ORNL/TM-13203

Computer Science and Mathematics Division

Mathematical Sciences Section

SECURE PVM

Thomas H. Dunigan and Nair Venugopal

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367
thd@ornl.gov venu@cs.utk.edu

Date Published: August 1996

Research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Lockheed Martin Energy Research Corp.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-96OR22464

Contents

SECURE PVM

Thomas H. Dunigan and Nair Venugopal

Abstract

This research investigates techniques for providing privacy, authentication, and data integrity to PVM (Parallel Virtual Machine). PVM is extended to provide secure message passing with no changes to the user's PVM application, or, optionally, security can be provided on a message-by-message basis. Diffie-Hellman is used for key distribution of a single session key for n-party communication. Keyed MD5 is used for message authentication, and the user may select from various secret-key encryption algorithms for message privacy. The modifications to PVM are described, and the performance of secure PVM is evaluated.

1. Introduction

PVM, Parallel Virtual Machine [8], is a message-passing system that allows a collection of heterogeneous computers on a network to function as a virtual parallel computer. 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. Historically, PVM has been used by researchers to speed up serial computing applications by harnessing the power of workstations and supercomputers connected by a network. As PVM's popularity has grown, it is now used in commercial applications and in applications where some of the data may need to be protected. This report describes describes extensions made to PVM to provide privacy, authentication, and integrity to the data passed between PVM tasks.

The development of secure PVM had the following design goals:

- compatible with PVM architecture
- transparent
- either full-time or on-demand encryption
- selectable encryption technology
- extensible
- compatible with other Department of Energy (DOE) security projects

PVM can be compiled and run on a wide variety of UNIX systems without requiring system privileges. Our secure PVM design tries to stay within those architectural guidelines. The implementation should be transparent, in that a user should not have to change their application to incorporate the features of a secure PVM. On the other hand, it is likely (and will be shown) that encryption slows PVM performance, so the user should be able to turn encryption on or off on a message-by-message basis. For performance, or encryption strength, or legal reasons, a user may also wish to select the form of message encryption used. The PVM implementer should be able to add other encryption modules or hashing algorithms to the secure PVM implementation. Finally, the funding for secure PVM is part of a larger DOE project looking at key distribution and user authentication. Secure PVM should seek to utililize the proposed authentication and key distribution technologies of these other research tasks.

This report is a subset of a larger research effort [19] that looked at several research questions in designing and implementing secure distributed applications. Research issues studied in developing a secure PVM include:

- key generation (random numbers)
- key distribution
- n-party session key
- crypto API selection
- crypto hash functions for authentication
- crypto algorithm performance

This report summarizes the choices made within these research issues subject to the design goals of secure PVM.

The following section explores the vulnerabilities of PVM and some of the counter-measures that can be deployed to reduce these vulnerabilities. Section 3 describes extensions to PVM message passing that were made to provide authentication, integrity, and privacy. Section 4 analyzes the performance of secure PVM, and the final section looks at the limitations of the current implementation.

2. Vulnerabilities and countermeasures

2.1. PVM architecture

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 [13] is the authoritative reference for the design and implementation of PVM Version 3. The PVM User's Guide [8] 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 2.1); it provides a point of contact, authentication, process control,

