To the Graduate Council:

I am submitting herewith a thesis written by Nair Venugopal entitled "The Design, Implementation, and Evaluation of Cryptographic Distributed Applications: Secure PVM." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Tom Dunigan, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

The Design, Implementation, And Evaluation Of Cryptographic Distributed Applications: Secure PVM

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Nair Venugopal

May, 1996

Acknowledgments

I would like to thank my advisor, Dr. Tom Dunigan, for the help and guidance he provided all through my academic career at UT-Knoxville. I thank the other committee members, Dr. Bill McClain and Dr. David Straight, for their comments and assistance in developing this thesis. I would also like to express my appreciation to Al Geist and Bob Manchek for their time and assistance in making this work possible.

I am indebted to my parents S.N.K. Nair and Vilasini Nair, for their love and support. I also thank Biju, Lathu and Jayasree for still liking me, though I have given them many reasons why they shouldn't. Finally, I would like to thank all my friends at Knoxville, especially Mohamad Eljazzar. It has been a privilege to share an office with him for the past two and one-half years.

Abstract

This research investigates techniques for providing privacy, authentication, and data integrity to message passing in distributed applications. Various software mechanisms for message hashing and encryption are evaluated, including techniques for key generation and key distribution. Different crypto-APIs' are evaluated, and the distribution of a single session key for n-party communication is implemented.

A secure version of PVM (Parallel Virtual Machine) is developed using Diffie-Hellman, MD5, and various symmetric encryption algorithms to provide message privacy, authentication, and integrity. The modifications to PVM are described, and the performance of secure PVM is evaluated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The prevalence of computers and computer networks has greatly influenced the way people work. Many computing environments now exist in which frequent and substantial parts of the activities involve communication among computers linked by *open* networks. Most of the networks (and internetworks) used for these activities are open in the sense that they are vulnerable to eavesdropping and interference from unauthorized intruders. Depending on the sensitivity of the information and the type of tampering, the potential damage can be significant.

Cryptography has been known for centuries and used by the military to protect sensitive or secret information from unauthorized personnel while the information was delivered via unsecured channels. Encryption permits the establishment of a data channel that is less open than the underlying internetwork, by arranging that only authorized parties can create, inspect, and/or modify the data. Only those who possess the correct decryption key can decipher the encrypted information. Cryptography can also be used to protect the integrity of data by a process called message authentication and verification. This process involves the calculation of a message checksum, which is then sent along with the message to a recipient. The recipient can recompute the checksum and verify that the message was not modified in transit.

Currently there are two dominant cryptographic techniques: secret-key cryptography and public-key cryptography (§ 2.1). Security services based on these mechanisms assume keys to be distributed prior to secure communication. In secret-key cryptography, a secret key is established and shared between the two parties, and the same key is used to encrypt and decrypt messages. If the two parties are in different physical locations, they must trust a courier, or some transmission system to establish the initial key and trust this third-party not to disclose the secret key they are communicating. In a public-key system, a user makes use of a pair of keys: a public key and a private key. The two keys are uniquely related so that the public key of a user can be made public without revealing any information about the private key. The private key of a user is usable only by its owner. Messages encrypted with the private key can only be decrypted with the public key and vice versa. Since the private key is not shared, public-key cryptography makes key management easier, allowing two parties to establish a secure channel without having to share a secret.

In *distributed computing*, a collection of computers connected by a network functions as a single entity in solving computational problems. By connecting several machines together, one has access to more compute power, memory and I/O bandwidth. One of the biggest advantages of distributed computing is the low cost relative to the computational power available, since the computers and network already exist at a typical university or industrial site; and much of the processor power goes unused[1].

Current distributed computing systems, for all their virtues, make it difficult to "reliably" limit access to sensitive data. Hosts often unselectively broadcast data to distant and unpredictable places, remote login facilities unintentionally open access to intruders, and distributed file systems often assume that all machines to which they provide service are trustworthy. To reduce these risks, cryptographic techniques can be used to limit data access while still taking advantage of insecure networks and services.

[1] Especially at night and weekends.

1.1 Objectives and Significance

Network security on the continually expanding Internet is a topic of increasing importance. The concomitant security problems in the Internet, coupled with the wide range of services and applications it supports, makes IP-based[2] networks a good choice for understanding and exploring network security issues. This research focusses on developing and evaluating various alternatives for adding security to distributed parallel applications that execute over unsecured IP-based networks. Specifically, this research addresses the following questions.

1. What UNIX[3] software systems are available for inter-host user authentication?
2. What encryption mechanisms (public and secret key) are suited to distributed applications?
3. What mechanisms exist for key management in distributed applications?
4. How does one achieve data integrity in a message-passing distributed application?
5. What is the effect of encryption/authentication on application performance?

In order to get a better understanding of the design tradeoffs involved in developing secure distributed applications, this research focussed on extending the Parallel Virtual Machine (PVM) to include support for cryptographic authentication, data integrity, and encryption. PVM[4] has become the *de facto* standard for distributed computing worldwide [ZG95]. PVM is a message passing system that allows a collection of heterogeneous computers on a network function like a distributed operating system. PVM supplies the functions to automatically start up tasks on the logical distributed-memory computer and allows the tasks to communicate and synchronize with each other.

[2]IP: Internet Protocol [Pos81].
[3]UNIX is a trademark of X/Open.
[4]Developed at the Oak Ridge National Lab (ORNL) [GBD+94]

1.2 Methods and Assumptions

Currently, PVM depends on UNIX and the standard TCP/IP protocol suite for security. For remote process initiation, the user either has to send his password on the remote host in clear-text or make the remote host "trust" the host running the master `pvmd` (using `rsh`). Both these approaches have concomitant risks. Trusted hosts can be impersonated by exploiting existing vulnerabilities in IP-based networks. These vulnerabilities include attacks based on *IP source routing*, *DNS database corruption* and *TCP sequence number prediction* [Bel89] [CERT96a]. Also, an established session can be intercepted and co-opted by an attacker by IP-splicing[5] attacks [Neu95] [Jon95].

In secure PVM, the above risks can be reduced by using existing mechanisms like Kerberos (§ 2.3.1) or new software like SSH or STEL (§ 2.4.2). The user could require that the set of machines used in his/her PVM application use Kerberos for user authentication and for `rsh` services used in spawning slave PVM daemons. Instead of Kerberos, SSH or STEL can also be used to securely spawn processes on remote hosts.

Once the `pvmds` are up and running, applications use the PVM infrastructure to exchange messages between PVM tasks running on different hosts in the virtual machine. This communication is insecure in the sense that an attacker can inspect/modify the contents of these messages. PVM's security is enhanced by adding cryptographic extensions to facilitate *authentication* and *encryption* of PVM messages. Authentication ensures that no unauthorized party can impersonate a PVM process as if it were part of the virtual machine. Encryption ensures that no unauthorized party can discover the contents of PVM messages.

The PVM slave startup protocol is extended to include a Diffie-Hellman (§ 2.1.2.2) key exchange. By means of this key exchange, the master `pvmd` can distribute a secret PVM session key to all the slaves. Once all the `pvmds` have access to this PVM session key, all

[5] Use of one-time passwords do not hinder this attack, since the connection is hijacked after the user authentication phase.

messages sent between PVM hosts can be encrypted and authenticated. The PVM user can either choose to encrypt all messages while starting PVM or selectively enable encryption for essential messages.

This study is based on several assumptions. Those that are not discussed later are presented here.

- Publicly available implementations of encryption algorithms like DES, IDEA, RC4 and one-way hash functions like MD5 are used in this research. It is assumed that these implementations are correct.

- This research is concerned with inter-host security issues. Intra-hosts security issues like the reliability of the operating system services are not addressed in this research. This is because intra-host security can be subverted by a malicious user with *super-user* privileges on the machine.

- This research does not address the issue of authenticating a user to a remote host. User authentication mechanisms like the Kerberos or DCE infrastructure are assumed to be available on the PVM hosts.

1.3 Thesis Overview

The next chapter surveys related literature in the areas of cryptography, key management, and network security. Chapter 3 discusses various issues involved in the design of a cryptographic application. In Chapter 4, the implementation and performance of secure PVM is described. The last chapter summarizes work done in this research and identifies areas were further studies would be beneficial.

Chapter 2

Survey of Related Work

It is possible to include secure communication at various levels in a communication protocol hierarchy. At the physical layer, security can be achieved by various non-cryptographic techniques that prevent tampering with the communication medium itself. At the data link layer, it is possible to encrypt all traffic on each link using a key which is shared by all nodes directly connected to that link. This is called *link encryption*; it protects against intruders from outside the community that shares that data link, but it does not distinguish authorized parties within the community. When a communication path is formed over a network consisting of multiple data links, link encryption allows intrusion by members of the "trusted" community of *every* data link traversed by the path.

A natural place for implementing *end-to-end* encryption policies is the network layer, since there is direct node-to-node addressing of packets. Transport-layer encryption differs from network-layer encryption primarily in its ability to provide a finer granularity of protection. However, both network and transport layer encryption are subject to network "discontinuities" due to the use of firewalls and protocol translators. This perhaps suggests that real end-to-end encryption can be accomplished only at the higher layers. Also, when the underlying infrastructure does not provide adequate security, application-layer encryption often becomes the only possible solution. Application-layer security is the focus of this research.

The first two sections look at various aspects of cryptography and authentication. The remaining sections review related work: protocols and software systems that provide authentication, integrity and confidentiality services at different layers of the standard network hierarchy.

2.1 Cryptography

Cryptography, as applied to data communications, is a matter of taking the original message and producing an encrypted version by using a special piece of information known only to the sender and receiver. The original message is called the *plaintext*; the special information is called the *key*, and the resulting message is called the *ciphertext*. The process for producing ciphertext is called *encryption*. The reverse of encryption is called *decryption*. Encryption (and decryption) can be performed by either computer software or hardware. Common approaches to encryption include writing the algorithm on a disk for execution by a CPU; placing it in ROM or PROM for execution by a microprocessor; and isolating storage and execution in a computer peripheral device (*e.g.*, PCMCIA card).

2.1.1 Secret Key Encryption

In conventional cryptographic systems, the sender and receiver know a single key in common and keep this key secret from everybody else. Such an arrangement is called a *secret (or symmetric) key cryptosystem*. Mathematically, a secret-key cryptosystem consists of an encryption system E and a decryption system D. The encryption system E is a collection of functions E_K, indexed by *keys* K, mapping some set of *plaintexts* P to some set of *ciphertexts* C. Similarly the decryption system D is a collection of functions D_K such that $D_K(E_K(P)) = P$ for every plaintext P. That is, successful decryption of ciphertext into plaintext is accomplished using the same key (index) as was used for the corresponding encryption of plaintext into ciphertext.

One of the biggest problems that cryptosystems have to face is the secure distribution of the secret encryption keys. In order to be useful, cryptosystems need to be able to securely disseminate the necessary encryption keys to all parties who need them. In many systems that use secret-key cryptography, the keys need to be distributed by manual means that are *outside* the normal functioning of the system. This approach is not very scalable nor flexible and limits the use of secret-key cryptosystems. Public-key cryptography (see Section 2.1.2) offers an elegant solution to this problem.

Schneier [Sch96], and Stallings [Sta95] survey a number of secret-key cryptographic algorithms. The following sub-sections deal with two of the most popular ones, DES and IDEA.

2.1.1.1 DES

The best known example of a secret key cryptosystem is the U.S. Data Encryption Standard (DES) [US 77]. DES was designed by IBM in the early 1970s in response to a National Bureau of Standards[1] request. The key length in DES is 56 bits[2].

DES operates on a 64-bit block of plaintext. First, the 64-bit plaintext block passes through an initial permutation that rearranges the bits to produce a *permuted* input. The block is then split into a left half and a right half, each 32 bits long. This is followed by sixteen rounds of identical operations (called `Function f`), in which the data are combined with the subkeys. After the sixteenth round, the left and right halves are joined and passed through a final permutation (the inverse of the initial permutation) to produce the 64-bit ciphertext.

In each round, the key bits are shifted and then 48-bit subkeys are selected from the 56

[1] Now known as the National Institute of Standards and Technology (NIST).

[2] The function expects a 64 bit key as input. Every eighth bit is used for parity checking and is ignored.

bits of the key. The right half of the data block is expanded to 48 bits via an expansion permutation, combined with 48 bits of a shifted and permuted key via an *exclusive-OR*, substituted for 32 new bits using a substitution algorithm and then permuted again. These four functions make the `Function f`. The output of `Function f` is then combined with the left half via another *exclusive-OR*. The result of these operations becomes the new right half; the old right half becomes the new left half. These operations are repeated 16 times, making 16 rounds of DES (see Figure 2.1).

If B_i is the result of the i-th iteration, L_i and R_i are the left and right halves of B_i, K_i is the 48-bit key for round i, `f` is the function that does all the substituting/permuting and XOR-ing with the key, then a round looks like:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_i \oplus f(R_{i-1}, K_i) \end{aligned}$$

The process of decryption with DES is essentially the same as the encryption process. The ciphertext is input to the DES algorithm, but the subkeys K_i are used in reverse order.

Ever since the adoption of DES as a federal standard, lingering concerns have existed against its short key-length (56-bits). In 1981, Diffie [Sch96] estimated that a special-purpose DES-cracking machine costing \$20 million could recover a key in two days based on "brute-force" search. In 1993, Wiener [Wie93] presented a design for an "economical" DES-cracker costing \$1,000,000 that will find any DES key in about 7 hours, with an average search taking 3.5 hours.

Given the potential vulnerability of DES to brute force attacks, there has been considerable interest in finding more "secure" alternatives. One of these approaches is to use multiple encryption with DES, using multiple number of keys. Tuchman [Tuc79] proposed a triple encryption method that uses of only two keys[3]. The function follows an *encrypt-decrypt-encrypt* (EDE) sequence:

[3]There are many variants to the 3-DES algorithm, including using 3 distinct keys 168-bit key.

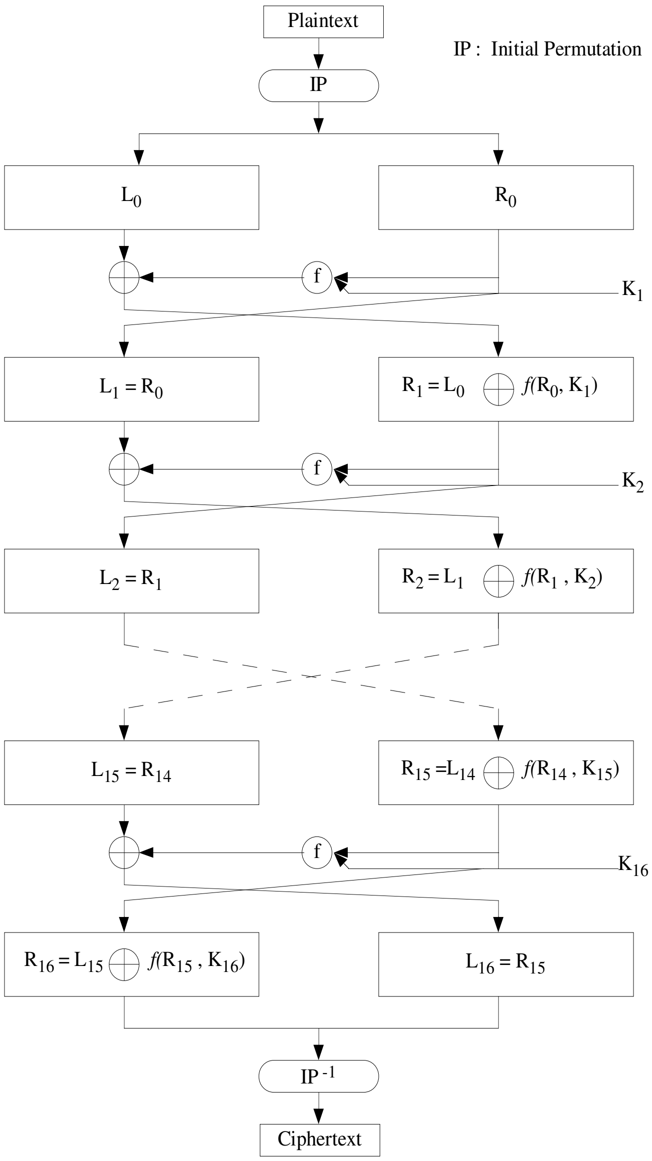

Figure 2.1: Overview of DES
Source: [Sch96]

$$ciphertext = E_{K_1}[\ D_{K_2}[\ E_{K_1}[\ plaintext\ ]]]$$

There is no cryptographic significance to the use of decryption for the second stage. However, this approach allows users to decrypt data encrypted by users of the “single” DES algorithm (by making $K_1 = K_2$ in the above equation), thereby preserving the existing investment in DES software and equipment.

It is expected that 3-DES will soon become a *de-facto* standard for secret-key encryption. It has already been adopted for use in the key management standards like ANS X9.17 and for Privacy Enhanced Mail (PEM)[4].

2.1.1.2 IDEA

International Data Encryption Algorithm (IDEA) was developed by Xuejia Lai and James Massey of the Swiss Federal Institute of Technology in 1990. It is one of a number of conventional algorithms proposed in recent years to replace DES. It uses a 128-bit key to encrypt data in blocks of 64 bits.

The design philosophy behind the algorithm is one of “mixing operations from three different algebraic groups” [Sch96]. IDEA uses three operations,

- Exclusive-OR (XOR)
- Addition modulo 2^{16} , and
- Multiplication modulo $2^{16} + 1$.

The IDEA algorithm consists of eight rounds, followed by a final transformation function. Each round makes use of six 16-bit sub-keys, whereas the final transformation uses four sub-keys, for a total of fifty-two sub-keys. These fifty-two sub-keys are derived from the original 128-bit key. For encryption, first the 64-bit input is broken into four 16-bit sub-blocks. These four sub-blocks become the input to the first round of the algorithm. In each

[4]ANS X9.17: American National Standard for Financial Institution Key Management.

round, the four sub-blocks are XOR-ed, added and multiplied with one another and with six 16-bit sub-keys. Between rounds, the second and third sub-blocks are swapped. Finally, the four sub-blocks are combined with four sub-keys in an output transformation and a 64-bit ciphertext is produced. Decryption is similar to encryption, except that the sub-keys are reversed and are slightly different. The decryption sub-keys are either the additive or multiplicative inverses of the encryption sub-keys. [LM91] describes the encryption/decryption mechanism in detail.

IDEA is patented in Europe and United States, though no license fee is required for non-commercial use.

2.1.1.3 Cipher Types and Modes

There are two basic types of symmetric (secret-key) algorithms: block ciphers and stream ciphers. *Block ciphers* operate on blocks of plaintext and ciphertext (usually 64 bits but sometimes longer). Since most messages do not divide cleanly into 64-bit (or bigger) blocks, the last block has to be *padded*, such that the total length of the message becomes a multiple of the block-size. Block ciphers are generally used to encrypt "static" entities like files, or data in message-oriented communication protocols (*e.g.,* PVM messages). *Stream ciphers* operate on streams of plaintext and ciphertext one bit or byte at a time (*e.g.,* encryption of a telnet session). With a block cipher, the same plaintext block will always encrypt to the same ciphertext block, using the same key. With a stream cipher, the same plaintext bit or byte will encrypt to a different bit or byte each time it is encrypted.

A cryptographic *mode* usually combines the basic cipher, some sort of feedback, and some simple operations. FIPS-81 [US 80] specifies four modes of operations to cover all the possible applications of encryption for which DES can be used. However, these "operation modes" can apply to any secret-key algorithm.

Electronic Codebook (ECB) Mode

In this mode, each block of plaintext is encrypted into a block of ciphertext. It is called 'electronic codebook' because, for a given key, there is a unique ciphertext for every 64-bit block of plaintext. This makes it theoretically possible to compile a code book of plaintexts and corresponding ciphertexts. For a message longer than 64-bits, the message is broken into 64-bit blocks, padding the last block if necessary.

Since each plaintext block is encrypted separately, the ECB mode is ideal for "random-access" file operations. Encryption/decryption on different blocks can be carried out in parallel, since there are no dependencies between different blocks of the plaintext/ciphertext. Bit errors in the ciphertext, when decrypted, will cause the entire plaintext block to be decrypted incorrectly, but will not affect the remainder of the plaintext. However, if the same plaintext block appears more than once in the message, the corresponding ciphertext generated by encryption is always the same. This makes the ECB mode of encryption less secure for highly structured long messages, since it may be possible for a cryptanalyst to exploit these regularities.

Cipher Block Chaining (CBC) Mode

Chaining adds a feedback mechanism to a block cipher. The results of the encryption of the previous blocks are fed into the encryption of the current block, thereby making each ciphertext block dependent on all the previous plaintext blocks in addition to the current plaintext block.

In CBC mode, the plaintext is XOR-ed with the previous ciphertext block before it is encrypted. After a plaintext block (P_i) is encrypted, the resulting ciphertext is stored in a feedback register to be used as an input for encrypting the next plaintext block. For decryption, a ciphertext block(C_i) is decrypted normally and also saved in a feedback register. After the next block is decrypted, it is XOR-ed with the results of the feedback register. This operation can be mathematically represented as follows:

$$Encryption: \quad C_i = E_K(P_i \oplus C_{i-1})$$

$$Decryption: \quad P_i = C_{i-1} \oplus D_K(C_i)$$

To produce the first block of ciphertext, an *initialization vector (IV)* is XOR-ed with the first block of plaintext. On decryption, the IV is XOR-ed with the output of the decryption algorithm to recover the first block of plaintext. With the addition of the IV, identical plaintext messages encrypt to different ciphertext messages. The IV does not have to be unique or kept secret; it can be transmitted in the clear with the ciphertext. According to Schneier, " while the IV should be unique for each message encrypted with the same key, it is not an absolute requirement" [Sch96].

The CBC mode is useful for encrypting files and long messages. In the CBC mode, a single bit error in the ciphertext affects one block and one bit of the recovered plaintext. The decrypted block corresponding to the block containing the error is completely garbled. The next block has a 1-bit error in the same bit position as the error. Subsequent blocks are not affected by the error, so CBC is *self-recovering.*

Cipher Feedback (CFB) Mode

With CBC mode, encryption cannot begin until a complete block of data is received. This limits the use of CBC mode encryption/decryption for data-streams. To resolve this issue, block ciphers can be implemented as self-synchronizing stream ciphers[5]; this is called the *cipher-feedback* mode.

