X.509 were taken from OpenSSL 0.9.6-beta1.

Initially, the benchmarks in section 6.2.2 were also measured on that same machine. For reasons that will be discussed in section 6.3, the experiment ended up being run on a second machine. The second machine was a Sun Ultra 60 running Solaris 2.8 with 1024 MB of physical memory and 2 GB of swap. The compilers and libraries were the same as on the first machine.

6.2 Experimental Questions

Experiments were performed to find two answers. The first answer we were looking for was the optimal value of k, the order of the CVT. The second answer we were seeking was how the EFECT prototype (using the optimal k) compares to a comparable X.509 system in terms of performance in an off-line, bulk verification scenario. This scenario was meant to model the daily computation performed by a merchant verifying many customer certificates (credit cards).

6.2.1 Order of the Certificate Verification Tree

The order k of the CVT determines the branching factor of the tree. Since a non-root node must have at least k search keys and no more than $2k$ search keys, the branching factor is at least $k+1$ and at most $2k+1$. This branching factor determines the length of a hash path from a tree of a given size and affects the running time of the tree operations.

Hash Path Length

It is possible to compute the number of hashes in a hash path from a CVT as a function of k, the order of the tree, and n, the number of certificates in the tree.

$$hashpathlength(n,k) = (k+1.1)(\frac{\ln n - \ln k}{\ln(k+1)} + 1) + 0.1 \tag{6.1}$$

$$hashpathlength(n,k) = (2k+1.1)(\frac{\ln n - \ln 2k}{\ln(2k+1)} + 1) + 0.1 \tag{6.2}$$

Equation 6.1 describes the length of a hash path (in number of hashes) for a minimally full B+tree while equation 6.2 is the analogous equation for a full B+tree.

These two length equations are plotted against n for various k in figure 6-1. The figure shows that a k of 1 minimizes the length for full trees while a k of 2 minimizes the length for minimally full trees. As one goes from full to non-full trees, the optimality of $k = 1$ quickly disappears, with that hash path length rising to almost the length at $k = 4$. Given this behavior, the k of 2 should minimize the length of the hash path for realistic trees.

The theory was supported by an experiment measuring the average lengths of hash paths on trees with 300,000 certificates and various values of k. Table 6.1 indicates that the k of 2 does indeed minimize the average hash path length. Note that averaging was necessary because the hash paths from non-full B+trees do not have the same length.