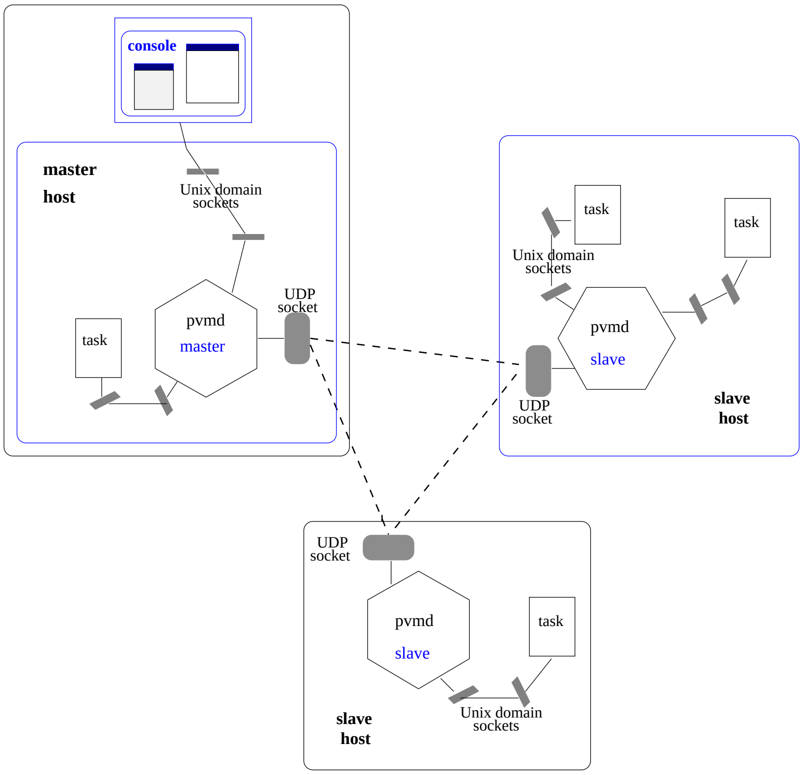

Figure 2.1: *Partial anatomy of PVM*

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 [13].

[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.

2.2. PVM vulnerabilities

Currently, PVM depends on UNIX and the standard TCP/IP protocol suite for security. Vulnerabilities exist in the mechanisms used by PVM in starting up remote daemons and in the communication between PVM tasks on different hosts. This study is based on several assumptions. First, 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. Second, this research does not address the issue of authenticating a user to a remote host. User authentication mechanisms like the Kerberos [17] or DCE [10] infrastructure are assumed to be available on the PVM hosts.

The standard PVM implementation provides three mechanisms to start a slave `pvmd` on a remote host.

- `rexec`
- `rsh`
- manual startup

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`). 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 host on which the slave `pvmd` is being started. With "password-sniffing" attacks becoming very common [5], it would be quite easy for an attacker to capture the PVM user's password if `rexec` is used. Manual logins and daemon startup are also subject to sniffing attacks.

Trusted hosts and *rsh/rlogin* can eliminate password sniffing, but 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 [3] [6]. Also, an established session can be intercepted and co-opted by an attacker by IP-splicing[3] attacks [15] [11].

In secure PVM, the above risks can be reduced by using existing mechanisms like Kerberos [17] or new software like SSH [18] or STEL [20]. The user could require that the set of machines used in his/her PVM application use Kerberos

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

for user authentication and for `rsh` services used in spawning slave PVM daemons. 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. Instead of Kerberos, SSH or STEL can also be used to securely spawn processes on remote hosts.

The other vulnerability in standard PVM is that PVM messages between hosts are not protected. 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) [7] key exchange[4]. The slave startup protocol was extended to accomplish 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. Venugopal [19] analyzes these various algorithms in terms of speed, cryptographic strength, and licensing. Figure 2.2 illustrates the relative performance of these algorithms. The tests were run on a Sun SPARC5 with 96MB of memory uder SunOS 4.1.4. The compiler used was gcc, and optimization was enabled with a "-O" flag.

By using a modular cryptographic API[5], 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 details of these extensions are described in the following section.

3. Implementation of Secure PVM

The security 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

[4]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.

[5]Based on the SSH distribution.