A block algorithm in CFB mode operates on a queue the size of the input block. Initially, the queue is filled with an initialization vector (IV), as in CBC mode. The queue is encrypted and the left-most eight bits of the result are XOR-ed with the first 8-bit[6] character of the plaintext, to become the first 8-bit character of the ciphertext. The same eight bits are also moved to the right-most eight bit positions of the queue and all the other bits move eight positions to the left. The eight left-most bits are discarded. Then the next plaintext character is encrypted in the same manner. Decryption is the reverse of this process. For

[5] In a self-synchronizing stream cipher, each keystream bit is a function of a fixed number of previous ciphertext bits.

[6] The most commonly used size of each unit for CFB mode is 8 bits.

both encryption and decryption, the block algorithm is used in its encryption mode.

$$Encryption: \quad C_i = P_i \oplus E_K(C_{i-1})$$

$$Decryption: \quad P_i = C_i \oplus E_K(C_{i-1})$$

The IV in CFB mode should be unique (unlike in CBC mode where the IV should be unique but does *not* have to be). If the IV is not unique, a cryptanalyst can recover the corresponding plaintext. The IV must be changed with every message.

In n-bit CFB, a single bit error in the ciphertext will affect the decryption of the current and following $m/n - 1$ blocks, where m is the block size. 8-bit CFB is generally the mode of choice for encrypting streams of characters when each character has to be treated individually, as in a link between a terminal and a host.

Output Feedback (OFB) Mode

Output-feedback (OFB) mode is a method of running a block cipher as a synchronous stream cipher[7]. It is similar to the CFB mode, except that the n bits of the previous output block are moved into the right-most positions of the queue. On both the encryption and decryption sides, the block algorithm is used in its encryption mode. Since the feedback mechanism is independent of both the plaintext and the ciphertext streams, the OFB mode is sometimes referred to as *internal feedback*. The OFB mode IV should be unique but does not have to be secret.

If n is the block size of the algorithm and S_i is the state (which is independent of either the plaintext or the ciphertext), n-bit OFB works as follows:

$$Encryption: \quad C_i = P_i \oplus S_i; \quad S_i = E_K(C_{i-1})$$

$$Decryption: \quad P_i = C_i \oplus S_i; \quad S_i = E_K(C_{i-1})$$

In OFB mode, a single-bit error in the ciphertext causes a single-bit error in the recovered plaintext. This makes OFB useful for digitized analog transmissions (like video/voice),

[7] In a synchronous stream cipher, the keystream is generated independent of the message stream [Sch96].

where the occasional single-bit error can be tolerated. Since the OFB mode relies on synchronization between the shift registers at the encryption and decryption end for correct functioning, a mechanism to detect and correct synchronization loss should exist. Analysis of OFB mode [DP83] has demonstrated that OFB mode should be used only when the feedback size is the same as the block size.

2.1.2 Public Key Encryption

The concept of public-key cryptography was invented by Whitfield Diffie and Martin Hellman [DH76], and independently by Ralph Merkle [Mer78]. They demonstrated that a message can be encrypted using one key and decrypted by another. The two keys are related in such a way that a knowledge of one key does not make it possible to figure out the other key. This permits one key, the *public* key, to be widely known[8], while the corresponding *private* key is known only to a single user. Mathematically, public key cryptography is based on trap-door one-way functions. A *trap-door* one-way function is a special type of one-way function that is easy to compute in one direction and hard to compute in the other direction unless certain trapdoor information is known.

RSA[9] and Diffie-Hellman are two well-known public key cryptosystems. The following subsections describes the functioning of these two cryptosystems.

2.1.2.1 RSA

The best known example of a public key cryptosystem is RSA. RSA was developed at MIT in 1977 by Ronald Rivest, Adi Shamir and Len Adelman [RSA78]. The public and private keys are functions of a pair of large prime numbers.

To generate the key-pair, two large prime numbers[10], p and q, are chosen. The product,

[8] There remains the problem of verifying the authenticity of the public key.

[9] The name is derived from the initials of the inventors.

[10] The two primes, p and q, which compose the modulus, should be of roughly equal length; thus if one chooses to use a 512-bit modulus, the primes should each have length approximately 256 bits.

$n = pq$ is computed. Then the encryption key, e, where e and $((p-1)*(q-1))$ are relatively prime, is randomly chosen. In the next phase, Euclid's algorithm for *greatest common divisor* (gcd) calculation is used to compute the decryption key, d, such that

$$e * d = 1 \ (mod(p-1)*(q-1))$$

That is,

$$d = e^{-1} \ (mod(p-1)*(q-1))$$

The numbers e and n are the public key; the number d is the private key. Once the key-pair is computed, the two primes, p and q are no longer needed.

To encrypt a message m, it is first divided into numerical blocks such that each numerical block has a unique representation modulo n. The encrypted message, c, will be made up of similarly sized message blocks, c_i, of about the same length. The encryption formula is :

$$c_i = {m_i}^e \ (mod \ n)$$

To decrypt a message, each encrypted block c_i is taken and m_i is computed:

$$m_i = {c_i}^d \ (mod \ n)$$

RSA gets its security from the difficulty of factoring large numbers[11]. Recovering the plaintext from one of the keys and the ciphertext is *conjectured* to be equivalent to factoring the product of the two prime numbers.

RSA is about 1000 times slower than DES in hardware and in software it is about 100 times slower. The RSA algorithm is patented in the United States, but not in other countries. The U.S. patent will expire in September 2000.

2.1.2.2 Diffie-Hellman

The Diffie-Hellman (DH) key exchange was the first published public-key algorithm [DH76] . The DH key exchange provides a mechanism by which two parties can negotiate

[11] Currently, large numbers in public key cryptography refer to integers which are 512 bits or longer in size.

a secret session key, without fear of eavesdroppers. This system gets its security from the difficulty of calculating discrete logarithms in a finite field, as compared with the ease of calculating exponentiation in the same field.

In the DH key exchange, there are two publicly known numbers: a prime number q and an integer α (called the *generator*) that is a primitive root[12] of q. Two users i and j do a DH key exchange as follows. User i selects a random integer $X_i < q$ and computes $Y_i = \alpha^{X_i} mod\ q$. Similarly, user j independently selects a random number $X_j < q$, and computes $Y_j = \alpha^{X_j} mod\ q$. Each side keeps the X value private and makes the Y value available publicly to the other side. Now user i computes the key as $K = (Y_j)^{X_i} mod\ q$ and user j computes the key as $K = (Y_i)^{X_j} mod\ q$. By the rules of modular arithmetic, these two calculations produce identical results (*ie.* $\alpha^{X_i X_j} mod\ q$). This "shared" key can be used for encrypting messages between the two parties using conventional secret key encryption.

The Diffie-Hellman key exchange algorithm is patented in the United States. The U.S. patent will expire in April 1997.

2.1.2.3 Hybrid Cryptosystems

A misconception about public-key encryption is that it has made secret key encryption obsolete. It should be noted that secret key cryptosystems are much faster than any known public key cryptosystem when encrypting large bodies of plaintext. According to Whitfield Diffie [Dif88], "The restriction of public-key cryptography to key management and signature applications is almost universally accepted". A judicious combination of the two approaches is often adopted to achieve the performance of the former with the flexibility of the latter. Typically, public-key encryption is used to exchange a secret key. The subsequent encryption of data using this key is done with a secret-key algorithm.

[12] A primitive root of a prime number p is one whose powers generate all the integers from 1 to p - 1.

2.1.3 Secure One-Way Hash Functions

A one-way hash function, *H(M)*, operates on an arbitrary length message M to generate a fixed-length hash value, h. It has the following properties.

- Given M, it is easy to compute h.
- It is irreversible. Given h, it is hard to compute M such that $H(M) = h$.
- It is collision-resistant. It is hard to find two random messages, M and M', such that $H(M) = H(M')$.

These properties make one-way hash functions useful mechanisms for authenticating messages and for ensuring data integrity.

MD5 and SHA are two well-known one-way hash functions. They are both based on MD4 [Riv92a]. MD4 is a block-chained digest algorithm, computed over the data in phases of 512-byte blocks. The first block is processed with an initial seed, resulting in a digest that becomes the seed for the next block. When the last block is computed, its digest is the digest for the entire stream. This chained seeding prohibits the parallel processing of the data blocks. MD4, while not broken yet, has been shown to be "potentially" vulnerable by Bosselaers and Boer [dBB92], and Biham [Bih93].

2.1.3.1 MD5

The MD5 message digest algorithm [Riv92b] is an extension of the MD4 [Riv92a] message-digest algorithm. According to its inventor, Ronald Rivest, "MD5 is slightly slower than MD4, but is more *conservative* in design. MD5 was designed because it was felt that MD4 was perhaps being adopted for use more quickly than justified by the existing critical review; because MD4 was designed to be exceptionally fast, it is *at the edge* in terms of risking successful cryptanalytic attack. MD5 backs off a bit, giving up a little in speed for a much greater likelihood of ultimate security".

MD5 processes the input data of arbitrary length (less than 2^{64} bytes) in 512-bit blocks, divided into sixteen 32-bit sub-blocks and produces as output a 128-bit *fingerprint* or *message digest* of the input First, the message is padded so that its length is 64 bits short of being a multiple of 512. Then, a 64-bit representation of the message's length (before the addition of the padding bits) is appended to the result to make the length an exact multiple of 512. The main loop of the MD5 algorithm has four rounds[13], with each round using a different operation 16 times. Each operation performs a non-linear function based on XOR, AND, OR and NOT operators. This loop gets executed for as many 512-bit blocks as are in the message. The output of the algorithm is a set of four 32-bit blocks, which concatenate to form a single 128-bit *message digest.*

2.1.3.2 SHA

NIST, along with the NSA, designed the Secure Hash Algorithm (SHA) [NIS93] for use with the Digital Signature Standard [NIS92]. Like MD5, SHA is very similar to MD4.

SHA works on messages of length less than 2^{64} bytes as input and produces a 160-bit message digest. First, the message is padded to make it a multiple of 512 bits long. For padding, it uses the same mechanism employed as in MD5. The main loop of the SHA algorithm has four rounds of 20 operations each[14]. It processes the message 512 bits at a time and continues for as many 512-bit blocks as are in the message.

Since SHA produces a 160-bit hash, it is more resistant to brute-force attacks than 128-bit hash functions like MD5. A flaw was apparently found in the original specification of SHA and a newer specification has been released. The newer version is not inter-operable with the older version.

[13] MD4 had three rounds.
[14] MD5 has four rounds of 16 operations each.

2.2 Authentication

In order to get access to services in a distributed environment, entities (users and hosts) should be able to prove that they are who they "claim" to be – *i.e.,* authenticate themselves. The authentication process usually consists of identification and verification of the entity's claim [WL92]. *Identification* is the process whereby an entity claims a certain identity, while *verification* is the process whereby that claim is checked. Three ways by which users and hosts can be authenticated are:

- By what you *are*; identifying an individual by some of his biological characteristics, *e.g.,* a retinal scan or a fingerprint.
- By what you *have*; proving one's identity by presenting a physical credential, *e.g.,* a credit card.
- By what you *know*; identifying an individual by something he (and only he) knows, *e.g.,* a login password.

Though each of these techniques have their weaknesses, authentication procedures can be strengthened by combining them.

2.2.1 User-Host Authentication

The authentication method most widely used in network environments today consists of asking users to prove their identity by demonstrating knowledge of a secret they know, typically a password. Since the host just has to be able to distinguish between valid and invalid passwords, it stores one-way functions of the passwords instead of the password itself for "enhanced" security. However, since the passwords are transmitted in the clear over the network when a user attempts to log in to a remote host, this authentication mechanism becomes vulnerable to *passive attacks* [CERT94]. A passive attack relies on being able to passively monitor information (*eavesdrop*) being sent between other parties. Moreover, such systems are also vulnerable to off-line *dictionary attacks* in which the attacker attempts to guess the hashed password by comparing it with a compiled dictionary of hashed strings (obtained by using the same one-way hash function) corresponding to commonly used passwords.

2.2.1.1 One-time passwords

In a one-time password system, the user's secret password never crosses the network during login, thereby eliminating the risk of "*eavesdropping*". Since one-time passwords do not get re-used for authenticating future sessions, they are useless to an eavesdropper.

S/KEY[15] is a one-time password system which uses secure one-way hash functions to generate a finite sequence of single-use passwords from a single secret known only to the user. The security of this system is based on the premise that it is computationally infeasible to invert one-way hash functions. The functioning of the S/KEY authentication system is discussed in detail in Section 2.4.4.

Time Based Cards (*e.g.*, SecurID[16]) offer another mechanism to implement one-time password systems. Such cards have an on-board clock. To gain access to a protected resource, a user types his Personal Identification Number (PIN), followed by the current access code displayed on the card. The access code is the output of a function of the current time. The host can compute the proper value that the card should be displaying and therefore can determine whether or not a legitimate card is being used. Such systems require the client and server to share the same notion of time.

Challenge-Response cards are smart cards that contain not only memory, but also processing capabilities. They are typically initialized with a secret key which is used to encrypt challenges to the card. In order to prove that the user has the correct card, the system sends a random number to the card and expects the card to algorithmically transform the

[15] S/KEY is a trademark of Bellcore.
[16] SecurID is a trademark of Security Dynamics.

number into a *response* number. The user keys in this response. The system performing the authentication can determine if the card has the correct key by checking if the response keyed-in matches the expected response.

2.2.1.2 Trusted Third-Parties

In this model, the host does not rely solely on the credentials supplied by the user seeking authentication. Also, the secret key of the user never crosses the network. Instead, both parties rely on a third entity, called a Key Distribution Center (KDC), to vouch for each other's identity. The KDC accomplishes this by sharing a secret key with each entity and is able to verify the identity of the entity based on this shared knowledge (authentication using trusted third-party systems are discussed in detail in Section 2.3).

2.2.2 Host-Host Authentication

The dominant form of host-to-host authentication on the Internet today relies on the network itself to provide authentication information. Network authentication comes in two flavors, address-based and name-based. For the former, the source's IP address is used as an authenticator. This approach is prone to attacks where the source address is modified by the attacker (*e.g.,* TCP sequence number guessing attacks [Bel89]). Name-based authentication is even weaker since it requires a host to trust the the network's name service infrastructure. By corrupting the name server's data, an attacker will be able to subvert name-based authentication mechanisms [CERT96b].

Cryptographic techniques provide a stronger basis for host-host authentication. They rely on the possession of some *secret* key. Since it is not feasible for each host to store keys for every other host (requiring $O(n^2)$ keys), a commonly adopted solution is to have a trusted third-party. In a secret-key based cryptographic system, a Key Distribution Center (KDC) shares a *secret* key with each host and can act as an intermediary in the authentication process. In a public-key based cryptographic system, a Certificate Distribution Center

(CDC) or a Directory Service (*e.g.,* X.500) can reliably distribute the host's authentic public key. Key distribution mechanisms are surveyed in detail in Section 2.3.

2.2.3 Message Authentication

Two mechanisms for achieving authentication of messages are Message Authentication Codes and Digital Signatures.

2.2.3.1 Message Authentication Codes (MACs)

A MAC is a key-dependent mechanism by means of which one can achieve authenticity without secrecy. Unlike a data checksum which is designed to catch errors due to "noise" in the communication channel, a MAC is a cryptographic checksum intended to resist forgery. This method requires the two parties to share a secret-key. It can either be based on a symmetric cryptosystem or on one-way hash functions.

The DES-CBC-MAC, as defined in FIPS-113 [US 85], is an example of a MAC based on symmetric cryptosystems. The 64-bit residue resulting from the application of the DES-CBC algorithm on the message is used as the message authentication code. The DES-residue is sent to the receiver along with the message. The recipient encrypts the message using the same shared-key and checks if the resulting residue corresponds to the one received from the sender.

A one-way hash function, along with a shared-secret key, can also be used as a message authentication code. This method takes advantage of a one-way hash function's collision/inversion-resistant properties and combines it with the knowledge of a secret key to produce a message authentication code. An example of this method would be take an MD5 hash of the concatenation of the secret key and data.

2.2.3.2 Digital Signatures

A digital signature [NIS92] is an electronic analogue of a written signature in that the digital signature can be used in proving to the recipient that the message was, in fact, signed by the originator. They can also be used to detect unauthorized modifications to data. Digital signatures make use of public-key cryptography. The private key of the sender is used to generate the signature. The signature is verified by the recipient by using the corresponding public key of the sender.

Since public key algorithms are often too inefficient to sign long messages, a one-way hash function is used in the signature generation process to obtain a condensed version of the data (called a message digest). The message digest is then input to the public-key algorithm to generate the digital signature. The digital signature is sent to the recipient along with the message. The recipient generates a hash of the message using the same hash function as the sender and verifies the signature using the sender's public key.

The recipient of the signed data can also use a digital signature in proving to a third party that the signature was in fact generated by the signatory. This feature is known as *non-repudiation.*

2.3 Key Distribution Systems

Security services based on cryptographic mechanisms assume keys to be distributed prior to secure communications. The secure management of these keys is one of the most critical elements when integrating cryptographic functions into a system, since any security concept will be ineffective if the key management is weak. Many key-distribution systems also provide authentication services.

2.3.1 Kerberos

Kerberos [SNS88] is a secret-key based system for providing authentication and key distribution in a networked environment. It was originally designed at MIT as a part of Project Athena. The basic protocol is derived from one originally proposed by Needham and Schroeder[NS78] [DS81]. As use of Kerberos spread to other environments, changes were needed to support new policies and patterns of use [BM90]. To address these needs, the design of Version 5 of Kerberos (V5) began in 1989 [KNT94]. Though Kerberos Version 4 (V4) still runs at many sites, V5 [KN93] is considered to be the standard Kerberos.

Kerberos allows a process (a client) running on behalf of a *principal* (a user or service) to prove its identity to a verifier(an application server) without sending data across the network that might allow an attacker or verifier to subsequently impersonate the principal. This is accomplished without relying on authentication by the host operating system, without basing trust on host addresses, without requiring the physical security of all the hosts on the network, and under the assumption that packets traveling along the network can be read, modified and inserted at will. Kerberos optionally provides integrity and confidentiality for data sent between the client and the server[17].

The Kerberos implementation consists of one or more authentication servers running on physically secure hosts. The authentication servers maintain a database of principals (i.e., users and servers) and their secret keys. Code libraries provide encryption and implement the Kerberos protocol. In order to add authentication to its transactions, a typical network application adds calls to the Kerberos library, which results in the transmission of the necessary messages to achieve authentication.

The authentication process proceeds as follows: A client sends a request to the authentication server (AS) requesting *credentials* for a given server. The AS responds with these credentials, encrypted in the client's key. The credentials consist of (1) a *ticket* for the server

[17] Many applications use Kerberos's function only upon the initiation of a stream-based network connection and assume the absence of any "hijackers" who might subvert such a connection.

and (2) a temporary encryption key (often called a *session key*). The client transmits the ticket (which contains the client's identity and a copy of the session key, all encrypted in the server's key) to the server. The session key (now shared by the client and server) is used to authenticate the client, and may optionally be used to authenticate the server. It may also be used to encrypt further communication between the two parties or to exchange a separate sub-session key to be used to encrypt further communication.

The Kerberos protocol consists of several sub-protocols (or exchanges). There are two methods by which a client can ask a Kerberos server for credentials. In the first approach, the client sends a cleartext request for a ticket for the desired server to the AS. The reply is sent encrypted in the client's secret key. Usually this request is for a ticket-granting ticket (TGT) which can later be used with the ticket-granting server (TGS). In the second method, the client sends a request to the TGS. The client sends the TGT to the TGS in the same manner as if it were contacting any other application server which requires Kerberos credentials. The reply is encrypted in the session key from the TGT (see Figure 2.2).

To verify the identities of the principals in a transaction, the client transmits the ticket to the server. Since the ticket is sent "in the clear" (parts of it are encrypted, but this encryption doesn't thwart replay) and might be intercepted and reused by an attacker, additional information is sent to prove that the message was originated by the principal to whom the ticket was issued. This information (called the *authenticator*) is encrypted in the session key and includes a timestamp. The timestamp proves that the message was recently generated and is not a "replay". Encrypting the authenticator in the session key proves that it was generated by a party possessing the session key. Since no one except the requesting principal and the server know the session key (it is never sent over the network in the clear), this guarantees the identity of the client.

The integrity of the messages exchanged between principals can also be guaranteed using the session key (passed in the ticket and contained in the credentials). This approach provides detection of both replay attacks and message stream modification attacks. It is

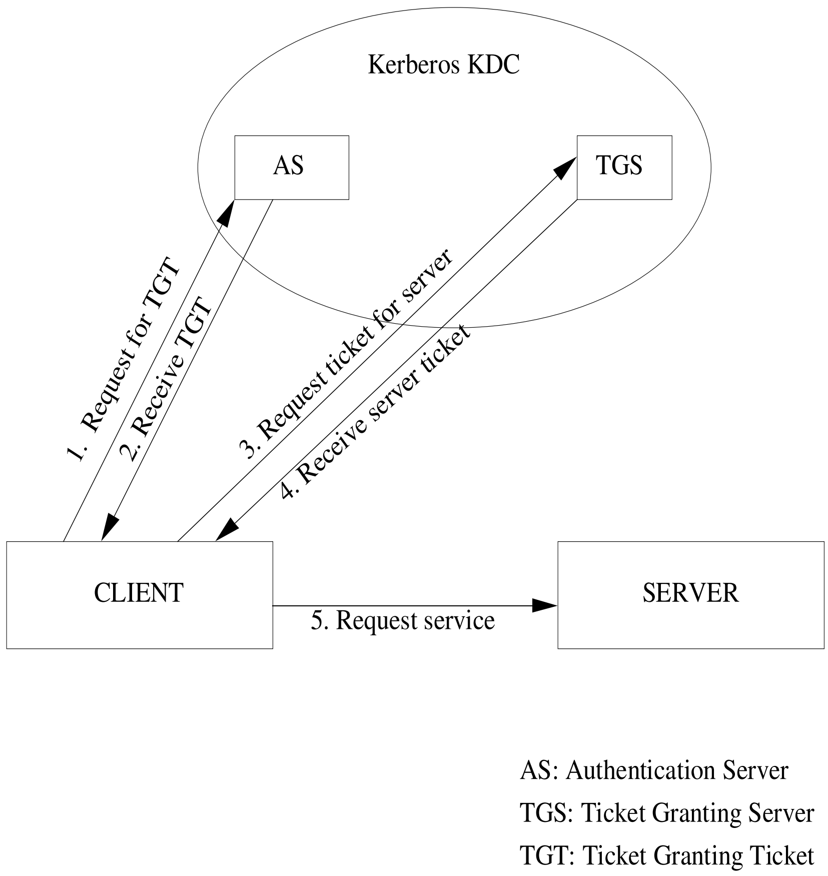

Figure 2.2: Authentication protocol in Kerberos.

accomplished by generating and transmitting a collision-proof checksum (elsewhere called a hash or digest function) of the client's message, keyed with the session key. Privacy and integrity of the messages exchanged between principals can be secured by encrypting the data to be passed using the session key passed in the ticket and contained in the credentials.

