Dottorato di Ricerca in Informatica

XV ciclo

Università degli Studi di Salerno

Network Security

Sk-DNSSEC:

an alternative to the Public Key scheme

Syncfiles:

a secure file sharing service for Linux

Aniello Del Sorbo

Novembre 2002

Coordinatore:	Relatore:
Prof. A. De Santis	Prof. G. Persiano

Contents

Preface **vii**

Contributions of this work **viii**

I Network Security **1**

Security Overview **2**

1 Network Security **3**

1.1 Internet 3

1.2 Threats to Networks 4

1.3 Cryptography 4

1.3.1 Symmetric Algorithms 6

1.3.2 Public Key Algorithms 6

1.3.3 Secret Sharing 6

1.3.4 Threshold Cryptography 7

1.3.5 Proactive Security 8

II DNS Security **9**

2 The Domain Name System **10**

2.1 What it is 10

2.2 The Domain Name Space 11

2.3 How it works . . . 11
2.4 Weakness: threats to the DNS . . . 12
2.4.1 Cache poisoning . . . 12
2.4.2 Client flooding . . . 13
2.4.3 DNS Dynamic Update Vulnerabilities . . . 13
2.4.4 Information leakage . . . 13
2.4.5 Compromise of DNS server's Authoritative Data . . . 14
2.5 DNSSEC Extensions . . . 14
2.5.1 Objectives . . . 14
2.5.2 Performance Considerations . . . 16
2.6 Sk-DNSSEC . . . 16
2.6.1 Goals . . . 16
2.6.2 Symmetric Certificates . . . 17
2.6.3 The Protocol . . . 17
2.7 Implementation Issues . . . 21
2.7.1 The format of a symmetric certificate . . . 21
2.7.2 The format of the $Info$ field . . . 22
2.7.3 The format of the Key field . . . 22
2.7.4 The format of the $Nonce$ field . . . 23
2.7.5 The $DNS\_RootCert\_Req$ request format . . . 23
2.8 Implementation . . . 24
2.8.1 The DNS Library . . . 26
2.8.2 The NAMED daemon . . . 26
2.9 Analisys . . . 29
2.10 Performances . . . 30
2.10.1 Stored information . . . 30
2.10.2 Network traffic . . . 31
2.10.3 Computational Time . . . 31
2.11 Conclusions and future works . . . 32

III File System Security **35**

3 Network Filesystems **36**

3.1 NFS 36

3.1.1 Weaknesses 37

3.2 Secure Network Filesystems 37

3.2.1 CFS 37

3.2.2 TCFS 37

3.3 Secure file sharing of encrypted files 38

3.3.1 Sharing in cryptographic distributed file-system 38

3.3.2 Threshold cryptography 39

3.4 Syncfiles 40

3.4.1 The Proposal 41

3.4.2 The Protocol 41

3.4.3 Access revokation and key refreshing 44

3.4.4 Active and passive attacks 45

3.5 Implementation for Linux 45

3.5.1 Applications 48

3.5.2 Performance Considerations 49

3.6 Conclusions and future directions 50

Overall Considerations **51**

References **52**

List of Figures

2.1 Resolving `www.isc.org` 11
2.2 The *Symmetric Certificate* shared between name servers X and Y. 17
2.3 Checking a query 27
2.4 Signing an answer 28
2.5 Verifying a response 29
2.6 Making of a query 29

3.1 An overview of the key retrieving process. 42
3.2 The structure of a `wkl` node. 46
3.3 Structure of a decryption request. 46
3.4 Code fragment for the implementation of the `procfs`. 47
3.5 Code fragments for the kernel thread that issue decryption requests. 48
3.6 Starting the decryption daemon. 50

Preface

This thesis proposes a valid alternative to the *Internet Software Consortium* (ISC) DNSSEC extensions in securing the DNS system and proposes a way to securely limit and log accesses to group encrypted secure resources. The document is divided into three parts: Part I will briefly discuss topics such as network security and a bit of theory of cryptography; in Part II is shown that one of the most important piece of the Internet is easy to hack and shows what researchers are doing to make it more secure and thus reliable (DNSSEC), also our solution (Sk-DNSSEC) is presented and analysed; Part III will present a useful service that provides means for securely share a resource (tipically a file) and will discuss about its implementation in the Linux Operating System. Moreover, it is shown how the *Transparent Cryptographic File System* will use this service to securely share files among a group.

Contributions of this work

Sk-DNSSEC is a project leaded by Prof. G. Ateniese and it is developed at the Computer Science Departement of The Johns Hopkins University in Baltimore, MD, USA. It is funded by the National Science Foundation (NSF). In this thesis we will briefly describe Sk-DNSSEC and we will discuss in details its implementation and performances.

TCFS is a project leaded by Prof. G. Persiano and it is developed at the Computer Science Departement of the Universitá degli Studi of Salerno, Italy. It was born seven years ago. Many new features were added during these seven years. *Syncfiles* is the last of these features and in this thesis its protocol is described and fully analysed as well as its implementation and performances.

Part I

Network Security

Security Overview

During its life Computer Science did a lot of improvements on several aspects: CPUs are very powerful, memory is very cheap and networks are very fast. Today we can count on an affordable, yet very powerful, computer that we can connect to the Internet almost at no cost and almost everywhere in the world.

Because of the increased reliance on powerful, networked computers, to manipulate and store our personal information and to help run businesses, a particular attention has grown around the practice of network and computer security.

Many individuals, and, unfortunately, many organizations think about computer security as a not very important issue; computer security is something that has been overlooked in favor of power and productivity. It is seldom, and many agree with this, to take the right security measures before connecting a computer, or a LAN, to an untrusted network (such as the *Internet*).

In this era an estimated 400 Million people use or have used the Internet. Everyday the Computer Emergency Response Team (*CERT*, created in the 80s to alert computer users of network security issues at Carnegie Mellon University) reports an estimated 140 major incidents of vulnerabilities exploits. Considering that the estimated worldwide economic impact of the three most dangerous Internet Viruses of the last two years, was a combined US $13.2 Billion, computer security has become a quantifiable and justifiable expense for all IT budgets.

Chapter 1

Network Security

1.1 Internet

In the late 1960s, the U.S. Department of Defense's Advanced Research Projects Agency, ARPA and later DARPA, founded the ARPAnet. ARPAnet was an experimental wide area computer network connecting research organizations in the United States. Originally ARPAnet served to allow government contractors to share expensive or scarce computing resources. But since the beginning users of ARPAnet used it to collaborate to each other. Collaboration ranging from sharing files and software as well as exchanging mails.

In the early 1980s TCP/IP (Transmission Control Protocol/Internet Protocol) was developed and quickly became the standard host-networking protocol on the ARPAnet. It was built into the University of California at Berkeley's BSD Unix, a virtually free operative system. This was the beginning of everything. Internetworking became cheaply available to many more organizations than were previously though by the ARPAnet folks. The number of computers connected to each other by the ARPAnet, that became a backbone of a confederation of local and regional networks based on TCP/IP, i.e. the backbone of *Internet*, rapidly increased.

In 1988 DARPA decided to stop the ARPAnet project and the NSFNET (founded by the National Science Foundation) became the new Internet's backbone. Today Internet is compound by multiple commercial and not-commercial backbones and connects millions of hosts around the world and it is still growing at a very fast and chaotic rate.

1.2 Threats to Networks

Experts discover new security vulnerabilities in software applications as well as in network protocols almost every day. These newly discovered vulnerabilities may be due to flaws in software or they may be the result of software configuration errors. Hackers or other malicious individuals can exploit these vulnerabilities to gain access to network assets. Administrators must spend a lot of time and energy just staying informed about and dealing with new vulnerabilities. Failure to defend against the key threats to data and network assets can result in disaster. The first and last word in security for many companies is firewall. Many administrators regard the firewall as a magic bullet that will somehow make their networks imperverious to risk. A firewall is a necessary and important part of any security program, however, by itself, a firewall can do anything against a network flaw. If an administrator configures a firewall so that DNS packets can pass thru it and reach the internal DNS server (an usual configuration) then a flaw in the DNS protocol (and there are many as we will see in Section 2.4) may let hackers by-pass the firewall and enter within the network.

Many network protocols were originally designed without security in mind. The Internet was, and is today, an *insecure* network. As the Internet grew and a lot of people became "connected", it was used for many new purposes. People started to use the Internet to share personal information with friends, IT companies started to make investements in new Internet based technologies, E-Commerce became a reality. None of this would be possible without putting "security" into new or existing network protocols. The science of **Cryprography** was born[1] from these efforts in making protocols more secure and thus reliable.

1.3 Cryptography

We will briefly discuss, in this section, what cryptography is. Suppose a user (*the sender*) whishes to send a message to another user (*the receiver*), but both the partecipants also want that nobody else can read the message. Let us first introduce some terminology. A message is *plaintext* (or *cleartext*), usually denoted by M. A *ciphertext*, usually denoted by

[1] Indeed, Cryptography has a very long history as Egyptians made a limited use of it some 4000 years ago. See Khan's *The Codebreakers* for a complete non-technical account on the subject.

C, is a message that has been disguised in such a way as to hide its content (this process is called *encryption*, while the opposite process, turning the ciphertext back to cleartext, is called *decryption*). **Cryptography** is the science that studies secure ways to disguise cleartexts into ciphertexts (and viceversa). Such *ways* are called *encryption/decryption algorithms*, or, briefly, *ciphers*. Most encryption algorithms base their secureness on the fact that the algorithm itself is kept secret and only known by the sender and the receiver. This is a major drawback that does not permit a wide use of the algorithm itself, as, otherwise, anyone can read a message intended only for a particular receiver. Instead a good encryption algorithm should be public and base its robustness on a *key*. This is a value that, given a cleartext M, a function (the encryption algorithm) E and a function (the decryption algorithm) D it computes $C = E_k(M)$ and $M = D_K(C)$ and the following property holds:

$$M = D_k(E_k(M))$$

A **Cryptosystem** is an algorithm, plus all possible cleartexts, ciphertexts, and keys. The science *opposite*[2] to cryptography (that is the science that studies how to break an encryption algorithm) is called **Cryptanalysis**. The new branch of mathematics encompassing both cryptography and cryptanalysis is called **Cryptology**.

Cryptography provides:

- **Confidentialty**: it should be possible to encrypt a message M to obtain $C = E_k(M)$ in such a way that only the receiver can decrypt C to obtain $M = D_k(C)$ (where E is the process of encrypting a cleartext and D is the process of decrypting a ciphertext).
- **Authentication**: it should be possible to be sure of the identities of both the sender and the receiver of a message. A malicious user should not be able to masquerade itself as one of them.
- **Integrity**: it should be possible to verify that a message was not altered during the process of sending it. A malicious user should not be able to modify it without the fact being noticed.
- **Nonrepudiation**: it should be impossible for a sender to assess that he never sent a message after having sent it.

[2]Indeed, Cryptography and Cryptanalysis cannot really be divided. One cannot design a cryptosystem without thinking if it is easy or not to break it.

There are two general types of key-based algorithms, *symmetric key* and *public key* ones.

1.3.1 Symmetric Algorithms

The algorithms where the encryption key can be calculated from the decryption key (even if usually the encryption and decryption keys are the same) are called **Symmetric Algorithms**. Both the sender and the receiver agree upon a key before starting to communicate in a secure manner. As long as the communication needs to remain confidential, the key **must** be kept secret.

1.3.2 Public Key Algorithms

The algorithms where the encryption key is different from the decryption key (and also it is infeasible to calculate one from another) are called **Public-key Algorithms**, because the encryption key is often made public. By using a public key algorithm anyone can encrypt a message using an user public key P_k, but only the user itself can decrypt it using its own private key S_k (the only key that **must** be kept secret).

1.3.3 Secret Sharing

A (n, k) secret sharing scheme is an algorithm to split a secret S taken from some finite domain D into n shares $h_1, \cdots, h_n$ such that the following properties hold

1. Given any set of t shares $h_{i_1}, \cdots, h_{i_k}$ the secret S can be efficiently reconstructed.
2. Any set of $t-1$ shares $h_{i_1}, \cdots, h_{i_{k-1}}$ does not give any information on the secret S (in the sense that any secret $s \in D$ is equally likely).

There exist various implementations of (n, k) secret sharing schemes. We next describe the scheme based on polynomial interpolation due to A. Shamir [29].

Let S be the secret to be shared, integers $u_1, , , u_n$ represent identities associated to the group members and let $t \leq n$ the stated threshold. We assume S and the users' identities to be taken from the finite field GF(2^m) for some integer m and all arithmetic is intended to be performed in the field.

