

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

ATM SECURITY VIA "STARGATE" SOLUTION

by

Katrina Hensley
and
Fredrick Ludden

September 1999

Thesis Advisor:

Geoffrey G. Xie

Approved for public release; distribution in unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE : ATM Security Via "Stargate" Solution			5. FUNDING NUMBERS
6. AUTHOR(S) Hensley, Katrina and Ludden, Fredrick			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution in unlimited.			12b. DISTRIBUTION CODE
13. ABSTRACT (maximum 200 words) In today's world of integrating voice, video and data into a single network, Asynchronous Transfer Mode (ATM) networks have become prevalent in the Department of Defense. The Department of Defense's critical data will have to pass through public networks, which causes concern for security. This study presents an efficient solution aimed at authenticating communications over public ATM networks. The authenticating device, "Stargate," utilizes a high speed, low level authentication protocol that offers the low cost, flexibility, and extensibility of software, while still capable of yielding performance comparable to hardware-based authentication.			
14. SUBJECT TERMS Authentication, Asynchronous Transfer Mode (ATM), Key Management, Security, Networking			15. NUMBER OF PAGES 89
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

Approved for public release; distribution in unlimited.

ATM SECURITY VIA "STARGATE" SOLUTION

Katrina Hensley
Captain, United States Marine Corps
B.A., University of Oklahoma, 1991

Fredrick Ludden
Captain, United States Army
B.S., Virginia Tech, 1989

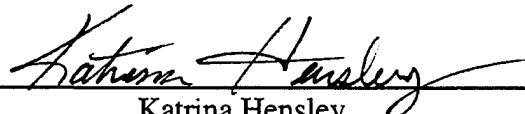
Submitted in partial fulfillment of the
requirements for the degree of

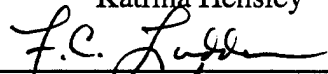
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the


NAVAL POSTGRADUATE SCHOOL
September 1999

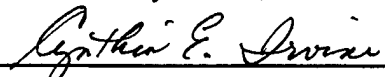
Authors:


Katrina Hensley


Fredrick Ludden

Approved by:


Geoffrey G. Xie, Thesis Advisor


Cynthia E. Irvine, Second Reader


Dan Boger, Chair, Department of Computer Science

ABSTRACT

In today's world of integrating voice, video and data into a single network, Asynchronous Transfer Mode (ATM) networks have become prevalent in the Department of Defense. The Department of Defense's critical data will have to pass through public networks, which causes concern for security. This study presents an efficient solution aimed at authenticating communications over public ATM networks. The authenticating device, "Stargate," utilizes a high speed, low level authentication protocol that offers the low cost, flexibility, and extensibility of software, while still capable of yielding performance comparable to hardware-based authentication.

TABLE OF CONTENTS

I. INTRODUCTION.....	1
A. MOTIVATION.....	1
B. EXECUTIVE SUMMARY.....	2
C. THESIS OUTLINE.....	2
II. BACKGROUND.....	3
A. INTRODUCTION.....	3
B. ATM.....	3
C. ATM SECURITY.....	6
1. Threats.....	6
2. Existing Technology.....	6
D. LOW-LEVEL AUTHENTICATION PROTOCOLS.....	8
III. DESIGN AND IMPLEMENTATION OF "STARGATE" SOLUTION.....	11
A. INTRODUCTION.....	11
B. PROTOTYPE ENVIRONMENT.....	11
1. Hardware.....	11
a. ATM Cell Generator and Receiver.....	11
b. Washington University Gigabit ATM Switch.....	11
c. ATM Switch controller.....	12
2. Software.....	12
a. ATM Cell Generator and PVC creation.....	12
b. ENI155P ATM NIC Driver.....	12
c. Gigabit Network Switch Controller.....	13
d. Jammer.....	13
e. NetBSD Native ATM networking protocol.....	13
f. MD5 Message Digest Algorithm.....	15
g. Network Time Protocol (NTP).....	15
C. STARGATE CONCEPTUAL DESIGN.....	17
1. Relationship to CellCase Technology.....	17
2. Relationship to LLPA Protocol.....	18
3. Relationship to WUGS Technology.....	18
D. STARGATE IMPLEMENTATION.....	18
1. Testbed Layout.....	18
2. "Public Network" Simulation Layout.....	19
3. Cell Authentication.....	20
a. Signing.....	20
b. Authentication.....	21
4. Key Management.....	22
a. Key Distribution Center responsibilities.....	23
b. Stargate key management responsibilities.....	27
IV. PERFORMANCE EVALUATION.....	31
A. TEST DESIGN AND DATA COLLECTION.....	31
B. RESULTS.....	32
V. LESSONS LEARNED AND RECOMMENDATIONS.....	35
A. AUTHENTICATION LESSONS LEARNED.....	35
1. Setting up the Testbed.....	35
2. Authentication Performance Issues.....	36
B. KEY MANAGEMENT LESSONS LEARNED.....	36
1. Language Integration.....	37

2. Communication Integration	37
C. SUGGESTED FURTHER RESEARCH	38
APPENDIX A. JAMMER SCRIPT	39
APPENDIX B. BSD KERNEL MODIFICATION CODE	41
A. <i>README</i>	41
B. <i>IF_ATM.H</i>	42
C. <i>IF_ATMSUBR.C</i>	48
APPENDIX C. ATM AUTHENTICATION CODE	57
A. <i>ATM_AUTH.H</i>	57
B. <i>ATM_AUTH.C</i>	57
APPENDIX D. KEY MANAGEMENT CODE	63
A. <i>KDC.JAVA</i>	63
B. <i>STARGATED.C</i>	68
LIST OF REFERENCES	69
BIBLIOGRAPHY	71
INITIAL DISTRIBUTION LIST	73

LIST OF FIGURES

Figure 1. ATM Cell Formats	4
Figure 2. AAL5 Framing and Segmentation	5
Figure 3. CellCase Concept	7
Figure 4. Conceptual Stargate Design	17
Figure 5. Physical Layout	19
Figure 6. Logical Layout	20
Figure 7. Signing Operation	21
Figure 8. Authentication Operation	22
Figure 9. Key Distribution Center Conceptual Design	23
Figure 10. Key Management Timing Relationships	26
Figure 11. Multiple Stargate Timing Relationship	27
Figure 12. Key Window Concept	28
Figure 13. Timing Implementation of Throughput Test	32
Figure 14. ATM Authentication Module Performance	34

LIST OF TABLES

Table 1. Performance Data for <i>sign()</i> and <i>auth()</i> Functions.....	33
--	----

ACKNOWLEDGMENT

The authors would like to thank Geoffrey Xie for providing guidance and mentorship as well as expert assistance with “C” and mathematical formulas. Special thanks goes to Cary Colwell, who provided invaluable assistance throughout our thesis work. He was always available to listen, troubleshoot, or provide brilliant insight without which we would have been lost.

Captain Ludden would also like to extend special thanks to his wife, Terri, and family who put up with late nights and odd schedules while he completed his studies. He would also like to give a special tribute to William R. Stevens, who recently passed away, for his unsurpassed technical expertise in the BSD TCP/IP network protocol.

Captain Hensley would also like to extend special thanks to her husband, David, for his constant support. His encouragement was the foundation for success.

I. INTRODUCTION

A. MOTIVATION

The Department of Defense (DoD) is continuing to increase its use of the Internet and other unsecured networks to pass data. Throughout the world, the Department of Defense is upgrading its telecommunication networks to provide support for voice and video in addition to text and image data. However in many cases, the budgets for maintaining the network infrastructure are not increasing; therefore, finding solutions that maximize efficiency and performance at a reasonable cost is driving network policies and acquisitions. The emergence of broadband services, such as Asynchronous Transfer Mode (ATM), will allow the same network to support voice, data and video. The capability to support all forms of information on one network allows for easier management and reduced costs of installation and maintenance. With an optimized ATM network, IT managers can gain up to 95% efficiency of the network [Ref. 1]. Furthermore, ATM offers guaranteed Quality of Service, important for critical DoD applications/users. ATM also offers an additional advantage, which is of particular interest to the Department of Defense. This advantage is ATM's ability to carry data for existing legacy applications while laying the foundation for emerging IP and multimedia applications.

The growth of ATM is on the rise by civilian companies and DoD. The Defense Information Services Agency (DISA) has proceeded with standards for implementing ATM infrastructures for the support of the Defense Integrated Services Network (DISN). Another ATM network initiative is the Navy Wide Internet (NWI), which is underway in San Diego, California. Furthermore, the majority of commercial telephony and data networks are already connected by ATM networks operated by companies like MCI WorldCom and AT&T.

The reality is that the DoD can not own the entire path upon which its data must travel. Therefore, public ATM networks will be used to link DoD sites together. The use of public ATM networks increases the concern about security of DoD's sensitive data. To combat possible security problems, several solutions have been proposed for networks that can be adapted to work with the ATM infrastructure. These solutions serve as a basis for the Stargate concept and will be discussed in Chapter II.

The security threats that Stargate is designed to thwart are those attacks that affect the integrity and authenticity of network traffic, and also to some extent the availability of network resources. Integrity ensures that only authorized parties are allowed to modify transmitted messages. Unauthorized editing, replaying or changing the status of the transmitted message must be detected. Authentication ensures that the origin of a message is correctly identified and that identity is not false. Examples of "active" attacks that affect integrity, authenticity and availability include attacks such as IP spoofing, identity theft, E-mail forgeries, playback attacks, E-mail or IP flooding, and inserting viruses into messages. While other attacks do exist, the Stargate device is not meant to be an all-encompassing security solution. Stargate is intended to be an inexpensive first line of defense which may be used in conjunction with other security techniques such as strong encryption to provide data confidentiality.

B. EXECUTIVE SUMMARY

This thesis focuses on ATM cell origin authentication and authentication key management for end-to-end transmissions across public networks. The goal of this thesis is to demonstrate a low-cost, high-speed method for authenticating ATM cells via a device called Stargate. This device will apply digital signatures to ATM traffic, which can be deciphered by the destination Stargate device to verify the source and integrity of the ATM traffic. Cells that can not be authenticated successfully will be thrown away. Additionally, an authentication solution involving keys must demonstrate successful key management. Another goal of this thesis is to study the management of dynamic authentication key tables. Specifically, this thesis looks at efficiency and synchronization issues involved with creating and maintaining authentication tables between a Central Authority, which manufactures authentication keys, and the Stargate device.

C. THESIS OUTLINE

This thesis is broken into five parts. Chapter II will give an overview of ATM and ATM security, to include the influences that contributed to the Stargate design and key management technique. Chapter III focuses on the design and implementation of the Stargate device. Chapter IV will focus on the performance evaluation of the Stargate device in a small testbed network. Finally, Chapter V will focus on the lessons learned and recommendations concerning additional research work involving Stargate.

II. BACKGROUND

A. INTRODUCTION

This chapter discusses background information on ATM, ATM security, and some existing proposals for establishing secure communications through public networks such as the Internet. Section B describes the ATM protocol. Section C summarizes the existing and proposed security solutions aiming to secure communications over ATM networks. Low-level authentication protocols that provide the framework for our research are discussed in Section D.

B. ATM

Asynchronous Transfer Mode (ATM) is a connection-oriented high speed, low delay switching technology using short, fixed-size packets called "cells". ATM is a transport technology for all types of data (text, voice, video, image, etc.). The asynchronous nature of the technology refers to its non-periodic transmission of data (bursty traffic) that are being transmitted across the ATM network. Primarily, the asynchronous label refers to voice and video data.

ATM networks typically consist of a set of end hosts connected by ATM links to an ATM switch. An ATM switch receives data from the hosts connected to it and forwards the data to the destination. The destination host can either be an end-point or it can be an intermediate ATM switch on the path from the data's source to its eventual destination.