The Kerberos protocol is designed to operate across organizational boundaries. A client in one organization can be authenticated to a server in another. Each organization wishing to run a Kerberos server establishes its own *realm*. The name of the realm in which a client is registered is part of the client's name and can be used by the end-service to decide whether to honor a request. By establishing *inter-realm* keys, the administrators of two realms can allow a client authenticated in the local realm to use its authentication remotely. The exchange of inter-realm keys registers the ticket-granting service of each realm as a principal in the other realm. A client is then able to obtain a ticket-granting ticket for the remote realm's ticket- granting service from its local realm. When that ticket-granting ticket is used, the remote ticket-granting service uses the inter-realm key to decrypt the ticket-granting ticket and is thus certain that it was issued by the client's own TGS.

Realms are typically organized hierarchically. Each realm shares a key with its parent and a different key with each child. If an inter-realm key is not directly shared by two realms, the hierarchical organization allows an authentication path to be easily constructed. However, in order to make communication between two realms more efficient, intermediate realms may be bypassed to achieve cross-realm authentication through alternate authentication paths.

Limitations of Kerberos are discussed in detail in [BM90]. Though most apply to Kerberos V4 and have been addressed in V5, a few fundamental ones remain. Kerberos is not effective against password guessing attacks; if a user chooses a poor password, then an attacker guessing that password can impersonate the user. Similarly, Kerberos requires a trusted path through which passwords are entered. If the user enters a password to a program that has already been modified by an attacker (a Trojan horse), or if the path between the user and the initial authentication program can be monitored, then an attacker may obtain sufficient information to impersonate the user. Kerberos can be combined with other techniques, like one-time passwords and public-key cryptography, to address these limitations.

Source code releases for V4 and Beta V5 Kerberos are freely available from the MIT, however, MIT does not officially support these releases. Several companies have taken reference implementations from MIT and provide commercially supported products[18].

[18] Information on the free releases and the supported versions can be obtained by sending a message to info-kerberos@mit.edu.

2.3.2 SPX

SPX [TA91] is an experimental authentication system developed by the Digital Equipment Corporation. It is a component of Digital's Distributed Authentication Security Service (DASS) architecture [Kau93]. SPX performs a function very similar to Kerberos, but unlike Kerberos, it is based on public-key authentication. Each SPX *principal* (user or server) has an RSA key pair as opposed to a DES key in Kerberos.

SPX uses the X.509 syntax for public key certificates and originally envisaged using the X.500 directory service for the distribution of certificates[19]. But since X.500 has not been deployed extensively, the designers invented their own certificate distribution service, called the *Certificate Distribution Center* (CDC). Unlike Kerberos, which stores an association between users and their DES keys, SPX stores each user's certificate and the user's private RSA key encrypted with a DES key (derived from the user's password). Since the user's password is not stored on the CDC in any form and since the RSA private key is stored only in an encrypted form, a CDC does not have to be as "secured" as a Kerberos KDC.

SPX makes use of two types of certificates. Normal certificates, where the certifying authority (CA) issues a certificate for an end-user, and Trusted Authority (TA) certificates where the end-user signs a certificate binding the CA's name to its key. Although SPX uses a hierarchy of certificates, the root of the hierarchy does not need to have it's key widely known and trusted. Instead, each CA below the root maintains a TA certificate that binds the root's key with its name.

When an user logs into a workstation using the SPX system, a random RSA key-pair (called *delegating key-pair*) is generated for him. The workstation sends a request to the CDC indicating that it wants the information that the CDC has stored for the user. The CDC returns the user's certificate, a TA certificate for the certifying authority, and the user's permanent RSA private key (encrypted by the user's password). The user is prompted for

[19] A certificate consists of a user name, a public key, the name of the entity which has signed it and other bookkeeping information.

a password which is used to decrypt the encrypted RSA private key. It then uses the user's permanent public key (contained in the certificate) to verify the TA from which it obtained the user's certificate.

For authentication and key exchange, SPX principals can act in two capacities: (1) the entity seeking to be recognized as authentic (*claimant*), and (2) the entity seeking to authenticate the claimant (*verifier*). To obtain a service, a user principal (claimant) requests a certificate for the server principal (verifier) from the CDC. It then verifies the authenticity of this certificate using the public keys of its trusted authorities (TA). The claimant then generates an authenticating DES key for this session and uses its claimant credentials to make an *authentication token* consisting of:

- its name,
- its ticket (a delegation public key signed by the user's long term private key),
- the authenticating DES key encrypted by the verifier's public key,
- the signature on the enciphered DES key using the delegation private key, and
- an *authenticator*, which is a timestamp and a cryptographic checksum computed using the authenticating DES key.

The token is sent to the verifier which recovers the encrypted DES key by using its private key. It then verifies the validity of the authenticator in the token received. It also checks the signature on the encrypted DES key using the public key in the ticket. At this time the verifier knows that it has a good authentication, but not the identity of the principal who made it. It uses the "claimed" identity to obtain the appropriate certificate from the CDC and finds one that contains the public key that can verify the signature on the ticket. If this certificate can be verified using the verifier's trusted certificate, the claimant's identity is accepted. For mutual authentication, the verifier sends an authenticator (a timestamp and a cryptographic checksum computed on the authenticating DES key) to the claimant.

If the authenticator is good, the claimant is assured of the identity of the verifier (only the verifier knows the private key required to extract the DES key in the first place).

For key revocation, SPX supports the X.509 revocation list mechanism. This provides a dated, signed list of revoked certificate serial numbers as an attribute of each CA and would be read from the CDC along with certificates as needed. This is not included in the initial implementation. Currently, the only form of revocation available is to delete certificates and user entities from the CDC database.

The largest drawback of SPX is that it remains an experimental system that has not been widely deployed. It is likely that the overall performance of a large distributed system using SPX may compare poorly with one using Kerberos, since the RSA operations involved in an SPX authentication exchange are extremely compute intensive.

2.3.3 Diffie-Hellman

The Diffie-Hellman (DH) key exchange (§ 2.1.2.2) provides a mechanism by which two parties can negotiate a shared session key, without fear of eavesdroppers. It does not require the presence of a centralized key distribution system and provides an elegant solution to the key distribution problem. Each party participating in a DH handshake exchanges their public keys with each other and computes the shared DH key. After each side generates the DH key, it is used to encrypt a session key which is then shared between the communicating parties. Each party extracts the session key from the encrypted payload and uses this session key to encrypt data using conventional secret key encryption.

The Diffie-Hellman key exchange protocol can easily be extended to work with three or more users. Each user generates a long-lasting private value X_i and calculate a public value Y_i. These public values, together with global public values for q and α are stored in some central directory. At any time, user j can access user i's public value, calculate a secret key and use that to send an encrypted message to user i. If the central directory is trusted, then this form of communication provides both confidentiality and authentication. Since only i and j can determine the key, no other user can read the message (confidentiality). Recipient i knows that only user j could have created a message using this key (authentication)[20].

One of the problems with the standard Diffie-Hellman protocol is that the public keys exchanged between the participating entities are not authenticated. Due to this, encrypted channels created using Diffie-Hellman key exchanges are vulnerable to *man-in-the-middle* attacks. Though there are techniques for secure transmission of authentication information while using this algorithm (based on the Interlock Protocol [RS84] [BM92] [BM93]), all of them require prior transmission of authenticated data.

A promising development is the use of X.509 certificates, RSA digital signatures, and Secure DNS resource records to authenticate DH public keys. This would facilitate the secure use of Diffie-Hellman.

2.3.4 X.509

X.509 [C.C88b] defines the authentication framework for the CCITT X.500 series of recommendations [C.C88a] [Sta95] that define a *directory service*. In addition, X.509 also defines a generic set of security services that can be adopted by other applications. X.509 was initially issued in 1988 and was subsequently revised in 1993 to address some of the security concerns documented in [Col90].

The X.500 directory may serve as a repository of public-key certificates, where each certificate contains the public key of a user and is signed with the public key of a trusted certification authority. The user certificates in the X.509 scheme (see Figure 2.3) are assumed to be created by some trusted certification authority (CA) and placed in the directory by the CA or by the user. The directory server itself is not responsible for the creation of public keys or for the certification function; it only provides an easily accessible location for users to obtain certificates. The user certificates generated by a CA have the following properties.

[20] However this technique does not protect against replay attacks.

- Any user with access to the public key of the CA can recover the public key of users served by the CA. The public key of the CA is used to verify the digital signature, which guarantees the integrity of the certificate information held for each user.

- No party other than the CA can modify the a user's certificate without it being detected.

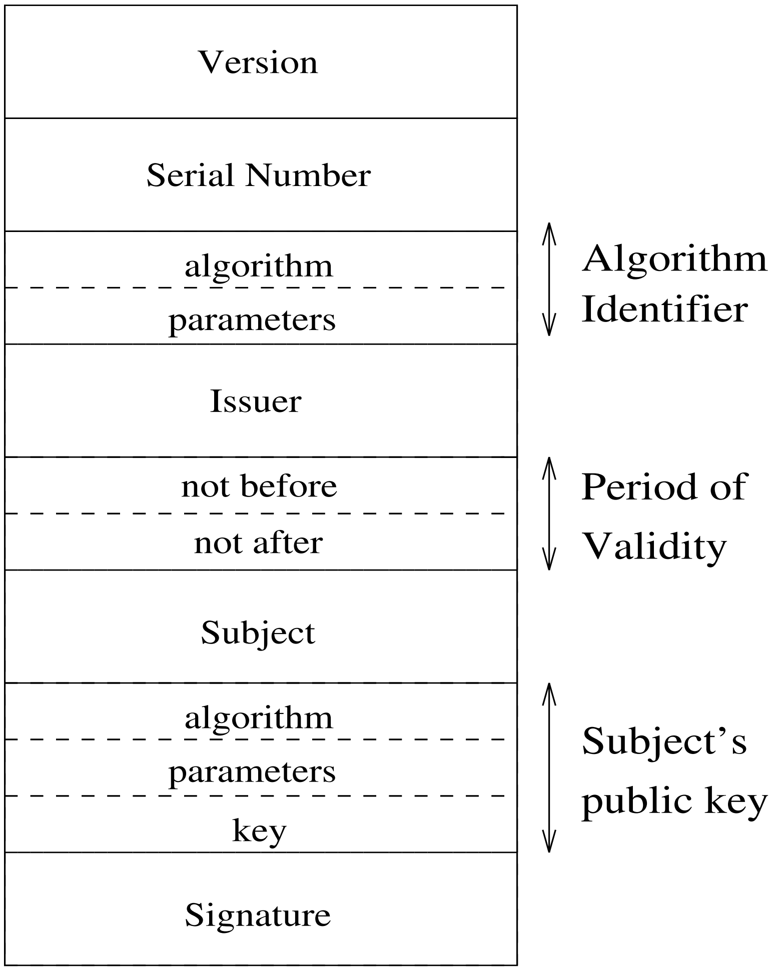

Figure 2.3: An X.509 certificate
Source: [Sta95]

Since it is not practical for a large community of users to subscribe to the same CA, X.509 supports the existence of multiple CAs, each of which securely provides its public key to some fraction of users. In order to reciprocally authenticate each other, users must obtain the complete forward and return *certification paths* from the directory[21]. In order to avoid delays, implementations are designed to cache the results of directory lookups for later reuse.

To handle instances such as the security of the private key of a user or CA getting compromised, X.509 makes provisions for the revocation of certificates. Each CA maintains a list of all revoked (but not expired) certificates issued by the CA, including both those issued to users and to other CAs. Each certificate-revocation list is also signed by the issuer and posted on the directory service. When a user receives a certificate in a message, the user is expected to determine whether the certificate has been revoked by querying the directory service.

X.509 also includes one-way and mutual authentication procedures, based on public-key digital signatures, that are intended for use across a variety of applications.

2.4 UNIX Security Systems

This section surveys how different UNIX security systems provide confidentiality, authentication and integrity services. In order to compare and contrast different systems, an attempt is made to classify systems based on the layer in the OSI model where support for security is provided. Since this research focussed mainly on cryptographic software solutions, data-link layer encrypting devices like the DEC Ethernet encrypting devices are not considered in this survey.

2.4.1 Network layer security

Traditionally, the network layer is the lowest end-to-end layer and is a natural place to provide end-to-end security. Since the current Internet Protocol (IPv4) [Pos81] does

[21] A certification path is a list of certificates that allows a user to get the public key of another user; each item in the list is a certificate for the CA of the next item on the list.

not provide explicit support for cryptographic security, this is achieved by encapsulating encrypted IP datagrams inside other IP datagrams and by providing authentication of the clear-text IP headers and data using one-way hash functions. This approach makes it possible to protect the secrecy and integrity of data in an internetworking environment without affecting higher level protocols and applications. Providing security services at the network layer also avoids the need to replicate the security functions in multiple transport protocols or user programs. Currently, network-level security mechanisms are mainly used to prevent passive-attacks by creating a secure "tunnel" between two hosts separated by an insecure network.

2.4.1.1 swIPe

swIPe was designed by Ioannidis and Blaze [IB93] to complement IP functionality by adding the necessary security features without altering the structure of IP. This is accomplished by encapsulating IP datagrams within IP datagrams of a new IP Protocol type. These new datagrams carry the payload of the original IP datagram, enough header information to reconstruct them at the remote end and any additional security information that may be needed. swIPe provides a clean model for host-to-host encryption and authentication. swIPe system consists of three conceptual entities on top of the ordinary IP mechanisms: the *security processing engine*, the *key management engine*, and the *policy engine*.

The *policy engine* is responsible for examining outgoing packets to determine whether they require swIPe processing, examining incoming packets to determine whether they are to be accepted and deciding the exact nature of processing. Currently swIPe follows a simple policy of deriving security associations on a host-by-host basis.

The *key management* engine establishes the session cryptographic variables used by the security processing engine. It also communicates with the key management engines on other hosts to establish key associations and is responsible for managing any required secure key exchanges. Though swIPe can employ both public-key and private-key methods for key management, it currently supports only static key distribution.

The *security engine* applies the actual authentication, integrity and confidentiality processing on individual datagrams, as controlled by the policy engine and using keys provided by the key management engine.

For output processing, the policy engine examines the IP packets and determines if the packet requires encryption, authentication or both. If so, a swIPe header is generated; the security processing engine obtains the keys from the key management engine, applies the appropriate encryption and authentication algorithms and sends the resulting encapsulated packet for delivery. For input processing, the swIPe policy engine examines the incoming IP datagram; if the packet is already a swIPe packet, it is passed to the security engine, which processes it in a manner analogous to the swIPe processing of outgoing IP packets. Otherwise, the policy engine determines whether the packet is admissible without authentication/encryption and if so, passes it on to IP for regular input processing.

swIPe under UNIX is implemented using a virtual network interface. The part of the implementation that process incoming and outgoing packets are entirely in the kernel; parameter setting and exception handling, however, are managed by user level processes. Though the current implementation is not stable enough for general distribution, it provides a vehicle for experimenting with alternate security algorithms, policies and key-management strategies.

2.4.1.2 SKIP

Simple Key-Management for Internet Protocols (SKIP) [AP95] was designed and developed by Sun Microsystems as a means to provide enhanced security at the network layer for host-host communications. Although its security architecture is similar to the one in swIPe, SKIP also offers an interesting solution to the *key-distribution* problem.

In SKIP, each host (principal) has a certified public key associated with its name. Each host uses a Diffie-Hellman key exchange to share a key which is computable based solely on the knowledge of the other principal's public key certificate. A master key, or *key-encrypting key* (K_{ij}), is derived by taking the low-order key-size bits from the shared key. An individual IP packet is encrypted (or authenticated) using a randomly-generated packet key (K_p). K_p is encrypted by the master key (K_{ij}) and sent *in-band* with the IP datagram to the destination host. The two-level key hierarchy reduces the exposure of the master key, making cryptanalysis more difficult. This also makes it possible to use a new packet encrypting/authenticating key (K_p), should the old one get compromised.

The SKIP system consists of three main components.

- Key management is carried out by a *key-manager daemon* (running in user-space), which maintains a database of certificate information about peers. It is also responsible for the DH key exchange and the computation of the keys (K_{ij} and K_p) required for encryption/authentication.

- A SKIP *data crypt engine* , which in conjunction with the key-manager, provides bulk data encryption/decryption services to kernel clients.

- A SKIP streams module, located between the IP layer and the network interface, maintains a policy engine which decides if the packets are passed in the clear, dropped, or sent to the bulk data crypt engine for encryption/decryption.

All communications between the SKIP kernel and the key manager take place using a pseudo device driver. Solaris[22] SKIP operates entirely transparently requiring no modifications either to existing applications or OS software. During the system boot process, the SKIP streams module is inserted in between the IP layer and the network interface using standard streams operations. The SKIP module intercepts all packets entering and leaving the IP layer.

[22] Solaris is the name of the UNIX operating system running on Sun machines.

SKIP is currently being used in proprietary Sun products like SunScreen, to provide a secure encrypted "tunnel" between two hosts.

2.4.2 Transport layer security

Most of the transport level security packages provide "secure" alternatives for common connection oriented services like `telnet`, `rlogin` and `rsh`. In these packages, traffic between the client and server can be encrypted, thereby eliminating the risk of passive monitoring.

2.4.2.1 SSH

Secure Shell (SSH) [Tat95] is a program developed by Tatu Ylonen of Finland. It is intended as a replacement for `rlogin`, `rsh` and `rcp`. It provides strong authentication and secure communications over insecure channels. Additionally, SSH provides secure X connections and secure forwarding of arbitrary TCP connections. The authentication mechanism in SSH is based on RSA. The user's RSA public key is stored on the server machine under the user's home directory. The user's private key is stored in an encrypted form on the user's local machine. Each server host and the SSH server program on each host[23] have their own RSA key-pair.

The software consists of a server program (`sshd`) running on a server machine and a client program (`ssh`) running on a client machine. When the client initiates a connection, the server accepts the connection and responds by sending back its version identification string. The client parses the server's identification and sends its own identification. The purpose of the identification strings is to validate that the connection was to the correct port, declare the protocol version number used, and to declare the software version used on each side. If either side fails to understand or support the other side's version, it closes the connection.

[23]The purpose of the separate server key is to make it impossible to decipher a captured session if the server machine gets compromised at a later time.

After the protocol identification phase, both sides switch to a packet-based binary protocol to authenticate each other. The server starts by sending its host's public key and the server public key to the client. The client then generates a 256-bit session key, encrypts it using both public keys (received from the server), and sends the encrypted session key, selected cipher type and other information to the server. Both sides then turn on encryption using the selected algorithm and key. The server then sends an encrypted confirmation message to the client.

For user-host authentication, the client tells the server the public key that the user wishes to use. The server checks if this public key is admissible. If so, it generates a random number, encrypts it with the user's public key and sends the value to the client. The client then decrypts the number with its private key, computes an MD5 checksum from the resulting data and sends the checksum back to the server. The server computes the checksum from its copy of the data and compares the checksums. Authentication is accepted if the checksums match.

Though SSH secures the network connection, it does not protect the user against anything that compromises the host in some other way. An attacker with super-user privileges on the SSH host can subvert SSH. Also, SSH assumes that an attacker cannot modify the public keys stored under the user's home directory on the server host.

2.4.2.2 Secure Telnet variants

There are several encrypting `telnet` programs available today. The most standardized one is a Kerberized `telnet` from MIT. However it requires the existence of a full Kerberos environment. Since this requires additional effort, Kerberized `telnet` is not ubiquitous yet.

STEL(*Secure TELnet*) [VTB95], was developed at the University of Milan, Italy. It consists of a client (`stel`) which is run by users and a server (`steld`) which is a standalone daemon running with superuser privileges. STEL is intended to act as a "surrogate" replacement for `telnetd`, `rlogind` and `rshd`. The session keys used for encryption are derived from a Diffie-Hellman(DH) key exchange procedure. STEL uses the Interlock Protocol [RS84] to defeat *man-in-the-middle* attacks on the DH key exchange. Upon establishing a secure channel, the user has a variety of methods (SecurID, S/Key and standard UNIX passwords) to authenticate himself. For encryption, currently STEL supports the DES, 3-DES and IDEA algorithms. An interesting feature of STEL is that it provides support for an S/Key key distribution center, which allows the S/Key keys to be centralized in a single place[24]. STEL does not currently use standard telnet option negotiation which would have made it more interoperable with other secure telnet implementations.

AT&T Bell Labs has developed an encrypting `telnet` [BB95] program for internal use. Like STEL, it uses Diffie-Hellman key-exchange to create a session-key. However, it uses challenge/response devices to authenticate the key. Key exchange, encryption, and challenge/response parameters are all negotiated and transmitted via extensions to the `telnet` options mechanism. `telnet` and `telnetd` first determine that they have encryption capability (using the WILL/WONT, DO/DONT protocol) and then negotiate keys as sub-options using the SEND/IS mechanism. In order to prevent an active attacker from hijacking the session in progress and forcing a return to cleartext or a change to a different key by injecting bogus DO/DONT, WILL/WONT sequences, the key exchange protocol can occur at most once per session. Once encryption has commenced, `telnetd` refuses to revert to cleartext mode or change keys. DES in CFB mode is used as the encryption algorithm.