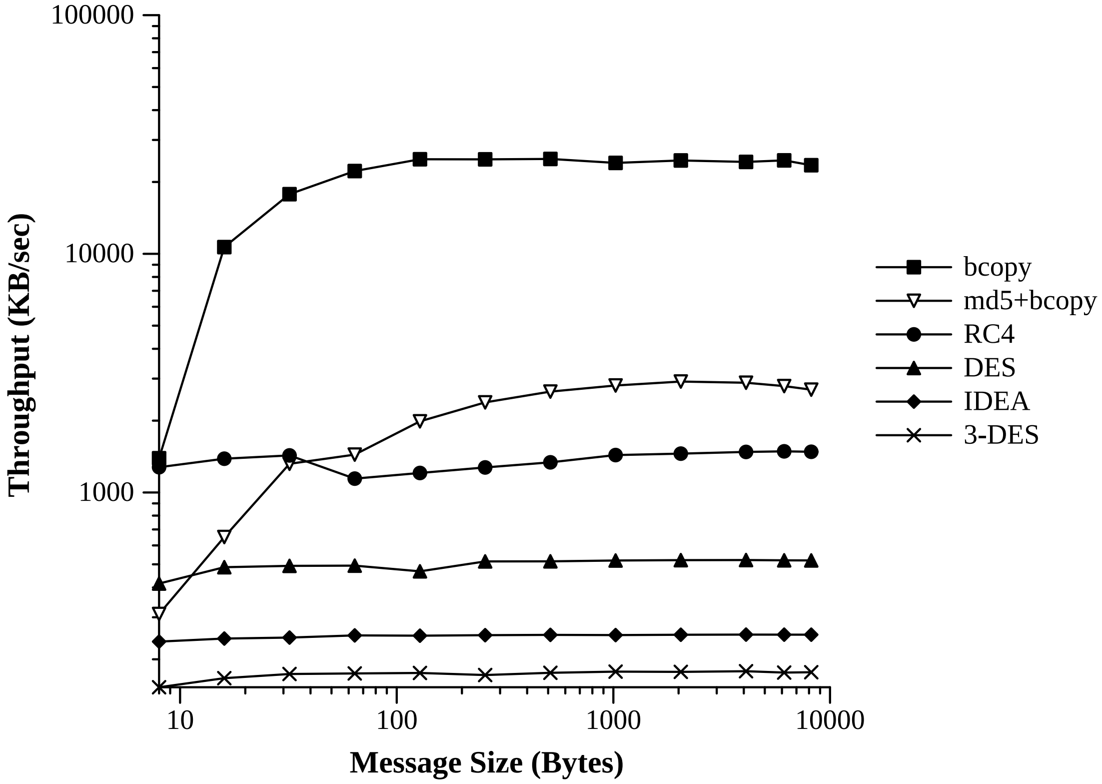

Figure 2.2: *Encryption/Hashing performance within a process*

describe the extensions made to standard PVM for enhanced security.

3.1. Changes to the user interface

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[6] to `pvm_mkbuf()` (or `pvm_initsend()`[7]). Table 3.1 lists the possible values for the encoding format.

Table 3.1: *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 );
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

[6] 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()`.

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

the encryption field in the message fragment and encrypts the data accordingly. Only messages destined for remote hosts get encrypted (§ 3.2).

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.

3.2. Internal changes

To provide message integrity, authenticity, and privacy a number of internal changes were made to PVM. The source for the PVM daemon was modified to support the new command line arguments, and new source was added to support key generation, key distribution, encryption, and authentication. Several of the internal data structures were modified and the message headers were expanded. Extensions were made to `libpvm` to add encryption and authentication. Details of these internal changes are desribed in more detail in the following sections.

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 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 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[8]. 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, [19]) on the DH key exchange protocol. However, such attacks are not trivial to mount[9]; 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 3.1 shows a host being added to the virtual machine[10]. 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[11]) and are not yet added to the host table. The master forks the *pvmd'*, passing it a list of hosts and commands to execute[12]. The *pvmd'* uses `rsh` or manual startup to start each pvmd, pass it parameters, and gets configuration data back from the newly started slave.

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[13]. When several hosts are added at once, startup is done in

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

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

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

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

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

[13] 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.