To compute the shares the following algorithm is executed.

1. Randomly pick $a_1, , a_{k-1} \in \mathrm{GF}(2^m)$ and consider the polynomial

$$P(x) = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + ... + a_1x + S.$$

2. The shares $h_1, \cdots, h_n$ are computed by evaluating the polynomial $P(\cdot)$ at the points $u_1, \cdots, u_n$; *i.e.*,

$$h_i = P(u_i) \quad 1 \leq i \leq n.$$

Given shares $h_{i_1}, \cdots, h_{i_k}$ corresponding to members $u_{i_1}, \cdots, u_{i_k}$ the secret is reconstructed as follows.

1. Compute the Lagrange interpolating polynomial

$$I(z) = \sum_{j=1}^{k} h_{i_j} L_{k,j}(z)$$

where

$$L_{k,j}(z) = \prod_{l=1, l\neq j}^{k} \frac{z - u_l}{u_{i_j} - u_l}.$$

2. Output $I(0)$.

It is easy to see that if t shares are available then the reconstruction algorithm always succeeds. Suppose now that only $t-1$ shares $h_{i_1}, \cdots, h_{i_{k-1}}$ corresponding to members $u_{i_1}, \cdots, u_{i_{k-1}}$ are available. Then simple algebra shows that for any possible secret $s \in \mathrm{GF}(2^m)$ there exists a polynomial $I_s(x)$ that is consistent with the shares and such that $I_s(0) = s$. Thus, $t-1$ shares leave the secret undetermined.

1.3.4 Threshold Cryptography

The idea of threshold cryptography is to protect information (or computation) by fault-tolerantly distributing it among a cluster of cooperating computers. Consider the fundamental problem of threshold cryptography, a problem of secure sharing of a secret. A secret sharing scheme allows one to distribute a piece of secret information among several servers in a way that meets the following requirements:

1. no group of corrupt servers (smaller than a given threshold) can figure out what the secret is, even if they cooperate;

2. when it becomes necessary that the secret information be reconstructed, a large enough number of servers (a number larger than the above threshold) can always do it.

A very useful extension of secret sharing is function sharing. Its main idea is that a highly sensitive operation, such as decryption or signing, can be performed by a group of cooperating servers in such a way that no minority of servers is able to perform this operation by themselves, nor would they be able to prevent the other servers from performing the operation when it is required.

In many real-life situations, we don't believe that any given person can be trusted, and we may even suspect that a big fraction of all people are dishonest, yet it is reasonable to assume that the majority of people are trustworthy. Similarly, in on-line transactions, we may doubt that a given server can be trusted, but we hope that the majority of servers are working properly. Based on this assumption, we can create trusted entities. A good example of an application whose security could be greatly improved with a threshold solution is a network Certification Authority, a trusted entity that certifies that a given public key corresponds to a given user. If we trust one server to perform this operation, then it is possible that as a result of just one break-in, no certificate can any longer be trusted. Thus it is a good idea to distribute the functionality of the certification authority between many servers, so that an adversary would need to corrupt half of them before he can forge a certificate on some public key.

1.3.5 Proactive Security

Proactive security combines the ideas of distributed cryptography (secret sharing seen above) with the refreshment of secrets. In usual secret sharing schemes an attacker need to multiple locations in order to learn the secret. As the secret does not change, the attacker has a lot of time to mount this attack. The natural solution that comes in mind is to change the secret often. But this is not always feasibile. Thus, what will be changed are the shares of the secret, leaving the secret as is, in such a way che whatever information an attacker obtained from obsolete shares is of no use with the new ones. Similarly, to avoid the gradual destruction of the information by corruption of shares it is necessary to periodically recover lost or corrupted shares without compromising the secrecy of the recovered shares. These are the core properties of the *Proactive Security*.

Part II

DNS Security

Chapter 2

The Domain Name System

2.1 What it is

The Domain Name System (*DNS*) provides a mechanism for resolving human memorable host names into the Internet Protocol (*IP*) address. In a general view it is a hierarchical distributed database that allows the storing and the retrieving of Resource Records (*RRs*). One such RR is the IP corresponding to a particular host name.

As you can imagine it plays a very important and critical role in Internet, but, despite that, its protocol is very weak, also because it inherits all the weaknesses of the underlying IP protocol. Without DNS, the only way to reach other computers on the Internet is to use their numerical network addresses. But using them is not very user-friendly, thus the DNS is relied upon to retrieve an IP address by just referencing a computer system's Fully Qualified Domain Name (*FQDN*), that, basically, is a DNS host name that represents *where* to resolve this host name within the DNS hierarchy (we will see soon how).

The threats to DNS are due in part to the lack of authenticity and integrity check of the data held into the database and to the fact that most protocols rely their access control mechanism on the correspondence between the host name and the IP of that host name. Correspondence, as we now know, that is provided by an unsecure protocol: the DNS protocol. In a bit we will give an overview of the most common security weaknesses of the DNS and an overview of the new security extensions (*DNSSEC*) being worked on by the Internet Engineering Task Force's (*IETF*) DNSSEC Working Group (*WG*).

2.2 The Domain Name Space

The DNS can be seen (and actually it was designed in this way) as a tree whose root node is called the *root* (or the *root domain*). Each FQDN is just a path from the root (the "`dot`" domain) to the leaf (the "*host name*"). For example "`www.isc.org`" has "`www.isc.org.`" as its FQDN[2] where the last dot is represented by the root of the DNS tree and each *domain* (like "`.org`" and "`.isc`") are represented by the nodes traversed by the path down to the leaf (the host name "`www`"). This inverted tree is called the "*domain name space*" (See Figure 2.1). When a host (`stub` in the figure) need to know the IP of another host, the FQDN is used to traverse the tree to find the node (the name server) among whose leaves there is the looked for host (for example "`www`" among the host names of the domain "`.isc.org`").

2.3 How it works

Suppose a host wants to navigate thru the web pages of the `www.isc.org` site. It will ask to its stub resolver the IP address of `www.isc.org`. In its turn the stub resolver will forward (1) the query to the local domain name server, that will start looking for the answer. First

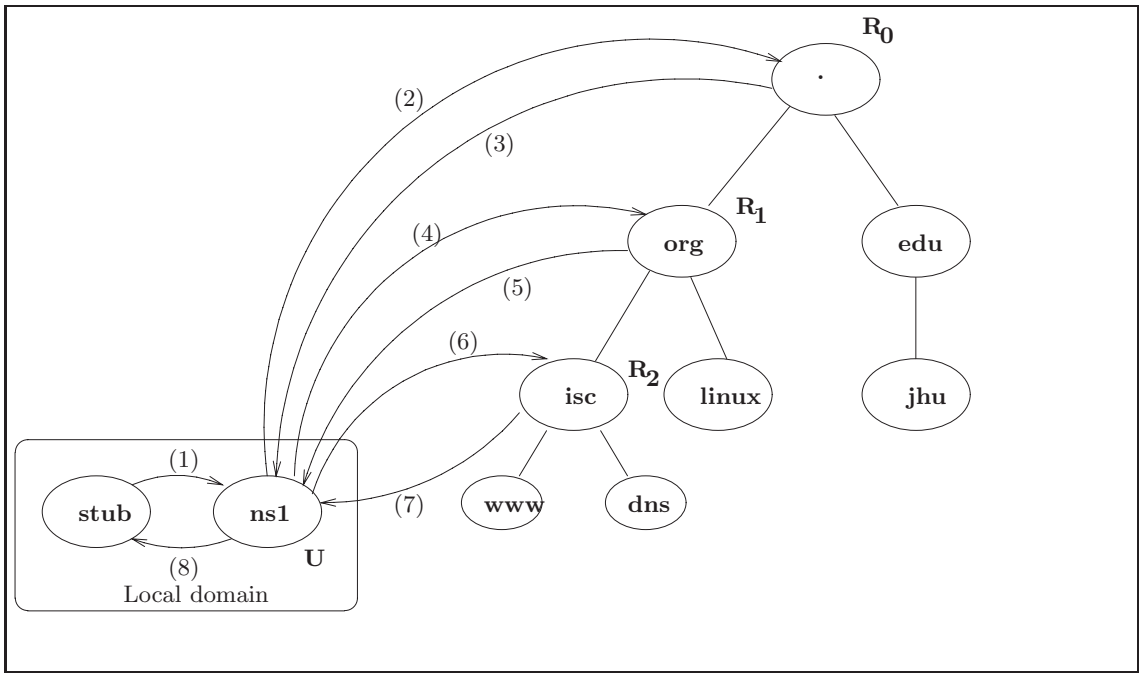

Figure 2.1: Resolving `www.isc.org`

the name server of the root domain "." is asked (2) for the translation, this will refer (3) the inquirer to the name server of the "`.org`" domain that in its turn will refer (5) the inquirer to the name server of the "`.isc`" domain that, eventually, being authoritative for that domain, will reply (7) with the requested, if available, IP of "`www.isc.org`". Got the final answer, the local domain name server will send back (8) the answer to the stub resolver and then to the web browser.

2.4 Weakness: threats to the DNS

In the original DNS specification there is no attempt to make the protocol secure. This is due to the fact that it is intended to transport public information as a means of communicating data. So the concept itself of restricting access to this data is considered not part of the DNS protocol, and that is right.

We need to focus our attention not on how to restrict access to the information stored in the DNS database, but on attempting to make sure that the information is reliable (*integrity*) and coming from the right place (*authenticity*). False information within the DNS can lead to unexpected and potentially dangerous exposures. Next we will take a look to main categories where the majority of the DNS weaknesses fall[1]:

- Cache Poisoning
- Client Flooding
- Dynamic Update vulnerability
- Information Leakage
- Compromise of DNS Server's Authoritative Data

2.4.1 Cache poisoning

Whenever a DNS server does not have the answer to a query within its cache (or the server is not authoritative for that response), the DNS server can pass the query onto another DNS server on behalf of the client. If the server passes the query onto another DNS server that has incorrect information, whether placed there intentionally or unintentionally, then cache poisoning can occur[13]. Malicious cache poisoning is commonly referred to as DNS spoofing[14].

2.4.2 Client flooding

Client flooding occurs when a client system sends out a query, but receives and accepts thousands of DNS responses from the attacker. The attack success is based upon lack of authentication of these responses. The attack is made to appear as if it is originating from the expected name server, but, without strong authentication, the client does not have the capabilities to verify the origin of the responses[15]. This attack can be used instead of DNS spoofing when attempting to host name spoof an application[16].

2.4.3 DNS Dynamic Update Vulnerabilities

RFC 1035 expects DNS zones to change slowly, thus it defines a static DNS where changes take place only in the zone files on the primary server and tipically through a manual process. DNS Dynamic Updates is a modification to RFC 1035 that allows Dynamic Updating of DNS information contained within a zone as long as several prerequisities are met. Protocols, such as Dynamic Host Configuration Protocol (DHCP) can make use of DNS Dynamic Update protocol to add and delete RR on demand. These updates take place on the primary server for the zone. The update take the form of additions and deletions[4].

The DNS Dynamic Update protocol has provisions to control what systems are allowed to dynamically update a primary server. Even if it is employed, it is a weak form of access control and is vulnerable to threats such as IP spoofing of the system performing the updates or compromise of the system. An attacker, who is able to successfully accomplish either, can perform a variety of dynamic updating attacks against the primary server. They can range from Denial-of-Service attacks, such as deletion of records, to malicious redirection, for instance, by changing IP address information for a RR being sent in update[5].

2.4.4 Information leakage

Other threats to the DNS include zone transfers that can leak information concerning internal networks to a potential attacker. Frequently, host names can represent project names that may be of interest or reveal the operating system of a machine[18]. Blocking zone transfer proves to be a futile effort in preventing such leaks of information. An

intruder can make use of DNS tools to automatically query, one by one, every IP address in a domain space in an attempt to learn the DNS host names or to find IP addresses that are not assigned. The latter motive of uncovering unused IP addresses may allow an intruder to use IP spoofing to masquerade as a host of a trusted network. If a system trusts an entire IP network, rather then specify every host that it trusts, then that system may be vulnerable to an attack using unassigned IP addresses.

2.4.5 Compromise of DNS server's Authoritative Data

Other threats against DNS servers include the threat when an attacker gains administrative privileges (e.g. root on Unix systems) with the intent of modifying zone information for which the server is authoritative. This is achieved thru other vulnerabilities on the server not necessarily related to DNS. Careful configuration of the DNS server can provide some protection to these threats. Such things as using the latest version of *BIND*, minimazing the number of other services offered on the same machine, limiting access to just administrators, employing split DNS technology, etc. are crucial to a safer DNS service for an organization. There are many sources to assist in such configuration including[18], [17], and any relevant CERT advisories. Unfortunately this does not provide strong protection against tampering of the data in the DNS files on the server. The appropriate security measures needed to provide adeguate protection whitin the DNS could be accomplished through the DNSSEC.