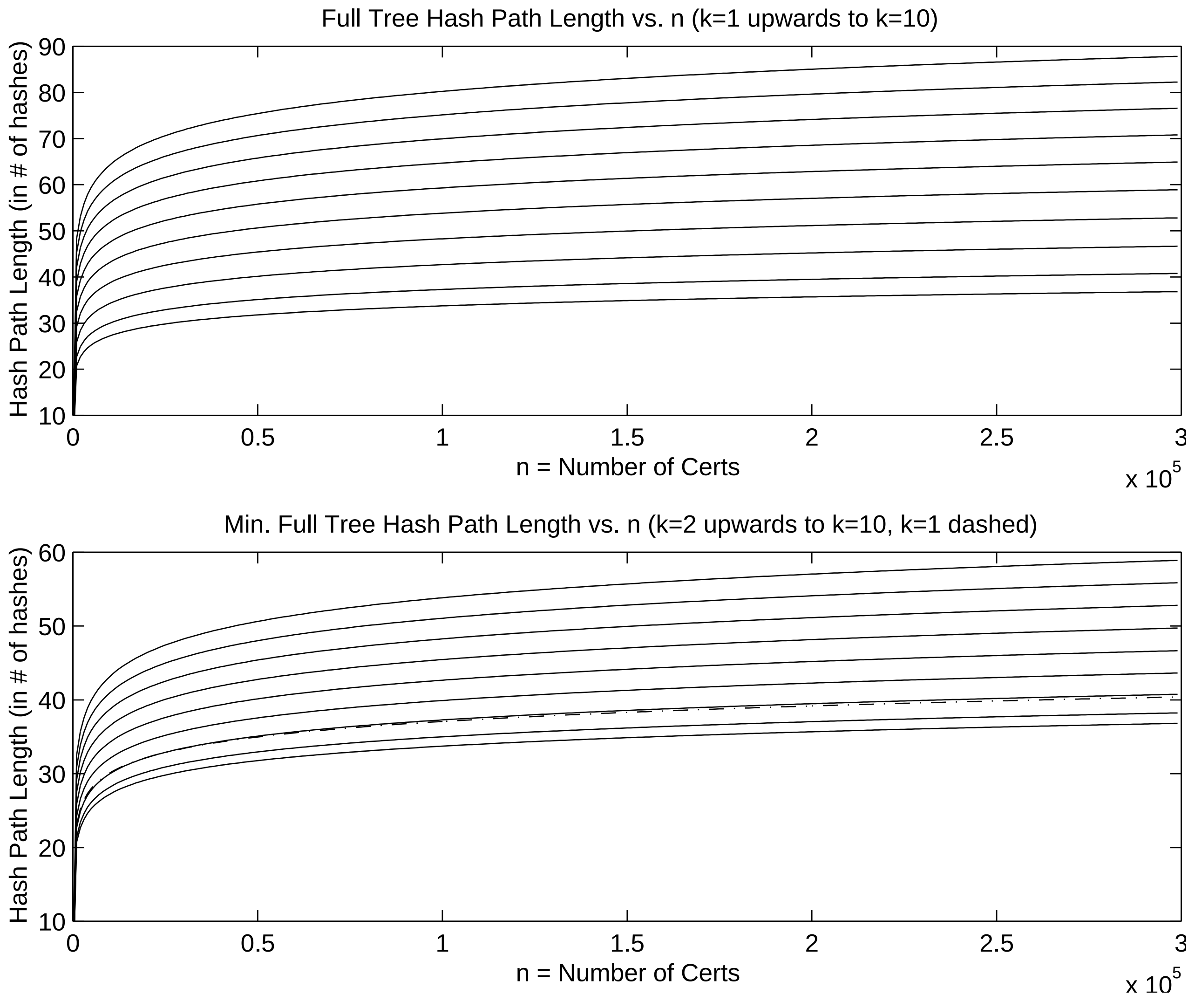

Figure 6-1: Plots of Hash Path Length vs. Tree Size (for various k's and tree fullnesses)

k	**Average Hash Path Length (in bytes)**
1	868.4
2	**804.5**
3	842.1
4	891.4
5	960.9
6	1056.2
7	1097.2
8	1126.3
9	1177.0
10	1241.8

Table 6.1: Experimentally Measured Avg. Hash Path Lengths for $n = 300,000$

Tree Search Speed

The search speed running time was not formally analyzed. A real implementation of a CVT would perform explicit management of a memory cache of nodes (see section 7.1). The formal analysis would basically be a cache-miss analysis of the search algorithms parameterized on Z, the size of the memory cache. Since the prototype implementation does not do this memory caching explicitly, its performance is intrinsically bound to the performance of the virtual memory subsystem of the underlying operating system.

A formal analysis of the interaction between the CVT and Solaris's virtual memory subsystem was not feasible, so an empirical analysis was performed instead. Speed measurements were taken for the various tree operations (building, hashing, and search) on trees with 300,000 certificates and various values of k. For each value of k, the time to build the tree from scratch was measured, the time to hash the entire tree was measured, and the total time required to do one hash path search for each key in the tree was measured. This whole process was repeated in a sequential loop ten times for each value of k ($k = 1, k = 2, ..., k = 10, k = 1, k = 2, ...$) The final time values for each k were obtained by averaging over the 10 trials for that k. Figure 6-2 shows the results of these measurements.