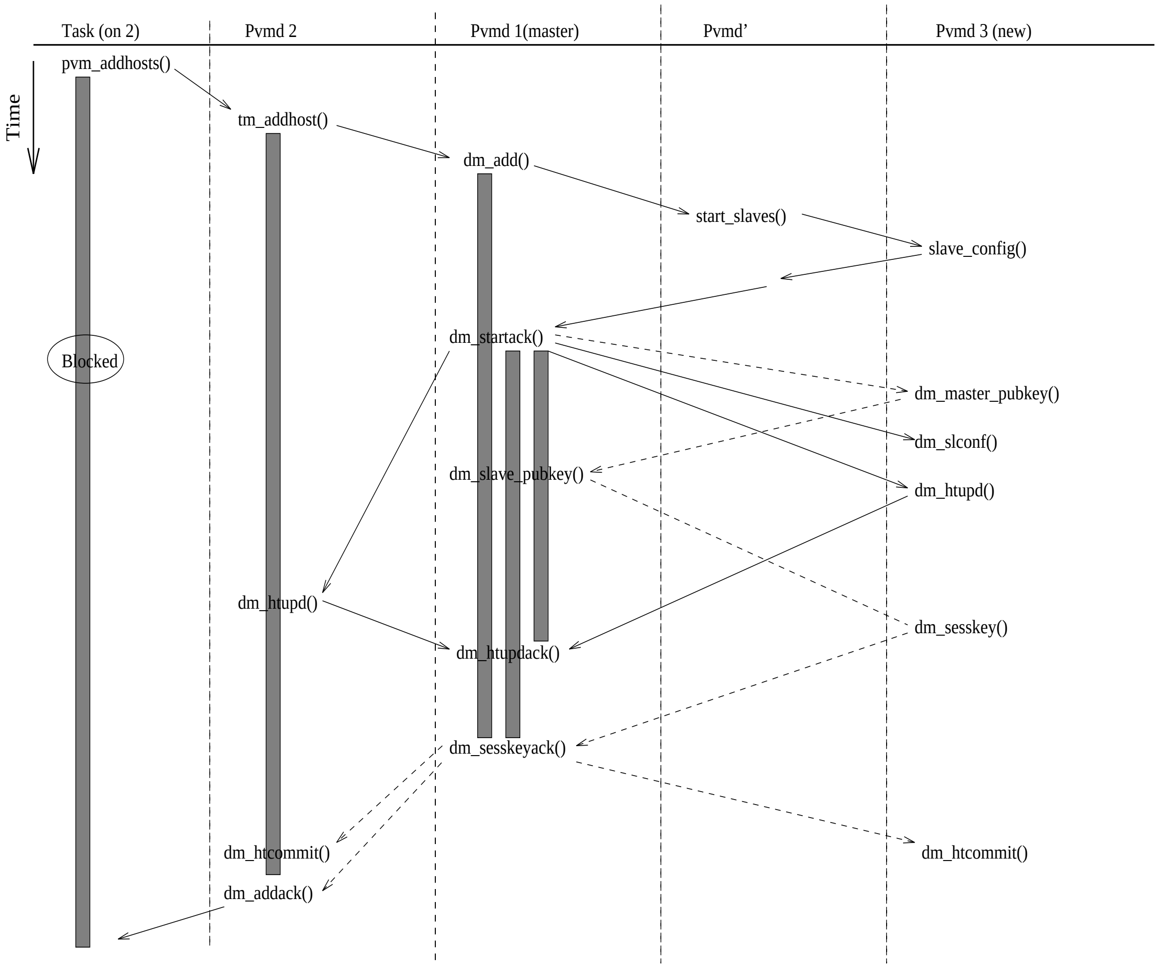

Figure 3.1: *Timeline of key-exchange operation*

parallel[14]. 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.

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

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.

PVM messages The pvmd and libpvm use the same message header (see Figure 3.2). *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* field is reserved for future use[15]. No modifications were made to the message header in secure PVM.