2.5 DNSSEC Extensions

The DNSSEC extensions[8] are an attempt to remove, or at least limit, the security weaknesses of the DNS protocol that we showed above. These extensions are provided by a Working Group (*WG*) formed in 1994 by the IETF and are designed to be interoperable with non-security aware implementations of the DNS protocol.

2.5.1 Objectives

Their primary goal is to provide authentication and integrity for the data held into the DNS database and they achieve this via digital signature schemes based on public key cryptography (See 1.3.2). Since this data is considered public data, there is no need

for access control and confidentiality. Although there is no mechanism to provide access control and confidentiality, the DNSSEC WG did not eliminate the ability to provide support for it. The WG defined a new set of RRs to hold the security information needed to the DNS security. This new set is designed to be extensible and each application can decide to use the information contained in this new set as they wish, in particular to provide confidentiality. Thus the DNS can be become a worldwide public key distribution mechanism (PKI). Moreover, issues such as cryptographic export should not be solved worldwide, so the DNS provides means to have multiple keys for different cryptographic algorithm thus allieviating this problem. Summarizing the DNSSEC extensions provide three basic services:

- Key Distribution
- Data Origin Authentication
- Transaction and Request Authentication

Key distribution service

The DNSSEC extensions make use of a KEY RR to verify the authenticity of the DNS zone data. This KEY RR is not limited to the DNSSEC scope, but can be used for different purposes since the key distribution service supports several types of keys and keys algorithms.

Data origin authentication

This is the crux of the DNSSEC design since this service tries to mitigate threats such as cache poisoning. The RRsets within a zone are cryptographically signed so that resolvers and name servers can trust the data received as response.

Transaction and request authentication

With this service name servers and resolvers can be sure that the received data is in response to the original query and that it comes from the server they queried to. The signature of the RRsets in a response is produced by concatenating the query and the response itself. In that way security aware servers can verify the transation. Also, with this service, it is possible to provide a secure mechanism for DNS Dynamic Updates.

2.5.2 Performance Considerations

The performance of these new extensions is an important issues and several aspects of DNSSEC are targeted to alleviate the overhead with processing them. To verify the signature of an RR just retrieved, a new query should be issued to retrieve the signature itself and the key needed to verify it. This is not the most efficient way to retrieve the signature so, whenever it is possible, this query is avoided by putting the required keys and signatures for the verification togheter with the retrieved RRset. The basic idea of the DNSSEC extension is to sign a DNS message in its entireness by means of a public-key scheme. But this may not be practical on a large scale. Thus they employed a more practical scheme by signing each RRset as described in[8]. The process of signing each RRset (a zone) is done off-line so no signature has to be computed when a response has to be sent.

2.6 Sk-DNSSEC

In this section we will analise our alternative to the DNSSEC approach to the problem of the weakness of the DNS protocol. Our alternative is based on a symmetric key scheme so we will refer to it as Sk-DNSSEC[12]. From now on we will refer to the IETF DNSSEC approach as Pk-DNSSEC. We will also take a look to Sk-DNSSEC implementation issues and we will make some performance considerations.

2.6.1 Goals

As stated above Pk-DNSSEC provides three basic services:

- Key distribution
- Data origin authentication
- Transaction and request authentication

Sk-DNSSEC will provide the same services but keeping an eye on speed and space performances. Also we implemented Sk-DNSSEC in such a way that it will be fully interoperable with Pk-DNSSEC (See Section 2.9).

2.6.2 Symmetric Certificates

$$P_{XY} = Info(P_{XY}), E_{K^1_{XY}}(K_{XY}, MAC_{K^2_{XY}}(Info(P_{XY}, K_{XY}))$$

$Info(P_{XY})$ contains information about the certificate
$K_{XY} = (K^1_{XY}, K^2_{XY})$ is the secret-key pair shared by X and Y
K^1_{XY} is the key used for encryption
K^2_{XY} is the key used for symmetric signature (MAC)

Figure 2.2: The *Symmetric Certificate* shared between name servers X and Y.

We introduce now the concept of a **symmetric certificate** P_{XY} that will be used to assure that a certain answer is to be trusted. Each query to an authoritative name server will contain a $Nonce$ (a random number that helps preventing several kind of attacks, such as replay attacks) compound by a random value and a timestamp. Moreover, along with each query will be sent, besides the nonce, a *symmetric certificate*. This can be viewed like a locked envelope containing informations shared between two consecutive DNS name servers needed by each of them to add another link to the chain of trust from the root name server down to the final authoritative name server. This symmetric certificate is sent to the querying name server along with the answer indicating the new name server to ask to and the inquirer should send it as is to this new name server.

The symmetric certificate contains informations about the symmetric certificate itself, such as the identities of both the peers and the creator of the certificate, inception and expiration dates, informations about the encryption and authentication algorithms and so on and the keys the queried server should use to sign the response (See Figure 2.2 and Section 2.7.1.) In the next section we will see in details how the symmetric certificates are used to ensure authenticity and integrity of a DNS message.

2.6.3 The Protocol

Let us take a fast look on how the Sk-DNSSEC protocol works. We assume that each name server of the DNS tree has a *master_key* $K_{R_X R_Y}$, a symmetric key that is shared between itself (R_X) and his slave name server R_Y (one key for each slave name server.) For example the node `.org` will share a *master_key* with the slave name server of the

`.isc.org` domain. The root name server has also an asymmetric key pair (*private_key*[1]) besides its *master_key*. This root *private_key* will be used to start the chain of trust.

Let see what happens when a name server U (the resolver) is queried, by a stub resolver, to translate a host name (suppose someone wants to connect to `www.isc.org`) in its IP address. We suppose that the resolver is not authoritative for that domain, so it will begin by querying the root name server R_0:

$$R_0 \leftarrow U : P_{R_0U}, DNS\_Query, Nonce_0$$

The root name server is not authoritative for that domain thus will refer the resolver to R_1 the name server of the `.org` domain. The root name server R_0 will generate a new symmetric certificate P_{R_1U} and will send it to the resolver along with the normal DNS response indicating the new name server R_1 to query to:

$$U \leftarrow R_0 : P_{R_1U}, DNS\_Ans, E_{K^1_{R_0U}}(K_{R_1U}, MAC_{K^2_{R_0U}}(DNS\_Ans, Nonce_0, K_{R_1U}))$$

The new symmetric certificate P_{R_1U} is encrypted with the *master_key* $K_{R_0R_1}$, shared between the root name server and the `.org` name server R_1, and contains a new pair of symmetric keys K_{R_1U} that will be used by the new name server to encrypt and sign the response that in its turn it will send to the resolver U (when this will query it):

$$P_{R_1U} = Info(P_{R_1U}), E_{K^1_{R_0R_1}}(K_{R_1U}, MAC_{K^2_{R_0R_1}}(Info(P_{R_1U}), K_{R_1U}))$$

Moreover, along with the response and the symmetric certificate, the root name server will send the keys K_{R_1U} to the resolver U, encrypted and signed with the key pair K_{R_0U} shared between itself and the resolver (see below.)

With this mechanism, the root name server will agree both the resolver U and the name server R_1 about a symmetric key pair K_{R_1U}, so that both can communicate over a secure channel and also authenticate themselves. The resolver U will find the keys in the response from the root name server while the name server R_1 will find them into the symmetric certificate P_{R_1U}. We stress that the root name server will send the symmetric certificate to the resolver and not to the `.org` name server. Is the resolver itself that will send it to the `.org` name server along with the query, as we are going to show.

When the resolver receives the response from the root name server it will check the signature and, if it is valid, decrypt the message to obtain the key pair K_{R_1U} that it has

[1]Note that here we are indicating the key pair, not only the private key.

to use to verify the response from the name server (the `.org` one) it is going to query to. This new query will carry the symmetric certificate, as is, just received from the root name server along with a nonce $Nonce_1$ in order to prevent attacks such as replay attacks. In general, for any $0 \leq i \leq n$, the query from the resolver U to the generic name server R_i is:

$$R_i \leftarrow U : P_{R_iU}, DNS\_Query, Nonce_i$$

while the response from the name server R_i to the resolver U is:

$$R_i \rightarrow U : P_{R_{i+1}U}, DNS\_Ans_i, E_{K^1_{R_iU}}(K_{R_{i+1}U}, MAC_{K^2_{R_iU}}(DNS\_Ans_i, Nonce_i, K_{R_{i+1}U}))$$

where:

$$P_{R_{i+1}U} = Info(P_{R_{i+1}U}), E_{K^1_{R_iR_{i+1}}}(K_{R_{i+1}U},$$

$$MAC_{K^2_{R_iR_{i+1}}}(Info(P_{R_{i+1}U}), K_{R_{i+1}U}))$$

The seek for the final answer will continue in this way until the authoritative name server is found. The response from the last server R_n will not carry any new symmetric certificate, since no new server need to be queried:

$$R_n \leftarrow U : P_{R_nU}, DNS\_Query, Nonce_n$$

$$R_n \rightarrow U : DNS\_Ans_n, MAC_{K^2_{R_nU}}(DNS\_Ans_n, Nonce_n)$$

The key pair shared between the root name server R_0 and a generic resolver U are kept only by the resolvers themself. The root name server has no need to store them (thus saving a lot of space) since the keys are sent to it by the resolvers, stored into a symmetric certificate called the *root certificate* P_{R_0U}. This certificate is requested by the resolver the first time it contacts the root name server by using a *DNS_RootCert_Req* request:

$$R_0 \leftarrow U : PE_{R_0}(PH, K_1, K_2, DNS\_RootCert\_Req)$$

obtaining the following as response:

$$R_0 \rightarrow U : P_{R_0U}, E_{K_1}(K_{R_0U}, MAC_{K_2}(K_{R_0U}, P_{R_0U}))$$

This request is encrypted with the root name server's public key along with a temporary key pair K_1 for the encryption and K_2 for the MAC and a protocol header PH which

should minimally contain the identities of both the resolver and the root name server, life time of the encryption and a $Nonce$.

When the root server receives such a request it will decrypt it, using its own private key, get the key pair K_1, K_2 from it and generate the *root certificate* P_{R_0U} that will be sent to the resolver along with a newly generated key pair K_{R_0U} encrypted and signed with the temporary key pair K_1, K_2. This new key pair is the key pair shared between the root name server and the resolver and, as said before, are stored only by the resolver along with the *root certificate*. Of course the *root certificate* is encrypted with the root own private key that is not shared with anyone else.

If mutual authentication and protection for DNS requests are needed then, for any $0 \leq i \leq n$, the first message

$$R_i \leftarrow U : P_{R_iU}, DNS\_Query, Nonce_i$$

becomes

$$R_i \leftarrow U : P_{R_iU}, DNS\_Query, Nonce_i, MAC_{K^2_{R_iU}}(DNS\_Query, Nonce_i)$$

Obviously the server U should autenticate itself first. This needs to be done only once, for instance together with the request for the DNS root symmetric certificate. The server U should sign the public-key encryption (the first message) by computing:

$$SIGN_U(PH_1, PE_{R_0}(PH, K_1, K_2, DNS\_RootCert\_Req))$$

where PH_1 is a protocol header similar to PH which also contains the sentence clearly stating that the signature is computed over an encryption in accordance with the DNS protocol. The public-key of U may be embedded in a certificate signed by some certification authority recognized by the root server. Finally, the root will set an appropriate flag inside the symmetric certificate to inform other nodes that the resolver was authenticated. This entire task can be performed by any downward name server and not necessarily by the root.

The last step of the resolving process, when the resolver U has to send the final answer back to the stub resolver, could be securely performed via the TSIG[11] mechanism.

2.7 Implementation Issues

In the next sections we will show how we integrated the Sk-DNSSEC protocol in the ISC's BIND implementation of the DNS protocol. First we will show the format of the Sk-DNSSEC messages (queries and responses) exchanged between DNS name servers.