The search measurement is the most important as it measures the time to extract

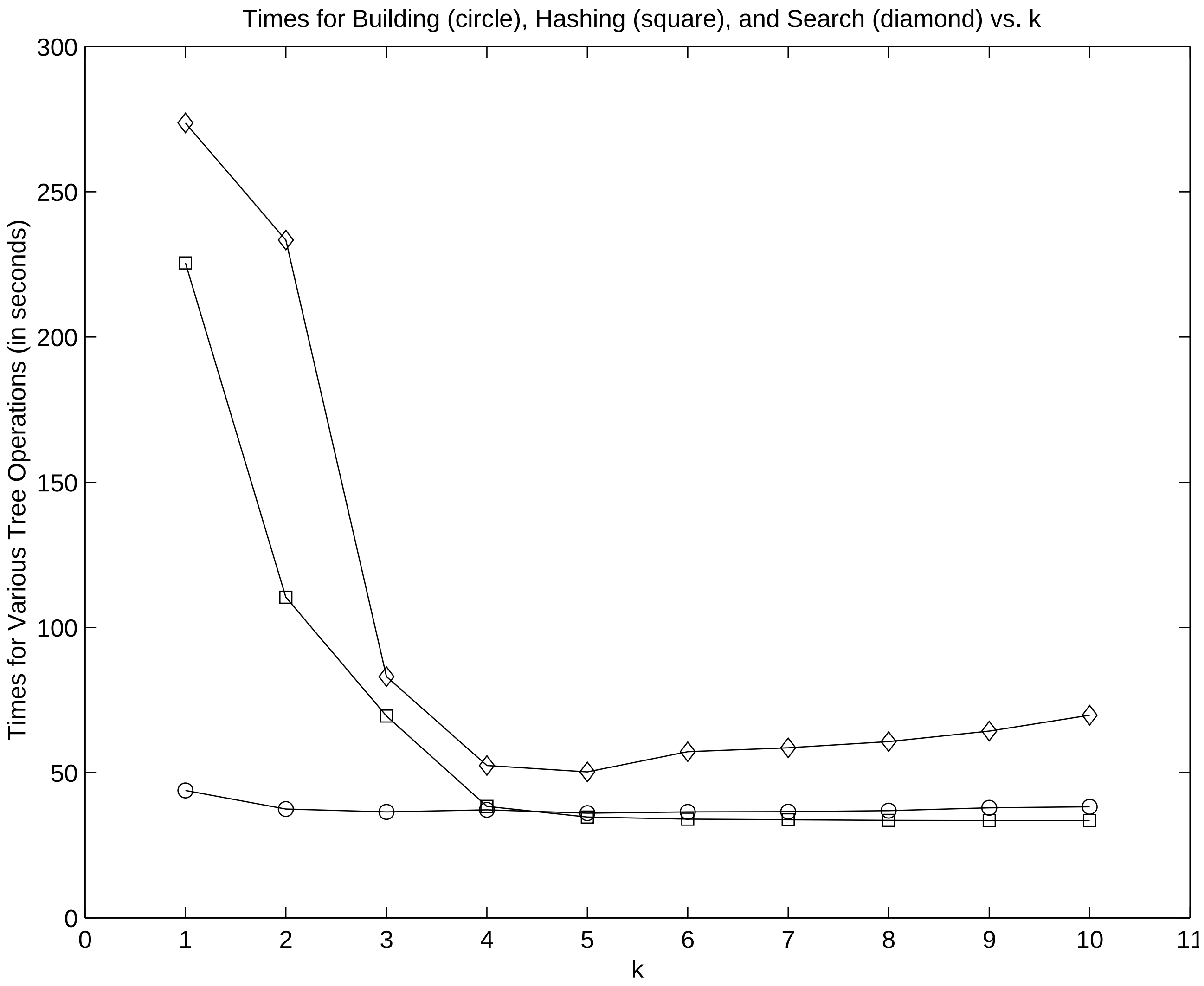

Figure 6-2: Plots of Speeds of Various Tree Operations vs. k (for $n = 300,000$)

a hash path given a search key. In EFECT, this search operation occurs much more frequently than tree building or hashing of the tree. In terms of speed performance, the search time is the most important thing to optimize. This speed optimization must be balanced against the hash path size optimization discussed previously.