	Byte 0 1 2 3
0	Code
4	Encoding
8	Wait Context ID
12	(reserved for checksum)

Figure 3.2: *PVM Message header*

PVM daemons communicate with one another through UDP sockets. UDP is an unreliable delivery service that 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

[15]Secure PVM uses a packet checksum/hash.

has a packet header with the requisite information (see Figure 3.3). Multi-byte values are sent "most significant byte first", *i.e.*, in (Internet) *network byte order.*

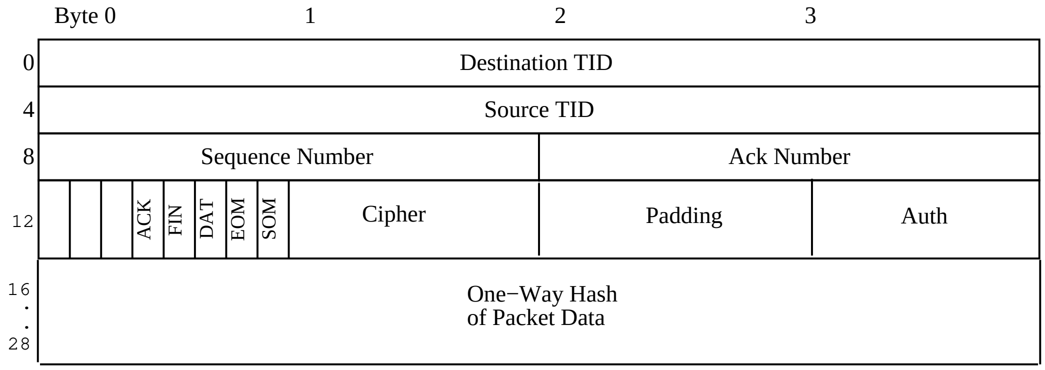

Figure 3.3: Pvmd-Pvmd *packet header for secure PVM*

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.

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 that receives the packet computes the message digest from the relevant fields[16] 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, `netoutput()` 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 [14]. 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.

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

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[17] 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[18]. `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 3.3) 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.

PVM Key Generation PVM uses the Diffie-Hellman key exchange to distribute the secret session key among all the pvmds. 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[19] implemented using the GNU Multiple-Precision (GMP) package.

[17] Based on SSH's cipher API.

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

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

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-id's, 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*[20]. 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.

3.3. PVM and Kerberos

During the initial stages of this research, the integration of PVM with Kerberos and/or DCE [21] was investigated. For this purpose, a Kerberos Version 5 Beta 5 KDC was installed along with the Kerberized versions of the Berkeley 'r' commands[22].

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[23]. 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, *forward-*

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

[21]DOE has several projects based on Kerberos and DCE.

[22]Kerberos V5 Beta5 was the latest release available when this research started.

[23]This suggests the need for a well-known PVM port on each machine.

able tickets do not function correctly in the Kerberos V5 Beta5 distribution[24] and hence this could not be tested.

Kerberos does provide a programming interface for developing secure applications, and if all of the hosts participating in a PVM application were running Kerberos, then the Kerberos API could be used by PVM to provide message integrity, authenticity, and privacy. At this time, Kerberos is not widely deployed, so our implementation of secure PVM is self-contained and does not depend on Kerberos services. DCE is based on Kerberos V5 and authentication and key management services are roughly equivalent. The DCE programming interface is RPC-based and does not fit the PVM programming model, though DCE 1.1 will provide a GSS-based programming interface [12].

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.

4.1. Comparison of Pvmd Slave Startup Times

The slave pvmd startup protocol (§ 3.2) was extended to do the Diffie-Hellman (DH) handshake and to distribute the PVM secret session key among all the pvmds. Figure 4.1 shows the time taken to start up one to eight slaves in parallel. Two sets of startup times, one for standard PVM and one for PVM with security extensions are plotted side by side.