Each Sk-DNSSEC query from a generic resolver U to a generic name server R_i will be sent in the following format:

	about 94 *bytes*	12 *bytes*
DNS_Query	P_{R_iU}	$Nonce_i$

where *DNS_Query* is the original, untouched DNS query in the format described by [2], P_{R_iU} is the symmetric certificate in the format we are going to show and $Nonce_i$ is, of course, the nonce. Each response from a server to the resolver will be sent in the following format:

	about 94 *bytes*	$6 + ENCKey_{Len} + MACKey_{Len}$	*usually* 20 *bytes*
DNS_Ans	$P_{R_{i+1}U}$	$Key\ K_{R_{i+1}U}$	$Signature$

where, as shown in Section 2.6.3, both the key $K_{R_{i+1}U}$ and the $Signature$ are encrypted with the key K_{R_iU} shared between R_i and U.

As you can see, actually, we will attach our messages at the end of a DNS message (query or response.) Next implementations of the Sk-DNSSEC protocol will an abstract Resource Record (RR) like the TSIG one.

2.7.1 The format of a symmetric certificate

As stated in the previous sections, the symmetric certificate is the core of the Sk-DNSSEC protocol. It is very short in length (about 94 bytes) and it consists of information about itself (the $Info$ field, see 2.7.2), the keys that two name servers U and R_{i+1} will share to securely communicate (the Key field, see 2.7.3) and a signature from the parent name server stating that the certificate is valid. The format is as follows: at the beginning there is the $Info$ followed by the encryption, under the key $K^1_{R_iR_{i+1}}$ shared between the servers R_i and R_{i+1}, of both the Key field and the $Signature$:

34 *bytes*	$(6 + ENCKey_{Len} + MACKey_{Len})$ *bytes*	*usually* 20 *bytes*
$Info(P_{R_{i+1}U})$	$Key\ K_{R_{i+1}U}$	$Signature$

2.7.2 The format of the $Info$ field

The $Info$ field contains the identifications of the name servers from the creator of the certificate to the server whom it is destinated to, all the informations needed to properly decrypt and verify it and the inception and expiration dates. Three are the name servers involved during the life time of a certificate: the creator of the certificate itself (i.e. the parent name server R_i), whom the certificate is destinated to (i.e. the delegated name server R_{i+1}) and the resolver U. We will use their IP addresses to identify each of them. The IP address of the network interface the resolver U sent its query from will be used to identify it; the parent name server will use the IP address of the network interface it got the query from as its ID, while the IP address of the delegated server will be taken from the NS RDATA of the DNS answer. We call the creator's ID the *CreatorID*, the resolver's ID the *SenderID* and the delegated name server's one the *ReceiverID*. The inception and expiration dates are in seconds since the epoch. The fields ENC_{Alg} and MAC_{Alg} specify the encryption and digest algorithms respectively, while ENC_{Len} and MAC_{Len} their length in bytes. The $Info$ format is the following:

Bytes	*Field Description*
4	$CreatorID$
4	$SenderID$
4	$ReceiverID$
4	*Inception Date*
4	*Expiration Date*
1	ENC_{Alg}
2	ENC_{Len}
1	MAC_{Alg}
2	MAC_{Len}
4	$CertificateID$
4	$KeyID$

2.7.3 The format of the Key field

A Key field is divided in two parts: one for the encryption key and one for the digest key. Both the first parts have the same format: one byte indicating the algorithm the

key refers to and two bytes with the key length followed by the keys themselves. Actually Sk-DNSSEC only supports the *Blowfish* encryption algorithm and the *HMACMD5* digest algorithm, but it is very easy to support other kind of algorithms. The Key format is as follows:

$Bytes$	$Field\ Name$
1	ENC_{Alg}
2	$ENCKey_{Len}$
$ENCKey_{Len}$	$ENC\ Key$
1	MAC_{Alg}
2	$MACKey_{Len}$
$MACKey_{Len}$	$MAC\ Key$

2.7.4 The format of the $Nonce$ field

When a query from the resolver U is sent, a nonce is attached to it. The nonce is used as a defense against attacks such as replay attacks and it is compound by a random value *Randval* and a timestamp as follow:

$Bytes$	
4	$Randval$
4	$Seconds$
4	$Nanoseconds$

2.7.5 The $DNS\_RootCert\_Req$ request format

When the resolver U contacts for the first time a root name server, or when the cached certificate is expired, a new chain must be built starting from the root domain name server. Indeed, in the case the certificate expired, we could restart the chain from the last valid certificate, but, as the resolver has only a locally restricted view of the entire DNS tree, we have no way to know which is the last valid certificate. Moreover, these certificate are going to expire soon as well. If this is the first time we contact the root domain name server a *DNS_RootCert_Req* will be made in order to obtain its *root_certificate*. The format of the *DNS_RootCert_Req* is the following:

PH	K_1	K_2

where the Public Header PH has the following format:

4 *bytes*	4 *bytes*	8 *bytes*	12 *bytes*
U_{Addr}	$Root_{Addr}$	TTL	$Nonce_0$

All the *DNS_RootCert_Req* is encrypted using the public key of the root domain name server. The meaning of the fields U_{Addr}, $Root_{Addr}$, TTL and $Nonce_0$ are simple to understand. The key pair (K_1, K_2) is a temporary key pair (K_1 for the encryption and K_2 for the digest) that the root name server must use in order to encrypt and sign the response:

P_{R_0U}	K_{R_0U}	MAC_{K_2}

where both K_{R_0U} and MAC_{K^2} are encrypted with K_1.

2.8 Implementation

The most widely used implementation of the DNS protocol is the *BIND* (*Berkeley Internet Name Domain*) implementation from the non-for-profit ISC corporation (*Internet Software Consortium*).

The DNS can be split into two major components:

- a **Domain Name System** server (`named`)
- a **Domain Name system** client (the resolver library)

The *BIND* implements these two components[2] providing tools for verifying the proper operation of the DNS server. The DNS server daemon `named` listens for queries from resolvers. Resolvers can be stub resolvers (like the DNS resolver library provided with *BIND*), or intelligent resolvers (like the DNS server itself when acting as a resolver on behalf of a stub resolver or the `dig` utility).

We choose the *BIND* implementation of the DNS protocol to implement our protocol into because of its wide use around the world and because it is one of the best available implementation of the DNS protocol itself. Moverover, and most important, actually the BIND is the only one to implement Pk-DNSSEC. This is important because we want

[2]Indeed, the resolver library is implemented in the `libc` library, but the new BIND versions includes a new *Light Weight resolver library* that should substitute it.

to make Sk-DNSSEC be fully interoperable with Pk-DNSSEC [12] (See also Section 2.9 below).

The code is very well written and it is organized in several different modules:

- The *DNS tools* (`named`, `dig`, ...)
- The *DNS library*
- The *ISC library*
- The *Light Weight stub resolver library*
- The *OMAPI library*

Each of the tools provided (among which there are the `named` server and the `dig` utility) will be linked against the library they need to use. For example the `dig` executable will be linked against the *DNS library* as well as the *ISC library*.

The **DNS library** is the real DNS protocol implementation. It implements query lookups, message parsing functions as well as RRsets handling functions and the DNSSEC extensions. These functions are called by top level applications such as `named`, `dig` and so on.

The **ISC library** implements a lot of functions to handle basic objects like *buffers*, *log*, *threads* and *tasks* handling functions. These functions are used by all other entities that compose the *BIND*, such as the *DNS library*, the *Light Weight stub resolver library* and so on.

The **Light Weight stub resolver library** implements a lookup service that should be used by the new generation of stub resolvers (like the one into the *libc* library). It will forward the lookup requests to a local resolver daemon (`lwresd`) that will perform the lookups on behalf of the caller and will send, through the library, the responses back to the application.

The **OMAPI library**, implements the command/control API protocol.

We will not care about the *Light Weight stub resolver library* or about the *OMAPI library* since our protocol is an addition to the basic DNS protocol. Thus we need to focus our attention on the *DNS library* and on the *DNS applications* that make use of it, in particular we need to take a look at the client and server side of each application we need to modify. We will make use of the powerful tools provided by the *ISC library*.

2.8.1 The DNS Library

We decided to use a bottom-up approach to implement our code. First we started looking at the basic objects we are dealing with: the *nonces*, the *keys*, the *info header* and the *public header*; then we implemented a set of API to manipulate each of them, thus facilitating the implementation of complex objects like the *symmetric certificates*. We added the implementations of these APIs to the *DNS Library*.

Each object is represented by a struct. A set of function to create, set, read and destroy them have been implemented. As an example we will show the C struct for the $Nonce$:

```
struct sk_nonce {
        isc_uint32_t     randval;
        isc_time_t       timestamp;
        dns_messageid_t  query_id;
        isc_mem_t        *mctx;
};
```

A pair of function to write the struct to and read it from a buffer have been implemented too. These are needed since all these objects are sent on and received from the network. Some of the objects (nonces, certificates and keys) needs to be stored and retrieved at a later time. For example, the $Nonce$ is sent along with a query and it needs to be checked when the response for this query is received. Since the resolver code does not passively wait for the answer, but, instead, in the meanwhile it will handle other queries and responses, we needed to bind the $Nonce$ to the query, store it and retrieve it when the query it is bound to receives a response. A list indexed by IP addresses has been used to store and retrieve these kind of objects. The IP address used as index is the IP address of the name server the resolver U is talking with. Thus, as an example, the function to create a nonce is `sk_nonce_new` while the function to bound it to a $QueryID$ is `sk_nonce_setid`, the function to read it from the received buffer is `sk_nonce_fromtext` and the one to retrieve it, after a response is received, is `sk_nonce_get (server_addr)` and so on.

2.8.2 The NAMED daemon

The core of the *BIND* DNS implementation is the `named` daemon. Each node in a DNS tree runs this daemon that will wait for queries, look for answers and send back responses as well as transfer zones. Each node has its own configuration, i.e. its own zones for which it is authoritative for and its own list of name servers to delegate queries to. The file `named.conf` describes the `named` configuration. For each zone a different file (such as `zonename.conf`) is used to describe the zone itself (i.e. the core association between FQDNs and IPs, its viceversa, and so on ...).

When the named daemon is launched it reads the named.conf file and each, if any, of the zone files it is authoritative for, then it sits down and waits for queries from remote or local clients. When a query is received it performs a number of checks to see if the query if well formed, according to the DNS protocol, and then starts looking for a response. If it does not hold any answer for this query, it will perform a recursive or iterative query (according to the named.conf configuration file) looking for a name server that knows the answer (see Section 2.6.3). We will now take a look

```
/*
 * Handle an incoming request event from the dispatch (UDP case)
 * or tcpmsg (TCP case).
 */
static void
client_request(isc_task_t *task, isc_event_t *event) {
        ns_client_t *client;
        isc_buffer_t *buffer;

        ...
        client = event->ev_arg;
        buffer = &event->buffer;
        ...
        result = dns_message_parse (client->message, buffer, 0);
        ...

        /*
         * We expect a symmetric certificate to verify and a nonce.
         * Save the nonce.
         */
         sk_dnssec_log ("Checking for SK_DNSSEC...");
         result = sk_query_verify (buffer, client->message->id, client->mctx);
         if (result != ISC_R_SUCCESS) {
                 ns_client_error(client, result);
                 goto cleanup_serverlock;
         }

 ...
}
```

Figure 2.3: Checking a query

on where to put entry points for our code into the named source code. First we will consider the point of view of a DNS name server acting as a server, next we will consider the point of view of a DNS name server acting as a client on behalf of a stub resolver. The named daemon can be seen as compound by three main components. We will call the *client side* of the named daemon, the piece of code that will handle incoming queries and outgoing responses; *resolver* the piece of code that will handle the seek of an answer to a query and *server side* the piece of code that will handle outgoing queries on behalf of the stub resolver. Clearly we need to put entry points for our code in these three main components of the named daemon. When the named daemon is acting as a server, we need to:

- check if a query carries a *symmetric certificate* or a *DNS_RootCert_Req*

- look for a *symmetric certificates* to put in a message

Checking if a query carries a *symmetric certificate* is done into the client side of the `named` daemon (See Fig 2.3). If the query contains such a certificate then the client is *Sk-DNSSEC*-aware and we need to jump on our code to verify the query. Once that the query and the certificate are been verified, the daemon will seek for an answer. If it does not have an answer for this query it should

```
void
sk_answer_sign (sk_ns_client_t *client, isc_buffer_t *dest)
{
        ...
        cert = sk_cert_serverget (&sender_addr,
                &receiver_addr);
        if (!cert) {
                ...
                result = sk_cert_new (&sender_addr, &receiver_addr,
                        &creator_addr, keysR1_U, keysR0_R1, &cert, mctx);
        }

        ...
        result = sk_answer_totext (dest, cert, nonce, keysR0_U, mctx);
        ...
}
```

Figure 2.4: Signing an answer

at least know the next name server to query to, and, in this case, we need to put the *symmetric certificate* for this server into the response. If such a certificate does not exist then it must be created (See Section 2.6.3 and refer to Fig 2.4). This job is done into the resolver. The client side will then put this new certificate into the response and will send it back to the inquirer.