SRA, from Texas A&M University [SHS93], is another secure `telnet` package. It is based on Secure-RPC [Mic88] and uses Diffie-Hellman key exchange to negotiate a session key. This session-key is used only to transmit the user's login and password; the remainder of the session is not protected. Since the modulus size for the DH key exchange is just 192 bits, the key-exchange procedure is vulnerable to cryptanalysis.

[24] S/Key currently requires a separate key database for each machine.

2.4.2.3 Kerberos 'r' commands

The Kerberos distribution from MIT contains enhanced versions of `rlogin` and `rsh` which authenticate users by taking advantage of the Kerberos infrastructure. The client (`rlogin/rsh`) uses the ticket-granting ticket for the user (obtained by using `kinit`) to get a ticket from the KDC for the desired service (`host` [25]) on the server machine. It then presents this ticket to the '`host`' service on the destination host which allows the connection if the ticket is authenticated. Kerberos Version 5 (beta) supports encryption of `rsh` and `rlogin` sessions.

2.4.3 Session layer security

Though the transport layer is the natural place to secure individual network connections, application level firewalls and multi-hop login sessions (*e.g.* via terminal servers) sometimes make it necessary to implement security at a higher layer. Also, some protocols like Secure-RPC are designed to operate over multiple transport mechanisms. This makes the session layer a suitable place to secure networked communications.

2.4.3.1 Sun's Secure-RPC

Sun Microsystems's RPC (ONC RPC) protocol [Mic88] is based on the remote procedure call model [Bir84]. In the remote procedure call model, one thread of control logically winds through two processes: the caller's (client) process and a server's process. The caller process first sends a call message to the server process and waits (blocks) for a reply message. Once the reply message is received, the results of the procedure are extracted and caller's execution is resumed. On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message.

[25] Kerberos Version 5 uses a common service 'host', for all the Berkeley 'r' commands.

Since remote procedure calls can be transported over insecure networks, provisions for authentication of caller to service and vice-versa are provided as a part of the ONC RPC protocol. Security and access control mechanisms can be built on top of this message authentication. There are currently three authentication *flavors* that can work with ONC RPC.

- UNIX authentication
- DES authentication
- Kerberos Version 4 authentication

UNIX authentication (called *AUTH_UNIX* or *AUTH_SYS*) attempts to provide security based on the UNIX user-id (`uid`) and group-id (`gid`) of the client process. This form of authentication can be easily faked since there are no mechanisms to provide cryptographically secure authentication.

Unlike UNIX authentication, **DES authentication** (called *AUTH_DES*) does have a verifier so that the server can validate the client's credential and vice-versa. The contents of this verifier is primarily an encrypted timestamp. The server can decrypt this timestamp and if it is close to the real time[26], the server knows that the client must have encrypted it correctly. The only way the client could encrypt it correctly is to know the *conversation key* of the RPC session. The conversation key is a DES key which the client generates and passes to the server in its first RPC call. The conversation key sent to the server is encrypted using a shared DES-key (derived from a Diffie-Hellman (DH) public key exchange[27] between the client and server). The client too must check the verifier returned from the server (the encrypted timestamp it received from the client, minus one second) to be sure it is legitimate. If confidentiality is required, the conversation key can be used to encrypt messages between the client and server. It should be noted that the public keys used in the Secure-RPC DH

[26] The client and the server need the same notion of the current time in order for this to work.

[27] The base and 192-bit modulus for the Diffie-Hellman key exchange are constant and are 'hard-coded'.

key exchange are unauthenticated and hence are vulnerable to active attacks.

Authentication using the **Kerberos Version 4** protocol (called *AUTH_KERB4*) was added recently to the ONC RPC specification[28]. This mode of authentication requires the hosts, on which client and server RPC processes run, to be members of a Kerberos V4 realm. Conceptually, AUTH_KERB4 based authentication in Secure-RPC is very similar to AUTH_DES authentication. The major difference is that Kerberos-based authentication takes advantage of the fact that Kerberos tickets have the client's name and conversation key encoded in them, thereby avoiding the need for a separate key exchange process to derive a conversation key.

2.4.3.2 Encrypted Session Manager (ESM)

The Encrypted Session Manager (`esm`) [BB95], from AT&T Research Labs is a system which provides privacy and confidentiality at the session layer. Although the `telnet` protocol is a natural place to define network session security, it is not always possible to run `telnet` directly between arbitrary trusted endpoints. Application level firewalls, multi-hop login sessions (*e.g.,* via terminal-servers), and non-TCP/IP connections like `kermit` and `tip`, sometimes make it necessary to have security at a higher layer than the transport layer.

`esm` exploits the BSD "pseudo-tty" mechanism to provide a layer under which everything between the user's local and remote login sessions are transparently encrypted and decrypted. When first invoked from an interactive shell, `esm` provides a transparent pseudo-terminal session on the local machine, passing all the I/O from the terminal session to the shell session. When invoked in the "server mode" from within an existing ESM session (the second session is typically on a remote machine), the two ESM processes automatically encrypt all traffic passed between them. The encryption key is derived via a 1024-bit

[28] Authentication support in Secure-RPC using Kerberos is not specified in RFC 1057.

Diffie-Hellman key exchange[29] initiated by the remote server. Since the typical user-to-host session traffic is "stream-oriented", `esm` uses 3-key triple DES in Cipher Feedback (CFB) mode for encryption/decryption. ESM is distributed as a part of the Cryptographic File System (CFS) package available from AT&T Bell Laboratories[30].

2.4.4 Application layer security

Security is provided at the application layer, when the underlying layers cannot provide the requisite security services to the application. S/Key is a "one-time password" system that protects user passwords against passive attacks. PEM/PGP are different mechanisms used for secure electronic mail over the Internet and provide excellent case studies of public-key cryptographic systems. TCP-Wrappers is a package which uses host names/addresses to authenticate service-requests coming from remote hosts. Kerberos offers code-libraries and application programming interfaces that enable users to develop new applications which can take advantage of the security services offered by the Kerberos infrastructure. The following subsections describe how these applications provide enhanced security services by building up on the services offered by other security systems.

2.4.4.1 S/KEY

The S/KEY authentication system is a scheme that protects user passwords against passive attacks. With the S/KEY system, only a "single-use" password ever crosses the network. The user's secret pass-phrase never crosses the network at any time, including during login or when executing other commands requiring authentication such as the UNIX commands `passwd` or `su`. S/Key can be easily and quickly added to almost any UNIX system without requiring any additional hardware.

[29] The public keys used in the Diffie-Hellman key exchange are not authenticated, thereby making the exchange vulnerable to active attacks.

[30] Matt Blaze (mab@research.att.com) is the contact person at AT&T Bell Labs.

The S/KEY system is based on the MD4 Message Digest algorithm[31]. The S/KEY one-time passwords are 64-bits in length. This is believed to be long enough to be secure and short enough to be manually entered when necessary. The S/KEY secure hash function consists of applying MD4 to a 64-bit input and folding the output of MD4 (128-bits long) with exclusive-OR to produce a 64-bit output.

There are two sides to the operation of the S/KEY one-time password system. On the client side, the appropriate one-time password must be generated. On the host side, the server must verify the one-time password and permit the secure changing of the user's secret pass-phrase.

The client's secret pass phrase may be of any length and should be more than eight characters[32]. Since the S/KEY secure hash function described above requires a 64-bit input, a preparatory step is needed. In this step, the pass phrase is concatenated with a seed that is transmitted from the server in clear text. This non-secret seed allows a client to use the same secret pass phrase on multiple machines (using different seeds) and to safely recycle secret passwords by changing the seed. A unique sequence of one-time passwords is produced by applying the secure hash function multiple times to the output of the preparatory step (called S). That is, the first one-time password is produced by passing S through the secure hash function a number of times (N) specified by the user. The next one-time password is generated by passing S though the secure hash function N-1 times. An eavesdropper who has monitored the transmission of a one-time password would not be able to generate any succeeding password because doing so would require inverting the hash function. The one-time password generated by the above procedure is 64 bits in length. Entering a 64-bit number is a difficult and error prone process. The one-time password is therefore converted to, and accepted as, a sequence of six short (1 to 4 letter) English words. Each word is chosen from a dictionary of 2048 words. Interoperability requires at

[31]The Naval Research Labs (NRL) has generated a functionally similar system, "OPIE", with support for both MD5 and MD4.

[32]To make dictionary attacks more difficult.

all S/KEY system hosts and calculators use the same dictionary.

A function on the host system that requires S/KEY authentication is expected to issue an S/KEY challenge. This challenge give the client the current S/KEY parameters - the sequence number and seed. The host system has a file (on the UNIX reference implementation, it is `/etc/skeykeys`) containing, for each user, the one-time password from the last successful login, To verify an authentication attempt, it passes the transmitted one-time password through the secure hash function[33] one time. If the result of this operation matches the stored previous one-time password, the authentication is successful and the accepted one-time password is stored for future use[34].

2.4.4.2 PEM

PEM is the Internet Privacy-Enhanced Mail standard adopted by the Internet Architecture Board (IAB) to provide secure electronic mail over the Internet. The PEM protocols provide for encryption, authentication, message integrity and key management.

PEM is compatible with the X.509 authentication framework. The key-management infrastructure establishes a single root for all Internet certification. The Internet Policy Registration Authority (IPRA) establishes global policies that apply to all certification under this hierarchy. Under the IPRA root, are the Policy Certification Authorities (PCAs), each of which establishes and publishes its policies for registering users or organizations. Each PCA is certified by the IPRA. Beneath the PCAs, CAs certify users and subordinate organizational entities.

A PEM message consists of the user's message, in either plaintext or ciphertext, and associated PEM headers. For message integrity and authentication, PEM uses a message integrity code (MIC) which is calculated over the entire message. The algorithm used to calculate the MIC (currently PEM supports MD2 and MD5) is specified in the PEM header.

[33] Clients and hosts must use the same secure hash function to interoperate.
[34] This verification technique was first suggested by Leslie Lamport [Lam81].

Though PEM supports both asymmetric or symmetric encryption techniques for authentication, the asymmetric approach is much more common. For 'asymmetric-encryption' authentication (RSA is the only algorithm supported), the MIC is encrypted with the originator's public key, forming a digital signature. The recipient verifies the signature using the originator's public key, which is either obtained via the X.500 directory service or by a signed certificate included in the message header. For 'symmetric-encryption' authentication (currently DES and 3-DES are supported), the MIC is encrypted with a secret symmetric key shared by the originator and recipient.

Message encryption is an optional service for PEM messages. If a message is encrypted, the message header includes a header that indicates the encryption algorithm used. Currently only DES-CBC mode encryption is supported. PEM makes a distinction between data encrypting keys (DEKs) and interchange keys (IKs). An IK can either be a symmetric key 'securely' shared between the originator and recipient, or a public/private key pair with the public key being 'reliably' shared between the originator and recipient. A DEK is a one-time session key used to encrypt message text.

Despite its name, PEM is not a program for exchanging private e-mail. RIPEM, written by Mark Riordan, is a popular implementation of the PEM standard.

2.4.4.3 PGP

Pretty Good Privacy (PGP) is a confidentiality and authentication service, originally designed by Phillip Zimmermann, which can be used for secure electronic mail. It uses IDEA for data encryption, RSA for key management and digital signatures, and MD5 for message integrity.

PGP's random public keys use a probabilistic primality tester and get their initial seeds from measuring the user's keyboard latency while typing. PGP generates random IDEA keys by distilling a secret seed value and a timestamp through IDEA and using the output as the key.

An interesting aspect of PGP is its distributed approach to key management. There are no certification authorities; PGP instead uses a "*a web of trust*". Every user generates and distributes his own public key. Users sign each other's public keys, creating an interconnected community of PGP users. PGP does not specify a policy for establishing trust; instead it provides mechanisms for associating trust with public keys and leaves it to the user to decide the degree of trust. Each user keeps a collection of signed public keys in a file called a *public-key ring*. Each key in the ring has a key-legitimacy field that indicates the degree to which the particular user trusts the validity of the key. Since key distribution is entirely *ad-hoc*, revocation of keys is a problem in PGP. It is impossible to guarantee that no one will use a compromised key.

PGP-encrypted messages have layered security. The only thing a cryptanalyst can learn about an encrypted message is who the recipient is. Only after the recipient decrypts the message does he learn who signed the message, if it is signed. In contrast, PEM reveals quite a bit of information about the sender, recipient and message in the unencrypted header.

2.4.4.4 Kerberos

The Kerberos distribution includes code-libraries which provide APIs for adding authentication, integrity and confidentiality to an application[35] (section 2.3 gives a detailed description of the Kerberos authentication process). The KRB_SAFE message may be used by clients requiring the ability to detect modifications of messages they exchange. It achieves this by including a keyed collision-proof checksum of the user data and some control information. The checksum is keyed with an encryption key (usually the last key negotiated via subkeys, or the session key if no negotiation has occurred). The KRB_PRIV message may be used by clients requiring confidentiality and the ability to detect modifications of exchanged messages. It achieves this by encrypting the messages and adding control information.

An interesting development is the specification of a standard applications program-

[35]Adding Kerberos support to an application is called *Kerberizing*.

ming interface called the Generic Security Services Applications Programming Interface (GSS-API) which can interact both with Kerberos and SPX. Programmers who write their applications to conform to the GSS-API (§ 3.1.2) will be able to compile them either with SPX or with Kerberos with minimal programming changes required to switch from one system to another. The GSS-API specifications are still evolving since the initial document, RFC-1508 [Lin93], was identified to have some deficiencies.

2.4.4.5 TCP-Wrappers

TCP-Wrappers is a TCP/IP daemon wrapper package written by Wietse Venema of Eindhoven University of Technology, Netherlands. It is an example of a system in which host-host authentication of network services gets implemented at the application level. It can be used to monitor and filter incoming requests for the `telnet, ftp, systat, finger, rlogin, rsh, exec, tftp, talk,` and other network services spawned by a "super server" such as `inetd`. It supports both 4.3 BSD-style sockets and System V.4-style Transport Layer Interface (TLI). The wrappers report the name of the client host and of the requested service; it does not exchange information with the client or server applications and impose no overhead on the actual conversation between the client and server applications. TCP-Wrappers also supports optional features such as: access control to restrict what systems can connect to what network daemons; additional protection against hosts that pretend to have someone else's host name or host address.

2.5 Distributed Computing Environment

Open Software Foundation's Distributed Computing Environment (DCE) provides a broadly supported, vendor-neutral[36] infrastructure for building distributed applications [Kha94]. The services provided by DCE include support for RPC, a directory service,

[36] DCE has been implemented on multiple flavors of UNIX and also on VMS.

security services, and a distributed file system. This section looks at the security services provided by DCE.

In DCE, authentication, data integrity, and confidentiality are provided by a slightly modified version of Kerberos Version 5 (Kerberos is discussed in detail in Section 2.3). Kerberos by design does not address the problem of *authorization* (*i.e.,* does an authenticated client have the right to perform the service it is requesting). DCE provides this service with *access control lists (ACLs).* When a service receives a request, that request typically contains the *privilege attribute certificate* (PAC) of the requestor. This PAC identifies who made the request and what groups he belongs to. A component of the server called the ACL manager compares the information requestor's PAC with the entries on the ACL of the desired object. Access is allowed if they match correctly.

In typical distributed environment, most clients perform most of their communication with only a small set of servers. This locality of reference is made explicit in DCE with the notion of a *cell.* A cell has no fixed size; a cell's size, both in the number of machines and in geographical extent, is determined by the people administering the cell. Although DCE allows communication between clients and servers in different cells, it optimizes for the more common case of intra-cell communication. To obtain intra-cell binding information for services, DCE provides a Cell Directory Service (CDS). When a client wishes to locate a server within its own cell, it *imports* that information from the CDS server. When a server wishes to make its binding information available to clients, it *exports* that information to one of its cell's CDS servers. Different cells can be linked together via existing global directory service infrastructures like DNS and X.500.

Every cell runs at least one security server process. The services supported by a security server include

- Registry service: The cell's registry database stores entries for all the cell's users (principals), all of its groups and all of its organizations.

- Key distribution service: This is essentially Kerberos's authentication and ticket granting services.

- Privilege service: This service is responsible for providing privilege attribute certificates (PACs).

The security server must run on a secure machine, since the registry on which it relies contains a secret key, generated from a password, for each principal in the cell. Although the keys are encrypted while stored on disk, the key to decrypt them is also stored on the machine, since DCE requires access to each principal's key.

Since the DCE effort is supported by virtually every vendor, it is likely to become the standard platform for distributed applications in a multi-vendor environment.

2.6 Security in IP*v6*

The Internet Engineering Task Force (IETF) started its effort to select a successor to IPv4 [Pos81] in late 1990 when projections indicated that the Internet address space would become an increasingly limiting resource. Several parallel efforts then started exploring ways to resolve these address limitations while at the same time providing additional functionality. The Internet Protocol Next Generation (IPng) was recommended by the Area Directors of the IPng Working Group of the IETF in July 1994 [BM95]. In November 1994, the Internet Engineering Steering Group (IESG) approved the recommendation and made the protocol a Proposed Standard with the formal name IPv6 [DH96]. The changes from IPv4 to IPv6 fall primarily into the following categories:

- Expanded Addressing Capabilities
- Header Format Simplification
- Improved Support for Extensions and Options
- Flow Labeling Capability

- Enhanced Security

This section describes the proposed security enhancements provided by IPv6.

There are two cryptographic security mechanisms for IP. The first is the *Authentication Header (AH)* [Atk95a] which provides integrity and authentication without confidentiality. The second is the *Encapsulating Security Payload (ESP)* [Atk95b] which always provides confidentiality, and (depending on algorithm and mode) may also provide integrity and authentication. The two IP security mechanisms may be used together or separately. The use of these headers will increase the IP protocol processing costs in participating end systems and also will increase the communication latency[37]. However, these costs are expected to be offset by the enhanced security provided by these mechanisms.

The concept of a *Security Association* is fundamental to both the IP Encapsulating Security Payload and the IP Authentication Header. A Security Association normally includes the authentication algorithm and algorithm mode being used with the AH; the encryption algorithm, algorithm mode and transform being used with the ESP; the key(s) used for the encryption and authentication algorithms; and other optional parameters. [Atk95c] specifies the all the parameters involved in a Security Association. The combination of a given Security Parameter Index (the SPI is a 32-bit pseudo-random value identifying the security association for an IPv6 datagram) and Destination Address uniquely identifies a particular Security Association.

2.6.1 Authentication Header (AH)

The AH is designed to provide integrity and authentication without confidentiality to IP datagrams. It might also provide non-repudiation, depending on the cryptographic algorithm being used. The absence of the "confidentiality-requirement" for the AH ensures that the implementations of the AH will be widely available on the Internet, even in locations

[37] This is due to the time taken for computing and verifying the AH and the time taken for encryption and decryption of the payload if ESP is used.

where the export, import, or use of encryption is regulated.

The authentication data carried by the IP Authentication Header is calculated using a message-digest algorithm either by encrypting the message-digest or "keying" the message-digest directly. It is calculated using all the fields in the IP datagram which do not change in transit. Fields or options which need to change in transit are considered to be zero for the calculation of the authentication data. All IPv6-capable hosts are *required* to implement the IP Authentication Header with the MD5 algorithm using a 128-bit key.

2.6.2 Encapsulating Security Payload (ESP)

The IP Encapsulating Security Payload (ESP) seeks to provide confidentiality and integrity by encrypting the data to be protected and placing the encrypted data in the data portion of the IP Encapsulating Security Payload. It may also provide authentication, depending on which algorithm and algorithm mode are used. The IP Authentication Header (AH) may be used in conjunction with ESP to provide authentication.

The first component of the ESP payload consist of the unencrypted field(s) of the payload. The second component consists of encrypted data. The field(s) of the unencrypted ESP header inform the intended receiver how to properly decrypt and process the encrypted data. The encrypted data component includes protected fields for the ESP security protocol and also the encrypted encapsulated IP datagram.

There are two modes under ESP. In *Tunnel-Mode* ESP, the original IP datagram is placed in the encrypted portion of the Encapsulating Security Payload and that entire ESP frame is placed within a datagram having unencrypted IP headers. The information in the unencrypted IP headers is used to route the secure datagram from origin to destination. The second mode, which is known as the *Transport Mode*, encapsulates an upper-layer protocol (*e.g.,* TCP or UDP) inside ESP and then prepends a cleartext IP header.

2.6.3 Key Management in IPv6

There are two keying approaches to IP. The first approach, called *host oriented keying*, has all users on $Host_A$ share the same key for use on traffic destined for all users on $Host_B$. The second approach, called *user-oriented keying*, lets $User_i$ on host A have one or more unique session keys for its traffic destined for host B; such session keys are not shared with other users on host A.

Key management in IP*v6* is made orthogonal to the AH and ESP security protocol mechanisms. This decoupling[38] permits several different key management mechanisms to be used. More importantly, it permits the key management protocol to be changed or corrected without unduly impacting the security protocol implementations. Work is ongoing in the IETF to specify a standard key management protocol. Photuris [Wil95], SKIP [Ash95] and ISAKMP [Dou95] are three proposals for an Internet Standard key management protocol.

[38]The only coupling between the key management protocol and the security protocol is with the Security Parameter Index (SPI).

Chapter 3

Design Issues in Crypto Systems

In the design of a "secure" distributed application, a number of issues have to be considered and carefully studied. The mechanism adopted for key management (§ 2.3) is very important. Other factors that need to be considered are the choice of good encryption algorithms and a generic cryptographic API to use them, a safe method to generate random numbers to be used as session keys, the optimal key-lengths required to be protected against brute-force attacks on the encryption algorithm used, and a mechanism to generate the large numbers required in public key cryptography. These issues are discussed in the following sections in this chapter.