The optimal search speed happens when $k = 5$, but it is not significantly faster than the speed at $k = 4$. Since the hash path gets smaller with smaller k around this point, we chose the value of $k = 4$ for the tests against X.509 in the next section (in order to have good speed and small certificates). Luckily, the speeds for tree building and hashing are also near optimal at $k = 4$.

6.2.2 Relative Performance of EFECT and X.509

It is claimed in [EFECT] that verification in the EFECT scheme is much faster than verification in the X.509 scheme. This experiment was designed to investigate that claim in an off-line verification scenario.

Off-line Bulk Verification Scenario

The off-line verification scenario is as follows. An X.509 verifier verifies the signature on the day's CRL once at the beginning of the day. Then for each X.509 certificate, the X.509 verifier verifies the signature on the certificate and scans the day's CRL for that certificate's serial number. An EFECT verifier verifies the signature on the root of the day once at the beginning of the day. Then for each EFECT certificate, the EFECT verifier verifies that the certificate's hash path hashes up to the signed root of the day. Each verifier will verify many certificates during the day (thousands of verifications). All verification is off-line except for the initial verification of the CRL or the signed root.

This experiment was meant to model what a busy merchant would do in a day of verifying something like a credit card certificate. The model was chosen because it is a model in which verification time savings will be most pronounced (the per-certificate time saved is multiplied by the large number of verifications performed). Another reason this model is relevant is because EFECT certificates can be significantly smaller than X.509 certificates when grouped together, allowing more of them to fit on a smart card (the credit card being verified). EFECT's good performance in this experiment would indicate an area where EFECT has multiple significant advantages over X.509.

Benchmark Details

The benchmark had two main phases:

1. For each of 300,000 users, certificates were created and written to disk for each of the three schemes to be tested: EFECT with SHA-1, X.509 with DSA and SHA-1, and X.509 with RSA and SHA-1.

2. For each of 300,000 users, the X.509 DSA certificate was verified, the EFECT certificate was verified, and the X.509 RSA certificate was verified (in that order). All verifications were timed using a microsecond timer.

The certificate bodies for all three schemes were identical: a fixed PGP 1024-bit public RSA key with the given user's name prepended. Since we were not interested in extracting information from the certificates themselves, the X.509 format was not actually used. Verification times depend mainly on the length of the data in question, not on its specific content. This certificate format was chosen for simplicity. Had the actual X.509 format been used, the results would not have been significantly different.

For EFECT, the root was signed using a 2048-bit DSA key. For X.509 with DSA, the certificates and the CRL were signed using the same 2048-bit DSA key as used in EFECT. For X.509 with RSA, the certificates and the CRL were signed using a 2048-bit RSA key. The two CRL's were identical (except for the signatures). They each contained the same 10% of the certificate serial numbers (every tenth serial number was on the list). The serial numbers were not actually written on the certificates, but were implied by the sorted order of the usernames.

EFECT verification time includes the time to hash up the path to the alleged root plus the time to compare the alleged root to the signed root. X.509 verification time

Certificate Scheme	**Mean Verification Time (in μs)**	**Standard Deviation (in μs)**
EFECT with SHA-1	160.0	8.3
X.509 with DSA and SHA-1	73656.7	9612.9
X.509 with RSA and SHA-1	6259.5	2995.9

Table 6.2: Mean Verification Times and Standard Deviations for EFECT and X.509

includes the time to verify the digital signature plus the time to do a binary search on the CRL for the certificate's serial number.

EFECT verification times do not include the time to verify the DSA signature on the root. X.509 verification times do not include the time to verify the signature on the CRL. Since these verifications only happen once per day, the differences in their times are not important.