Data are transmitted over an ATM network in "ATM cells". A cell is a fixed-size 53 byte data structure that contains 48 bytes of data and 5 bytes of control information. Figure 1 depicts the two possible cell structures; the User-Network Interface (UNI) and the Network-Network Interface (NNI). The UNI cell structure is used between the user and the switch. The NNI cell structure is used between switches. Each cell's control information includes a "virtual circuit" number. This number is used by ATM switches to determine where the cell should be sent, and it is used by the receiving end hosts to determine which process' buffer should receive the data. The generic flow control (GFC) field of the UNI cell structure supports simple multiplexing implementations.

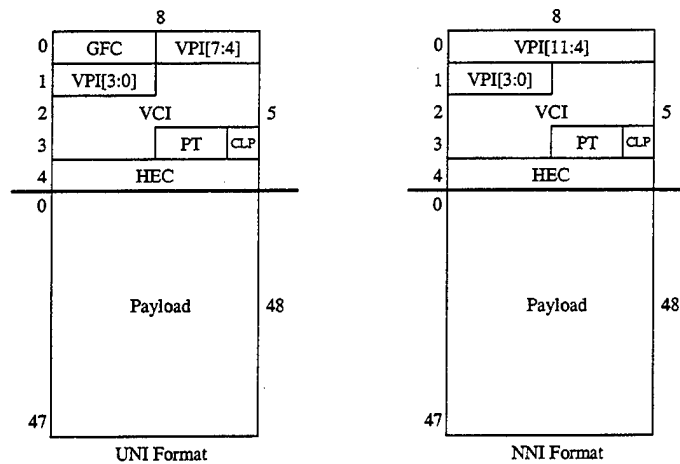


Figure 1. ATM Cell Formats.

The virtual circuit number is composed of two numbers: the virtual circuit identifier (VCI) and virtual path identifier (VPI). The numbers in brackets after VPI refer to the bit numbers (e.g., VPI[3:0] means that bits 3, 2, 1, and 0 are contained in this field). All data sent over an ATM network are associated with a virtual circuit. There are two types of virtual circuits: permanent virtual circuits (PVCs) and switched virtual circuits (SVCs). PVCs are usually set up in an ATM switch by a network administrator. SVCs are connections that are established “on demand” through the use of complex signaling protocols.

PT or payload type discriminates between a cell payload carrying user information and one carrying management information, such as signaling mechanisms to establish SVCs. Cell Loss Priority (CLP) indicates the loss priority of an individual cell. Either the end user or the network may set this bit. A value of 0 in the CLP field means that the cell is of the highest priority and least likely to be discarded by the network during periods of congestion. Header Error Control (HEC) provides error checking for the header only.

The 48 byte data area of an ATM cell is quite small when compared to the data area of an Ethernet packet. To address this problem, ATM includes a number of “ATM

adaptation layers" (AALs). The most widely used framing methods are AAL0 and AAL5. AAL0 allows a host to send and receive individual ATM cells. AAL5 allows a host to send and receive frames up to 64KB in size. When a host sends a AAL5 frame, the ATM host's network interface segments it up into ATM cells. When the cells arrive at the receiving host, these cells are reassembled into a frame by the receiving machine's ATM network interface. AAL5 allows hosts to send and receive frames and not have to worry about how to package data into small ATM cells. Figure 2 illustrates the framing of user data into AAL5 frames and the segmentation of those frames into ATM cells. An eight byte trailer is appended to every AAL5 frame. This trailer consists of a variable length PAD such that the entire frame is a multiple of 48 bytes so that it can be directly segmented into cell payloads. User-to-User (UU) indication field is used for the transparent transfer of user-to-user information. The Common Part Indicator (CPI) is used to align the trailer to 64 bits. The length and CRC are similar in function to a standard TCP/IP packet.

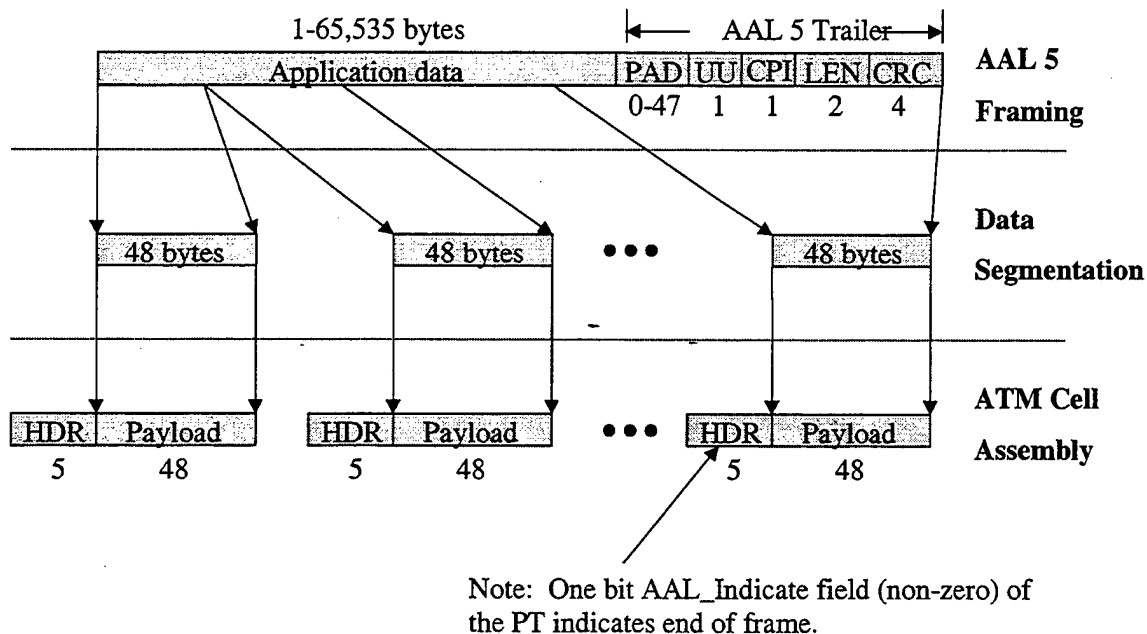


Figure 2. AAL5 Framing and Segmentation.

One advantage of ATM is its use of virtual circuits which makes it easier to provide network performance guarantees to individual applications. Each active virtual circuit on an ATM network can be allocated a fixed portion of the network's bandwidth.

If a host attempts to exceed the allocated bandwidth for a virtual circuit, the ATM switch may drop the cells rather than allow the host to congest the network and affect other circuits. ATM delivers the cells at several standard speeds including 155 Mbps, 622 Mbps, 1.2 Gbps, and 2.4 Gbps.

C. ATM SECURITY

1. Threats

ATM networks are able to carry a variety of types of information - voice, video, and data - which can be of a private or sensitive nature. Therefore, like any other communication network, ATM networks are vulnerable to some of the same threats and attacks. They are listed below:

- Violation of data secrecy through eavesdropping.
- Unauthorized modification or corruption of information.
- Impersonation of authorized sender/recipient by masquerading.
- Repudiation of a message sent/received.
- Denial of service by blocking or saturating the network.

To counteract these threats, ATM networks require security services such as information confidentiality, integrity, authentication, access control and non-repudiation. The first service, confidentiality, requires some sort of encryption to render the payload unreadable to malicious persons. The last two, non-repudiation and denial of service, are not addressed by our solution. The second and third, integrity and authentication, can be provided by the same encryption mechanism that provides confidentiality. However, good encryption incurs a high overhead in computation and thus can affect throughput. In an attempt to overcome this limitation, encryption algorithms have been moved into hardware. This solution has the drawback of increased cost. We believe that integrity and authentication can be supported by software mechanisms that yield high throughput and are inexpensive and thus provide a first-line of defense to a more costly encryption scheme.

2. Existing Technology

For the Stargate device to be successful, the technology has to be at least as efficient as existing commercial products, yet cheaper to encourage the promulgation of

the device throughout the Department of Defense. To accomplish this goal, there were several innovations that influenced the Stargate design. This chapter introduces each of these influences.

Many existing security devices are designed with an IP network in mind. One impressive device, called CellCase, is marketed by Celotek Corporation. This non-key agile device provides a hardware solution for encrypting data that are passed between two different, "trusted" networks, using a public ATM network as the link between them. Figure 3 shows the conceptual design of a virtual private network using the CellCase solution. A hardware encryption device positioned between each private ATM LAN and the public ATM network provides the full spectrum of security services, except for prevention of denial of service. Each CellCase node ranges from \$40,000 to \$52,000 depending on speed and level of security. CellCase45 operating at T3 speed with single DES encryption retails for approximately \$40,000 while CellCase155 operating at OC-3 speed with Triple DES encryption is approximately \$52,000. Both claim to operate at full link speed (T3 or OC-3) while handling 35 secure calls per second with a hardware encryption overhead of approximately 20 μ sec for each call [Ref. 2].

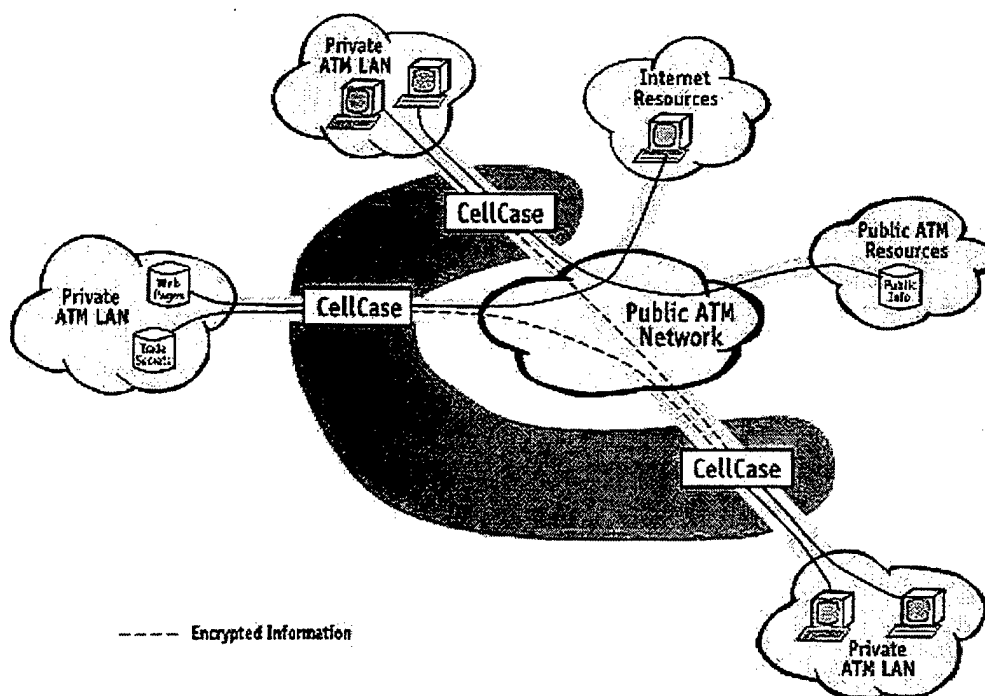


Figure 3. CellCase Concept.

The KG-189 is another available encryption device which was developed by the National Security Agency. The KG-189 provides services similar to CellCase at OC-3(155 Mbps) and OC-12(622 Mbps) speeds. It is also quite expensive, costing from \$48,000 (non-redundant OC-3) to \$63,000 (redundant OC-12).

The NSA also has two new devices in development, FASTLANE and TACLANE. These are key agile in-line network security products that provide confidentiality, data integrity, and authentication (via an add-on called FIREFLY). FASTLANE is designed to operate at network speeds from DS-1(1.54 Mbps) up to OC-12(622 Mbps). The price for these devices was not specified, but should be similar to previous hardware encryption devices. TACLANE is a slower and more rugged version designed for use over twisted pair at the DS-1(1.54 Mbps) and DS-3(25 Mbps) rates. TACLANE is expected to cost approximately \$8000 per device.