3.1 Crypto API's

Until recently, the integration of cryptographic functionality into application software has required that developers tightly couple the application to the cryptographic module. This approach forces each new combination of application and cryptography to be treated as a distinct development and does not facilitate interoperability among different cryptographic implementations. The use of a standardized Cryptographic Application Programming Interface (CAPI) provides the following advantages:

- Application programmers will need to learn only one set of cryptographic service calls for multiple cryptographic applications.

- Cryptographic modules from different software developers, which conform to this interface standard, may be interfaced to a given application without requiring modification to the application program.

The compelling case for a modular cryptographic interface has given rise to the development of numerous proposed CAPI standards. These proposals include the GSS-API(IETF), the GCS-API(X/Open), and Cryptoki(RSA). Each of these CAPIs was designed to support significantly different levels of security awareness; with the GSS-API requiring very little cryptographic awareness and Cryptoki requiring extensive knowledge of underlying cryptography. A cross-organizational team from the NSA developed the following criteria [Tea95] to evaluate different CAPIs.

Algorithm Independence: The CAPI must provide access to a large number of choices to current and future cryptographic algorithms. This property gives an application access to any cryptographic algorithm supported by the underlying cryptomodule, enhancing interoperability.

Application Independence: The CAPI must provide cryptographic service to a wide variety of applications being written today, and to many new ones in the future. This will give the CAPI widespread use and longevity. An application-independent CAPI should be equally suitable for connection-oriented applications (*e.g.* file transfer) and for store-and-forward applications (*e.g.* electronic mail).

Cryptomodule Independence: In providing its cryptographic service, a CAPI should be able to support any current or future cryptomodule with equal ease. The application should not need to know the specifics of the underlying cryptographic implementation; *e.g.*, the application need not know whether or not the cryptography is provided in hardware or software.

Degree of Cryptographic Awareness: A complete CAPI should support both cryptographic-aware and cryptographic-unaware applications. For a majority of applications, a minimal degree of cryptographic knowledge is needed by the developer. However, for applications like key management, the programmer is expected to have a higher degree of cryptographic knowledge.

Legacy support: Legacy support is defined as providing the extensibility, so that current cryptography can be supported in future systems.

After evaluating different CAPIs based on these criteria, the NSA team concluded that there was no single CAPI which adequately met all requirements. Rather than recommending the selection of single CAPI, they suggested that a combination of different widely accepted proposals be adopted. These proposals include the GSS-API, GCS-API, and Cryptoki. The main difference between these CAPIs is in the amount of cryptographic knowledge required by both the application programmer and the user.

> "GSS-API provides the safest interface, but the most limited capability to manipulate cryptography. Cryptoki and GCS-API provide applications with more capabilities to manipulate the cryptography that increases the ability of the application to misuse the interface. Since the majority of applications will be cryptographic unaware, the bottom-line recommendation is for applications to use GSS-API. Only if the application absolutely needs to be cryptographic aware should GCS-API or Cryptoki be considered for use by the application" [Tea95].

3.1.1 Cryptographic Service Calls

A well-designed cryptographic API should be able to perform all standard cryptographic operations without strict dependency on any algorithm or I/O mechanism. This would enable an application programmer to incorporate cryptography into his application without worrying about the specifics of any algorithm. A typical set of CAPI routines would include the following.

Set_Context(): This function initializes the security context for a specific encryption algorithm. The security context consists of the algorithm being used, the cipher mode of the algorithm, the key used for encryption/decryption and the initialization vector(s) used. The security context and the status are returned to the calling program.

Set_Context(context, algorithm, key, key_len, initialization_vector, status);

Encipher(): This routine enciphers a given length of plaintext data using a specific algorithm and mode. The security context for the algorithm should be set prior to invoking this function. The ciphertext, the length of the ciphertext and the status are usually returned to the calling program. For a block cipher, depending on the mode of operation, some padding may be added to the input plaintext data. Hence the length of the ciphertext may be greater than the length of the plaintext.

Encipher(context, mode, ciphertext, ciphertext_len, plaintext, plaintext_len, status);

Decipher(): This call decrypts the ciphertext of given length in the specified algorithm and mode using the appropriate key. As in Encipher(), security context for the algorithm should be set prior to invoking this function. The decrypted plaintext, the length of the plaintext and the status are returned to the calling program.

Decipher(context, mode, plaintext, plaintext_len, ciphertext, ciphertext_len, status);

Compute_Hash(): This routine computes the data authentication code (DAC) on a data of given length using a specified algorithm and key. The computed DAC and resulting status are passed back to the calling program.

Compute_Hash(key, algorithm, data, data_len, dac, status);

Verify_Hash(): This function first invokes Compute_Hash() to compute the DAC on the given data using the specified algorithm and key. The resulting DAC is compared with the input DAC to check if they match. If the DACs are identical, verification succeeds. The result of the verification is returned to the calling program.

Verify_Hash(key, algorithm, data, data_len, dac, status);

Generate_Key(): This call generates a pseudo-random number of specified length. A seed value can be used to reset the random number generator (rng) to a random starting point. The result of this function can be used as a key for encryption/decryption.

Generate_Key(rng, seed, len, random_key, status);

Routines having the above functionality were used to incorporate cryptographic services into secure PVM. In this research, the cryptographic libraries provided with Kerberos, SSH, Cryptolib, RSAref, Crypto++ and SSLref were studied[1].

3.1.2 Generic Security Service Application Programmer Interface

Since this research required access to cryptographic functions only at a relatively high level, the GSS-API seemed to be the appropriate choice . The GSS-API is a portable set of functions that allow application programmers to create secure applications without learning the particular elements of any of the underlying authentication systems available to them. The GSS-API can provide generic authentication, encryption and integrity checking services to network applications. These applications include those built on the client-server model and those that use peer-to-peer communications.

The GSS-API provides a library into which both sides of an application can make calls for basic security services. The side initiating the communication (usually the client) begins

[1]Some of these libraries have newer implementations available now.

by acquiring credentials for its system according to whatever mechanism available to it[2]. All GSS-API's functionality is based on the ability of entities to prove that they are who they say are – i.e., authenticate themselves. When the client side of the application initiates contact with another entity, such as the server, it does so using a GSS-API function that allows the programmer to specify the particular security options that the connection should use. The authenticated connection between two entities is called a *context*. Having established such a context, the two entities can then use the GSS-API to process data for exchange; GSS-API processing always includes integrity checking and can also include encryption. The protocol also offers methods for extracting logging information from the transactions, so that the application programmer can have a record of the various interactions that take place.

After the two entities complete their exchange of data, either can destroy the context on its end. This action creates a message or *token*. The entity that has destroyed its end of the context then passes the token to the other entity, which indicates that the other end should destroy its side of the context.

Quite a few deficiencies have been identified in the initial GSS-API specification published in RFC-1508 [Lin93]. The IETF working group on Common Authentication Technologies (CAT) is working on addressing these deficiencies.

3.1.3 SSH's Crypto-API

One of the goals of this research was to compare the performance of different encryption algorithms. The GSS-API seemed to be the most suitable CAPI, since it requires minimal knowledge of underlying cryptography and has interfaces for both private-key and public-key cryptosystems. However, the GSS-API was not chosen for this research because of the following reasons.

[2]The GSS-API does not provide the security mechanisms; it relies on existing mechanisms (*e.g.,* Kerberos, X.509) that are available.

1. The initial GSS-API presented in RFC1508 [Lin93] is not complete and its specifications are still evolving.

2. The lack of implementations of different algorithms supported by this API[3].

3. Concerns about excessive degradation in performance for applications using the GSS-API[4].

For this research, the crypto-API in the SSH[5] distribution was used. This API implementation includes support for a number of different algorithms and offers a reasonably generic and portable interface to the underlying crypto-libraries. The algorithms supported by the current implementation are:

DES: DES is used in CBC mode. The key is taken from the first 8 bytes of the session key[6]. The least significant bit of each byte is ignored resulting in 56 bits of key data.

3-DES: This consists of three independent DES-CBC ciphers. The data is first encrypted with the first cipher, then decrypted with the second cipher, and finally encrypted with the third cipher. All these operations are performed in CBC mode. The key for the first and third ciphers are taken from the first 8 bytes of the session key; the key for the next cipher from the next 8 bytes of the session key.

IDEA: The key is taken from the first 16 bytes of the session key. IDEA is used in CFB mode[7].

RC4: This algorithm is the alleged RC4 cipher posted to the Usenet in 1995. It is widely believed to be equivalent with the original RSADSI RC4 cipher. The first 16 bytes of

---

[3] The Kerberos V5 beta distribution has a GSS-API library for DES. However, I have not come across a GSS-API implementation of IDEA.

[4] Jaspan [Jas93] reports slowdowns of a factor of ten when using the GSS-API to implement secure remote procedure calls; though his study does not reveal why performance is reduced [Fos95].

[5] The URL for the SSH distribution is http://www.cs.hut.fi/ssh.

[6] The minimum length of the session keys used in this research is 16 bytes.

[7] Historically, most of the publicly available IDEA implementations have been in CFB mode, starting with the IDEA implementation in the PGP distribution. A secret-key cipher operating on the same block size in CBC and CFB modes has comparable performance [Sch96].

the session key are used as the key.

The generic design of the SSH crypto-API makes it also possible to add cryptomodules for other encryption algorithms. The application can choose the suitable algorithm based on the speed or strength of encryption required. Details about the relative performance of the different encryption algorithms specified above can be found in Section 3.5.

3.2 Random Number Generation

Security systems today are built on increasingly strong cryptographic algorithms that foil pattern analysis attempts. However, the security of these systems is dependent on generating secret quantities which are used as passwords, session keys, initialization vectors, and nonces[8] in cryptographic protocols. Since the security of the protocol depends crucially on the unpredictability of the secret quantities, it is vital that these quantities be generated from an unpredictable random-number source. In this research, random numbers are used in the generation of the public/private key pairs used in the Diffie-Hellman key exchange and the session keys used for message encryption (§ 4.2.7).

The only truly random number sources are those related to physical phenomena such as the rate of radioactive decay of an element or the thermal noise of a semiconductor diode. Barring the use of external devices, computer programs that need random numbers must generate these numbers themselves. However, since CPUs are deterministic, it is impossible to algorithmically generate truly random numbers.

In the absence of truly random number sources, cryptographic applications make use of *Pseudo Random Number Generators* (PRNG) for random number generation. A PRNG is a function which takes a certain amount of true randomness, called the *seed*, and generates a stream of bits which can be used as if they were true-random, assuming the adversary is computationally limited and that the seed is large enough to thwart brute force attacks by

[8] A nonce is a quantity which any given user of a protocol uses only once.

that adversary. A PRNG function has two properties.

- It looks random; *i.e.*, it passes all the statistical tests of randomness.
- It is unpredictable; *i.e.*, it must be computationally infeasible to predict what the next random bit will be, given complete knowledge of the algorithm generating the sequence and all of the previous bits in the stream.

A good method to select seed values for the PRNG is an essential part of a cryptographic application. If the seed values for the PRNG can easily be guessed, the level of security offered by the program is diminished significantly, since it requires less work for an attacker to decrypt an intercepted message[9]. A description of the PRNG and the method used to generate seed values for the PRNG are described in Section 4.2.7.

3.2.1 Limitations of Some Random Number Generation Techniques

A computer's clock is often used as a seed for a PRNG. However, computer clocks provide significantly fewer real bits of unpredictability than might appear from their specifications. Different hardware configurations running the same version of an operating system may provide different resolutions in a clock. This makes designing portable applications difficult since the designer does not always know the properties of the system clocks that the code will execute on [ECS94].

Use of a hardware serial number such as an Ethernet address may also provide fewer bits of unpredictability. Such quantities are usually heavily structured and subfields may have only a limited range of possible values or values easily guessable based on approximate date of manufacture or other data. In the case of Ethernet addresses, the adversary can also find out the address by using protocols like ARP[10] if he is on the same subnet as the target host.

[9] The attack on Netscape's implementation of the SSL protocol by Goldberg and Wagner was based on predicting the seed used for random number generation.

[10] Address Resolution Protocol, [Plu82]

It is possible to measure the timing and content of mouse movement, key strokes, and similar user events. This is a reasonable source of unguessable data with some qualifications. On some machines, inputs such as key strokes are buffered. Even though the user's inter-keystroke timing may have sufficient variation and unpredictability, there might not be an easy way to access that variation. Another problem is that no standard method exists to sample timing details. This makes it hard to build standard software intended for distribution to a large range of machines based on this technique.

Another strategy that can give a misleading appearance of unpredictability is the selection of a quantity randomly from a database like the USENET archives and assuming that its strength is related to the total number of bits in the database. For an adversary with access to the same database, the unguessability rests only on the starting point of the selection. The same argument applies to selecting sequences from the data on a CD ROM or Audio CD recording or any other large public database.

3.2.2 Sources for Randomness

In the absence of a reliable hardware source, the best way to generate random numbers is to find a lot of seemingly random events and *distill* randomness from them using a strong mixing function. Such a function will preserve the randomness present in any of the sources even if other quantities being combined are fixed or easily guessable. This randomness can then be stored in a pool that applications can draw on as needed. Sources for randomness in UNIX systems include

- timing between keystrokes
- actual mouse position
- air turbulence within a sealed disk drive
- CPU load
- contents of the kernel tables
- arrival times of network packets

- access & modify times of /dev/tty
- /dev/audio without a microphone attached
- input from a microphone/camera
- /dev/random

Encryption algorithms like DES and one-way hash functions like MD5 can be used as mixing functions; each bit of output produced by these algorithms is dependent on a non-linear function of all input bits. In this research, 3-DES is used as a mixing function to distill randomness from readily available sources on the local machine (§ 4.2.7).

3.2.3 Key Generation Standards

Several public standards are now in place for the generation of keys. Two of these are described below. Both use DES but any equally strong or stronger mixing function could be substituted.

3.2.3.1 US DoD Recommendation for Key Generation

The United States Department of Defense has specific recommendations for password generation [ECS94]. They suggest using DES in 64-bit Output Feedback Mode as follows:

- Use an initialization vector determined from the system clock, system ID, user ID, current date and time.
- Use a key determined from system interrupt registers, system status registers, and system counters.
- And as plaintext, use an externally generated 64-bit quantity such as eight characters typed in by a system administrator.

The 64-bit ciphertext generated as output in DES Output Feedback Mode can be used as a key. If keys with larger key-lengths are required, a multiple number of keys can be generated by the above process, and the resulting output can be concatenated together.

3.2.3.2 ANSI X9.17 Psuedo Random Number Generation

ANSI X9.17 specifies one of the strongest PRNGs currently available [ABA85]. A number of applications employ this technique, including financial security applications and PGP. This method generates a sequence of random keys as follows:

- Input: Two pseudo random inputs to drive the generator. One is a 64-bit representation of the current date and time, DT_i, which is updated on each number generation. The other is a 64-bit seed value, S_i, which is initialized to some arbitrary value and updated during the generation process.
- Key: A secret encryption key, K, used only for random number generation.
- Output: The output is a 64-bit pseudo random number, R_i, and a 64-bit seed value for the next iteration, S_{i+1}.

If DES is the encryption algorithm used, and DES_K denotes DES encryption using 'K' as the key, R_i (which is derived as shown below) can be used as a key.

$$R_i = DES_K[\ DES_K[\ DT_i \oplus S_i]]$$

$$S_{i+1} = DES_K[\ DES_K[\ DT_i \oplus R_i]]$$

3.3 Key Length

The degree of protection obtained by encryption depends on the quality of the cryptosystem, the reliability of its implementation, and the total number of possible keys that can be used to encrypt the information. A symmetric-key algorithm is considered strong if there is no "shortcut" that allows the opponent to recover the plaintext without using brute force to test keys until the correct one is found; and if the number of possible keys is sufficiently large to make such an attack infeasible [BDR+95].

There is no definitive way to examine a cipher and determine whether an easier alternative than brute force attack exists to break the cryptosystem. However, most of the popular encryption algorithms have been extensively studied in public literature[11] (notably DES) and are widely believed to be of high quality. Thus the length of the key can be used to estimate the upper bound on the system's strength.

Current cryptography uses symmetric-key cryptosystems for encrypting data and public-key cryptosystems to manage the keys used by symmetric systems. If an application uses both symmetric and public-key cryptography, the key lengths for each type of cryptography should be chosen so that it is equally difficult to attack the system via each mechanism. The strength of today's public-key encryption algorithms are based on the *conjectured* difficulty to factor large numbers that are the product of two large primes. Breaking these algorithms do not involve trying every possible key; instead it involves trying to factor the large number into its constituent primes. Mathematical techniques have evolved to make public key cryptosystems subject to shortcut attacks and hence such systems must use keys much bigger than the lengths used in symmetric-key algorithms.

The optimal key length for encrypting data varies from application to application. The following factors should be considered while deciding the strength of the encryption required for an application.

- The value of the data to be protected.
- The period for which the data should be protected.
- The resources available to potential adversaries who want access to the secret data.

Studies by Blaze *et al.*, indicate that modest increases in computational cost can produce vast increases in security. Encrypting information using a symmetric cipher very securely (*e.g.,* with 128-bit keys) typically does not require a lot more computing than encrypting it weakly (*e.g.* with 40-bit keys). They point out that there is no practical or economic reason to design hardware and software to provide differing levels of encryption for different messages, and that the most prudent approach would be to use the strongest encryption required for any information that is stored or transmitted by a secure system [BDR+95].

[11] Peer-review is essential to weed out any hidden flaws in a cryptographic algorithm.

The Blaze study recommends using a minimum key-length of 90 bits to make brute-force attacks on symmetric cryptography infeasible.

Table 3.1 lists the average time taken to crack keys of various lengths using hardware costing $100,000 in 1995. This time decreases linearly with respect to increases in the cost of the hardware used for brute force cracking.

Table 3.1: Estimates for a brute-force attack on symmetric cryptosystems using hardware costing $100K in 1995. Source: [Sch96]

Key Length	Time	
40	2	seconds
56	35	hours
64	1	year
80	70,000	years
112	10^{14}	years
128	10^{19}	years

3.4 Big-Number Libraries

Public-key cryptosystems (§ 2.1.2) are either based on the difficulty of factoring large numbers that are the product of two large primes (*e.g.,* RSA) or on the difficulty of calculating discrete logarithms in a large finite field (*e.g.,* Diffie-Hellman). If these numbers are too small, such systems can be broken. Today's dominant public-key cryptosystems use numbers whose size ranges from 512 bits to 2048 bits. Since standard hardware on current computers do not support more than 64 bits of integer precision, there exists a need for mathematical software libraries which facilitate the use of multiple-precision (and sometimes infinitely extendable) integers.

A multiple-precision (MP) library generally consists of routines to manipulate integer, rational and natural numbers, including functions to initialize and allocate space for MP variables; to perform standard arithmetic operations on these variables; and to free the dynamic space used by these variables after use. The limit of the precision is set by the dynamic memory available in the computer.

There are a number of publicly available implementations of MP libraries. In this research, the GNU MP library[12] was used. The GNU MP library is compatible with the MP library available on many BSD derived UNIX systems. This library was used to generate the random numbers required in this research (§ 4.2.7).

3.5 Encryption/Authentication Performance

The encryption of data-streams in a distributed application will degrade performance since it involves additional computation. To quantify the effects of encryption, a number of performance tests were constructed. The compiler used was gcc, and optimization was enabled with a "-O" flag. The Sun SPARC5s used in most of the tests were running SunOS 4.1.4 and had 96MB main memory. The tests were conducted on a moderately busy Ethernet (10 MBit/sec) segment (the Computer Science department). In order to reduce the effects from extraneous factors, the tests were carried out in the late evening (after 10:00 PM) when the network traffic is likely to be less.

3.5.1 Comparison of DES Operating in Different Cipher Modes

In this test, DES was run in the four different operating modes: ECB, CBC, CFB and OFB[13]. This test was carried out to find out how the DES performed in different modes. The same test was run a number of machines, with different architectures, and running different operating systems[14]. Repeated encryptions were carried out on a 8192-byte string to measure the performance of different DES modes. The block size used for ECB and CBC modes is 64 bits. For the CFB mode, encryption was carried out both 8 bits and 64 bits at

[12] This package is available for ftp from any of the usual GNU ftp-sites.

[13] Using Eric Young's `libdes` package.

[14] The performance of DES on the different platforms should not be compared with one another since the machine-architectures and operating systems used for this test are different. The goal of this test is to show that encryption speeds with DES in CBC or CFB mode, with the same block size, have similar performance.

a time. For the OFB mode, encryption was carried out 8 bits at a time.

It can be noted from Figure 3.1 that the speeds for 64-bit CBC and 64-bit CFB encryption are comparable. This is in agreement Schneier's studies on symmetric ciphers [Sch96] where he reports that encryption in different modes for a symmetric cipher are comparable if the number of bits encrypted at a time are the same. This result is relevant in this research because the IDEA implementation used in this research encrypts 64 bits at a time in CFB mode, whereas DES is used in 64-bit CBC mode.