For each of the schemes, the 300,000 verification times were averaged. These means (and their standard deviations) are shown in table 6.2.

None of the verification times include disk accesses. Variations in disk access times would have been nearly as large as the verification times themselves, making the timing data useless.

Actually, the first version of this experiment was aborted when it was discovered that the directory being accessed was an NFS mount rather than a local hard disk. This was discovered because the experiment had taken an order of magnitude longer to run than had been predicted, and still was not finished. When this was discovered, the experiment was moved to an isolated machine with a large local disk. It is worth noting that the non-local file system affected performance of the system far more than the certificate verification scheme used.

6.3 Conclusions

The parameter k, the order of the EFECT CVT, affects both the size of the hash paths associated with EFECT certificates and the speed at which a server can extract those paths from the tree. Unfortunately, space optimality and speed optimality occur at different values of k. Fortunately, the two optimal values were not too far apart. The calculations and experiments from section 6.2.1 indicate that smaller k results in shorter hash paths for $k \geq 2$. Section 6.2.1 also showed that a k of 5 provides the fastest hash path search time,[1] but that $k = 4$ was approximately as fast and would result in smaller hash paths. This k of 4 was used in the benchmark comparing EFECT and X.509.

Certificate verification is significantly faster in the EFECT scheme than in the X.509 scheme. EFECT verification is about 460 times faster than DSA X.509 verification and about 39 times faster than RSA X.509 verification. However, it is worth noting that these speed savings are less significant in any verification system that stores certificates on a hard disk. This is because the time difference between verification schemes is on the order of the time required for a hard disk access. In a verification system accessing a networked file system, the verification time savings become completely insignificant compared to the amount of time spent accessing the file system.

So although EFECT is indeed faster for verification, its novel properties (as described in section 3.4) are probably more important than any verification speed gains. It would be wise for authors designing future schemes to notice this fact: further verification speed gains probably are not meaningful in real systems.

[1]Note that this k of 5 was empirically determined, and that a system that implements the explicit node caching described in section 7.1 would probably have a different optimal value of k.

Chapter 7

Future Work

This chapter discusses aspects of the prototype that could be improved by future work. These ideas were considered but rejected due to time constraints.

7.1 Memory Cache Manager

For optimal performance, the certificate verification tree routines should utilize a custom memory cache manager that would move tree nodes between memory and disk appropriately. This cache manager would be tailored to the tree algorithms, having an appropriate replacement policy and performing occasional prefetching of nodes.

The tree routines would have to request nodes from the cache manager before using those nodes, and would relinquish nodes when through with them. The cache manager would pin a node in memory upon request, only moving it back to disk after the application relinquished it. The cache manager could also keep non-requested nodes in memory as per its prefetching strategy and replacement policy. Between a routine's requesting a node and relinquishing that node, the cache manager would provide the routine with access to the node through accessor functions using some handle to identify the node. By routing the node accesses through accessor functions, the cache manager could easily keep track of dirty nodes and access patterns. This information would make the cache manager more efficient because it would not have to write non-dirty nodes back to disk, and it would have more detailed information available for prefetching and the replacement policy.

Prefetching would only make sense during horizontal tree traversal. This is because during vertical traversal, there is no way to predict which child will be accessed. During horizontal traversal, all siblings of a given node are generally accessed (to gather the appropriate hashes for verifying a certificate). Given this, a good prefetching strategy would be to prefetch sibling groups during horizontal traversal. The accessor functions should be written in such a way as to make it easy for the cache manager to determine the traversal direction of the calling routine.

The replacement policy could be similarly tuned to the access patterns of the tree routines. It is known that the tree searches go down the tree to the bottom and come back up those same nodes, traversing siblings at each level. Given this traversal pattern, the cache manager knows that once the routine has moved up past a given node, that node will not be used again.

Tuning the replacement policy to this access pattern would result in a lower percentage of cache misses than that achieved by the operating system and its general purpose replacement policy for its virtual memory system. Another reason the cache manager would increase performance is that it would use a page size appropriate for the size of a tree node, rather than a general purpose page size used by the operating system.