The preceding devices are all designed to protect an entire LAN. CryptoRunnerLE produced by Fore Systems is a hardware encryption device designed to protect one end-host. Based on the ForeRunnerLE 155C ATM NIC, it combines a cryptographic daughter card which is FASTLANE/TACLANE compatible. The cost of one unit should be approximately \$1000. This provides a lower cost solution for protecting individual computers and small LANs.

D. LOW-LEVEL AUTHENTICATION PROTOCOLS

While there are several authentication protocols in existence, such as IPsec, one protocol, LLPA, promises a low cost, flexible solution for authenticating IP traffic. LLPA stands for Link Layer Packet Authentication and it was recently developed at the Naval Postgraduate School in Monterey, California [Ref. 3]. LLPA is described as a high-speed authentication protocol for IP traffic. Although there are aspects of the protocol that have yet to be tested, i.e. key management, this protocol was a valuable basis for setting up the Stargate authenticator. The appeal of the LLPA protocol is its performance. Two of the key considerations for the Stargate concept are speed and cost; consequently, the ability to quickly authenticate traffic with software addressed these two concerns. To test the LLPA protocol, two categories of tests were developed. The first test category was designed to test "good" IP traffic, which referred to sending datagrams that were properly signed by the LLPA signing process. The second test category was designed to test "bad" IP traffic, which referred to sending datagrams that were not

properly signed by the LLPA signing process. The initial tests showed signing of IP traffic to take the average time of 153.26 μ s and authentication of "good" IP traffic to take the average time of 150.28 μ s, while it only took on average 0.16 μ s to authenticate typical "bad" IP traffic [Ref. 3]. Furthermore, the suggested approaches to key management were a useful guide for setting up and retrieving authentication keys. Specifically, the use of masks and key windows was designed to speed up performance. The concept of using masks avoids lengthy transmissions of key tables. The use of a key window that holds just three necessary keys avoids having to search large tables for correct keys.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DESIGN AND IMPLEMENTATION OF "STARGATE" SOLUTION

A. INTRODUCTION

This chapter discusses the conceptual design and actual implementation of our proposed solution. Section B is an overview of the hardware used in the test implementation. The software programs used are discussed in Section C. A conceptual overview of the Stargate idea is given in Section D. Finally, Section E describes the actual implementation constructed to test our security solution.

B. PROTOTYPE ENVIRONMENT

1. Hardware

a. ATM Cell Generator and Receiver

A Fore Systems ForeRunner LE 155Mbps ATM network interface card transmitting over OC-3 fiber optic cable was used as the cell generator and receiver for all development and testing. The cards were installed in two Dell brand Intel Pentium processor based systems, a dual Pentium 400 Mhz and a Pentium Overdrive 200 Mhz, operating under Windows NT 4.0. The newer system was significantly faster than the older one, which later caused a performance problem. Both systems were equipped with Ethernet network interface cards and attached to the local area network. IP over ATM was not implemented on these systems, so TCP/IP and ATM communications were completely isolated from each other.

b. Washington University Gigabit ATM Switch

The Washington University Switch (WUGS 20) is a high speed, multicast virtual circuit experimental ATM switch funded by the Defense Advanced Research Projects Agency and the National Science Foundation. It has eight ports, two dual OC-3 (155Mbps) line cards and six G-link (1.2Gbs) line cards. The open architecture enables experimental modification at all levels. The switch's external cell format follows the

ATM standard and therefore the switch can be integrated into both local area networks and wide area networks with little or no modification. One feature in particular, a novel nonblocking cell-recycling architecture, was used in the Stargate project to simulate a public ATM network. The switch has no internal processing capability, it therefore, requires an external controller in the form of a standard PC.

c. *ATM Switch controller*

An Efficient Networks ENI155P 155Mbps ATM network interface card with 512K of on-board RAM transmitting over OC-3 fiber optic cable was used in the ATM switch controller for the WUGS 20. The controller was a generic Pentium Pro 200Mhz processor based system with 128MB of RAM operating under NetBSD 1.32. The system was also equipped with an Ethernet network interface card and attached to the local area network. As with the cell generator and receiver systems, IP over ATM was not implemented, so TCP/IP and ATM communications were completely isolated from one another.

2. Software

a. *ATM Cell Generator and PVC creation*

The Fore Systems ForeRunner LE ATM network interface card came with a CD containing drivers and testing code. The cell generation utility provided by Fore Systems (*perf.c* and *sockutils.c*) created a PVC and generated a continuous stream of ATM cells framed with AAL5. The continuous stream was undesirable for development purposes, so the code was modified to transmit a single user specified text message. This allowed for greater ease of debugging during development. The cell payloads transmitted through the switch could then be intercepted and analyzed for correctness. Once development was completed, the continuous stream was again used to test the throughput and speed.

b. *ENI155P ATM NIC Driver*

The Efficient Networks ENI155P ATM network interface card driver *midway.c* is included in the NetBSD source code under */usr/src/sys/dev/ic/*.

The driver must be added to the kernel configuration file, usually the "generic" configuration file `/usr/src/arch/i386/conf/GENERIC` and then compiled into the kernel. Once compiled the new kernel file, `netbsd`, must be copied over the existing file in the root directory (`/`).

c. Gigabit Network Switch Controller

The purpose of this software is to control one WUGS and hide hardware details as much as possible. The Gigabit Network Switch Controller (GBNSC) monitors the state of the switch and provides access to all hardware details for client applications. Since the switch has no processing engine, a standard PC running GBNSC controls the switch. Access to the switch is through ATM cells transmitted by the controller. These special cells, called control cells, have special formats defined and are sent on VPI 0 VCI 32. The switch's internal routing tables and maintenance registers are modified and monitored by the control cells.

d. Jammer

Jammer is a script-based client utility used to access all the bits in the switch's tables and registers. It connects to GBNSC through a TCP IPC socket and issues pre-defined commands to ping the switch, read or write routing tables, read maintenance registers, or reset/clear the switch. Users can create Jammer scripts to automate routing table programming. See Appendix A for script used with our implementation.

e. NetBSD Native ATM networking protocol

BSD ATM provides support for ATM networking under a traditional BSD-based operating system. The networking subsystem of the BSD kernel is composed of three layers: the socket layer, the protocol layer, and the network interface layer. Transmitted data travels from the application through the socket and protocol layers to the network interface layer. Received data arrives at the network interface and are passed up towards the socket layer. All data in the networking subsystem are stored in a data structure called an "mbuf". There are two basic types of mbufs: small mbufs and large mbufs. Small mbufs contain 108 bytes of data and are used for small data or packet headers. Large mbufs typically contain either 2K or 4K of data. Mbuf structures can be linked together to form an "mbuf chain".

The socket layer has two main roles. First, it transfers the data between a user's address space and kernel layer mbufs. Second, it queues the data between the user and the kernel. If a process attempts to transmit too much data and its socket buffer becomes full, the socket layer will put the process to sleep until room is available.

All networking protocol processing is done at the protocol layer. ATM, TCP, UDP, and IP are all implemented in the BSD networking subsystem's protocol layer. When transmitting, the protocol layer receives data from the socket layer, adds the necessary headers, and passes the packet to the network interface layer for transmission. When receiving, the protocol layer dequeues packets from its input queue, and determines the destination of each packet. If the packet is to be forwarded to another host, then the protocol passes it back to the network interface layer. If the packet is bound for a local process, then the protocol layer enqueues the packet on the receiving process' receive buffer and notifies the socket layer that new data are available.

The network interface layer transfers packets between the networking hardware and the protocol layer. When transmitting, the network interface layer receives packets through its interface queue and transmits them on the network. When receiving, the network interface layer determines which protocol to pass the inbound packet to, enqueues the packet for the protocol, and then schedules a software interrupt to service the protocol.

ATM networking is integrated into the BSD kernel through a device-independent ATM networking layer and a device-specific driver for the Midway-based ATM card (ENI155P). The device-independent layer provides support for using IP over ATM through PVCs and also provides support for "native" mode ATM sockets to send and receive raw ATM cells or AAL5 frames. An ATM pseudo header structure is used to route the ATM packets through the BSD networking subsystem. This four-byte header consists of the virtual circuit number (VPI and VCI) and a set of flag bits. The first flag bit indicates whether AAL0 or AAL5 is being used. This pseudo header is needed because the normal ATM header is removed from each cell in hardware by the network interface layer. The pseudo header only exists in the protocol layer and is removed before it is passed up to the socket layer or down to the network interface layer.

The device-dependent layer of BSD ATM supports only ATM cards based on the Efficient Networks "Midway" ATM chipset. To transmit data, the protocol layer enqueues an mbuf chain on the network device's input queue and calls the device's start routine. The start routine immediately removes the outbound packet from the network

interface queue and inspects the packet's ATM pseudo header to determine on which transmit channel to enqueue the packet. Then the driver inserts a Transmit Buffer Descriptor (TBD) at the front of the packet and a trailer to the end of the data area so that it is the proper length. The TBD is read by the hardware to determine size and destination of the packet and then discarded. When a Midway card receives a complete AAL5 frame or an AAL0 cell into its on-board memory it puts the virtual circuit on a hardware-managed "service list" and generates a "receive" interrupt. The driver's interrupt handler responds by taking the virtual circuit off the hardware service list and placing it on a software managed service list. The software list is needed in case there is a shortage of memory resources. The driver allocates mbuf chains for each frame and then programs the Midway card to transfer the data from on-board memory to host memory. The mbuf chain receiving the data is placed on the receive queue and the driver removes the circuit from the software service list. The packet is pulled off the receive queue and passes it up to the protocol layer.

f. MD5 Message Digest Algorithm

The MD5 message-digest algorithm was developed by Ron Rivest at MIT [Ref. 4]. Until the last few years, when both brute-force and cryptanalytic concerns have arisen, MD5 was the most widely used secure hash algorithm. The block-chained algorithm takes as input a message of arbitrary length and produces as output a 128-bit message digest. The MD5 algorithm has the property that every bit of the hash code is a function of every bit of the input. The complex repetition of the basic functions in the algorithm produces results that are well mixed and it is unlikely that two messages, even if they exhibit similar regularities, will have the same hash code. Even so, from a cryptanalytic point of view, MD5 must be considered vulnerable to cryptanalysis or brute-force attack. Since MD5 is a 128-bit hash functions, it must either be replaced by a stronger algorithm which uses a longer hash function or, as we propose, be used for only a very short duration. The exact duration should be based on current computer capabilities.

g. Network Time Protocol (NTP)

Network Time Protocol (NTP) is a distributed computer clock synchronization protocol that has been in use for more than 20 years. The work done on

NTP has been through the cooperation of several people, but under the oversight of David L. Mills. While there are other synchronization protocols available, such as Digital Time Synchronization (DTSS), NTP is "the longest running, continuously operating application protocol" in the Internet today [Ref. 5]. Some of the appeal of NTP is also due to the many platforms to which it has been ported. Of specific import to Stargate, NTP has been ported to Windows NT and NetBSD. However, the build for Windows NT is less stable than that of the Unix versions.

NTP can be used in various modes. NTP is widely used in the classic client-server mode with a hierarchy built in to reduce network traffic and latency. NTP can also be used in symmetric mode by isolated networks, such as a peer to peer network. The symmetric mode is ideal for the Stargate prototype. Finally, NTP can operate in a broadcast mode if there are a large number of clients involved.

The standard time used by most nations of the world is Universal Coordinated Time (UTC), formerly known as Greenwich Mean Time (GMT). NTP uses UTC to synchronize "primary" servers via radio, satellite receiver or modem. These primary servers then adjust the clocks of secondary servers/clients. In order to correctly adjust clocks of secondary servers over a LAN or WAN, a time offset of the server clock relative to the client clock is computed by the client running NTP. In existence today, there are 79 public primary servers synchronized directly to UTC, located in every continent except Antarctica. There are over 100 public secondary servers synchronized to the primary servers and providing synchronization to more than 100,000 clients and servers in the Internet. Additionally, there are an unknown number of private servers utilizing NTP. The general model for discovering the clock offset starts with a server sending a message that includes its current clock value to the client, which could be another server or workstation. The client records its own current clock value upon arrival of the message. For accuracy, the client has to measure the server-client propagation delay. NTP measures the total roundtrip delay and assumes the propagation times are statistically equal in each direction. [Ref. 5]

Clock errors are due to variation in network delay and latencies in computer hardware and software (jitter), as well as clock oscillator instability (wander). According to NTP documentation, NTP in the majority of cases can keep clock synchronization within a few milliseconds on LANs and a few tens of milliseconds on WANs [Ref. 5]. This performance is acceptable for the Stargate project.