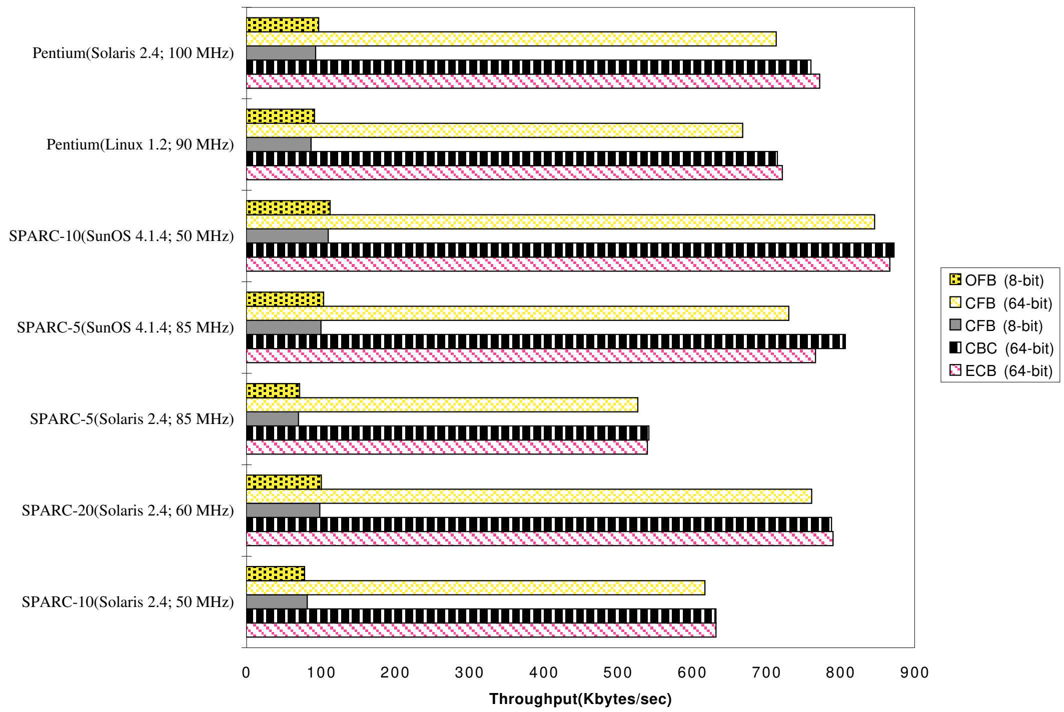

Figure 3.1: DES performance in different cipher modes

3.5.2 Encryption and One-Way Hashing Performance within a Process

This test was carried out to find the performance of different encryption algorithms on different data lengths within a process[15]. These results are compared with the speeds of a

[15] Using SSH's crypto-API implementation.

plain bcopy() for the same data lengths. To get a feel for the relative costs of encryption and authentication[16] the speed of a MD5+bcopy is also plotted on the same figure (see Figure 3.2). Throughput was computed by dividing the total amount of the data processed (*i.e.*, encrypted/hashed), for each message size, by the time taken. Since the speed of bcopy() is orders of magnitude faster than that of encryption algorithms, the graph is plotted with both axes set to logarithmic (base = 10) scale.

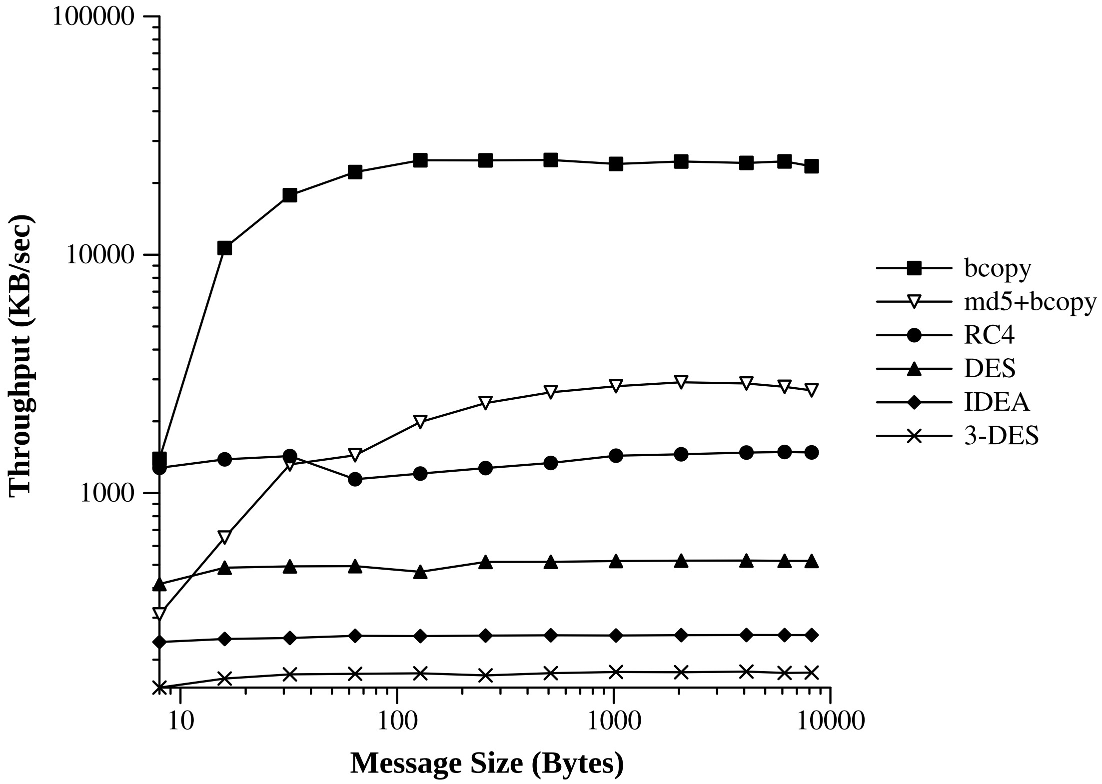

Figure 3.2: Encryption/Hashing performance within a process

[16]Message authentication is usually carried out using secure one-way hash functions like MD5.

3.5.3 UDP Throughput Performance of Encryption Algorithms

In this test, the throughput performance of different encryption algorithms for messages being sent between two hosts using UDP was measured[17]. In order to avoid having to synchronize the clocks of the two hosts or to approximate the offset, messages are sent round-trip, and the total time difference is divided in half. The timing is carried out only on one of the hosts with the assumption that messages should take roughly the same amount of time to go in each direction. To factor in the costs for encryption, messages are encrypted on one host and the ciphertext is sent to the other host. At the remote side, the message is decrypted and the plaintext is sent back. As expected, the UDP performance between two hosts degrades when messages are encrypted (see Figure 3.3).

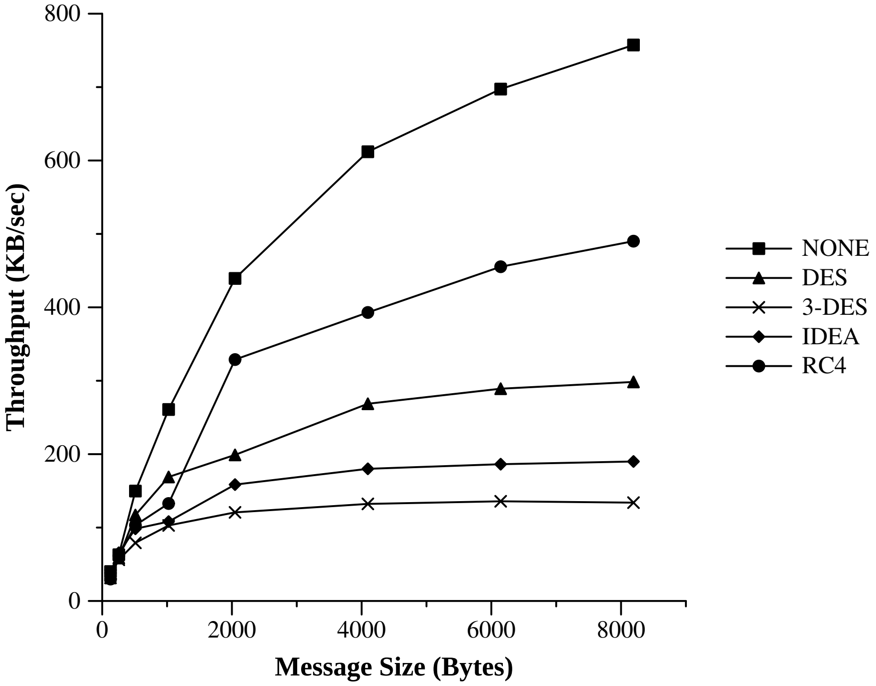

Figure 3.3: UDP throughput performance for different encryption algorithms

[17] UDP is the transport protocol used by PVM for pvmd-pvmd communication.

Chapter 4

Implementation of Secure PVM

In order to get a better understanding of the design tradeoffs involved in secure distributed applications, this research focussed on extending the Parallel Virtual Machine (PVM) to include support for cryptographic authentication, data integrity and encryption. This chapter first introduces relevant PVM concepts and then describes the extensions made to enhance PVM's security. The last section reports the results from performance tests carried out on the modified version of PVM.

4.1 Parallel Virtual Machine (PVM)

PVM is a message-passing system that permits a network of heterogeneous computers to work in parallel in solving both scientific and commercial applications. PVM is called a "virtual" machine since it joins physically separate and architecturally different machines over a network (LAN or WAN). It functions like a "loosely-coupled" distributed operating system, but runs on top of the existing operating system (*e.g.*, UNIX). In order to be highly portable, PVM uses the file-system and memory-management services provided by the underlying operating system. Manchek's thesis [Man94] is the authoritative reference for the design and implementation of PVM Version 3. The PVM User's Guide [GBD+94] is another useful source for information about PVM.

The PVM system consists of two parts. The first part is a daemon process, called `pvmd`, that resides on all the computers (hosts) making up the virtual machine. When a user wants to run a PVM application, he first creates a virtual machine by starting up PVM. This starts a `pvmd` process on each of the member hosts in the virtual machine. The `pvmd` serves as a message router and controller (see Figure 4.1); it provides a point of contact,

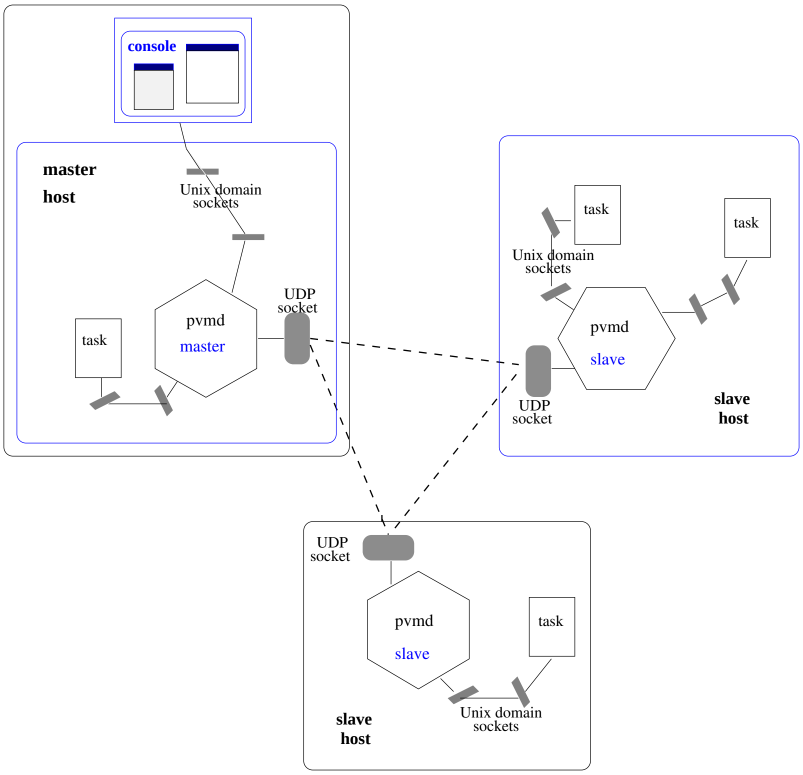

Figure 4.1: Partial anatomy of PVM

authentication, process control, and fault detection. The first `pvmd` (started by hand) is designated as the *master*, while the others (started by the master) are called *slaves*. Only the master can add or delete slaves from the virtual machine.

The second part of the PVM system is a library of routines (`libpvm`) that allows a task[1] to interface with the pvmd and other tasks. libpvm contains functions for packing and unpacking messages, and functions to perform PVM *syscalls* by using the message functions to send service requests to the pvmd.

PVM daemons communicate with each other through UDP sockets. A task talks to its local pvmd and other tasks through TCP sockets. UDP cannot be used for such communications since tasks cannot be interrupted while computing to perform I/O[2]. In order to improve latency and transfer rates, PVM 3.3 introduced the use of UNIX-domain stream sockets as an alternative to TCP for local communication. If enabled at compile time, stream sockets are used between the pvmd and tasks as well as tasks on the same host [Man94].

4.2 PVM Extensions for Enhanced Security

In this research, only security enhancements for inter-host communication were considered. This is because security in intra-host communication is associated closely with the authority of *root* on each machine and can be subverted by anyone with superuser privileges on the machine. However, some of the programming hooks provided for inter-host security can also be used for enhancing the security of intra-host communications (§ 4.2.4).

Due to known weaknesses in the standard protocols (rexec and rsh) used for starting slave pvmds on a host [CERT94] [CERT95], this research proposes using the Kerberos version of rsh or new protocols like SSH or STEL for securely spawning processes on a remote host. In order to set up secure communication among the hosts belonging to the virtual machine, a secret PVM session key needs to be established among the hosts. This secret key is generated on the master pvmd and distributed to each slave by means of a Diffie-Hellman (DH) key exchange[3]. The slave startup protocol was extended to accomplish

[1] A task is a unit of computation in PVM analogous to a UNIX process.
[2] UDP can lose packets even within a host, requiring retry (with timers) at both ends.
[3] Instead of Diffie-Hellman, public key mechanisms like the RSA implementation in PGP can also be used. This approach would require access to the PVM user's PGP key-rings on each of the participating hosts.

the DH key exchange.

The user can request secure communication either while starting PVM (*i.e.,* at the command-line), or selectively enable secure communication on a per-message basis (*i.e.,* by routines in `libpvm`). Regardless of the level of security chosen, the secret PVM session key is always passed on to the slave during daemon startup.

For encryption, the user can choose from secret-key algorithms like DES, 3-DES, IDEA, and RC4. For authentication and data integrity, the MD5 message digest algorithm is used. By using a modular cryptographic API[4], new algorithms can easily be added in the future. New fields were added to existing PVM data-structures to pass on cryptographic security related information among the participating hosts in the virtual machine.

The extensions were made to PVM release 3.3.9, which was the latest one available when work was started on this project. The following subsections describe the extensions made to standard PVM for enhanced security.

4.2.1 Starting Slave Pvmds

The first step involved in adding a host to the virtual machine is to start a slave pvmd on that host. The goal is to get a process running on the new host, with enough identity to let it be fully configured and added as a peer. The standard PVM implementation provides three mechanisms to start a slave pvmd.

- `rexec`
- `rsh`
- manual startup

From a security viewpoint, the use of `rexec` is no longer feasible to start a slave pvmd, since `rexec` requires the user's password to be sent in the clear over the network to the

[4] Based on the SSH distribution.

host on which the slave pvmd is being started. With "password-sniffing" attacks becoming very common [CERT94], it would be quite easy for an attacker to capture the PVM user's password if `rexec` is used.

In the last few years, there has been a large number of attacks (*e.g.*, source routing attacks) on the `rsh` protocol [Bel89]. In order to contain this risk, a number of sites are disabling the use of Berkeley-`rsh` to spawn processes on a remote host. However, a mechanism for spawning processes on a remote host without being prompted for a password is still quite attractive. The Kerberos version of `rsh` offers a reasonable solution to this problem by modifying the `rsh` protocol to take advantage of the Kerberos's authentication infrastructure. Kerberized-`rsh` is a drop-in replacement for the Berkeley-`rsh` and eliminates most of the risks involved with using the `rsh` protocol for remote process initiation. If the slave pvmds need to be started using `rsh`, the Kerberized version of the `rsh` protocol should be used.

The manual startup option allows the user to log on to a remote host and start the pvmd by hand, and to type in the configuration. One-time password mechanisms like S/Key (§ 2.4.4) should be used to eliminate the risk from "password-sniffing" attacks. New protocols like SSH and STEL (§ 2.4.2) can also be used to securely start slave pvmds.

4.2.2 Key Distribution

Prior to secure communication between the hosts belonging to the virtual machine, a secret session key needs to be established among the hosts. The secret PVM session key is generated (§ 4.2.7) on the host running the master pvmd. The master pvmd distributes the PVM session key to each slave pvmd as follows:

1. While starting up a new slave pvmd, the master pvmd initiates a 1024-bit modulo Diffie-Hellman key exchange (§ 2.1.2.2) with the slave $pvmd_i$ to generate a shared secret key, DH_{key_i}, between the master and slave.

2. The master pvmd encrypts the session key using 3-DES with the DH_{key_i} shared with the slave $pvmd_i$ and sends the encrypted session key to the slave.

3. The slave-$pvmd_i$ decrypts the encrypted session key using its copy of DH_{key_i}.

The Diffie-Hellman (DH) key exchange requires each participating entity to exchange their public keys with each other. The slave pvmd startup protocol was extended to accomplish this exchange. When each pvmd (master or slave) is started up, it generates a public/private key-pair to be used for the DH key exchange[5]. The DH public keys exchanged between the master and slave pvmds are not authenticated. This makes it theoretically possible for an adversary to mount an active attack (*man-in-the-middle* attack, § 2.1.2.2) on the DH key exchange protocol. However, such attacks are not trivial to mount[6]; so there is still improved security with respect to standard PVM. Since messages are encrypted and authenticated in secure PVM, and the master pvmd will reject forged messages for starting up a new slave. This makes it more difficult for an intruder to mount a man-in-the middle attack on the secure PVM DH key exchange.

Figure 4.2 shows a host being added to the virtual machine[7]. A task calls `pvm_addhosts`(), to send a request to its pvmd, which in turn sends a `DM_ADD` message to the master (possibly itself). The master pvmd creates a new host table entry for each host requested, looks up the IP addresses and sets the options from the host file entries or defaults. The host descriptors are kept in a `waitc_add` structure (attached to a wait context[8]) and are not yet added to the host table. The master forks the $pvmd'$, passing it a list of hosts and commands to execute[9]. The $pvmd'$ uses `rsh` or manual startup to start each pvmd, pass it parameters, and gets configuration data back from the newly started

[5]The base and the modulos for the DH exchange are constant and are "hard-coded" in this implementation.

[6]Private communication with Randal Atkinson, rja@cisco.com (Jan 9, 1996).

[7]The dotted lines indicate the new messages added to the standard PVM slave startup protocol.

[8]Pvmds use a wait context (waitc) to hold state when a thread of operation must be interrupted.

[9]The shadow pvmd, $pvmd'$, is a process which runs on the master host and is used by the master to start new slave pvmds.

slave.

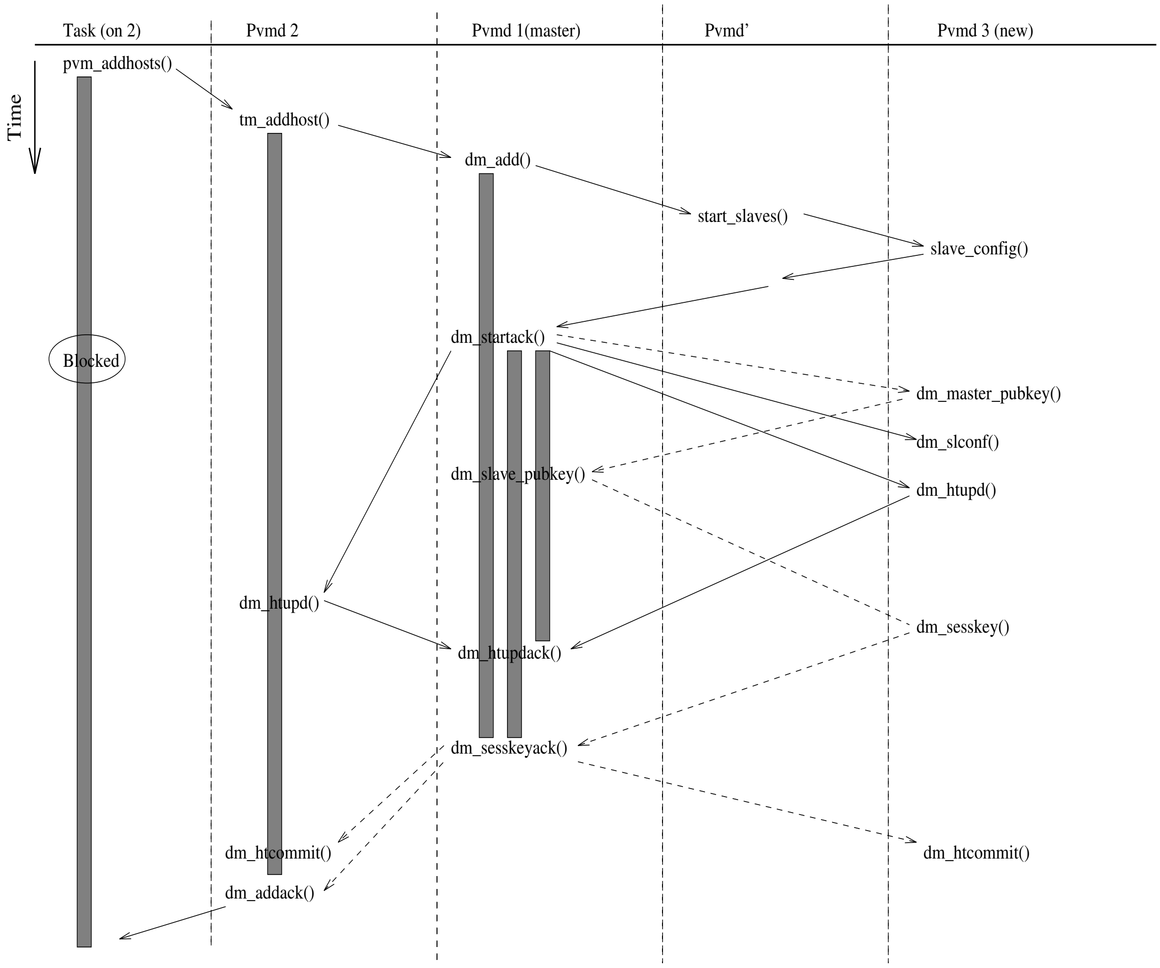

Figure 4.2: Timeline of key-exchange operation

The addresses of the master and slave **pvmds** are passed on the command line. The slave writes its configuration on standard output, then waits for an EOF from the *pvmd′* and disconnects. The slave runs on probationary status until it receives the rest of its configuration from the master **pvmd**. If it is not configured within five minutes (parameter `DDBAILTIME`), it assumes that there is something amiss with the master and quits. The protocol revision (parameter `DDPROTOCOL`) of the slave **pvmd** must match that of the master. This number is incremented whenever a change in the protocol makes its incompatible with the previous version[10]. When several hosts are added at once, startup is done in parallel[11]. The *pvmd′* sends the data (or errors) in a DM_STARTACK message to the master pvmd, which completes the host descriptors held in the wait context.