The cache manager would make it possible to run the directory server on a machine with a small virtual memory. We would never need too much virtual memory at any one time (just enough for a path down or across the tree). In the current prototype, the entire tree resides in virtual memory, making it impossible to serve large trees on machines with small virtual memories.

This cache manager optimization was not done due to time constraints. It would involve replacing the pointer fields in each node with a handle understood by the cache manager. It would also require making each tree node a fixed size, with search keys embedded in the nodes. Currently the search keys reside outside of the nodes with the nodes holding only pointers to them. Currently, the certificates also reside outside of the nodes. In a tree full of small atomic certificates, the certificates should probably be moved into the nodes. In a tree of public key certificates, the certificates should probably be nodes of their own as their size would be comparable to the size of non-leaf nodes.

Note that this optimization could change the empirically determined value of k from section 6.2.1, as the operating system's virtual memory performance may well have determined that value.

7.2 CA Servicing of User Requests

The current prototype has a fixed tree that it serves. Currently, there are no mechanisms for changing the tree other than the low-level C tree library routines for insertion and deletion. In a real system, the CA would want some high-level way to queue up certificate creation and revocation requests and incorporate those user requests into the next day's tree. This high-level user request queue interface would be implemented using the low-level tree interface.

The details of servicing user requests would depend heavily on the nature of the CA. A credit card company would already have mechanisms for processing new card and revocation requests. A credit card CA would then just add software to reframe the requests as tree operations for the next time period. Basically, this issue is not directly related to the EFECT scheme. It is just a matter of how a real company or organization processes input from its customers or members. The requests from the customers or members would just need to be mapped into tree operations, which would be queued up for the next time period.

At the beginning of the next time period, the CA would perform the necessary tree manipulations (insertion for issuing a certificate and deletion for revoking a certificate), and would rehash the necessary parts of the tree. Finally, it would timestamp and sign the top of this new tree.

7.3 CA to Directory Communication

In the current prototype, there is no separation between the CA and the certificate directory. The building, hashing, and signing of the tree are all currently done in the directory server. In a real system, the CA and the directory would be distinct entities, so there would need to be some protocol to move the tree between the two.

For efficiency, the CA could send a list of the incremental changes from the last time period's tree, rather than a copy of the entire new tree. The incremental change message would include the search keys of revoked certificates (to be deleted from the tree), and the search keys and bodies of newly issued certificates. This communication would be digitally signed by the CA to prevent forgery. Encryption would only be necessary for trees of sensitive information, not for trees of public keys. Assuming that there was agreement between the CA and the directories on the order in which to perform these tree operations, the directories and the CA's would end up back in synch for each time period without having to send the entire tree over the network. In the beginning, a new certificate directory would get an incremental update from nothing, basically a list of all search keys and certificates in the tree.

For tree rehashing, the two options would be to have the CA send over a signed list of the new hash values or to have the directories recompute these values locally. At first it seems wasteful to have the local directories recompute these hashes when the CA is already computing them. In fact, this rehashing is not an expensive computation. Doing the rehashing locally is fast enough for an off-line computation, saves bandwidth by not sending those hashes over the network, and simplifies our design by not requiring yet another protocol be spoken between the directories and the CA.

Finally, the CA would have to send the directories the timestamped signature for the new time period.

7.4 Smart Cards

One of the major uses for EFECT is in a credit card scenario, where the credit card company would run a CA, certifying the keys of the credit card holders. The credit card holders would keep their private keys and their certificates on smart cards. Stores would have smart card readers where customers would insert smart cards in order to make purchases. The smart card readers in stores would verify the certificate hashes on a customer's smart card, as well as performing some sort of challenge/response protocol with the card in order to verify possession of the private key corresponding to the certificate.