When the `named` daemon is acting as a client on behalf of a resolver, we need to:

- check if a response carries a *symmetric certificate* (when the queried name server does not have an answer but knows a better place where to look in).
- save the certificate
- look for right certificate when querying a name server

The server side is in charge of checking if a response, to an its own previous query, carries a certificate. If so it will check if the signature is valid, and in that case, save this new certificate. Also, along with the certificate, a key pair (K_1, K_2) is sent. This key pair is needed by the server to verify the signature of the next name server (see Section 2.6.3) and so it need to be saved, too (See Fig. 2.5).

When the server side of the `named` daemon is asked to send a new query to a name server (usually this is done when acting as a client on behalf of a stub resolver) it will look for the certificate bound to it and will put it along with the query. If no such a certificate is available, then it needs to make a *DNS_RootCert_Req* (See Fig 2.6.) We implemented two lists for storing

```
static void
resquery_response(isc_task_t *task, isc_event_t *event) {
        ...
        result = dns_message_parse(message, &devent->buffer, 0);
        ...
        result = sk_answer_verify (&devent->buffer, &addr, message->id, query->mctx);
        ...
}

isc_result_t
sk_answer_verify (isc_buffer_t *buffer, isc_netaddr_t *server,
        dns_messageid_t query_id, isc_mem_t *mctx)
{
        ...verify the signature...

        result = sk_cert_add (cert, receiver_addr);
        ...
        result = sk_server_addkeys (receiver_addr, &keysR1_U);
        ...
}
```

Figure 2.5: Verifying a response

```
isc_result_t
sk_query_put (isc_mem_t *mctx, isc_netaddr_t *server_addr,
                isc_buffer_t *buffer, dns_messageid_t query_id)
{
        ...
        cert = sk_cert_get (server_addr);
        if (!cert) {
                /* DNS_Root_Cert_Req */
                result = sk_request_totext (mctx, nonce, server_addr, buffer);
        } else {
                result = sk_query_totext (buffer, cert, nonce);
        }
        ...
}
```

Figure 2.6: Making of a query

the keys and the certificates. The former is indexed by the IP address of the peer. The latter is indexed by the IP address of both the peers. This happens because a key stored in a name server is shared between itself and another name server. Thus, we need only one IP address to index the key list. Instead, the *symmetric certificate* stored in a name server is not shared between this name server and another one, but between two consecutive name servers (usually different from the name server that is currently storing it), thus we need the IP addresses of both the name servers to correctly index the *symmetric certificate* list.

2.9 Analisys

We believe that our alternative to the solution to the weaknesses of the DNS system addresses some interesting issues:

Performance. The use of symmetric signature in Sk-DNSSEC makes it very efficient in verifying them, compared to Pk-DNSSEC, where the use of RSA to sign a response dramatically reduce performances.

Network traffic. Pk-DNSSEC will sign each RRset in a response. Thus increasing a lot the size of a DNS response (that, also, will not fit into the small UDP packets usually used by the DNS). Instead, Sk-DNSSEC will sign the entire message thus reducing the amount of data to be sent along with the response to just a small digest of the message.

Storage. Symmetric certificates are also very short and compared to the Pk-DNSSEC ones, with the same amount of space it is possible to store more certificates. Moreover, there is no need for NXT RRs, thus removing the need of ordering the zones and saving more space.

Replay attacks. In Sk-DNSSEC a signature cannot be reused thanks to the use of a *Nonce*. This is not true in Pk-DNSSEC where the only way to reduce exposure to this type of attacks is to use very close inception and expiration dates. Also the NXT RR is clearly susceptible to this attack, while Sk-DNSSEC is not (as it does not use it.)

Mutual authentication. This can be achieved very efficiently in Sk-DNSSEC since it requires only an additional MAC computation. BIND 8 and 9's access control lists demand mutual authentication to prevent IP spoofing attacks, and without it they are useless.

Confidentiality. By simply putting the query and the response directly into the encryption of Sk-DNSSEC it is possible to achieve confidentiality even if the pricinples on which the DNS is based do not consider it.

2.10 Performances

In the next sections we will try to compare the Sk-DNSSEC overhead with the Pk-DNSSEC one. We will briefly discuss the impacts of the amount of informations, needed by each of them to reach their goals, on network traffic and storage.

2.10.1 Stored information

It is impossible to talk about the exact bytes a DNS message is long, because of the variable length of domain names and number of children of each zone. According to [19] the size of a signed zone in Pk-DNSSEC is augmented of a factor of seven. In Sk-DNSSEC the signature is not pre-computed, but fastly computed on the fly, thus the size of the zone stored on the media does not increase.

Sk-DNSSEC only needs to store approximately 128 bytes (the size of a tipical symmetric key) for the parent and for each child node in the zone file.

2.10.2 Network traffic

There is a big difference in the sizes of messages between Pk-DNSSEC and Sk-DNSSEC as shown in [12]. This is due to the fact that Pk-DNSSEC with SIG RR has to return a public-key signature computed over an entire RRset along with the RRset itself (needed to verify the signature.) Thus, querying for n different types of RRsets would result in n public-key signatures along with the n corresponding RRsets. That means that a tipical Pk-DNSSEC response would not fit in an UDP datagram, thus increasing the number of truncated messages. Instead Sk-DNSSEC would reply in both cases with only one, small, signature, while each referral requires two signatures: one for the symmetric certificate and one for the actual signature. Assuming to use a block cipher with keys 128 bits long and a 128-bit MAC function, the total number of bytes that would be added to a DNS message is about 152 (See Section 2.7) that would perfectly fit into a 512-byte UDP datagram.

2.10.3 Computational Time

The use of symmetric encryption and MAC signatures greatly improves time performances of our implementation. Sk-DNSSEC needs to perform two on-the-fly computations in order to achieve authenticity of the data held in an answer: a symmetric certificate validation and generation, both done by the same server. The first needed when it recevies a Sk-DNSSEC aware query from a name server acting as a resolver, the latter when it needs to refer the resolver to a new name server (in this case it needs to generate a new symmetric certificate.) Both Sk-DNSSEC and Pk-DNSSEC need to add computational overhead in generating a signature (in Pk-DNSSEC this is done off-line, altough, for huge zones, this is **very** expensive) and in its verification (this is done by the resolver when it receives an answer from a server.) As you can imagine, this overhead is greater in Pk-DNSSEC than in Sk-DNSSEC as Pk-DNSSEC makes use of public-key signatures while Sk-DNSSEC uses faster MAC ones.

Summarizing Sk-DNSSEC needs less bytes, either in storing additional informations on a media and in network overhead, than Pk-DNSSEC to achieve the security same goals. Also, computational time is lesser in Sk-DNSSEC than in Pk-DNSSEC. In spite of these facts, Sk-DNSSEC cannot fully substitute Pk-DNSSEC. Whereas zones are static Pk-DNSSEC is more suitable than Sk-DNSSEC as the latter generates fresh signatures each time a query is received. Usually root zones are very static and these name servers are subject to a lot of queries per day. Thus the already signed RRsets of Pk-DNSSEC are more suitable (no additional computational time needed in generating

them.) Instead, whereas zones rapidly change (think also about dynamic updates) Sk-DNSSEC's faster signature generation is more suitable than the slowest counterpart in Pk-DNSSEC.

2.11 Conclusions and future works

As seen in the first sections of this chapter, the Domain Name System was designed without taking any kind of security measures in account. Altough it is one of the core points of the whole Internet, it is very weak and totally insecure. Today a lot of efforts are made in order to make it secure and thus reliable. Actually the biggest effort in securing the DNS is made by the ISC consortium and goes under the name of DNS Security Extensions (DNSSEC.) Check the site `http://www.dnssec.net` if you want to keep yourself up to date on this important topic.

The DNSSEC extensions fully fill up the security gap of the DNS, making it very reliable and secure. They achieve these goals by making use of digital signature schemes based on public-key cryptography. The major drawback of this choice is that a public-key signature is computationally slow and hude in space, thus signing a zone can be really a hard process (both in time and in space) whereas the zones are big and drastic network downgrades can happen whereas name servers receive millions of queries (root servers, for instance.)

We believe that Sk-DNSSEC protocol does not suffer of these drawbacks since it makes use of symmetric cryptography to sign DNS messages, and, since the symmetric keys are very short compared to the public-key ones, it has a little impact on storage and network performances. We stress that the problem in DNSSEC is not that servers cannot compute cryptographic functions fast enough. Usually they are very fast and expensive machines. The real problems in DNSSEC are the potential increase of network traffic due to larger DNS messages and difficulty of cryptographically verifying zone data (at the resolver side.)

An hybrid approach We designed the Sk-DNSSEC protocol to be fully interoperable with the Pk-DNSSEC one. The idea behind this is that root name servers usually handle static zones, where changes appear rarely. In such a scenario our protocol does not fit very well. Generating different signatures for the same data each time is not very performing. Also our protocols requires that root name server to do a public-key decryption at every $DNS\_RootCert\_Req$. Here the Pk-DNSSEC approach is more suitable since the zones are signed only once. Instead in top-level domains changes are frequent and our protocol fits better since the faster computational time and the shorter messages have lesser impact on performances and network traffic. The idea is to let some servers sign answer via public-key signatures which, at a certain point, will include subdomain's public-key used by Sk-DNSSEC to continue the chain of trust, via the protocol used for requesting DNS root certificates. For example a resolver may use the Pk-DNSSEC protocol to obtain the public-key of

the `.jhu.edu` domain name server and use it to make a *DNS_RootCert_Req* request to it thus switching to the Sk-DNSSEC protocol. This mechanism will relieve name servers (particularly root ones) from computing public-key decryptions.

Proactive security Usually a zone can have several name servers authoritative for it. A resolver that receives several name servers authoritative for the same zone has to choose one of them. BIND use the roundtrip time, or RTT, to make the choice. This means that in Sk-DNSSEC we had to make a design choice: how to handle the case that several name servers accept the same symmetric certificate from the parent name server. We have currently adopted a single strategy: the name servers, authoritative for the same zone, will share the same secret with the parent name server. This is not the safest, and indeed it is not safe, solution. We are actually studying proactive techniques to employ in such a scenario. By means of proactive security the secret shared with the parent name server is split in several shares. Given enough time, an attacker may recover enough shares to recompute the secret and start signing false DNS messages. Refreshing the shares by means of proactive techniques becomes an important task.

A threshold, intrusion tolerant signature scheme Both the two protocols, the Pk-DNSSEC one and our Sk-DNSSEC alternative, assure that queries and answers are correctly received and verified by the entities involved during a seek for an authoritative name server. But nothing can be done by both protocols when secret keys are stolen. This usually happens when a name server is fully compromised (when an attacker gains access to the host and to its secrets). Whatever countermisure is taken, if a secret key (doesn't matter if symmetric or asymmetric) is stolen, we lose. Try to imagine what could happen is an attacker is able to penetrate and steal the root name server's *private_key*. Both protocols will become useless. To mitigate the effects of such an event, one may think to use short term secret, by changing it frequently. But as soon as the new secret is stored on the penetrated host it is lost again.

We are investigating the use of *intrusion tolerant techniques* to make very difficult for an attacker stealing the root's *private_key*. Usually in a local area network there is more than one name server. This way, if one or more of them is temporarily unavailable (because it is down for manteinance or crash), the DNS service will continue to be available for the hosts in the local domain. These backup name servers could be used to build an intrusion tolerant system. The idea is to split the *private_key* among the backup name servers and to make them active in signing an answer to a client. We want that at least a chosen minimum number t of them must be on-line in order to make one of them able to sign an answer. This way an attacker must be able to fully compromise at least t name servers in order to be able to diffuse false, but trustworthy, information. Even fully compromising $t-1$ name servers will give no information, about the *private_key* to the attacker. Moreover each secret data given to each of the name servers should have a time to live.

Thus, the attacker must also be very fast (if the time to live is short) in compromising the t name servers.

Indeed a threshold, intrusion tolerant signature scheme would be very slow, and thus it is unsuitable whereas we need to respond to a lot of queries.

Part III

File System Security

Chapter 3

Network Filesystems

3.1 NFS

NFS, or the Network File System, was originally developed by Sun Microsystems in the 1980's as a way to create a file system on diskless clients. NFS provides remote access to shared file systems across networks. This means that a file system may actually be sitting on machine A, but machine B can mount that filesystem and it will look to the users on machine B like the file system resides on the local machine. In this way NFS is transparent to the user. NFS was also designed to be machine, operating system, network architecture, and transport protocol independent.