C. STARGATE CONCEPTUAL DESIGN

Stargate is a software-based authentication solution designed to provide transparent protection for autonomous private networks connected across a public network. Figure 4 illustrates the conceptual implementation of the Stargate solution. A single Stargate protects each private network. A data frame leaving the private network passes through a Stargate where it is signed using a unique key. The key's index and the computed digital signature are appended to the outgoing data frame and forwarded across the public network to the Stargate protecting the receiving private network. The key index contained in the data is used to retrieve the corresponding key and a new signature is created. The signature contained in the data is compared with the newly computed signature. If they match the original data are forwarded to the end host. If not, the data are discarded. This concept is scalable to protect any number of private networks.

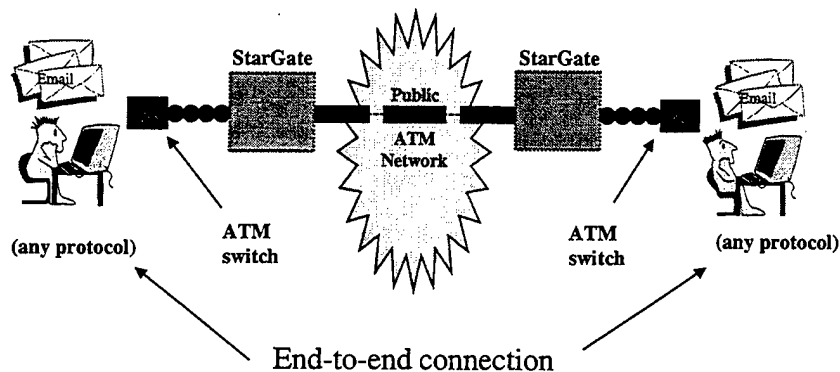


Figure 4. Conceptual Stargate Design.

1. Relationship to CellCase Technology

Stargate employs the same concept as the CellCase technology. Isolated private networks connected across a public network are each protected by a "black box". CellCase provides this protection with strong encryption performed in hardware. This

solution is both inflexible and expensive. We believe that a simple software solution will provide similar results at a fraction of the cost.

2. Relationship to LLPA Protocol

Stargate extends the LLPA protocol to ATM networks. LLPA authenticates IP packets. Porting LLPA to authenticate ATM cells is not trivial because of their small size (53 bytes). If each cell is authenticated, a large processing overhead is incurred, so grouping of the cells is required. If the groups are too large, then the authentication would introduce an unacceptable delay. Fortunately, ATM provides framing in the form of AAL5 which easily solves the problem. Using AAL5, the LLPA port is fairly straight forward since the NetBSD ATM protocol uses the same data structures and conventions as TCP/IP.

3. Relationship to WUGS Technology

The WUGS 20 became an important tool in our prototype implementation. It allowed us to simulate the data flow across a public network and also provided convenient access to the ATM cell stream. Since the WUGS 20 has no processing capability, it requires a controlling PC. This PC monitors and modifies the switch using Washington University's GBNSC and Jammer programs which are written for the NetBSD operating system. The controller provides a convenient platform on which to implement our Stargate solution since we have access, through NetBSD's open architecture, to the native ATM protocol and to the ATM cells which are passing through the switch. Additionally, NetBSD's open architecture allows us to modify the NetBSD kernel to include the signing and authentication module and the key management daemon.

D. STARGATE IMPLEMENTATION

1. Testbed Layout

Figure 5 depicts the physical layout of our test implementation. The computers, Pine and Cypress, simulate the source and destination end-hosts. They are connected to ports 1A and 1B of one of the WUGS 20's two dual OC-3 line cards. Stargate is

connected to port 0A of the other dual OC-3 line card which it shares with the WUGS controller. Cypress also acted as the Key Distribution Center (KDC) for the key management process. A separate TCP/IP path was created to emulate the network path between Stargate and the KDC.

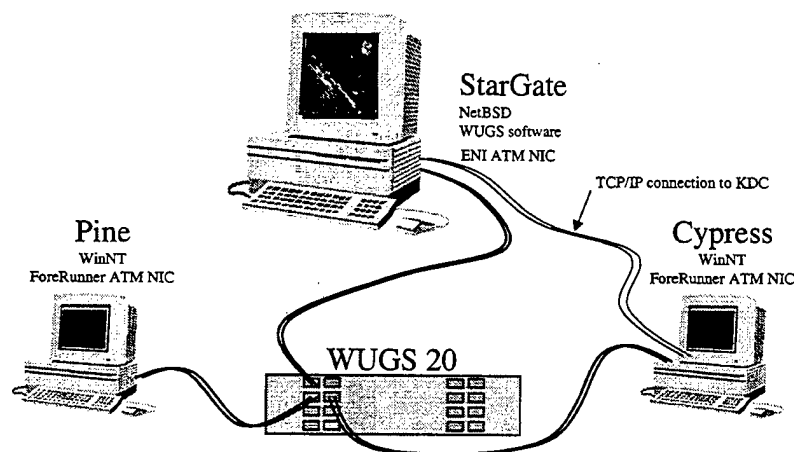


Figure 5. Physical Layout.

2. "Public Network" Simulation Layout

To simulate the conceptual design of two Stargates, sender and receiver, we used the WUGS 20 to recirculate the ATM cells. Figure 6 demonstrates the logical flow of the cells through the switch. First, ATM AAL5 frames were generated and sent from one end-host to the WUGS 20. The WUGS 20 routes the cells to Stargate. Stargate signs the AAL5 frame, changes the VCI by adding 100, and forwards the cells back to the WUGS 20. To simulate travel through a public ATM network infrastructure, the cells received on VCI 133/134 are recycled through the switch and back to Stargate. Now, acting as the receiving Stargate the frame is authenticated, the VCI is changed by subtracting 100, and then the frame is forwarded back to the switch (WUGS 20). The switch routes the cells to the receiving end-host.

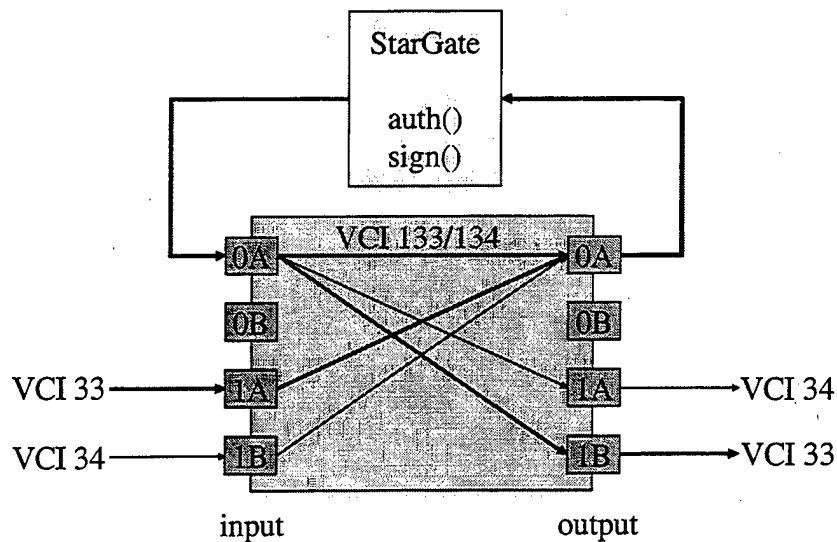


Figure 6. Logical Layout.

3. Cell Authentication

a. Signing

Based on the principles of LLPA. ATM cell signing was implemented by intercepting the ATM cells in the WUGS 20 controller. Figure 7 illustrates the procedure. After a complete AAL5 frame is received by the NIC, the headers are stripped in hardware and the data payload is passed up to the ATM layer process associated with that particular VPI/VCI combination. Data bound for VPI 0 VCI 32, the reserved controller circuit, are allowed to continue up the protocol stack. All other data begins the signing process by appending a 26 byte trailer to the AAL5 frame. The trailer consists of two bytes for version/option bits, four bytes for a sequence number, and four bytes for a key index provided by the key management daemon. The corresponding 16 byte key is appended next and the entire frame is hashed with the MD5 algorithm. The resulting 16 byte message digest is copied over the key and the AAL5 frame is put in the outgoing queue. The NIC processes the frame into individual ATM cells and transmits it through the public network.

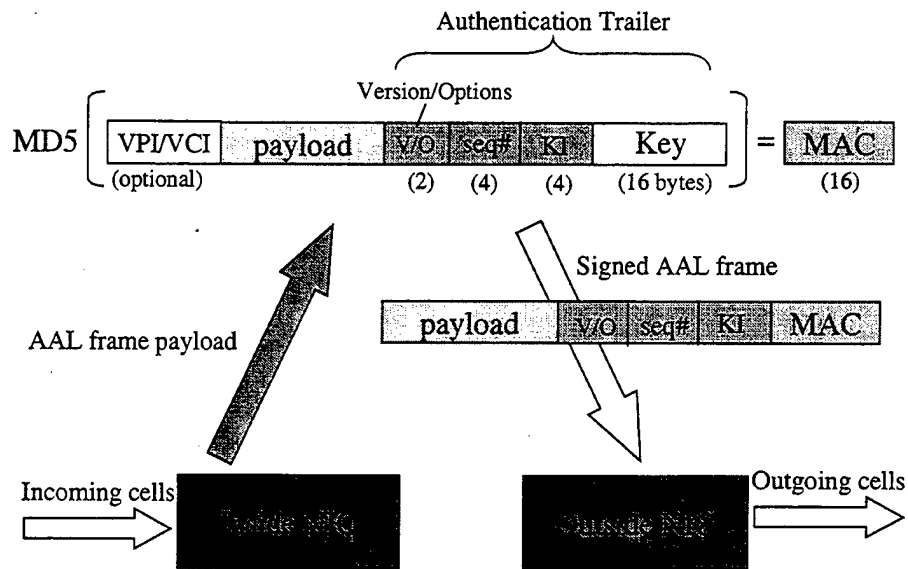


Figure 7. Signing Operation.

b. Authentication

The authentication process as illustrated in Figure 8 is similar to the signing process. Once a complete AAL5 frame is received, the hardware again strips the cell headers and forwards the payload up to the ATM layer. If the payload is not destined for the switch controller, the Key Index (KI) and message digest are extracted from the payload. The key management daemon provides a key based on the received KI. The message digest in the authentication trailer is replaced with the key and a new message digest is created by hashing the frame with the MD5 algorithm. The new message digest is compared with the received message digest. If they match, then the trailer is stripped and the AAL5 frame is placed in the outgoing queue for transmission to the end host. If the digests do not match, the frame is dumped.

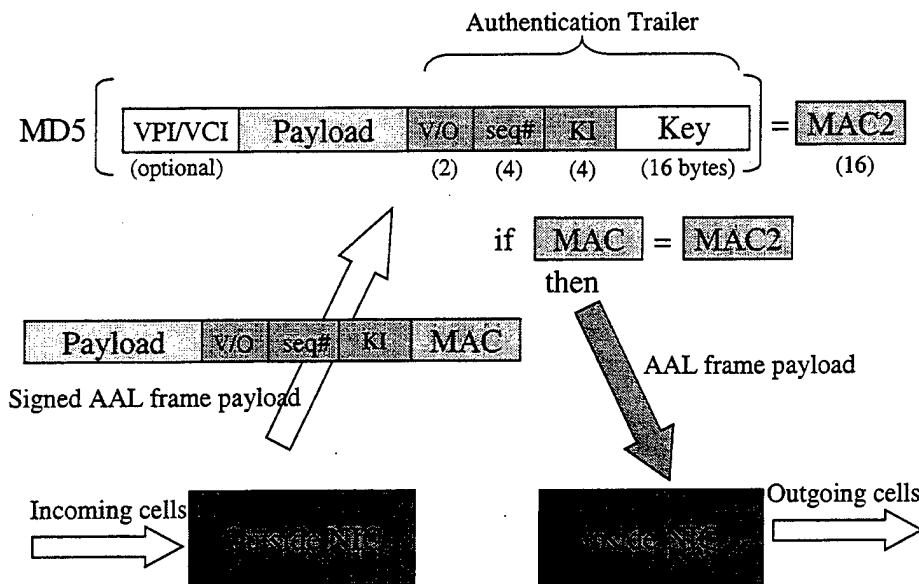


Figure 8. Authentication Operation.