The current prototype can only store downloaded certificates on the local disk of the machine running the EFECT client software. Eventually, the client software will need to be able to store the downloaded certificates onto a smart card. It may be possible to automatically download and save the user's certificates on the smart card in response to the insertion of the card into the card reader. This functionality would greatly simplify the user experience.

Also, the hash path verification routines currently found in the client software would need to be ported to the smart card reader. The challenge/response protocol would need to be written from scratch for the smart card reader as it does not exist in the current prototype.

7.5 Hash Path Compression

The current encoding of hash paths is not optimal for multiple certificates. When storing certificates that are nearby nodes in the same CVT, the hash paths will differ in only a few hashes at the bottom. This hash redundancy means that very good compression is possible. One such compression scheme would involve a bit flag for each hash denoting the hash as either a literal value or a reference to another existing hash. The reference would only have to be a few bits in length. By encoding redundant hashes as a few bits instead of 20 bytes, we would save significant amounts of space when storing adjacent certificates in the tree.

This path compression will be especially important when users have many certificates to carry on devices with small memories (smart cards, phones, other wireless devices, etc.).

Chapter 8

Summary

EFECT is a new certificate scheme that solves the certificate revocation problem more elegantly than do current schemes like X.509. Compared to X.509, EFECT also provides faster certificate verification, smaller certificates, fully untrusted certificate distribution, and better protection against CA key compromises.

The software described in this thesis implements the EFECT scheme and meets the prototype design requirements. The software is portable and modular, and it meets speed requirements. Also, it includes a graphical interface for the client.

The order of the EFECT certificate verification tree affects both the size of the EFECT certificates and the search performance of the tree. An order was found in this thesis that is nearly optimal for both certificate size and search speed.

The EFECT prototype was configured with this nearly optimal tree order, and an experiment was conducted to compare the relative performances of EFECT and X.509 in an off-line, bulk verification scenario. The results show that certificate verification is significantly faster in the EFECT scheme than in the X.509 scheme. It should be noted that although EFECT verification is faster, the time savings can easily be dwarfed by the time required to access a file system. EFECT's other advantages seem more important than improved verification speed in light of this realization.

The results in this thesis demonstrate that EFECT is a suitable replacement for X.509 in situations where the certified user population is large, situations where bulk verification takes place, or situations where certificates must be stored in limited space (wireless devices and smart cards). Hopefully after a few future modifications, this software will be deployed to provide the public key infrastructure in some of these scenarios.

Bibliography

[AR] Michel Abdalla and Leonid Reyzin. *A New Forward-Secure Digital Signature Scheme.* In Tatsuaki Okamoto, editor, *Advances in Cryptology—ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 116–129, Kyoto, Japan, 3–7 December 2000. Springer-Verlag.

[DH] Diffie and Hellman. *New Directions in Cryptography.* IEEE Transactions on Information Theory IT-22, 6 (Nov. 1976), 644–654.

[EFECT] I. Gassko, P. Gemmell, and P. MacKenzie. *Efficient and Fresh Certification.* In PKC2000, pp. 342–353.

[X509] R. Hously, W. Ford, W. Polk, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile.* RFC 2459.

[CRT] Paul Kocher. *A Quick Introduction to Certificate Revocation Trees (CRTs).* http://www.valicert.com/technology/

[Kohnfelder] Loren Kohnfelder. *Towards a Practical Public-key Cryptosystem.* Bachelor's thesis, MIT, May 1978.

[Merkle] R. Merkle. *A Certified Digital Signature.* Advances in Cryptology: CRYPTO '89, pp.218–238.

[CRS] S. Micali. *Efficient Certificate Revocation.* RSA Data Security Conference, San Francisco, California, January, 1997.

[23CRT] M. Naor, K. Nissim. *Certificate Revocation and Certificate Update.* Proceedings of Usenix '98.

[atomic] Narayan Raghu. *ATOMIC CERTIFICATES*. IETF Internet draft.

[SDSI] links to SDSI and SPKI materials can be found at http://theory.lcs.mit.edu/cis/sdsi.html