After the slaves are started, the master sends a DM_MASTER_PUBKEY message to each slave. The master also sends each slave a DM_SLCONF message to set parameters not included in the startup protocol. It then broadcasts a DM_HTUPD message to all new and existing slaves.

On receiving the DM_MASTER_PUBKEY message, the slave computes its copy of the Diffie-Hellman (DH) shared secret key using its private key and the master's public key. The slave then sends its public key to the master using a DM_SLAVE_PUBKEY message. Upon receiving the DM_HTUPD message, each slave knows the configuration of the new virtual machine.

On receiving the DM_SLAVE_PUBKEY from a slave, the master computes the DH key shared with this slave. It then encrypts the PVM session key with the DH key using 3-DES and sends the encrypted session key to the slave in a DM_SESSKEY message. On receiving the DM_SESSKEY message, the slave extracts the PVM session key from it. It then informs the receipt of the PVM session key by sending the master a DM_SESSKEYACK message.

The master waits for an acknowledging DM_SESSKEYACK message from each newly started slave, and then broadcasts a DM_HTCOMMIT message, shifting all pvmds to the new host table. Finally, the master sends a DM_ADDACK reply to the original request, giving the host IDs.

4.2.3 PVM messages

The pvmd and libpvm use the same message header (see Figure 4.3). *Code* is an integer tag which specifies the message type. Since libpvm can pack messages in different formats, it makes use of the *Encoding* field to specify the encoding style of the message. The pvmd always sets the Encoding field to use *foo* encoding. The pvmds use the *Wait Context* field to specify the wait ID (if any) of the *waitc* associated with the message. The *Checksum*

[10] The DDPROTOCOL value was incremented while modifying code for this research. This facilitates the detection of pvmd versions without security extensions attempting to join the virtual machine configuration.

[11] PVM can initiate the startup of five slaves concurrently.

field is reserved for future use. No modifications were made to the message header in secure PVM.

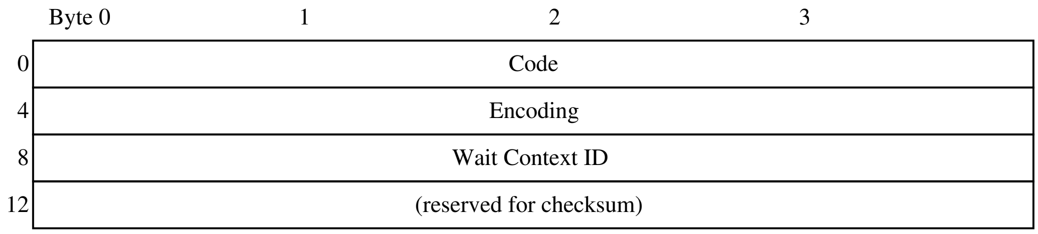

Figure 4.3: PVM Message header

PVM daemons communicate with one another through UDP sockets. UDP is an unreliable delivery service which can lose, duplicate, or reorder packets. An acknowledgement and retry mechanism is used by PVM to provide a reliable delivery service over UDP. UDP also limits packet length, so PVM fragments long messages. Messages are sent in one or more fragments, each with its own fragment header. The message header is at the beginning of the first fragment.

Each message fragment is sent in a separate UDP packet. In order to re-assemble packets back into a PVM message at the receiving pvmd, each packet has a packet header with the requisite information (see Figure 4.4). Multi-byte values are sent "most significant byte first", *i.e.*, in (Internet) *network byte order*.

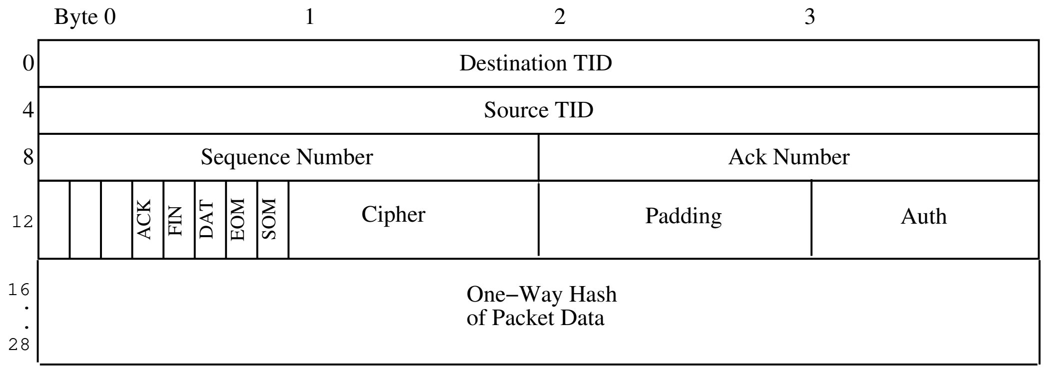

Figure 4.4: Pvmd-Pvmd packet header for secure PVM

The source and destination fields hold the TIDs of the true source and final destination of the packet, regardless of the route it takes. Sequence and acknowledgement numbers start at 1 and increment to 65535, and then wrap to zero. The *Flag* field conveys information about the packet like whether it is the first/last fragment of the message (SOM/EOM), whether any data is contained in the packet (DAT), whether the packet is an acknowledgment (ACK), or whether a pvmd is closing down the connection (FIN).

In order to pass on cryptographic security related information among the pvmds, a few new fields were added to the packet header. The *Cipher* field is used by the pvmd to specify the algorithm used to encrypt the data (0 means unencrypted) contained in the packet. If the encryption algorithm is block-oriented, the data will have to be padded to a multiple of the block size before being encrypted. The *Padding* field is used to convey the size of the padding (in bytes) present in the encrypted data, so that the receiving pvmd can correctly extract the data from the encrypted payload. The *Auth* field is used by the pvmd to indicate whether the data in the packet is authenticated using a one-way hash function (0 indicates un-authenticated). The last 16 bytes of the packet header are used to store the one-way hash of the data, if the packet is being authenticated.

4.2.4 Security Options available to the PVM user

The PVM user can either choose enhanced security services at daemon startup time or selectively enable them for any specific message sent between PVM hosts. He can choose from three different levels of security. They are

1. Encryption: The user is assured of message privacy and integrity.

2. Authentication: The user is assured of message integrity.

3. Default: The user gets access to the standard PVM; *i.e.*, services without cryptographic encryption/authentication support.

While starting PVM, the '-e' flag can be used to specify the encryption algorithm and the '-a' flag can be used to specify the one-way hash function to be used for authentication. For example,

1. Start PVM with DES encryption,
   pvm -edes
2. Start PVM with MD5 authentication,
   pvm -amd5

Since there is a performance cost associated with the encryption of messages, the user may choose to encrypt only essential messages. Support for this requirement is provided by overloading the "encoding format" parameter[12] to `pvm_mkbuf()` (or `pvm_initsend()`[13]). Table 4.1 lists the possible values for the encoding format.

Table 4.1: Encoding formats used in libpvm

Encoding	
PvmDataDefault	standard PVM
PvmDataRaw	standard PVM
PvmDataInPlace	standard PVM
PvmDataFoo	standard PVM
PvmDataDefault_CipherXXX	newly added
PvmDataRaw_CipherXXX	newly added
PvmDataInPlace_CipherXXX	newly added
PvmDataFoo_CipherXXX	newly added
† where XXX stands for DES/3DES/IDEA/RC4	

For example, to send a message which needs to be encrypted using DES, a PVM programmer would use the following code segment.

```
strcpy(buffer, "this is a secret message ");
bufid = pvm_initsend( PvmDataDefault_CipherDES );
```

[12] Libpvm provides functions to pack all primitive data types into a message. When creating a new message, the encoder set is determined by the "encoding format" parameter to `pvm_mkbuf()`.

[13] `pvm_initsend()` invokes `pvm_mkbuf()`.

```
info = pvm_pkstr( buffer );
msgtag = 3 ;
info = pvm_send( tid, msgtag );
```

When `pvm_send()` gets invoked, it checks to see if the message needs to be encrypted. If so, it marks all the fragments in the message for encryption by the local pvmd. When the local pvmd receives a message from `libpvm`, it checks the encryption field in the message fragment and encrypts the data accordingly. Only messages destined for remote hosts get encrypted (§ 4.2).

If the programmer has access to his own crypto-library and only needs access to a key "shared" among all the PVM hosts, he can invoke `pvm_getsesskey()` to get access to the shared PVM session key. This function sends a TM_GETSESSKEY message to the local pvmd and retrieves the PVM session key. The user can then use this key to encrypt the data and send the encrypted data *opaquely* across to the remote PVM host. At the remote end, his application can extract the PVM session key from its pvmd in the same way and use it to retrieve the encrypted data. This feature could be used to enhance the security of PVM direct TCP connections (*i.e.* , PvmRouteDirect) by using the PVM session key to encrypt and opaquely send data across to a remote task.

4.2.5 Authentication

Secure one-way hash functions are used to facilitate the authentication of data going across the network. The PVM packet header was extended by 16 bytes to include a 128-bit message digest of the packet data. The pvmd at the remote host which receives the packet computes the message digest from the relevant fields[14] in the packet and compares it with the message-digest which was included in the incoming packet. If they match, the remote pvmd is assured that the packet is authentic.

If authentication is requested by the user explicitly or implicitly (§ 4.2.6), `netoutput()`

[14]Currently the fields that are hashed include the source task-id, the destination task-id, the packet sequence number and the packet data.

checks the pk_authtype field in the packet, and invokes auth_hash() to generate a message digest of the packet data (currently, MD5 is the only one-way hash function supported). The message digest is calculated by auth_hash() as specified in the keyed-MD5 RFC [MS95]. The form of the authenticated message is

$$[< key >< keyfill >< data >< key >< MD5fill >]$$

The message digest is generated as follows.

1. The secret authentication key is padded with zeroes to the next 512-bit boundary.

2. The "filled" key is concatenated with the relevant fields of the packet structure (struct pkt) and concatenated with the original session key again. These fields include the source and destination task-ids, the packet sequence number, and the data contained in the packet.

3. A trailing pad with length to the next 512-bit boundary for the entire message is added by MD5 itself.

The PVM session key, shared by all the pvmds, is used as the key while computing the message digest.

netoutput() incorporates the message-digest generated by auth_hash() into the packet header, sets the *Auth* field, and adds the packet to the send-queue for the remote destination. On receiving a packet, netinput() examines the packet header to check if the *Auth* field is set. If so, it invokes auth_verify() to authenticate the packet. auth_verify() computes the message digest in the same way as auth_hash(), and checks if it matches the message digest included in the incoming packet. If they do not match, the packet is considered to be a bogus one and dropped after logging it to the PVM log file. Message replay attacks can be detected because the packet sequence number is also hashed in while generating the message digest. Duplicate packets are logged to the PVM log file and dropped without further processing.

4.2.6 Encryption

Secret-key encryption is used to ensure the privacy of messages sent across the network. The PVM session key, shared among all the pvmds, is used as the key for encryption/decryption. To facilitate the use of different algorithms for encryption, a modular cryptographic API is used. This API implementation[15] currently supports DES, 3-DES, IDEA and RC-4.

The encryption/decryption of data is handled on a per-packet basis. If the encryption field is set in `struct pkt`, the data portion of the packet is encrypted before the packet is queued for a remote-destination[16]. `netoutput()` checks the `pk_ctype` field in the packet structure to determine the encryption algorithm being used. It then invokes `encrypt_packet()` to encrypt the data contained in the packet. Before encrypting the data, `encrypt_packet()` pads the data to a multiple of the block-size used by the encryption algorithm. The *Padding* field in the packet header (see Figure 4.4) is used to indicate the amount of padding used.

On receiving a packet from a remote pvmd, `netinput()` inspects it to check if the payload is encrypted. If so, it invokes `decrypt_packet()` to recover the data. After decrypting the data, the *Padding* field is checked to see if data was padded to a multiple of the block-size prior to encryption and only the relevant portion is extracted. `netinput()` then sends an acknowledgment to the sending pvmd to indicate proper receipt of the packet and adds the packet to the reordering queue for further processing by `netinpkt()`.

If encryption is chosen, authentication is also implicitly enabled for sending messages between pvmds. This is because each block of ciphertext corresponds to some block of plaintext. Since the receiving host needs to know that the encrypted message is coming from an authentic source, the data also needs to be authenticated.

---

[15] Based on SSH's cipher API.

[16] Packet headers cannot be encrypted since the pvmds need to inspect them to make routing decisions.

4.2.7 PVM Key Generation

PVM uses the Diffie-Hellman key exchange to distribute the secret session key among all the pvmds (§ 4.2.2). For this purpose, each slave pvmd generates a public/private key-pair, and exchanges public keys with the master pvmd during startup time. The public/private key-pairs and the PVM session key used for securing pvmd-pvmd communication are obtained via a pseudo-random number generator[17] implemented using the GNU Multiple-Precision (GMP) package.

In order to achieve better performance, entropy is collected into a buffer from readily available sources on the local machine. These sources include the current system time, the host name and operating system version(*i.e.,* the output of `uname()`), the process and group-ids', the current working directory, and the access/create/modify times of frequently changing files. Other candidates for "entropy-sources" are the output of UNIX utilities like `netstat`, `vmstat`, `pstat`, `iostat` *etc*[18]. This buffer is hashed using MD5 to generate a 16-byte value. This MD5 hash is repeatedly concatenated to generate a 512-byte string which is then encrypted using 3-DES (used as a mixing function) to distill out a reasonably random string.

The PVM session key is generated during the startup handshake with the first slave pvmd connecting back to the master. Prior to generating the PVM session key, the Diffie-Hellman key shared between the master pvmd and the first slave pvmd connecting to the master is also used to seed the random number generating routines.

4.3 PVM and Kerberos

During the initial stages of this research, the integration of PVM with Kerberos was investigated. For this purpose, a Kerberos Version 5 Beta 5 KDC was installed along with

[17] Based on STEL's method for random number generation.

[18] It would have been great if something like Linux's /dev/random were available on all UNIX platforms.

the Kerberized versions of the Berkeley 'r' commands[19].

The PVM design, however, does not fit cleanly into the Kerberos model. In the Kerberos model, the KDC shares a secret key with each Kerberized service on a host. This requires the existence of a registered principal on each host[20]. In PVM, multiple users can simultaneously run their own isolated virtual machines with the pvmds running on any of the non-reserved ports on a host. This design was chosen in order to allow an user to install PVM without having any special (super-user) privileges on the machine. Due to this, a single trusted PVM principal cannot be used to function as the pvmd for all PVM users in a host.

One could perhaps extend the *forwardable ticket* concept in Kerberos V5 to create a session key shared among the pvmds. This shared session key could then be used to encrypt/authenticate messages sent between hosts. However, *forwardable tickets* do not function correctly in the Kerberos V5 Beta5 distribution[21] and hence this could not be tested.

4.4 Performance

All the PVM tests were carried out on Sun SPARC5's running SunOS 4.1.4 and containing 96MB of main memory. Experiments were carried out to determine the message passing performance of secure PVM relative to standard PVM 3.3.9. All machines were connected to the same Ethernet segment; packets were routed between hosts in a single hop without being forwarded through any routers.

4.4.1 Comparison of Pvmd Slave Startup Times

The slave pvmd startup protocol (§ 4.2.2) was extended to do the Diffie-Hellman (DH) handshake and to distribute the PVM secret session key among all the pvmds. Figure 4.5 shows the time taken to start up one to eight slaves in parallel. Two sets of startup times,

[19] Kerberos V5 Beta5 was latest release available when this research started.
[20] This suggests the need for a well-known PVM port on each machine.
[21] Joe Ramus (ramus@nersc.gov), private email.

one for standard PVM and one for PVM with security extensions are plotted side by side.

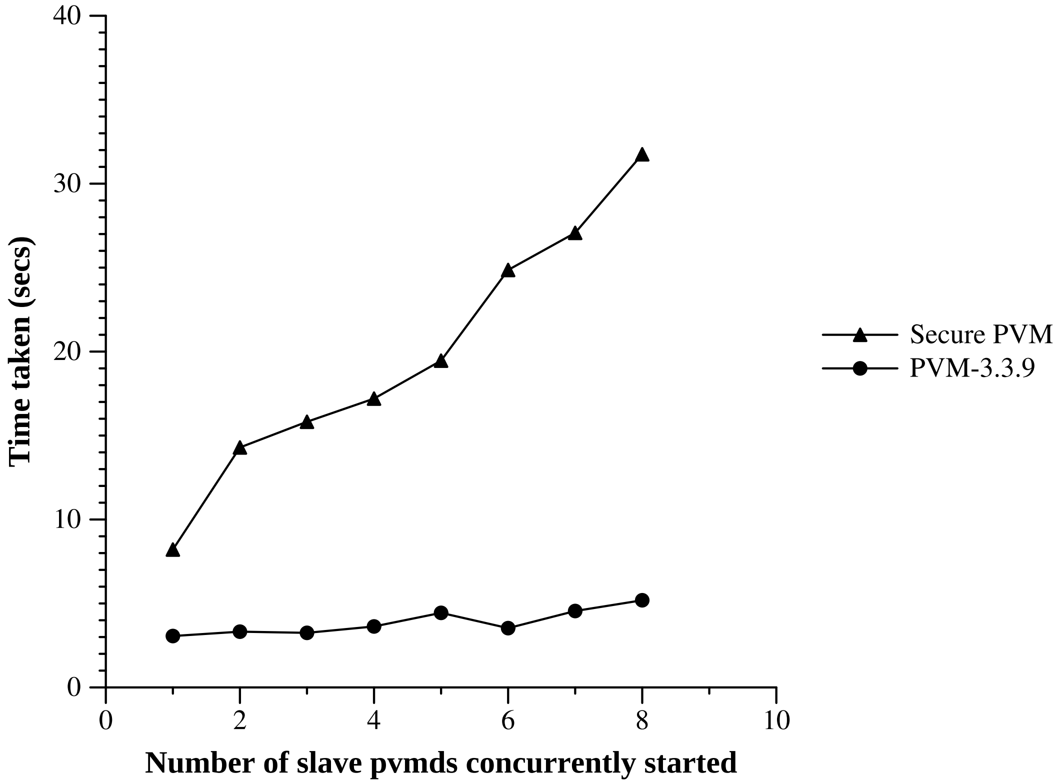

Figure 4.5: Time taken to start 1-8 slave pvmds

It can be seen that the startup times in secure PVM increases with the increase in the number of slaves added concurrently[22]. The factors contributing to the additional latency when compared with standard PVM are

- Public/private key computation on the slave pvmd.
- DH key computation on the slave pvmd.
- DH key computation on the master pvmd.
- 3-DES encryption and distribution of the PVM session key by the master pvmd.

As the number of slave pvmds started concurrently increases, the DH key computation on the master pvmd becomes the *bottleneck*.

[22] PVM can initiate the startup of five slave pvmds in parallel via rsh.

4.4.2 Comparison of PVM Throughput

Tests of throughput were run to find out the relative performance of different encryption/authentication algorithms used in secure PVM. The message-lengths were varied from 128 bytes to 4k bytes (the default PVM fragment size is 4kB). Figure 4.6 plots the bandwidths that can be achieved for traffic between two PVM hosts with standard PVM, secure PVM with authentication enabled, and secure PVM with encryption enabled[23]. In all the cases, "default" routing is used (*i.e.,* packets are routed via each hosts's `pvmd`). In order to avoid having to synchronize the clocks of the two hosts or to approximate the offset, messages are sent round-trip, and the total time difference is divided in half. It can be observed from Figure 4.6 that adding encryption/authentication extensions to PVM degrades the throughput performance.

A program called `timing.c` is provided with the PVM 3.3.9 distribution to get an estimate of PVM performance on different platforms. This program sends messages with lengths ranging from 100 bytes to 1 Mbytes from one PVM host to another. For each message sent, the sending host gets an acknowledgement which is 4 bytes long from the receiving host. The results from this test can also be used to get an estimate of the throughput performance between two PVM hosts. Table 4.2 shows the results obtained from running this program with standard PVM, secure PVM with authentication enabled, and secure PVM with encryption enabled.

---

[23] When encryption is turned on, authentication is implicitly enabled (§ 4.2.6).