It can be seen that the startup times in secure PVM increases with the increase in the number of slaves added concurrently[25]. 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.

[24] Joe Ramus (ramus@nersc.gov), private email.
[25] PVM can initiate the startup of five slave pvmds in parallel via `rsh`.

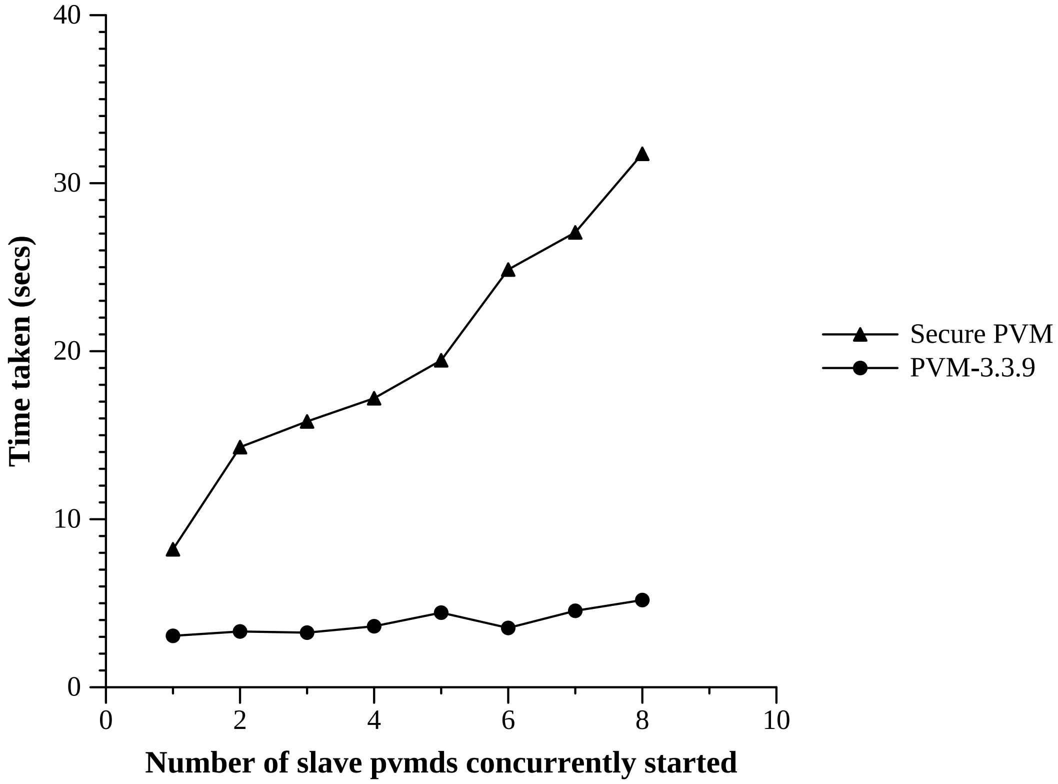

Figure 4.1: *Time taken to start 1-8 slave pvmds*

- 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*.

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 4.2 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[26]. 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

[26]When encryption is turned on, authentication is implicitly enabled (§ 3.2).

are sent round-trip, and the total time difference is divided in half. It can be observed from Figure 4.2 that adding encryption/authentication extensions to PVM degrades the throughput performance.