4. Key Management

One important factor to the success of the Stargate device is its key management. This section will discuss the methodologies for creating authentication tables and keys for optimal security. There are a few assumptions made in order to clarify the key management process. First, when a Stargate device is started, there is a logon process where each Stargate is authenticated to a managing Key Distribution Center, which acts as the Central Authority for authentication keys. Since this thesis did not deal with signaling, this logon process had to be assumed and therefore no security threats of impersonating the KDC were tested. Furthermore, normally the transmission of key tables and masks between a KDC and a Stargate device would require some form of heavy encryption, such as Triple DES [Ref. 6]. This encryption was not added to the transmissions for simplicity of testing and evaluation. Finally, timing between the KDC and Stargate device is critical. While timing issues are addressed when discussing synchronization of key tables and masks, it was assumed that the computer clocks between the KDC and Stargate are synchronized automatically with the use of NTP.

The best design for testing Stargate's key management process was a design that is very similar to the "Gateway" approach mentioned in the LLPA protocol [Ref. 3]. For

this thesis, the design of the key management process involved one Key Distribution Center (KDC) and one Stargate device. Figure 9 shows the conceptual design for broadening the key management process to two or more Stargate devices, which are handled by a single KDC. This lends Stargate devices to be “grouped” as peer devices and serviced by a single KDC. The rest of this chapter will discuss the purpose and key issues surrounding the KDC and Stargate independently.

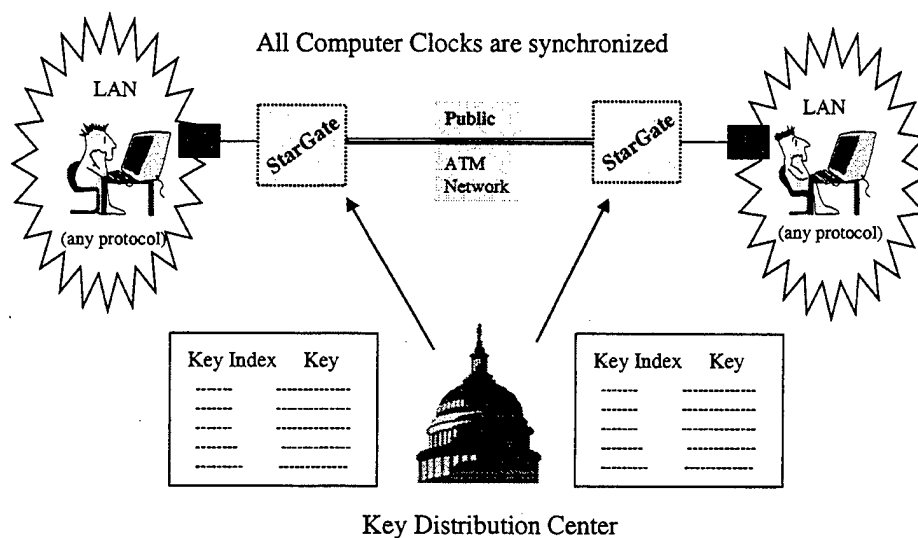


Figure 9. Key Distribution Center Conceptual Design.

a. Key Distribution Center responsibilities

The KDC is the keeper of the authentication keys. It has three main responsibilities: creation of authentication key tables, creation of masks, and establishment of timing. The KDC was written in Java. The selection of Java was due in large part for the benefit of platform-independence. The code for the KDC is found in Appendix D.

In parallel with the LLPA protocol, the Stargate’s authentication table is created with a 4-byte key index (KI) and 16-byte authentication key. By using a smaller key index, searching for the correct authentication key is much faster. The number of keys needed in the authentication key table is based on two important design decisions.

Those design decisions are (1) the lifetime for each key and (2) the time specified for replacing the key table. Due to the relative strength of the 128-bit key, it is obvious that the lifetime for each key would have to be smaller than the time required to break the key with a brute-force attack. The selected lifetime for each key was set at 2 minutes for testing purposes. After two minutes, the current key would be permanently discarded from the available keys in the key table.

The time for replacing the authentication key table is based on a distinction between the types of key authentication tables used in the key management process. First "base tables" are referred to as the complete tables sent from the KDC. When referring to generic "key table(s)" in this section, we are referring to the current authentication table that is being used by the Stargate device during the signing and authentication processes. For security and efficiency reasons, base tables are created less frequently than the current key table. At startup, the Stargate device from the KDC receives a base table and an initial mask. The current key table is then created by XORing the base table and mask, which is 20-bytes in length. This process is required so the base table can never be discovered by unauthorized users. Inevitably, these base tables will be replaced by the KDC and sent to the Stargate device at specified intervals, but current key tables can be more readily created with the use of randomly generated masks sent by the KDC. Replacing base tables is an expensive process because the tables are large. Conceptually, the interval for replacing base tables could be once a month.

The design decision was made to create an authentication table that would contain enough keys for a 24-hour period of service. Before the key table expires, the KDC will send a new mask and then the Stargate device can recreate a new, current key table. By replacing the key indexes and keys in the authentication table once a day, it would require fewer transmissions between the KDC and Stargate, but the time period for replacing the key indexes and authentication keys is variable based on the implementation of the KDC and Stargate(s). For our testing, changing key indexes and authentication keys every day, plus a 2-min key lifetime would require an authentication table of 720 KI/keys. A few keys are added as padding.

The randomness of key indexes, keys and masks is important. If the same key index or key appeared in the same table, security risks and unpredictable behavior by the Stargate device is expected. Java's random number generator creates the key table and masks. It is possible that over time, a pattern will emerge in the numbers a computer generates. To get truly random results, there are devices under development at various

universities and corporations, such as Intel, that harness thermal noise to produce random numbers [Ref. 7].

The final responsibility of the KDC involves the synchronization of key tables. The KDC has to tell the Stargate devices when to start using a new key table to ensure all Stargates are synchronized with each other as well as with the KDC. Due to clock drift and network latency, if a Stargate device started to immediately use a key table, then within a short time two Stargate devices would no longer be synchronized; therefore, keys would not match and traffic would be discarded. The starting time for all Stargates is chosen to be far enough into the future to overcome synchronization issues that are not associated with clock synchronization, i.e. network latencies. It is assumed that clock drift is within tolerances. For testing purposes, the start time was chosen to be 20 minutes into the future.

In addition to setting the start time for all Stargate devices, the KDC must also set the time for itself to issue the next mask. The size of the table and the start time interval are factors in calculating when the next transmission between the KDC and Stargate(s) will take place. The next transmission obviously has to happen prior to the expiration of the key table, but it also has to occur in enough time to set the "start time" before the last key is used. To accomplish this synchronization, a simple formula will suffice. The formula for determining consecutive transmissions, which is set to the constant variable, *sendWaitTime*, is:

$$\textit{sendWaitTime} = \textit{tableDuration} - \textit{startWaitTime},$$

where *tableDuration* is calculated as:

$$\textit{tableDuration} = \textit{keyLifeTime} * (\textit{numOfKeys} - 1),$$

where *numOfKeys* refers to the actual number of keys in the authentication table and *keyLifeTime* refers to the length of time for which each key will be used. The *sendWaitTime* is set to a constant time interval, meaning its value will not change once initialized; however, its initialization will vary based upon the input values for the above formula. To control consecutive transmissions, the main thread is put to sleep for the time interval specified by the *sendWaitTime* parameter.

The Stargate device has to be told by the KDC when to begin using a new table. This is required to overcome the delays associated with sending the table through the network. The starting time for Stargate to start using its new table is set to the variable, *startTime*, and is calculated as:

$$startTime = startTime + tableDuration,$$

but *startTime* has to be initially set to the current time of when the table is sent; otherwise, the Stargate may try to sign or authenticate traffic without the key table being ready for use. Figure 10 shows the relationship of the before mentioned timing variables on a timeline. This diagram will help visualize the interaction of the KDC and a single Stargate device. The scenario in Figure 10 is the same as the Stargate testbed scenario. Figure 11 shows the scenario of a Stargate device coming online with other existing Stargate devices that are all controlled by a single KDC. In this more complex scenario, the first *startTime*, base table and mask are identical to what the existing Stargate devices are using; therefore, the *startTime* appears on the timeline before the new Stargate is operational. The KDC is responsible for setting all of the timing variables, thus maintaining control over the key management process.

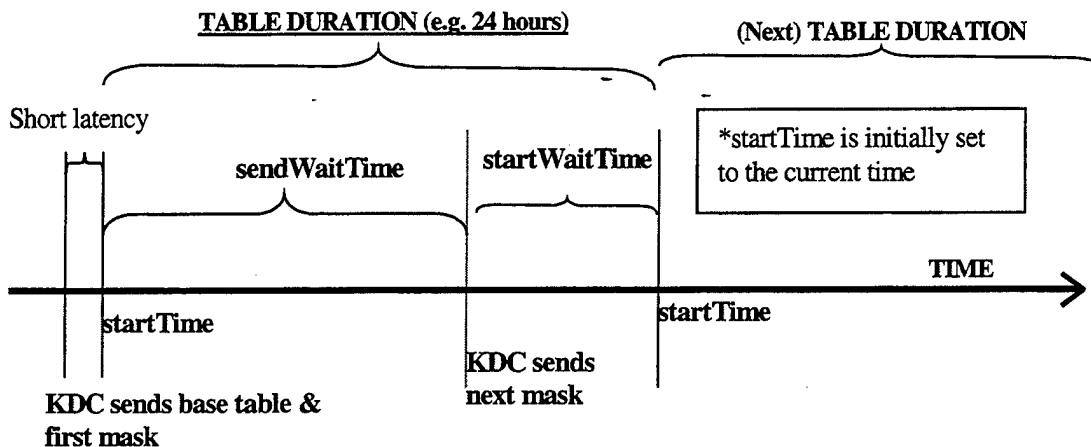


Figure 10. Key Management Timing Relationships.

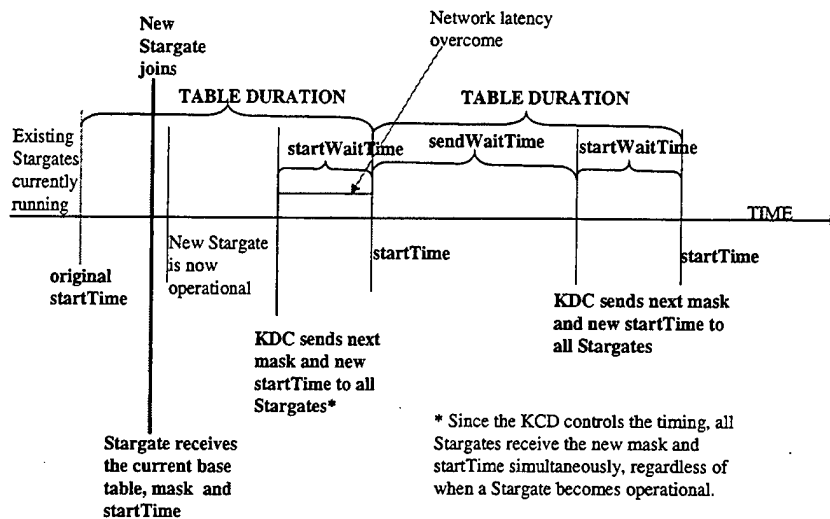


Figure 11. Multiple Stargate Timing Relationship.

b. Stargate key management responsibilities

Once the Stargate device receives its initial authentication table and subsequent masks from the KDC, the focus of key management changes to the Stargate device. Key management responsibilities for the Stargate device centers around interfacing with the authentication program. Specifically, getting the current key and verifying a key index are the two main responsibilities of the Stargate device. In order to perform either of these functions, the Stargate device uses a key window to speed up performance. As mentioned earlier, searching for a possible key is quicker by searching a key window that only contains three keys vice a table that contains hundreds of keys.