The primary functions of NFS are to export or mount directories to other machines, either on or off a local network. These diretories can then be accessed as though they were local. NFS uses a client/server architecture and consists of a client program, a server program, and a protocol used to communicate between the two.

The server program makes filesystems available for access by other machines via a process called exporting. File systems that are available for access across the network are often referred to as shared file systems.

NFS clients access shared file systems mounting them from an NFS server machine. When a file system is mounted, it is integrated into the directory tree.

The NFS mount protocol facilitates the functions that allow NFS clients to attach remote directory trees to a mount point in the local file system. A mount point is an empty directory or subdirectory, created as place to attache a remote file system. In order to mount a file system from an NFS server, a user needs an account on the machine where the file system resides. The NFS client passes the UID and GID of the process requesting the mount to the NFS server. The server then validates the request. Mount protocol also allows the server to grant remote access privileges to a restricted set of clients via export control.

There are currently three versions of NFS. The default version for most workstations is NFS2, although IRIX 6.2 and Soloris 2.5.1 support NFS3. NFS4 is currently being designed.

3.1.1 Weaknesses

Even if NFS was designed, like many other network protocols, without security in mind (or at least with a very basic idea of it), it is one of the most widely used network filesystem. NFS assues a very strong trust model: the user trusts both the file server and the network connecting his host to the file server. Administrators of the file server can easily access user's data without this even noticing it. Also, any local area network is very easy to tap. Moreover it is very easy to imporsonificate other users in NFS by simply modifying UID and GID into the request sent by the NFS client to the server, while this is travelling thru the network.

3.2 Secure Network Filesystems

3.2.1 CFS

The need for pushing cryptography at the operating system level has been advocated by M. Blaze[33] that proposed the Cryptographyc File System. In the CFS, the user can associate a password to a secure directory. A secure directory is created by using a ad-hoc tool and need to be attached to a special directory (`/crypt`) before its files can be accessed. The user need to supply the password relative to a directory each time the directory is attached. Once that is done, the user can access the content of his files. CFS suffers of the following minor drawbacks:

- Each directory is associated with a different key. This makes key management from the user's side quite cumbersome.
- Once a directory is encrypted, all of its files are encrypted too. This is not always necessary.
- CFS keeps all the vital management informations relative to the directory in special hidden files in the directory itself.

3.2.2 TCFS

Our Transparent Cryptographyc File Systems improves Matt Blaze's CFS by providing a deeper integration between the encryption service and the file system that results in a complete transparency by the user's point of view. TCFS was designed to provide a robust mechanism to securely share files at the lowest possible cost to the user. We tried to guarantee that secure files should not be readable

- by users other than the legitimate owner

- by tapping the commmunication lines between the user and the remote file system server
- by the superuser of the file system server

Also we required that all sensitive meta data should be hidden. This means that for each file, not only its content, but also its filename is encrypted. Moreover on the same file system space should be possibile for the users to easily distinguish private data from scratch data such as temporary data that do not require encryption.

3.3 Secure file sharing of encrypted files

In this section we describe the implementation of a basic secure file sharing facility for the Linux kernel and discuss its applications to distributed cryptographic file-systems.

Deciding which component of a distributed system is to be trusted is a crucial architectural and design decision in the development of a secure system. Assuming the existence of a trusted party greatly simplifies the design of secure systems and applications as the sensitive operations can be delegated to the trusted party.

Unfortunately, it is often the case that no party can be completely trusted as it is very difficult to assess whether, for example, a given host that is part of a distributed system can be corrupted or penetrated by an adversary. At the light of this observation, one possible solution is to share trust among n entities in such a way that a successful adversary must be able to penetrate several entities at once.

The approach of sharing trust among the several components of a distributed system has been first suggested by Deswarte, Blain and Fabre [24] that outlined an architecture for a cryptographic file system in which file keys are distributed across several servers. Following the lead of [24], we present here a simple file sharing mechanism for the Linux kernel. The main motivation of our research is to show that a distributed architecture based on sharing of keys is practical and that the delay is still acceptable.

For the sake of concreteness, in this paper we focus on the specific case of cryptographic distributed file-system. Our current implementation of this general principle is to be considered a first step toward a general integration in the Linux kernel of the principle of sharing trust among the several components of a distributed system.

3.3.1 Sharing in cryptographic distributed file-system

A distributed file system is an efficient and convenient way to share a file-system across a network [27, 25, 28]. Starting from the original proposal of a distributed file system, several issues regarding distributed file systems have been raised and addressed (fault tolerance, load sharing, ability to

stand network outages and others) and recently, motivated by security considerations, several proposals for cryptographic distributed file-systems have been put forth [20, 26, 21, 30, 28].

In this paper, we propose the concept of a *secure file sharing service*, discuss its applications to distributed and cryptographic file systems and describe its integration and implementation with the Linux kernel. A secure file sharing service allows the creator of a file to specify a set of legitimate readers (called the *trustees*) and a threshold t. In order to read a file, a trustee must have the approval of at least t other trustee. Moreover, no set of less than t trustee can access the file.

A secure file sharing service has several applications among which we briefly discuss two: key recovery and secure access logging. In a cryptographic file system files are encrypted using a key known to the owner of the file. Providing a mechanism for key recovery is crucial for otherwise a user that has lost his private key is locked out of his files. One solution would be to assume the existence of a trusted party that holds all the keys. If no such trusted party is available (which is often the case), then an alternative way would be to share the key among a set of n trustees in such a way that the original key can be reconstructed by pooling t shares together. The value of t must be chosen by keeping in mind two contrasting factors: large values of t make it difficult for corrupted trustee to pool together their shares and thus get access to the encrypted files of the user; small values of t have the advantage that the key can be reconstructed by the user even if a small numnber of trustees is on-line to provide help. In other words, a secure sharing file service has the effect of creating a virtual distributed trusted party out of n users.

A secure file sharing service is also useful to keep track of file accesses so that abnormal activity of a user can be easily detected. The nayïve approach consists in delegating the logging of the accesses to the file server (as part of the normal system logging activity). This approach works if the file server is local (*i.e.*, the file-system is not distributed) as we can safely assume that the local machine is trusted. For distributed file systems instead we cannot assume that the file server is trusted and thus what is actually logged is up to the goodwill of the superuser. In other words, a dishonest superuser could decide which accesses to log and which not to log. On the other hand, if a file is shared in a secure way then each time the file is read, permission to a quorum of trustees must be asked. Therefore, if the number of dishonest trustees is smaller than t, then at least one honest trustee is asked permission to read the file and the access is then logged.

3.3.2 Threshold cryptography

Recently, the idea of sharing keys among several parties has been recently adopted by Wu, Malkin and Boneh [32] that showed how to construct intrusion tolerant applications. More precisely, they showed how to share the private key of an Apache web server among n hosts so that the private key would not be exposed even in presence of successful penetration of less than a prescribed threshold t of servers. This is achieved by a distributed generation of a shared RSA key [22] which generates

an RSA public key and gives each of the n hosts one share of the corresponding private key. Thus no single hosts has knowledge of the private key and each time the web server is called to produce an RSA signature as part of the SSL handshake it does so by interacting with $t-1$ other hosts. Since the private key is not stored at any of the hosts a successful adversary must penetrate at least t hosts.

Even though our approach is similar to the one of Boneh there are several differences in the scenario and in the tools employed that we briefly discuss in the following. First of all, we observe that the work of [32] is at the application level. That is, they show how to modify specific applications (the ApacheSSl web server and OpenSSL CA) so to achieve intrusion tolerance. The process has to be repeated for any application that is made intrusion tolerant. Instead, we propose an operating system service (placed at the file system level) that can be called transparently by the applications that seek to access private data. In our approach, the web server's private key is stored encrypted at any of the hosts and the encryption key is shared among the n hosts. Each time the private key is needed the applications reads it from the locally mounted file system and the encryption key is recovered by the kernel by interacting with the other hosts in a completely transparent way. Moreover, we point out that [32] employs tools from Threshold Cryptography (in particular, shared RSA key generation) whereas we only use the simpler and more efficient tool of Secret Sharing (see Appendix 1.3.3). This implies that the parties must perform an initialization protocol in which the RSA key is generated and shared. Actually, the protocol must be performed for each different group for which a different RSA key is to be shared. Moreover, each party must keep an amount of information proportional to the number of groups he belongs. On the other hand our approach requires no set-up procedure. It is the file creator that decides the group of trustees for the file and, without any cooperation from the trustees creates the shared file. As a consequence, each trustee needs only to store one private key (i.e., his own private key) independently from the number of different groups he belongs to.

3.4 Syncfiles

We introduce now a basic secure file sharing service for the Linux operative system. This service will help applications in securely share a sensitive resource (tipically a file) among a group of selected users. Indeed, this service will help in limiting the access to the shared resource.

We suppose that the resource is cryptographically encrypted using a *resource_key*. What we do here is to split the key among the users by means of a secret sharing algorithm where the shared secret is the *resource_key*. Each piece of the key is called *share* and basically, it is given to each user of the group.

When a user needs to access the resource he has to contact the users of the group and ask each

of them for his share. Once he has collected the shares he becomes able to recompute the key and thus to access the resource.

The scheme just shown is not fault tolerant: if a user is not on-line, de facto denying access to his share, no one is able to recompute the key. By means of a *threshold* secret sharing algorithm, this scheme becomes fault tolerant: the *resource_key* is split in n shares (where n is the number of the members of the group) but at least $t <= n$ shares are needed to recompute the key, where t is called *threshold*.

3.4.1 The Proposal

Our scenario is a group of users that are working on their own workstation connected to each other by an unsecure network (i.e. the every day scenario of workstations connected to the *Internet*) wishing to share a key secured resource (for example an encrypted file).

A set of userland utilities has been developed to facilitate these users in accomplishing this job. These userland utilities will make use of a secure file sharing service developed in the Linux kernel. This service implements the basic scheme shown above, but strengthening the protocol to avoid the network security flaws. Moreover we will show how, with this protocol, accesses to a secure shared resource can be **logged**.

Moreover this service has been integrated into TCFS.

3.4.2 The Protocol

In this section, we will focus our attention on how to ensure that the shares requested over the unsecure network are securely sent to the requester and how the latter is securely **identified** and his access attempt securely **logged** (or at least his request for a share).

The protocol makes use of *public key cryptography* to assure these properties.

Each share of the *resource_key* will be encrypted with the public key of the member it belongs to and stored along with the resource itself. Thus only the owner of the share can access it by decrypting the share with his own private key. Also, he needs not to be present when a resource becomes securely shared among a group (only his public key needs to be available). Each time a user needs access to the resource he will broadcast a share request along with the encrypted share over the network asking the on-line users to decrypt their own share and to send back to him the decrypted share. Once that at least *threshold* users sent back their shares, the requester can recompute the *resource_keys*. Of course, with the scheme as it is, it is not assured either that the decrypted share will be read only by the requester (a malicious user could easily intercept it over the network), or the identity of the requester itself (and this invalidates the logging). Even more this protocol cannot assure that either the collected shares will be destroyed after the *resource_key* is recomputed or that the *resource_key* itself is destroyed when it is not needed anymore.

To prevent that a decrypted share, in the reply, is intercepted by a malicious user, the protocol has been adapted in the following way. The *resource_key* is recomputed by XORing a secret S with a random sequence R of bytes. Both the secret and the random sequence are then stored along with the resource, both encrypted with the public key of the user they belong to, the same as shown before. The requester will continue to ask for the decrypted share, end the latter will be replied in clear as usual. But now the malicious user cannot recompute the *resource_key* because he lacks the knowledge of the random sequence R. Instead, the requester (that now needs also to be a member of the group) can obtain the random sequence R by simply decrypting his own piece of the key (the secret S **and** the random sequence R of bytes).

Moreover to identify the requester the protocol has been adapted to sign each share request. This is done by computing a hash H of the request and by encrypting H with the private key of the requester. Thus the receivers of the request can verify the signature and be sure of the indentity of the requester.

Also this gives the possibility to make a trustworthy log of the request.

By checking the log of a single user of the group, one can only deduce that a given member of the group as requested a share for a particular file, but has no way to know if this member eventually gained access to the resource. Instead, by crosschecking the logs of other (not necessarily all) members of the group, one can reconstruct who replied to the share request and thus infer if the requester collected enough shares to recompute the *resource_key*.