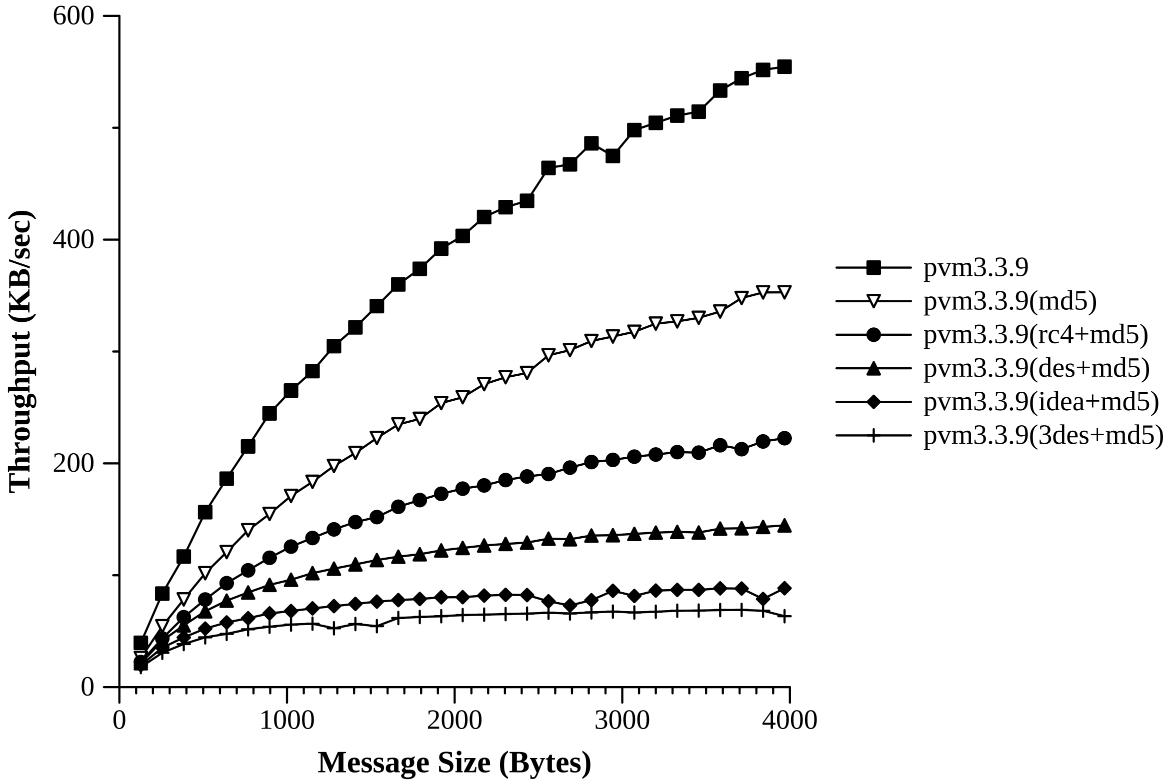

Figure 4.2: *Comparison of throughput performance between two PVM hosts*

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 4.1 shows the results obtained from running this program with standard PVM, secure PVM with authentication enabled, and secure PVM with encryption enabled.

Table 4.1: *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

5. Limitations and Further Work

This research focussed on extending PVM message passing to provide message integrity, authenticity, and encryption. The use of different encryption/authentication algorithms and their impact on the performance of PVM were studied. 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.

5.1. 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.

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 [4]. 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 or DCE 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.

5.2. 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 [13]. 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]. 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. We chose a custom API developed from the SSH source distribution for secure PVM. Future work should re-investigate using GSS [12] as the underlying API for secure PVM.

The Internet community has recognized the need for having an integrated security framework [1]. 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.

5.3. Legal Issues

Foreign accessibility to strong cryptography is considered to compromise communications intelligence. "According to the U.S. government, cryptography can be a munition" [16]. Most packages that 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. For the current status and availability of secure PVM, the reader is invited to visit *http://www.epm.ornl.gov/pvm*.

6. Acknowledgements

A special thanks to Al Geist and Bob Manchek of the PVM design team for their patient assistance in this work.

7. References

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

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

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

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

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

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

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

[8] 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.

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

[10] Wei Hu. *"DCE Security Programming"* . O'Reilly, 1995.

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

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

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

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

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

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

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

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

[19] Nair Venugopal. The design, implementation, and evaluation of cryptographic distributed applications: secure PVM. Technical report, University of Tennessee, 1996. Master's Thesis.

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