The key window contains the last key used, current key and future key. Figure 12 shows the key window concept. The reason for using three keys is discussed in the LLPA protocol [Ref 3]; however, the use of three keys has an enormous security benefit besides improving performance. For example, a key could not be used beyond its lifetime because it would no longer be in the search window. This is how Stargate reduces the effectiveness of flooding and/or playback attacks. The updating of the key window has to be a function of time. There are two possible methods for updating the key window.

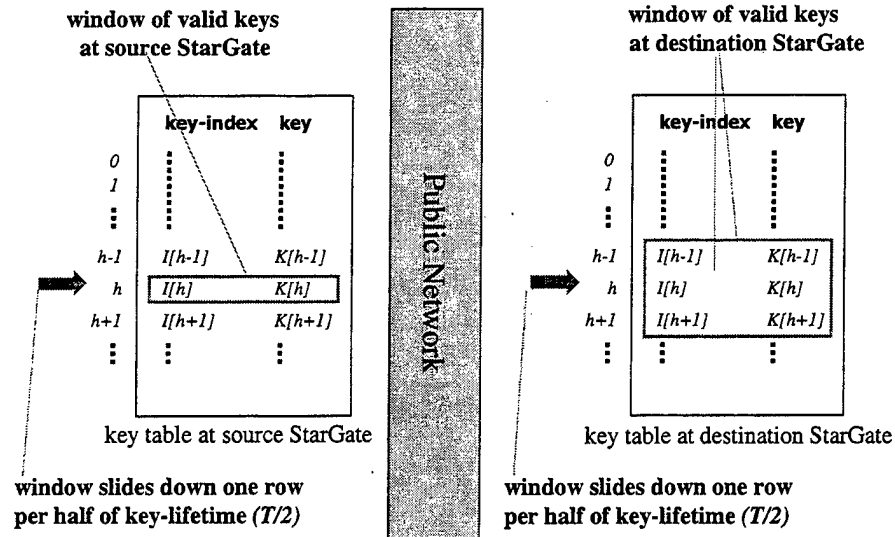


Figure 12. Key Window Concept.

The first method is to create a program that would rely on software interrupts (i.e. *Thread.sleep()*) to control when the key window would “move,” updating the current batch of keys from the overall key table. This method, although simple and requiring no any mathematical computation, was eliminated from consideration for several reasons. First, this method would have unpredictable behavior. With the possibility of another program running that might have a higher precedence, the time for the *move_window()* to actually run could be delayed. Secondly, once a delay is encountered, the delay will continue to manifest itself for each subsequent execution.

The second method, which was adopted, was to create a function that can be called by the *get_key()* and *veri_key()* that would update the key window using a mathematical computation. This computation is as follows:

$$(T - T_0) / p$$

where T refers to the current time, T_0 refers to the starting time of when the key table was put into use and p refers to the lifetime for the keys (e.g., 2 minutes). This computation would be performed to find the index of the key table and then the current key would be

set to this index. For authentication purposes, the 3 possible keys in the key window would include the key before and the key after the calculated index. For instance, if the key table has been in use for 6 minutes and the lifetime for each key is 2 minutes, then the current key would be retrieved at the third row of the key table. The drawback to this approach is the mathematical computation involved each time the authentication keys are needed. The benefit of this method, which outweighs the drawback, is that if a delay was encountered, it would not affect subsequent calculations.

The updating of the key window is accomplished by calling the *move_window()*. Both the *get_key()* and *veri_key()* call *move_window()* before doing anything else. The *get_key()* will return the current key for creating the message digest. The *veri_key()* is called by passing a key index as the function's argument. If the key index is found in the key window, the key is returned for the authentication process.

The Stargate device does one other important function. When the KDC passes it a new mask, the Stargate device will bitwise XOR the mask with each entry of the base table to create a new authentication key table. This new table is created as soon as the mask is sent, even though it will not be put to use until the start time is reached.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. PERFORMANCE EVALUATION

A. TEST DESIGN AND DATA COLLECTION

We tested our prototype using AAL5 frames generated by the software provided with the Fore Systems ForeRunner LE ATM cards (*perf.c* and *sockutils.c*). As discussed in Chapter III, Section D.1 and illustrated in Figure 5, Pine simulated the sending host and Cypress acted as the receiver. Pine created a PVC on VPI 0 VCI 33 to Cypress. The WUGS routing table was programmed according to the Logical Layout in Section D.2 and the illustration in Figure 6. The routing table on Port 1 routed VCI 33 to the switch controller (0A). The routing table on port 0 routed VCI 133 back to the switch controller (0A) and VCI 33 to the high side output of port 1 (1B).

We tested the throughput of two functions, *sign()* and *auth()*. Both functions are compiled into the NetBSD kernel as part of the *atm_auth.c* file. The kernel function *microtime()* was inserted in the ATM protocol function, *atm_input()* to measure beginning and end times for both *sign()* and *auth()*. *atm_input()* is contained in the *if_atmsubr.c* file. Figure 13 illustrates the points at which timing measurements were taken.

To factor out the CPU execution time of the *microtime()* function, two consecutive calls to the function were made prior to actual testing. The function's execution time was determined by subtracting the second time from the first. The execution time, 3 μ sec was then subtracted from all test measurements for each function call. Additionally, no computations were made during the execution of the test. All times were stored in a two-dimensional array during execution and upon completion of the required number of test iterations, an average time was computed.

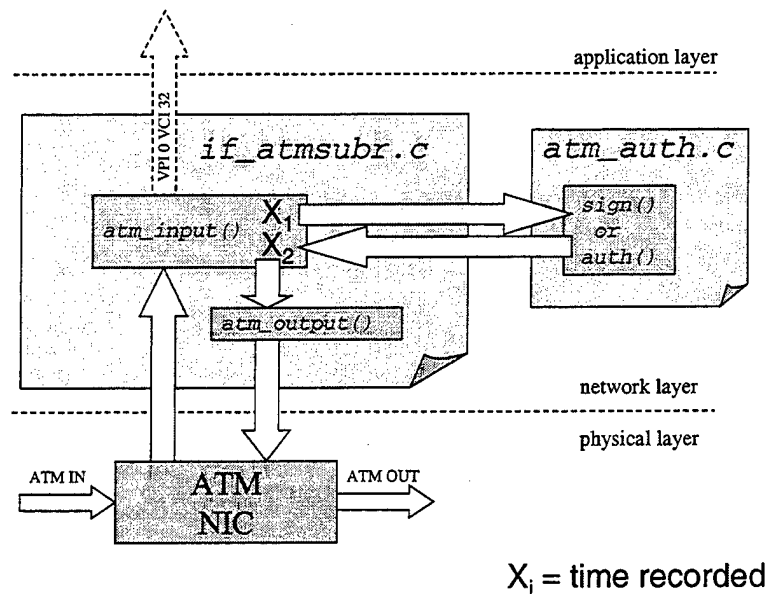


Figure 13. Timing Implementation of Throughput Test.

B. RESULTS

To compute the throughput of the function, the following formula was used.

$$(\text{Bytes/frame} * 8) / \text{average execution time}$$

Pine generated AAL5 frames from 32 bytes to 5120 bytes in length. The average execution time of the function and the computed throughput is listed in Table 1 for each frame size tested. A graphical presentation of the results is shown in Figure 14. We found, as expected, that the throughput was low for smaller frame sizes, but quickly grew as frame size increased. Throughput rose sharply up to approximately 1KB and then leveled off between 60 Mbps and 70 Mbps. This is comparable to the results observed with the LLPA implementation although slightly slower. We attribute this difference to a slightly different approach used to manipulate the mbufs and the authentication trailer. To eliminate the complicated logic required to process the four possible types of mbufs, Stargate copies the entire mbuf into a char array and then appends the authentication trailer. This made indexing, copying, and extracting authentication information much

simplier. Once signing is complete, the procbuf, with authentication trailer, is copied back to the mbuf using *m_copyback()* function which will extend the mbuf chain if required. This is a performance trade-off we made to simplify the coding and its impact is noticeable. The *sign()* function is slower than the *auth()* function because there is only one copy operation to the procbuf for authentication. It was not necessary to *m_copyback()* the authenticated mbuf since we only need to remove the trailer. We simply cropped the authentication trailer from the mbuf with *m_adj()*.

Number of bytes of data per frame	Avg. execution time for <i>sign()</i> function (μ sec)	Through put of <i>sign()</i> function (Mbps)	Avg. execution time for <i>auth()</i> function (μ sec)	Through put of <i>auth()</i> (Mbps)
32	39	6.56	34	7.53
64	40	12.80	35	14.63
128	47	21.79	41	24.98
256	59	34.71	53	38.64
512	85	48.19	79	51.85
1,024	154	53.19	135	60.68
1,536	217	56.63	191	64.34
2,048	278	58.94	245	66.87
2,560	342	59.88	305	67.15
3,072	399	61.59	360	68.27
3,584	460	62.33	413	69.42
4,096	526	62.30	474	69.13
4,608	585	63.02	545	67.64
5,120	647	63.31	609	67.26

Table 1. Performance Data for *sign()* and *auth()* Functions.

An interesting observation made during testing was that we were unable to drive the packet generation software fast enough to overwhelm Stargate's authentication software. However, at its best speed, Pine was producing cells fast enough to overwhelm Cypress. We observed that with less than half the raw CPU power of Pine, but comparable to Stargate, Cypress was dropping 75-85% of the transmitted frames. Yet, Stargate processed every packet correctly. While Stargate was only handling one connection, we feel that if implemented on a high end machine, the software will handle multiple connections with similar results. Another important limitation of our prototype is the ENI155P ATM card. It has only 512KB onboard RAM and has only 7 receive buffers. Efficient Networks makes a 2MB version of the ENI155P which handles many more connections simultaneously. By improving the component parts of the computer,

the performance of Stargate can be improved as well. This provides easy extensibility as computers evolve.