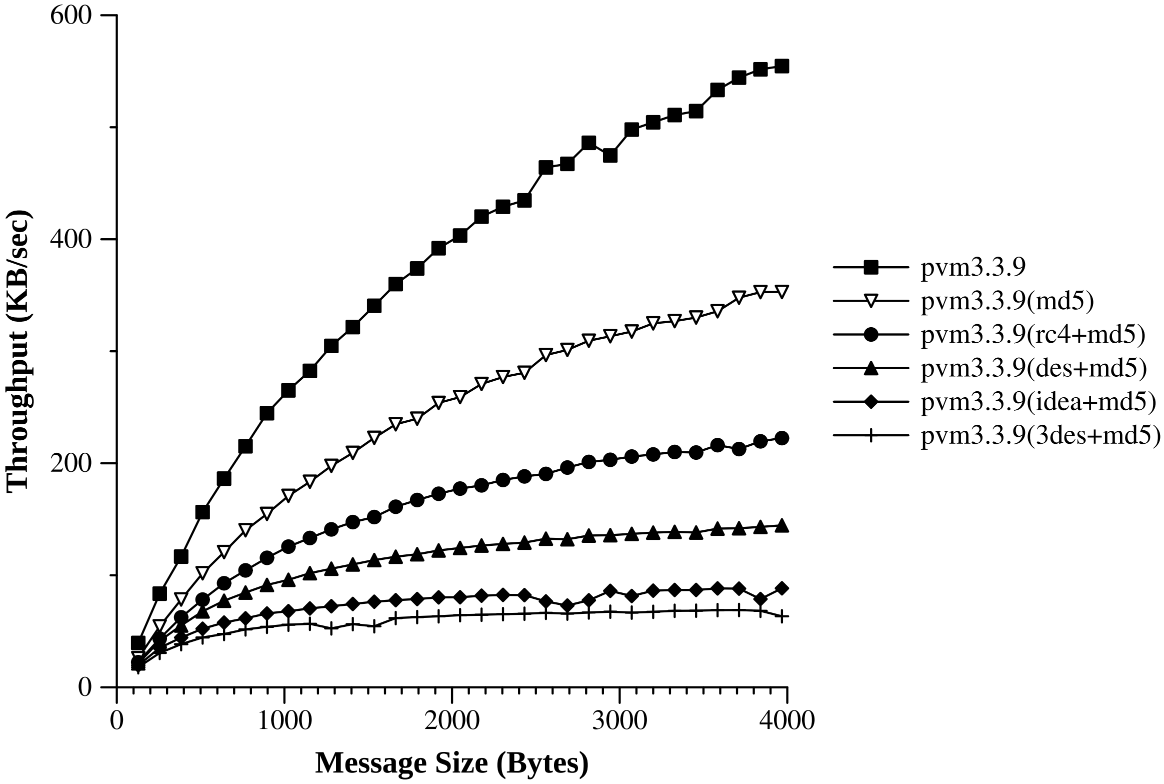

Figure 4.6: Comparison of throughput performance between two PVM hosts

Table 4.2: Results from `timing.c`

Message Size (bytes)	PVM (μsec)	PVM(md5) (μsec)	PVM(des+md5) (μsec)	PVM(rc4+md5) (μsec)
100	5,739	9,106	10,504	10,509
1000	6,668	10,627	16,804	13,458
10000	18,999	32,022	76,790	49,861
100000	147,219	276,958	677,735	427,962
1000000	1528,444	3113,104	6678,024	4346,369

Chapter 5

Summary and Recommendations

This research focussed on developing and evaluating various alternatives for enhancing the security of distributed applications that execute over insecure networks. The use of different encryption/authentication algorithms and their impact on the performance of distributed applications were studied by extending PVM to provide cryptographic support for data privacy and integrity. Since security services based on cryptographic techniques require the keys used for encryption/authentication to be distributed securely, various mechanisms for key distribution were also investigated. The following sections describe how the research issues identified in Section 1.1 are addressed in this research.

5.1 Research Summary

The design issues involved in adding security to a distributed application were discussed in Section 2.3 and in Chapter 3. These include the choice of a good mechanism for key distribution, the selection of a generic crypto-API, a reliable method for generating random numbers, the choice of an optimal key-length, and the use of auxiliary libraries to perform multiple-precision integer calculations. After an extensive survey of current literature and the approaches adopted in existing applications to enhance security, this research attempted to get a better understanding of the above issues by implementing security extensions to PVM. These extensions were made to PVM release 3.3.9, which was the latest one available when work was started on this project.

For inter-host user authentication, this research used existing mechanisms like Kerberos-`rsh` and new protocols like SSH or STEL. The use of `rexec` was avoided since it involves sending a cleartext user passwords across the network. For the PVM *manual-startup* option, this research used one-time password mechanisms like S/Key or OPIE.

The user can request secure communication either while starting PVM (*i.e.,* at the command-line), or selectively enable secure communication on a per-message basis (*i.e.,* by routines in `libpvm`). Regardless of the level of security chosen, the secret PVM session key is always passed on to the slave during daemon startup. The PVM session key, shared among all the pvmds, is used as the key for encryption/decryption.

The slave startup protocol was extended to include a public key exchange between the master pvmd and the newly spawned slave pvmd. This exchange was used to generate a shared secret key between the master pvmd and the slave using the Diffie-Hellman (DH) key exchange mechanism (§ 2.1.2.2). Once the DH key is generated, the master pvmd encrypts the PVM session key using the DH key and sends the encrypted session key to the slave. The newly spawned slave pvmd becomes a part of the virtual machine only after it acknowledges receipt of the PVM session key. It was observed in this research that the slave startup time increased almost linearly with the the number of slaves pvmds concurrently spawned. This is because the master pvmd is involved in the Diffie-Hellman key computation with each newly spawned slave, thereby "serializing" the concurrent spawning process[1].

New fields were added to the pvmd-pvmd packet header in order to specify the encryption algorithm, the amount of padding used (if any), the authentication algorithm and the hash of the packet data. To facilitate the use of different algorithms for encryption, a modular

[1]Only the master sf pvmd can add new slaves to the PVM system.

cryptographic API is used. This API implementation[2] currently supports DES, 3-DES, IDEA and RC4. The MD5 message digest algorithm is used for message authentication and to verify data integrity. By choosing secure PVM's encryption services, the user is assured of message privacy. He is also assured of the authenticity of the message since selecting encryption also implicitly enables authentication (§ 4.2.6). If the user only requires assurance of message authenticity, he can choose the "authentication-only" option. In either of the above cases, *replay-attacks* can be detected because the packet's sequence number is included in the authentication hash of the packet data. Also, if encryption is enabled, an attacker cannot send fake PVM messages to make PVM hosts execute arbitary commands[3].

This research needed access to random numbers to generate the public/private key pairs for the DH key exchange and for the PVM session key. The GNU MP library was used to perform the multiple-precision integer calculations required for generating the large random numbers. These random numbers were generated by collecting entropy from readily available sources on the local machine, and distilling this by using 3-DES as a mixing function (§ 4.2.7). Provisions have been made to add additional entropy in a clean and easy manner.

Since additional computation is involved, the encryption of a data stream in a distributed application will degrade performance. To quantify the effects of encryption, a number of performance tests were carried out as a part of this research. The results of these tests were presented in Section 3.5 and Section 4.4.

5.2 Limitations

This implementation of secure PVM has several limitations. Some of these are due to the fact that the security extensions are done at the application level. Others are intrinsic to the current PVM design.

[2] Based on SSH's cipher API.
[3] This is possible in standard PVM.

Since PVM is layered over the existing operating system, vulnerabilities in the operating system implementation can be exploited to subvert the security of secure PVM. For example, a malicious user with super-user access on a host which is a part of PVM can obtain the PVM session key from the operating system kernel.

The transport mechanism used for pvmd-pvmd communication is UDP. It is much easier to forge UDP packets than TCP packets, since there are no handshakes or sequence numbers [CB94]. Due to this threat, sites using firewalls often drop all UDP packets arriving at non-reserved ports (*e.g.,* port numbers > 1024). Secure PVM (and standard PVM) will not work with hosts behind firewalls having this policy.

To securely spawn slave pvmds, secure PVM uses Kerberized-`rsh` or new protocols like SSH or STEL. In order to use Kerberized-`rsh`, a Kerberos infrastructure should already exist. SSH and STEL also require system-administrator assistance for installation on a machine. Another issue that needs to be considered is that SSH and STEL are still quite new and have not undergone extensive public review. Bug-fixes to their implementations should be promptly applied in order to prevent malicious users from exploiting newly discovered security holes.

In the current design, the PVM session-key is passed on to the slave pvmd during daemon startup. The encryption and authentication of PVM packets exchanged between hosts does not begin until the slave pvmd acknowledges receipt of the PVM session key to the master. Also, there is no provision to change the PVM session key during a PVM session.

Another limitation is that the Pvmd packet headers are sent unencrypted across the network. This is because, the pvmds need to inspect the packet header for making routing decisions. Also, the packet header specifies the algorithm used for encryption and the amount of padding used. The pvmd needs access to this information to correctly decipher the encrypted data. However, portions of the packet headers are part of the message authentication code.

5.3 Future Work

In addition to inter-host communication via the pvmds on each host, PVM also supports direct communication between tasks on different hosts. Direct routing allows tasks to exchange messages via TCP, avoiding the overhead of forwarding through the pvmds [Man94]. In this research, encryption support is not provided for direct task routing. Instead, tasks on each host can obtain the PVM session key by sending a TM_GETSESSKEY message to the local pvmd, and use this key to encrypt and opaquely send data across to a remote task.

In order to facilitate changing the encryption key during a PVM session, secure PVM could have a key hierarchy as in SKIP (§ 2.4.1.2). A master (or key-encrypting) key could be exchanged during slave startup, and a separate packet-encrypting key could be used to encrypt individual packets. The packet-encrypting key would be encrypted using the master key and sent *in-band* in the PVM packet. Since it is sent *in-band*, it will be possible to change the key used for encrypting packet data during a PVM session.

By increasing the size of keys used for encryption, brute-force attacks on cryptosystems can be made "theoretically" infeasible. However, if good random number generation techniques are not used, attackers can exploit weaknesses in the key generation techniques to reduce the search-space for brute force attacks. Operating systems which provide efficient and easy access to randomness sources could help in generating sufficient entropy for seeding pseudo random number generators.

There is a pressing need for implementations of standard security APIs which are fast and suitable for use in distributed parallel applications. A high-performance library that includes implementations of both standard and specialized confidentiality and integrity mechanisms would be very useful to application developers.

The Internet community has recognized the need for having an integrated security framework [Atk95c]. By providing security services at the lower levels of the network hierarchy, *ad-hoc* application specific security solutions can be replaced by generic solutions. There would be no need for a secure version of PVM, if the network (IP) layer provided support for cryptographic security.

5.4 Legal Issues

Foreign accessibility to strong cryptography is considered to compromise communications intelligence. "According to the U.S. government, cryptography can be a munition" [Sch96]. Most packages which include cryptographic solutions have export restrictions associated with them. So considerable care needs to be exercised while making software using cryptographic algorithms publicly available.

Software and algorithms can be patented in the United States. A large number of public and secret key algorithms are patented, though some of them can be used freely for non-commercial purposes. Before incorporating cryptographic solutions into software packages, the legal issues associated with using them should be carefully examined.

Due to these reasons, the distribution of secure PVM will have to be controlled. It is likely that there will be two separate PVM distributions, one for standard PVM and the other for secure PVM.

BIBLIOGRAPHY

Bibliography

[ABA85] American Bankers Association. "American National Standard for Financial Institution Key Management", 1985.

[AP95] Ashar Aziz and Martin Patterson. "Design and Implementation of SKIP". In *INET'95*, June 1995.

[Ash95] Ashar Aziz. "Simple Key Management for Internet Protocols". Internet Draft - Work in Progress, November 1995. URL: ftp://ds.internic.net/internet-drafts/.

[Atk95a] R. Atkinson. "IP Authentication Header". *RFC 1826*, August 1995.

[Atk95b] R. Atkinson. "IP Encapsulating Security Payload (ESP)". *RFC 1827*, August 1995.

[Atk95c] R. Atkinson. "Security Architecture for the Internet Protocol". *RFC 1825*, August 1995.

[BB95] Matt Blaze and Steven Bellovin. "Session-Layer Encryption". In *The Fifth USENIX Security Symposium, Salt Lake City, Utah*, June 1995.

[BDR+95] Matt Blaze, Whitfield Diffie, Ronald Rivest, Bruce Schneier, Tsutomu Shimomura, Eric Thompson, and Michael Wiener. "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security ", November 1995. One-day meeting organized by the Business Software Alliance in Chicago.

[Bel89] S. Bellovin. "Security Problems in the TCP/IP Protocol Suite". *Computer Communications Review*, Vol. 19, no. 2:32–48, April 1989.

[Bih93] E. Biham. "On the Applicability of Differential Cryptanalysis to Hash Functions", March 1993. Lecture at EIES Workshop on Cryptographic Hash Functions.

[Bir84] Andrew Birrell. "Implementing Remote Procedure Calls'. *ACM Transactions on Computer Systems*, Vol. 2, n. 1:39–59, February 1984.

[BM90] S. Bellovin and M. Merritt. "Limitations of the Kerberos Authentication System". *Computer Communications Review*, Vol. 20, no. 5:119–132, October 1990.

[BM92] S.M. Bellovin and M. Merritt. "Encrypted Key Exchange". In *Proc. IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, May 1992.

[BM93] S.M. Bellovin and M. Merritt. "Augmented Encrypted Key Exchange". In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 244–250, November 1993.

[BM95] S. Bradner and A. Mankin. "The Recommendation for the IP Next Generation Protocol". *RFC 1752*, January 1995.

[CB94] William Cheswick and Steven Bellovin. *"Firewalls and Internet Security"* . Addison-Wesley Publishing Company, 1994.

[C.C88a] C.C.I.T.T. "Data Communications Networks Directory: Recommendation X.500", December 1988.

[C.C88b] C.C.I.T.T. "The Directory Authentication Framework: Recommendation X.509", December 1988.

[CERT94] Computer Emergency Response Team. "Ongoing Network Monitoring Attacks". CERT Advisory: CA-94:01, February 1994. URL: http://www.cert.org.

[CERT95] Computer Emergency Response Team. "IP Spoofing Attacks and Hijacked Terminal Connections". CERT Advisory: CA-95:01, January 1995. URL: http://www.cert.org.

[CERT96a] Computer Emergency Response Team. "BIND Version 4.9.3". CERT Advisory: CA-96:02, February 1996. URL: http://www.cert.org.

[CERT96b] Computer Emergency Response Team. "Corrupt Information from Network Servers". CERT Advisory: CA-96:04, February 1996. URL: http://www.cert.org.

[Col90] Colin Ianson and Chris Mitchell. "Security Defects in C.C.I.T.T. Recommendation X.509", December 1990.

[dBB92] B. den Boer and A. Bosselaers. "An Attack on the Last Two Rounds of MD4". In *Advances in Cryptology —Crypto '91 Proceedings*, pages 194–203. Springer-Verlag, 1992.

[DH76] Whitfield Diffie and Martin Hellman. "New Directions in Cryptography". *IEEE Transactions on Information Theory*, Vol. IT-22:644–654, November 1976.

[DH96] S. Deering and R. Hinden. "Internet Protocol, Version 6 (IPv6) Specification". *RFC 1883*, January 1996.

[Dif88] W. Diffie. "First Ten Years of Public Key Cryptography". In *Proceedings of the IEEE*, May 1988.

[Dou95] Douglas Maughan and Mark Schertler. "Internet Security Association and Key Management Protocol (ISAKMP)". Internet Draft - Work in Progress, November 1995. URL: ftp://ds.internic.net/internet-drafts/.

[DP83] D.W. Davies and G.I.P. Parkin. "The Average Size of the Key Stream in Output Feedback Encipherment". In *Cryptography, Proceedings of the Workshop on*

Cryptography, pages 263–279, Burg Feuerstein, Germany, 1983. Springer-Verlag. March29 - April2.

[DS81] D. Denning and G. Sacco. "Time stamps in Key Distribution Protocols". *Communications of the ACM*, Vol. 24, no. 8:533–536, August 1981.

[ECS94] D. Eastlake, S. Crocker, and J. Schiller. "Randomness Recommendations for Security". *RFC 1750*, December 1994.

[Fos95] Ian Foster. "ZIPPER: A Secure Communication Toolkit for High Performance NII Applications", September 1995. Argonne National Laboratory.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *"PVM - A Users' Guide and Tutorial for Network Parallel Computing"* . The MIT Press, 1994.

[GW96] Ian Goldberg and David Wagner. "Randomness and the Netscape Browser". *Dr. Dobb's Journal*, pages 66–70, January 1996.

[Hal94] Kara Hall. "The Implementation and Evaluation of Reliable IP Multicast". Master's thesis, University of Tennessee, Knoxville, 1994.

[IB93] John Ioannidis and Matt Blaze. "The Architecture and Implementation of Network-Layer Security Under UNIX". In *The Fourth USENIX Security Symposium, Santa Clara, California*, October 1993.

[Jas93] Barry Jaspan. "GSS-API security for ONC-RPC". In *Proc. Symposium on Network and Distributed Systems Security*, pages 144–151. IEEE Computer Society, 1993.

[Jon95] Laurent Joncheray. "A Simple Active Attack Against TCP". In *The Fifth USENIX Security Symposium, Salt Lake City, Utah*, June 1995.

[Kau93] C. Kaufman. "DASS: Distributed Authentication Security Service". *RFC 1507*, September 1993.

[Kha94] Raman Khanna. *"Distributed Computing"* . Prentice Hall, 1994.

[KN93] J. Kohl and B. Neuman. "The The Kerberos Network Authentication Service (V5)". *RFC 1510*, October 1993.

[KNT94] J.T. Kohl, B.C. Neuman, and T.Y. Tso. "The Evolution of the Kerberos Authentication System" . In *Distributed Open Systems*, pages 78–94. IEEE Computer Society Press, 1994.

[KOO95] Gene Kim, Hilarie Orman, and Sean OMalley. "Implementing a Secure rlogin Environment". In *The Fifth USENIX Security Symposium, Salt Lake City, Utah*, June 1995.

[KPS95] Charlie Kaufman, Radia Perlman, and Mike Speciner. *"Network Security: Private Communication in a Public World"*. Prentice Hall, 1995.

[Lam81] L. Lamport. "Password Authentication with Insecure Communication". *Communications of the ACM*, Vol. 24(11), November 1981.

[Lin93] J. Linn. "Generic Security Service Application Program Interface". *RFC 1508*, September 1993.

[LM91] X. Lai and J. Massey. "A Proposal for a New Block Encryption Standard". In *Proceedings, EUROCRYPT '90*, pages 389–404. Springer-Verlag, 1991.

[Man94] Robert J. Manchek. "Design and Implementation of PVM Version 3". Master's thesis, University of Tennessee, Knoxville, 1994.

[Mer78] Ralph C Merkle. "Secure Communications Over Insecure Channels". *Communications of the ACM*, Vol. 21, n. 4:294–299, 1978.

[Mic88] Sun Microsystems. "RPC: Remote Procedure Call Protocol specification version 2". *RFC 1057*, June 1988.

[MS95] P. Metzger and W. Simpson. "IP Authentication using Keyed MD5". *RFC 1828*, August 1995.

[Mul90] Sape Mullender. *"Distributed Systems"* . ACM Press, 1990.

[Neu95] Michael Neuman. "Monitoring and Controlling Suspicious Activity in Real-time With IP-Watcher". In *Proceedings of the 11th Annual Computer Security Applications Conference*, December 1995.

[NIS92] NIST. "The Digital Signature Standard, proposal and discussion". *Communications of the ACM*, Vol. 35(7), July 1992.

[NIS93] NIST. "Secure Hash Algorithm". Federal Information Processing Standard (FIPS) Publication - 180, May 1993.

[NS78] R. Needham and M. Schroeder. "Using Encryption for Authentication in Large Networks of Computers". *Communications of the ACM*, Vol. 21, no. 12:993–999, December 1978.

[Plu82] David C. Plummer. "An Ethernet Address Resolution Protocol". *RFC 826*, November 1982.

[Pos81] J. Postel. "Internet Protocol". *RFC 791*, September 1981.

[Riv92a] R. Rivest. "The MD4 Message-Digest Algorithm". *RFC 1320*, April 1992.

[Riv92b] R. Rivest. "The MD5 Message-Digest Algorithm". *RFC 1321*, April 1992.

[RS84] R.L. Rivest and A. Shamir. "How To Expose An Eavesdropper". *Communications of the ACM*, Vol. 27, no. 4:393–395, April 1984.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM*, Vol. 21, no. 2:120–126, February 1978.

[Sch96] Bruce Schneier. *"Applied Cryptography"* . John Wiley and Sons, Inc., second edition, 1996.

[SHS93] David R. Safford, David K. Hess, and Douglas Lee Schales. "Secure RPC Authentication (SRA) for TELNET and FTP". In *The Fourth USENIX Security Symposium, Santa Clara, California*, pages 63–67, October 1993.

[SNS88] J. Steiner, C. Neuman, and J. Schiller. "Kerberos: An Authentication Service for Open Network Systems". In *Usenix Conference Proceedings, Dallas, Texas*, February 1988.

[Sta95] William Stallings. *"Network and Internetwork Security"* . Prentice Hall, 1995.

[TA91] J.J. Tardo and L. Alagappan. "SPX: Global Authentication using Public Key Certificates". In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 232–244, May 1991.

[Tat95] Tatu Ylonen. "The Secure Shell (SSH) Remote Login Protocol". Internet Draft - Work in Progress, November 1995. URL: http://www.cs.hut.fi/ssh.

[Tea95] NSA Cross Organization CAPI Team. "Security Service API: Cryptographic API Recommendation", June 1995.

[Tuc79] W. Tuchman. "Hellman Presents No Shortcut Solutions to DES". *IEEE Spectrum*, July 1979.

[US 77] US National Bureau of Standards. "Data Encryption Standard". Federal Information Processing Standard (FIPS) Publication - 46, January 1977.

[US 80] US National Bureau of Standards. "DES Modes of Operation". Federal Information Processing Standard (FIPS) Publication - 81, December 1980.

[US 85] US National Bureau of Standards. "Computer Data Authentication". Federal Information Processing Standard (FIPS) Publication - 113, May 1985.

[VTB95] David Vincenzetti, Stefano Taino, and Fabio Bolognesi. "STEL: Secure TELnet". In *The Fifth USENIX Security Symposium, Salt Lake City, Utah*, June 1995.

[Wie93] M.J. Wiener. "Efficient DES Key Search", August 1993. Presented at the rump session of CRYPTO '93.

[Wil95] William Simpson and Phillip Karn. "The Photuris Session Key Management Protocol". Internet Draft - Work in Progress, November 1995. URL: ftp://ds.internic.net/internet-drafts/.

[WL92] T. Woo and S. Lam. "Authentication for Distributed Systems". *Computer*, January 1992.

[ZG95] Honbo Zhou and Al Geist. "Faster Message Passing in PVM". In *Proc. of the First International Workshop on High Speed Network Computing (HiNet-95), Santa Barbara, CA*. IEEE Press, April 1995.

Vita

Nair Venugopal (Venu) was born in Trivandrum, India on May 30, 1969. After initial schooling in different Indian cities including Hyderabad, Bangalore, and Trivandrum, he completed his high school education in June 1986. He graduated with a B.Tech. in Computer Science from the College of Engineering, Trivandrum in December 1990. He immediately started working for the National Center for Software Technology (NCST), Bombay, India and continued to work with NCST till July 1993. A strong desire to see the world brought him over to the United States of America where he joined the Masters program in Computer Science at the University of Tennessee, Knoxville in Fall 1993. After getting his M.S. he intends to head for Santa Clara, California to make sufficient money to fund his future travel urges.