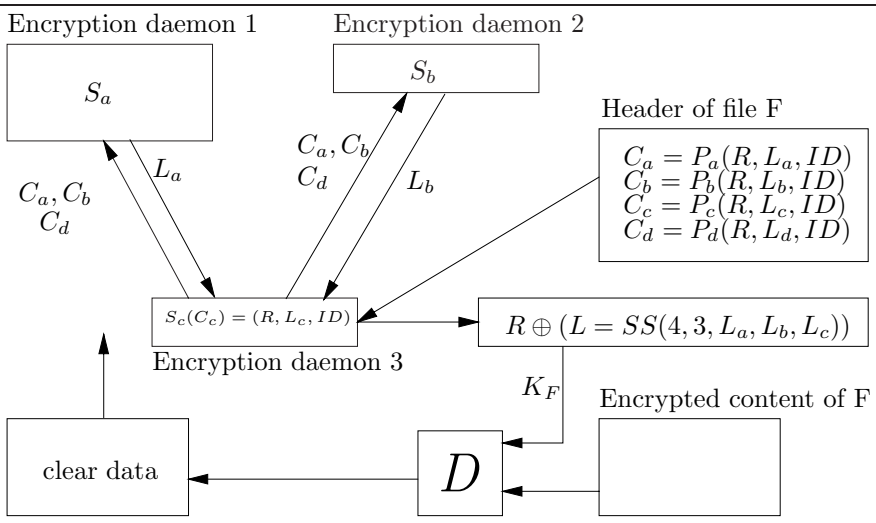

Figure 3.1: An overview of the key retrieving process.

File sharing example

We consider a group of n users each with his own pair of private/public key. We consider three different functionalities for shared files to be implemented: reading a shared file, writing a shared file and helping others in reading a shared file. We wish to enforce the following three properties:

1. a user can create or write a file of threshold $t \leq n$ without the help of the other members of the group; he only needs to know the public keys of the members of the group;
2. it is not feasible to read a shared file with threshold t without contacting at least $t-1$ other users;
3. no user outside the group can gain access to a shared file even if he receives help from all the members of the group.

We denote by $U_1, \cdots, U_n$ a group of n users or trustees and with P_i and S_i the RSA public and private key of U_i, respectively We use RSA only for the sake of concreteness. Our scheme will work with any public-key encryption algorithm. We also denote as $P_i(x)$ ($S_i(x)$) the RSA encryption (decryption) of x computed using P_i (S_i)

The creation of a shared file is obtained in the following way. The creator U of the file picks two random values L and R and computes the *file key* K_F for the file F to be shared as $K_F = L \oplus R$. File F is then encrypted using a symmetric encryption algorithm using K_F as key. Moreover, U picks an identifier ID for the file being created; ID consists of two parts: the security level associated with the file and an identifier of the file. Then, U computes a (n, t) secret sharing of L (see Appendix 1.3.3 for a brief description of secret sharing schemes) obtaining *clear-shares* $L_1, \cdots, L_n$. Finally, for $i = 1, \cdots, n$, U computes the *crypto-shares* of file F for user U_i as $C_i = P_i(R, L_i, ID)$. We stress that no other user is involved in this phase and shares need not to be sent to their owners: they are just stored along with file F (of course the C_i's are not encrypted using K_F).

Reading a file instead is achieved in the following way. We assume that each member of the group knows the address of the hosts at which other users reside (or, as in our implementation, they belong to the same local area network and are reached through a broadcast message). Each of these hosts runs a *decryption daemon* that accepts requests of decryptions of crypto-shares. A user willing to allow access to shared files provides his local decryption daemon with his own private key, a maximum level l^* of security for which decryption of crypto-share is allowed, a list of file identifiers for which decryption is allowed (despite having level larger than l^*) and a list of file identifiers for which decryption is denied (even if their level is lower than l^*). A decryption request (see Figure 3.3) contains one crypto-share C, the date at which the request has been originated, the id H of the key the key with which the share has been encrypted (that is the hash of the key), the public key P of the requester and the signature S of the crypto-share with respect to key P. Upon receiving a decryption request, the daemon performs the following checks:

1. a private key corresponding to the public key with id H is available to be used to decrypt the crypto-share;
2. the date of the request is not older than a fixed amount of time (in our implementation, decryption daemons do not accept requests older than half a second);
3. the signature S is correct;
4. the clear share (R, L, ID) obtained from crypto-share C carries an ID for which decryption is allowed;

If all checks are successful then the value L is encrypted using P and sent back. We stress here that in our scheme each user must keep secret only his decryption key independently of the number of groups he belongs to.

Notice that if the user belongs to the group then he can decrypt his crypto-share obtaining one clear share and the value R. Moreover, if at least $t-1$ clear shares are obtained from other members then the user can reconstruct the value of L and thus obtains the file key K_F. On the other hand, if the user does not belong to the group he can reconstruct L by asking t members of the group but will not be able to obtain R.

3.4.3 Access revokation and key refreshing

Preventing further access to a resource is a bit complicated. Once that a member of the group recomputed the *resource_key* there is no way (except trusting in his goodness) to stop him from storing the *resource_key*, the random sequence R and the user shares. Thus this scheme is not able to revoke to a user the access to a secure shared resource. The only way to solve this problem is to change the *resource_key*. But now a question arises: how often it needs to be changed? and following what policies?

Several strategies are possible according to the level of security desidered:

- the *resource_key* never changes;
- the *resource_key* is refreshed each time the resource has a write access to it (that is own key for each resource revision);
- the *resource_key* is renewed each time the resource is accessed (both for reading and writing)

The problem has been analysed and a conclusion has been reached: that a good tradeoff between performances and security is to change the *resource_key* (and then re-encrypting the resource) at every write access to the resource.

By going thru the following reasonings one can reach the same conclusion. If one would change the *resource_key* at every kind of access (both for reading and writing) maximum security and the

best revokation policy one could think of could be gained. But the price for this is a big reduction in performance as it would be needed to re-encrypt the resource too often.

By changing the *resource_key* when a user is dismissed from a group would give a good revokation policy but a stolen key (for example lost or disclosed by mistake, or not, by a member of the group) would be valid for too much time.

It is now quite obvious that a good solution would be to change the *resource_key* at every write access to the resource. Thus, revoking access to a dismissed member of the group is equivalent to changing the *resource_key* (and splitting it in a new decreased set of shares) and performing a write access (i.e. re-encrypting the resource). Moreover a stolen key will be valid only between two consecutive write accesses to the resource.

3.4.4 Active and passive attacks

As we have pointed out the decryption daemons encrypt their replies to decryption requests with the public key of the requester. This is not prevent users not in the group to access the file but instead is crucial to combat the following *passive* attacks. As it is obvious from the description of our scheme, if an attacker gains access to the private key of any user of the group, then the attacker can try to read any shared file by requesting the help of the other members of the group (the solution of [32] based on threshold cryptography suffers of the same drawback) Actually, there is no way of distinguishing a legitimate member of the group from an attacker that has access to the private key of a member of the group. However, any access performed by the adversary is going to be logged by the other members of the group.

A more subtle attack would be to compromise one private key (which gives the adversary one share of L and the value of R) and then wait for some other user (possibly the same user whose key has been compromised) to ask for $t-1$ other shares. In this way, the adversary has t shares of L (from which L can be reconstructed) and the value of R and can read the file without his activity being recorded. We can say that in this attack the adversary is completely passive.

3.5 Implementation for Linux

In this section we describe our implementation of a secure file sharing service for the Linux operating system. Let us first consider the creation process of a shared file.

The kernel offers a service that we call `MakeSharedFileKey` which receives as inputs the threshold t and the *level* of the shared file to be created (see later for discussion about the level). The `MakeSharedFileKey` returns the header of a shared file which includes the encrypted shares of the file key to be used to encrypt the file. The `MakeSharedFileKey` is invoked by a user application in one of three possible ways. The first and easiest way is via the `procfs` interface: we have created

```
/*
 * writing-key list structure
 */
struct _wkl {
        int                     uid;
        unsigned int            key_id;
        R_RSA_PUBLIC_KEY        *public;
        R_RSA_PRIVATE_KEY       *private;
        pub_list_t              *key_list;
        struct _wkl             *next;
        int                     nofk;
};
```

Figure 3.2: The structure of a `wkl` node.

```
/*
 * Message request structure.
 */
struct _dec_request {
        unsigned int key_id;                    /* hash of the public key */
        unsigned char enc_share[MAX_LENCRYPT];  /* encrypted share */
        unsigned int len_enc_share;             /* length of encrypted share */
        R_RSA_PUBLIC_KEY public;                /* applicant's public key */
        struct timeval date;         /* timestamp of request */
        unsigned char signature[MAX_LENCRYPT];  /* signature of request */
} dec_request_t;
```

Figure 3.3: Structure of a decryption request.

a directory called `SharedFiles` that contains a file for each service related to secure file sharing (see Fig. 3.4).

For the specific case of the `MakeSharedFileKey` we have a file called `MakeFileKey`. Each time a process writes into the file the `MakeSharedFileKey` is executed using the data written into the file as input. The output is then obtained by reading from the file. A second way would be to add an `ioctl` procedure to the underlying file-system. An application that wishes to create a shared file invokes the `ioctl` that, in turns, calls the `MakeSharedFileKey` procedure. Finally, it is possible to define a new device and issue read and write operations on the device to read the output and to pass the input to the procedure.

The `MakeSharedFileKey` procedure uses a local data structure the *writing-key list* (`wkl` in short) that holds the public keys of the trustee of the file (Fig. 3.2). Public keys are added to and deleted from the `wkl` via the `procfs` file-system.

Reading a shared file instead is more complicated. Users wishing to obtain access to a shared file invoke the `GetSharedFileKey` which reads the crypto-shares from the header of the shared file and sends one decryption request for each of them. In our implementation each decryption request is handled by a separate kernel thread (see Figure 3.5). Each time a thread obtains the clear share

```
struct proc_dir_entry *shared_dir, *make_filekey;

/* initialize /proc entries */
int init_shared_procfs (void)
{
        /* create the dir entry into the /proc file system */
        shared_dir = proc_mkdir ("SharedFiles");
        if (!shared_dir) {
                return -ENOMEM;
        }

        /* create the MakeFileKey entry in the /proc/SharedFiles directory */
        make_filekey = create_proc_entry ("MakeFileKey", 0222, shared_dir);
        if (!make_filekey) {
                return -ENOMEM;
        }

        /* this kernel module is the owner of the entries.
         * 'share_proc_makefilekey' will handle writes to this file */
        make_filekey->owner = THIS_MODULE;
        make_filekey->write_proc = share_proc_makefilekey;
        ...
}
```

Figure 3.4: Code fragment for the implementation of the `procfs`.

corresponding to the crypto-share for which it is responsible, a shared counter is incremented. When the counter hits t, then we known that enough clear shares have been obtained and the reconstructing phase can start. Once a sufficient number of decrypted shares has been received, the kernel computes the file key and returns it to the calling user-level application.

Decryption of shares is performed by a kernel-space daemon (`kshared`). Private keys are stored into the *reading key list*, a data structure handled by the `kshared`. Users provide their secret keys to the `kshared` by means of the `procfs` interface as seen above. In addition, user sends to the daemon instructions about *which* requests have to be satisfied on the basis of *file-id* and *level* contained in each request (Fig. 3.3). As we have briefly hinted above, at creation time each file is endowed with a file-id and a level. The level of the file denotes the level of security associated with the file (the higher the level the more confidential the file is considered) and the file-id identifies the file within the security level. The `kshared` daemon is given the maximum security level l for which shares will be decrypted and two lists: one containing identifier of files of level higher than l for which access is to be granted and one containing identifiers of files of level lower than l for which access is to be denied.

Each decryption request has the structure described in Figure 3.3. The `kshared` daemon first checks if he has access to the private key corresponding to the public key whose digest is part of the request. If he does then the share is decrypted and the level and the file-id are obtained. If user has instructed the daemon to allow reading of the file (according to the mechanism outlined

```
int
thread_decrypt_share (void *cryptoshare)
{
        ...
        /* sends decryption requests, waits for the answer and
         * processes received replies */
        ...
}

int
share_get_file_key (struct shared_header *h)
{
        ...
        int threads[num_of_users], share;
        ...

        /* Create a processing thread for each cryptoshare.
         * Put each share in 'args'. */
        threads[share]=kernel_thread (thread_decrypt_share, (void*)args,
               CLONE_SIGHAND);
        ...
}
```

Figure 3.5: Code fragments for the kernel thread that issue decryption requests.

above) the decrypted share is sent back to the requester.

3.5.1 Applications

We have developed a simple suite of applications on top of the kernel services we have introduced to allow users to access to shared files. At this stage, we are mainly interested in showing the viability of our proposal and did not try to offer a complete suite of applications.