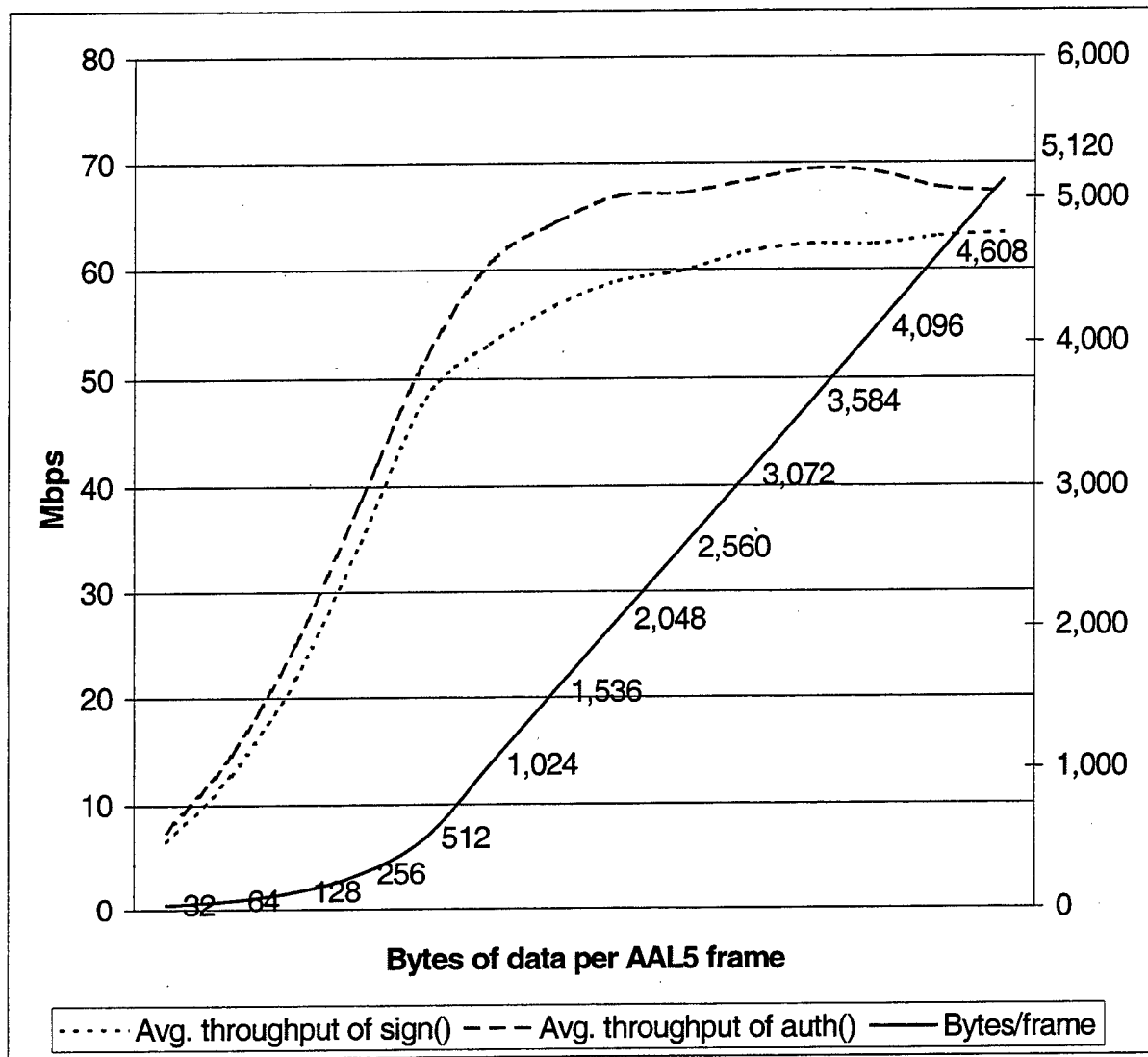


Figure 14. ATM Authentication Module Performance.

V. LESSONS LEARNED AND RECOMMENDATIONS

A. AUTHENTICATION LESSONS LEARNED

1. Setting up the Testbed

We started building the testbed with the idea that we could use one high-speed computer with two ATM network interface cards as our "Stargate". This did not work because we did not have access to the source code for the Fore Systems ForeRunner LE ATM cards. An alternative we considered was implementing the authentication module at the application layer, but we felt this was not optimal and would affect our test results. We then fell upon the WUGS 20 as an intriguing alternative since it is based on NetBSD and therefore open source software. Email exchanges with Fore Systems technicians confirmed that we would not be able to get the source code for the ForeRunner LE cards. We therefore turned our attention to the WUGS and NetBSD's ATM protocol implementation. Although Washington University used NetBSD to implement the original WUGS 20 controller, they had a Linux version available which would have been preferred because of our familiarity with Linux. After several weeks of frustration, we were unable to compile a working Linux based controller on the high end system, which eventually became our frame generator (Pine). We abandoned Linux and started over with the NetBSD version and attempted to compile the software on our high end, dual processor computer. Unfortunately, NetBSD is not as well supported as Linux and is therefore not as advanced in its support for new hardware. We were unable to use NetBSD with the new computer. Yet again, we started over using the older, slower computer from the LLPA implementation. Here we were successful. We set up NetBSD as the WUGS controller and dissected the kernel's ATM network protocol source code. Fortunately, the ATM network protocol is very similar to the BSD TCP/IP network protocol implementation. Stevens' books on TCP/IP were invaluable in deciphering and documenting the TCP/IP protocol [Ref. 8].

A thorough analysis of the ATM protocol identified several places that the ATM cells could be "intercepted" enroute up the protocol stack. One possibility was in the kernel's ENI155P driver code, but this was rejected since it would make the code

hardware specific. We settled on the function call, *atm_input()* from the ATM layer which allows us to create a hardware independent implementation based on the BSD ATM network protocol. Once this was decided, the LLPA code which is a BSD based TCP/IP kernel modification, was ported fairly easily to an ATM network kernel modification. A good knowledge of the BSD ATM network protocol helped immensely in this task.

2. Authentication Performance Issues

How could this software be made to execute faster? Since the creation of a message digest is integral to our solution, it is crucial that we use the fastest algorithm available. Hardware issues aside, we believe that MD5, in its standard form, is not the fastest solution for creating a message digest [Ref. 9]. MD5 is conveniently implemented in the NetBSD kernel, but other options exist such as HMAC, SHA-1, and RIPEMD-160 [Ref 6]. Although, we did not specifically measure the performance of MD5 or substitute other message digest algorithms, we think further investigation is warranted to determine the most advantageous method of creating the "digital signature".

Another possibility to increase performance would be to create a large RAM disk and load the Stargate system into memory to eliminate slower disk accesses. This approach is commonly used by high speed internet servers to provide increased system performance.

B. KEY MANAGEMENT LESSONS LEARNED

The lessons learned involving the key management process were primarily integration issues. The original plan was to use a high-level programming language, Java, to write the KDC and Stargate key management functions. Java development is generally quicker than with other languages because of its true object oriented approach enhanced in this case by the experience of the programmer. This quicker development coupled with Java being platform-independent appeared to make Java the perfect choice initially; however, integration with the kernel-level authentication program and communication between the KDC and Stargate device became significant hurdles to overcome.

1. Language Integration

While the Stargate key management object and KDC object worked together perfectly when written in Java, there was an integration issue when trying to get the Stargate object to work with the kernel-level authentication program, which must be written in C. Since the Java Classpath is a user environment setting, the Stargate program could not be initiated by the kernel during the bootup process. How was the kernel going to be able to retrieve key objects from the user environment? Researching the problem, we discovered that Java has addressed the problem of integrating native code with Java. The Java solution is called Java Native Interface (JNI). JNI acts as an interface between the C code and the Java code. The key to this process is building a shared object library out of the JNI code. In the Stargate testbed, building shared object libraries in NetBSD became very time consuming; therefore, in the interest of time this approach had to be abandoned. The only recourse was to rewrite the Stargate object in C and then compile the code into the kernel, so the authentication program could easily access the `get_key()` and `veri_key()` methods. While this solved the problem of integrating the key management code with the authentication code, it was anticipated that going back and integrating the communication between the new kernel-level Stargate key management object and the KDC object could be difficult.

2. Communication Integration

After rewriting the Stargate object using C, communication between the Stargate object and KDC did not work. The KDC was written using a Java ServerSocket and the Stargate object was written as a client in C. As initially written, the Stargate client would send a request to the KDC for either a table or a mask and then the KDC would respond accordingly. The KDC and Stargate could make a socket connection, but the KDC could not read what was being sent on the input stream. The programs for the KDC and Stargate were tested against Java clients and C servers respectively. Both programs worked as designed, but they would not work together. Through much testing, it was decided to change the Java program to be a Socket client instead of a ServerSocket. The Stargate object was rewritten to act as the server and run as a startup daemon, constantly listening for connections from the KDC. This scenario was successful in passing Strings, but there are formatting problems with putting the Strings into a useable format on Stargate, which have yet to be overcome.

C. SUGGESTED FURTHER RESEARCH

This exploratory study has only begun to uncover the growing body of knowledge on the authentication of ATM traffic. The following list is specific areas to the Stargate concept that were not addressed by this thesis, but would be valuable follow-on research topics:

- Expanding and evaluating multiple Stargates in a larger testbed environment.
- Implement clock synchronization.
- Evaluate other message digest algorithms.
- Evaluating Stargate's performance with streaming traffic, such as video or voice.
- Develop the secure signaling process between the KDC and Stargate or Stargate to Stargate.
- Research to improve the randomness for the generation of security keys/masks.

APPENDIX A. JAMMER SCRIPT

```
# Set port 1A->0A and 1B->0A
# and port 0A->1A or 0A->1B or 0A->0A based on VPI/VCI
# using VPI 0 and VCI 33/34 or VCI 133/134
# NOTE: OC-3 duplex card does not allow TX/RV on same VCI
# OBSCURE POINT: To access upper half of OC-3 duplex (B port),
# the line card uses bit VPI[7] (ie VPI >128 = B port).
```

```
# (1A->0A)          C C          V  V  B V V B A A
#                   V      Y Y    U U    V R    P  C  D P C D D D
#                   P X B R  C C C D D S P C B I  I  I I I I R R
#                   P I I C D 1 2 S 1 2 C T O R 1  1  1 2 2 2 1 2
#-----
write vcxt 1 33 1 2 1 0 0 0 0 0 0 0 0 0 0 0 0 33 0 0 0 0 0 0
write vpxt 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
```

```
# (1B->0A)          C C          V  V  B V V B A A
#                   V      Y Y    U U    V R    P  C  D P C D D D
#                   P X B R  C C C D D S P C B I  I  I I I I R R
#                   P I I C D 1 2 S 1 2 C T O R 1  1  1 2 2 2 1 2
#-----
write vcxt 1 34 1 2 1 0 0 0 0 0 0 0 0 0 0 0 0 34 0 0 0 0 0 0
write vpxt 1 128 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
```

```
# (0A->1A)          C C          V  V  B V V B A A
#                   V      Y Y    U U    V R    P  C  D P C D D D
#                   P X B R  C C C D D S P C B I  I  I I I I R R
#                   P I I C D 1 2 S 1 2 C T O R 1  1  1 2 2 2 1 2
#-----
write vcxt 0 34 1 2 1 0 0 0 0 0 0 0 0 0 0 0 0 34 0 0 0 0 0 1 0
```

```
# (0A->1B)          C C          V  V  B V V B A A
#                   V      Y Y    U U    V R    P  C  D P C D D D
#                   P X B R  C C C D D S P C B I  I  I I I I R R
#                   P I I C D 1 2 S 1 2 C T O R 1  1  1 2 2 2 1 2
#-----
write vcxt 0 33 1 2 1 0 0 0 0 0 0 0 0 0 0 128 33 0 0 0 0 0 1 0
```

```
# (0A->0A)          C C          V  V  B V V B A A
#                   V      Y Y    U U    V R    P  C  D P C D D D
#                   P X B R  C C C D D S P C B I  I  I I I I R R
#                   P I I C D 1 2 S 1 2 C T O R 1  1  1 2 2 2 1 2
#-----
write vcxt 0 133 1 2 1 0 0 0 0 0 0 0 0 0 0 0 0 133 0 0 0 0 0 0
write vcxt 0 134 1 2 1 0 0 0 0 0 0 0 0 0 0 0 0 134 0 0 0 0 0 0
write vpxt 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
```

#	(SET MR)	F	T	R	R	V											
#		I	S	C	C	P			S		H	S					
#		E	T	O	B	B	C		S	R	S		R	C			
#		P	L	G	F	D	H	N	R	L	L	L		E	L		
#		P	D	I	F	T	D	T	E	E	E	E		T	T		
#		-----															
	write mr	1	2	0	128	32	0	255	1	1	1	1	100000	0			
	write mr	0	2	0	128	32	0	255	1	1	1	1	100000	0			

APPENDIX B. BSD KERNEL MODIFICATION CODE

A. *README*

```
=====
                        Stargate
=====
*****
IMPORTANT! Stargate authentication runs in
the NetBSD kernel. Changes to the source code
could possibly corrupt the kernel/filesystem.
*****
```

The source code for Stargate is compressed in the file Stargate-0.1.tar.gz.

```
README
stargated.h
stargated.c
atm_auth.h
atm_auth.c
if.h
if_atmsubr.c
KDC.java
set_switch.stargate
```

Uncompress the files with a command similar to the one below.

```
gzip -dc the-file.tar.gz | tar -xvf
```

```
=====
                        Configuration Notes
=====
```

Stargate was developed on a Dell OptiPlex GXMT 5200 with a 200 Mhz Pentium Pro, ENI155P ATM card, 3COM 3C509 Ethernet card, and 128 MB RAM running NetBSD 1.3

To compile the stargate source into the NetBSD kernel:

- 1) Start with standard kernel compiled with ATM support and WUGS package installed.
- 2) cp (overwrite) if.h and if_atmsubr.c to /usr/src/sys/net.
- 3) cp stargated.h, stargated.c, atm_auth.h, and atm_auth.c to /usr/src/sys/netatm.
- 4) Modify Makefile in /usr/src/sys/arch/i386/compile/"name-of-your-kernel" to include dependencies for files in step 2.
- 5) Run make from same directory.