1. Application `addrkl` takes on the command line an integer l (that is, the maximum security level of a file for which decryption requests are to be honored) the name of a file containing a pair of RSA (public and secret) keys which are then passed to the kernel that adds the secret key to the `rkl` list. Moreover, the applications takes two optional lists of file ids which are the ids corresponding to files for which reading is allowed (despite their level higher than l) and files for which reading is disallowed (even though their level is smaller than l).

2. Application `addwkl` takes on the command line the name of a file containing a sequence of RSA public keys and adds them to the `wkl` of the users.

3. Application `sharedcat` instead takes on the command line the name of the shared file F and a file containing the user's secret key and asks the kernel to recover the file key K_F used to encrypt F. The kernel extracts from the header on the file the encrypted shares, queries

the remote decryption daemons and reconstructs the file key which is then passed back to `sharedcat`.

4. Application `sharedwrite` takes on the command line the name of a file and constructs a *shared* version of the file. The file is shared using the public keys present at the moment in the `wkl` associated with the user.

3.5.2 Performance Considerations

To test the viability of our approach we have measured the time needed to generate the header of a shared-file (that is, the time to generate the crypto-shares) and the time needed to reconstruct the key used to encrypt the shared file. Thus we have not measured the time it takes to encrypt/decrypt the file as it does not depend on our architecture but only on the efficiency of the encryption algorithm.

We have tested our system for values of $n = 5, 10$ and for values of t from 1 (corresponding to minimum security as any trustee can reconstruct the key by himself) to n (corresponding to maximum security but requiring that all trustees be simultaneously on-line to answer the decryption requests). The tests have been conducted on Ethernet 100Mb local area network and using Pentium IV machines running Linux kernel 2.4.19. We have used 1024 bit RSA keys generated using the RSARef library. Since we only had 5 machines available for the experiment, for values of $t > 5$ we had decryption daemons handling requests for more than one key. We stress that giving a daemon the key of more than one trustee defeats the purpose of the system as an adversary that penetrates one of the machine would obtain multiple shares. We have set up this scenario only to be able to test the performance of the system with a significant number of users.

As a result of the experiments we have noticed that the creation of a shared file does not show any significant delay for all values of n and t that have been tested (all the cases tested had a creation time of about .08 seconds for $n = 5$ and .12 seconds for $n = 10$). Indeed the computation is local and there is no need to interact with the decryption daemons. The situation is different when we try to read a file. Here our experiments shows that the time needed to reconstruct the key is essentially independent from n but grows with t as the work needed to reconstruct the key (see Appendix 1.3.3) depends only on the number of clear shares necessary. This is an important features of our scheme as it shows that the scheme is scalable with respect to the number n of trustees provided that the threshold is relatively small.

In the following table we report times for $n = 5$ and $t = 1, 2, 3, 4, 5$. Times are expressed in seconds of elapsed time and are the average of 10 different runs. Similar times are observed for $n = 10$ despite the fact that, some decryption daemons had to perform two RSA decryptions. For example, forthe case $t = 10$, each daemon had to perform two decryptions.

$n = 5$	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
	.282	.293	.302	.352	.392

To better investigate this phenomenon and measure the impact of the network latency on the performance, we have run the same experiments using *one* machine and thus only one *local* decryption daemon. In this case, the decryption of the shares had to be performed sequentially by the same daemon while in the case in which more daemons were available the decryptions could be performed in parallel. We have observed essentially the same figures for the time needed to reconstruct a key and thus we can conclude that the network latency is responsible for a large portion of the delay observed to reconstruct the keys. This is expected as the machines on which we conducted the experiments have very fast CPUs (at least 1.2GHz) and thus RSA decryption could be performed very quickly.

Finally, we also investigated the impact of the multi-threaded architecture on the performance. Thus we have developed a single-thread (albeit unstable) version of our scheme just to measure the overhead induced by the multiple threads. From the experiments conducted we have found out that the single threaded version is between 20% and 30% faster. We are unable to explain such a gap between the two versions.

3.6 Conclusions and future directions

Our work shows that sharing keys among users and then reconstructing them when needed is an effective way to share trust among the several components of a distributed system. We have considered the case of a distributed cryptographic file-system and have provided kernel support for the operations of creating a shared file and reconstructing a shared file. The next step will be to further integrate this service into the Linux kernel by developing a new distributed file-system that has sharing capabilities. We point out that the novel components of such a file-system (that is the capabilities of creating, writing and reading a shared file) have already been implemented as part of the work presented by this paper at the kernel level and thus all it is needed is integration with an existing distributed file system. A more ambitious goal would be to apply the ideas of sharing to aspects of a distributed system other than file-system.

```
alice@alicehost$ addrkl -l 10 -s alice.sec -p alice.pub
alice@alicehost$ cat /proc/sharedservice/get_rkl_ids
pub_id: 0x86569138
        level: 10
        items in the prohibited list 0
        items in the allowed list 0
```

Figure 3.6: Starting the decryption daemon.

Overall Considerations

The network of all the networks, this is the way Internet is often called, was designed keeping well distinguished its main goal (to be a reliable communication channel between two or more hosts that wants to communicate but that communicates using different networks topologies and protocols), from protecting the channel itself from malicious misuses. Besides that, Internet grew fast and until it became what we see today. It was this growth that attracted many eyes on Internet, eyes that were seeking for a new market, eyes that were seeking for new opportunities. New services are born every day, the same way new stores are born on our streets. It is obvious that all this chaos needed order and the order needs protection. Network security started looking for new technologies to make Internet a secure channel, a channel where should be possible to privately talk with others, to let them speak about private facts or to let them echange credit cart informations to complete a purchase. Cryptology provided all these tools, from mathematical formulas proved to protect data to the final implementation of it. As a new flaw in a protocol is discovered a new patch is made to close it, in the same way, whereas an old protocols lack, partially or totally, of security countermisures, new protocols where designed to substitute it or to make it secure. In spite of these efforts, today a lot of protocols are still insecure. Even main, core, ones like the Domain Name System protocol and the Network File system. In this document we discussed about security flaws in both the two protocols and provided valid countermisures we suggest to take to make them secure and thus reliable. Many folks tried to make them secure and a lot of valid alternatives were born from these efforts. We do not believe our alternatives be the final solution to these two old widely used protocols. We, indeed, believe our protocols be strong enough to be used in an every day scenario but Internet itself will choose what suits best its needs. A lot of work on both protocols still need to be done: testing, improvementes, new features must yet be integrated and implemented in both of them. We received a positive feedback from folks around the world on our work, and this makes us believe we are on the right direction. The National Science Foundation (NSF) provided funds for the Sk-DNSSEC project thus proving the effectiveness of our work.

References

[1] Diane Davidowicz, "*Domain Name System (DNS) Security*", 1999 (URL: `http://www.geocities.com/compsec101/papers/dnssec/dnssec.html`)

[2] Mockapetris P., RFC 1034, "*Domain Names - Concepts and Facilities*", 1987. (URL: `ftp://ftp.isi.edu/in-notes/rfc1034.txt`)

[3] Mockapetris P., RFC 1034, "*Domain Names - Implementation and Specification*", 1987. (URL: `ftp://ftp.isi.edu/in-notes/rfc1035.txt`)

[4] Vixie P., Thomson S., Rekhter Y., Bound J., RFC 2136, "*Dynamic Updates in the Domain Name System (DNS UPDATE)*", 1997. (URL: `ftp://ftp.isi.edu/in-notes/rfc2136.txt`)

[5] Eastlake D., RFC 2137, "*Secure Domain Name System Dynamic Update*", 1997. (URL: `ftp://ftp.isi.edu/in-notes/rfc2137.txt`)

[6] Elz R., Bush R., RFC 2181, "*Clarifications to the DNS Specification*", 1997. (URL: `ftp://ftp.isi.edu/in-notes/rfc2181.txt`)

[7] Bellovin S., RFC 2316, "*Report of the IAB Security Architecture*", 1997. (URL: `ftp://ftp.isi.edu/in-notes/rfc2316.txt`)

[8] Eastlake D., RFC 2535, "*Domain Name System Security Extensions*", 1999. (URL: `ftp://ftp.isi.edu/in-notes/rfc2535.txt`)

[9] Eastlake D., Gudmundsson O., RFC 2538, "*Storing Certificates in the Domain Name System*", 1999. (URL: `ftp://ftp.isi.edu/in-notes/rfc2538.txt`)

[10] Eastlake D., RFC 2541, "*DNS Operational Security Considerations*", 1999. (URL: `ftp://ftp.isi.edu/in-notes/rfc2541.txt`)

[11] Vixie P., Gudmundsson O., Eastlake D., Wellington B., RFC 2845, "*Secret Key Transaction Signatures for DNS (TSIG)*", 2000. (URL: `ftp://ftp.isi.edu/in-notes/rfc2845.txt`)

[12] Ateniese G., Mangard S., "*A new approach to DNS security (DNSSEC)*", 2001. (URL: `http://www.cs.jhu.edu/ ateniese/dnssec.html`)

[13] Computer Emergency Response TEAM "*CERT* Advisory CA-97.22*", "*Topic: BIND - the Berkeley Internet Name Daemon*", CERT, 1997. (URL: `http://www.cert.org/advisories/CA-97.22.bind.html`)

[14] Men and Mice, "*What is DNS Spoofing*", 1999. (URL: `http://www.menandmice.com/infobase/mennmys/vefsidur.nsf/index/6.2.1.1`)

[15] Garfinkel S., Spafford G., "*Practical Unix and Internet Security*", 2^nd Ed., Sebastopol, CA, O'Reilly and Associates, 1997, 473-475.

[16] Cheswick W.R., Bellovin S. M., "*Firewalls and Internet Security*", Reading Ma, Addison-Wesley, 1994, 28.

[17] Chapman D.B., Zwicky E.D., "*Building Internet Firewalls*", Sebastopol, CA, O'Reilly and Associates, 1995, 278-296.

[18] Larson M., Liu C., "*Using BIND: Don't get spoofed again*", SunWorld, 1997. (URL: `http://www.sunworld.com/swol-11-1997/swol-11-bind.html`)

[19] Albitz P., Liu C., "*DNS and BIND, 4th Edition*, O'Reilly, 2001

[20] Blaze M., "*A Cryptographic File System for UNIX*", First ACM Conference on Communication and Computing Security, pp. 158-165, Fairfax VA, 1993.

[21] Blaze M., "*Key Management in an Encrypting File System*", Proceedings of the Summer 1994, USENIX Conference, 1994.

[22] Boneh D., Franklin M., "*Efficient Generation of Shared RSA Keys*", Proceedings of Crypto '97, 425-439.

[23] Cattaneo G., Catuogno L., Del Sorbo A., Persiano G., "*The design and implementation of a Transparente Cryptographic File System*", Proceedings of the USENIX Annual Technical Conference 2001, Freenix track, Boston, MA, 2001.

[24] Deswarte Y., Blain L., Fabre J.C., "*Intrusion Tolerane in Distributed Computing Systems*", Proceedings of the IEEE Symposium on Research in Security and Privacy, 1991, 110-122.

[25] Howard J.H., "*An Overview of the Andrew File System*", Proceedings of the USENIX Winter Technical Conference, Dallas TX, 1998.

[26] Marières D., Kaminsky M., Kaashoek M.F., Witchel E., "*Separating key management from file system security*", Proceedings of 17th ACM Symposium on Operating System Principles (SOSP '99), Kiawah Island, South Carolina, 1999.

[27] Sandberg R., Goldberg D., Kleiman S., Walsh D., Lyon B., "*Design and Implementation of the Sun Network File System*", Proceedings of USENIX Association Conference, 1985, 119-130.

[28] Satyanarayanan M., "*Integrating Security in a Large Distributed System*", ACM Trans. Computer System, vol. 7, no. 3, Aug. 1989, 247-280.

[29] Shamir A., "*How to share a secret*", Comm. ACM v. 24 n. 11, 1979.

[30] Zadok E., Badulescu I., Shender A., "*CryptFS: a stackable vnode level encryption file system*", Columbia University Computer Science Report CUCS-02198, 1998.

[31] Stinson D., "*Cryptography: Theory and Practice*", CRC Press.

[32] Wu T., Malkin M., Boneh D., "*Building Intrusion Tolerant Applications*", Proceedings of the 8th USENIX Security Simposium, 1999, 79-91.

[33] Blaze M., "*A Cryptographic File System for UNIX*", First ACM Conference on Communication and Computing Security, Fairfax VA, 1993.