To setup and run stargate:

- 1) Boot Stargate machine
- 2) Run GBNSC <GBNSC config.port1> (See big red book for more info).
- 3) Run Jammer <Jammer 0.1 stargate 5551> (See big red book for more info).
- 4) Set switch tables using Jammer <include set_switch.stargate>.
- 5) Setup a PVC (VPI 0 VCI 33) between the end hosts.
- 6) Connect end hosts to ports 1A and 1B of the WUGS.
- 7) Stargate daemon is running waiting for key table from KDC.

KDC:

- 1) Start KDC.java on KDC machine <java KDC stargate> (must have TCP/IP connection to stargate).
- 2) KDC sends key table and periodic masks.
- 3) Stargate ready to authenticate traffic. (Only on VCI 33 and 34)

=====

General Notes

=====

Switch routing tables must be changed to handle other than VCI 32, 33, and 34. Logic in if_atmsubr.c must be changed as well.

Authentication will handle all four varieties of mbuf since we use m_copyback and m_adj. See Stevens TCP/IP Vol 2 for more info on m_buf functions.

B. **IF_ATM.H**

```
/*      $NetBSD: if.h,v 1.29 1997/10/02 19:41:57 is Exp $      */

/*
 * Copyright (c) 1982, 1986, 1989, 1993
 *   The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *   notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *   must display the following acknowledgement:
 *     This product includes software developed by the University of
 *     California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its contributors
 *   may be used to endorse or promote products derived from this software
 *   without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)if.h      8.1 (Berkeley) 6/10/93
 */

#ifndef _NET_IF_H_
#define _NET_IF_H_

#include <sys/queue.h>

/*
```

```

* Structures defining a network interface, providing a packet
* transport mechanism (ala level 0 of the PUP protocols).
*
* Each interface accepts output datagrams of a specified maximum
* length, and provides higher level routines with input datagrams
* received from its medium.
*
* Output occurs when the routine if_output is called, with four parameters:
*   (*ifp->if_output)(ifp, m, dst, rt)
* Here m is the mbuf chain to be sent and dst is the destination address.
* The output routine encapsulates the supplied datagram if necessary,
* and then transmits it on its medium.
*
* On input, each interface unwraps the data received by it, and either
* places it on the input queue of a internetwork datagram routine
* and posts the associated software interrupt, or passes the datagram to a raw
* packet input routine.
*
* Routines exist for locating interfaces by their addresses
* or for locating a interface on a certain network, as well as more general
* routing and gateway routines maintaining information used to locate
* interfaces. These routines live in the files if.c and route.c
*/
/* XXX fast fix for SNMP, going away soon */
#include <sys/time.h>

struct mbuf;
struct proc;
struct rtenry;
struct socket;
struct ether_header;

/*
 * Structure defining statistics and other data kept regarding a network
 * interface.
 */
struct if_data {
    /* generic interface information */
    u_char ifi_type;           /* ethernet, tokenring, etc. */
    u_char ifi_addrlen;       /* media address length */
    u_char ifi_hdrlen;        /* media header length */
    u_long ifi_mtu;           /* maximum transmission unit */
    u_long ifi_metric;         /* routing metric (external only) */
    u_long ifi_baudrate;       /* linespeed */
    /* volatile statistics */
    u_long ifi_ipackets;       /* packets received on interface */
    u_long ifi_ierrors;        /* input errors on interface */
    u_long ifi_opackets;       /* packets sent on interface */
    u_long ifi_oerrors;        /* output errors on interface */
    u_long ifi_collisions;     /* collisions on csma interfaces */
    u_long ifi_ibytes;         /* total number of octets received */
    u_long ifi_obytes;         /* total number of octets sent */
    u_long ifi_imcasts;        /* packets received via multicast */
    u_long ifi_omcasts;        /* packets sent via multicast */
    u_long ifi_iqdrops;        /* dropped on input, this interface */
    u_long ifi_noproto;        /* destined for unsupported protocol */
    struct timeval ifi_lastchange; /* last updated */
};

/*
 * Structure defining a queue for a network interface.
 *
 * (Would like to call this struct 'if', but C isn't PL/1.)

```

```

*/
TAILQ_HEAD(ifnet_head, ifnet);          /* the actual queue head */

/*
 * Length of interface external name, including terminating '\0'.
 * Note: this is the same size as a generic device's external name.
 */
#define IFNAMSIZ 16

struct ifnet {
    /* and the entries */
    void *if_softc;          /* lower-level data for this if */
    TAILQ_ENTRY(ifnet) if_list; /* all struct ifnets are chained */
    TAILQ_HEAD(, ifaddr) if_addrlist; /* linked list of addresses per if */
    char if_xname[IFNAMSIZ]; /* external name (name + unit) */
    int if_pcount;          /* number of promiscuous listeners */
    caddr_t if_bpf;          /* packet filter structure */
    u_short if_index;        /* numeric abbreviation for this if */
    short if_timer;          /* time 'til if_watchdog called */
    short if_flags;          /* up/down, broadcast, etc. */
    short if_pad1;           /* be nice to m68k ports */
    struct if_data if_data;   /* statistics and other data about if */
    /* procedure handles */
    int (*if_output)          /* output routine (enqueue) */
        __P((struct ifnet *, struct mbuf *, struct sockaddr *,
            struct rentry *));
    void (*if_start)          /* initiate output routine */
        __P((struct ifnet *));
    int (*if_ioctl)          /* ioctl routine */
        __P((struct ifnet *, u_long, caddr_t));
    int (*if_reset)          /* XXX bus reset routine */
        __P((struct ifnet *));
    void (*if_watchdog)       /* timer routine */
        __P((struct ifnet *));
    struct ifqueue {
        struct mbuf *ifq_head;
        struct mbuf *ifq_tail;
        int ifq_len;
        int ifq_maxlen;
        int ifq_drops;
    } if_snd;
    struct sockaddr_dl *if_sadl; /* pointer to our sockaddr_dl */
    u_int8_t *if_broadcastaddr; /* linklevel broadcast bytestring */
};

#define if_mtu if_data.ifi_mtu
#define if_type if_data.ifi_type
#define if_addrlen if_data.ifi_addrlen
#define if_hdrlen if_data.ifi_hdrlen
#define if_metric if_data.ifi_metric
#define if_baudrate if_data.ifi_baudrate
#define if_ipackets if_data.ifi_ipackets
#define if_ierrors if_data.ifi_ierrors
#define if_opackets if_data.ifi_opackets
#define if_oerrors if_data.ifi_oerrors
#define if_collisions if_data.ifi_collisions
#define if_ibytes if_data.ifi_ibytes
#define if_obytes if_data.ifi_obytes
#define if_imcasts if_data.ifi_imcasts
#define if_omcasts if_data.ifi_omcasts
#define if_iqdrops if_data.ifi_iqdrops
#define if_noproto if_data.ifi_noproto
#define if_lastchange if_data.ifi_lastchange

#define IFF_UP 0x1          /* interface is up */

```

```

#define IFF_BROADCAST 0x2 /* broadcast address valid */
#define IFF_DEBUG 0x4 /* turn on debugging */
#define IFF_LOOPBACK 0x8 /* is a loopback net */
#define IFF_POINTOPOINT 0x10 /* interface is point-to-point link */
#define IFF_NOTRAILERS 0x20 /* avoid use of trailers */
#define IFF_RUNNING 0x40 /* resources allocated */
#define IFF_NOARP 0x80 /* no address resolution protocol */
#define IFF_PROMISC 0x100 /* receive all packets */
#define IFF_ALLMULTI 0x200 /* receive all multicast packets */
#define IFF_OACTIVE 0x400 /* transmission in progress */
#define IFF_SIMPLEX 0x800 /* can't hear own transmissions */
#define IFF_LINK0 0x1000 /* per link layer defined bit */
#define IFF_LINK1 0x2000 /* per link layer defined bit */
#define IFF_LINK2 0x4000 /* per link layer defined bit */
#define IFF_MULTICAST 0x8000 /* supports multicast */

/* flags set internally only: */
#define IFF_CANTCHANGE \
    (IFF_BROADCAST|IFF_POINTOPOINT|IFF_RUNNING|IFF_OACTIVE|\
     IFF_SIMPLEX|IFF_MULTICAST|IFF_ALLMULTI)

/*
 * Output queues (ifp->if_snd) and internetwork datagram level (pup level 1)
 * input routines have queues of messages stored on ifqueue structures
 * (defined above). Entries are added to and deleted from these structures
 * by these macros, which should be called with ipl raised to splimp().
 */
#define IF_QFULL(ifq) ((ifq)->ifq_len >= (ifq)->ifq_maxlen)
#define IF_DROP(ifq) ((ifq)->ifq_drops++)
#define IF_ENQUEUE(ifq, m) { \
    (m)->m_nextpkt = 0; \
    if ((ifq)->ifq_tail == 0) \
        (ifq)->ifq_head = m; \
    else \
        (ifq)->ifq_tail->m_nextpkt = m; \
    (ifq)->ifq_tail = m; \
    (ifq)->ifq_len++; \
}
#define IF_PREPEND(ifq, m) { \
    (m)->m_nextpkt = (ifq)->ifq_head; \
    if ((ifq)->ifq_tail == 0) \
        (ifq)->ifq_tail = (m); \
    (ifq)->ifq_head = (m); \
    (ifq)->ifq_len++; \
}
#define IF_DEQUEUE(ifq, m) { \
    (m) = (ifq)->ifq_head; \
    if (m) { \
        if (((ifq)->ifq_head = (m)->m_nextpkt) == 0) \
            (ifq)->ifq_tail = 0; \
        (m)->m_nextpkt = 0; \
        (ifq)->ifq_len--; \
    } \
}

#define IFQ_MAXLEN 50
#define IFNET_SLOWHZ 1 /* granularity is 1 second */

/*
 * The ifaddr structure contains information about one address
 * of an interface. They are maintained by the different address families,
 * are allocated and attached when an address is set, and are linked
 * together so all addresses for an interface can be located.

```

```

*/
struct ifaddr {
    struct sockaddr *ifa_addr; /* address of interface */
    struct sockaddr *ifa_dstaddr; /* other end of p-to-p link */
#define ifa_broadaddr ifa_dstaddr /* broadcast address interface */
    struct sockaddr *ifa_netmask; /* used to determine subnet */
    struct ifnet *ifa_ifp; /* back-pointer to interface */
    TAILQ_ENTRY(ifaddr) ifa_list; /* list of addresses for interface */
    void (*ifa_rtrequest) /* check or clean routes (+ or -)'d */
        __P((int, struct rtentry *, struct sockaddr *));
    u_short ifa_flags; /* mostly rt_flags for cloning */
    short ifa_refcnt; /* count of references */
    int ifa_metric; /* cost of going out this interface */
};
#define IFA_ROUTE RTF_UP /* route installed */

/*
 * Message format for use in obtaining information about interfaces
 * from sysctl and the routing socket.
 */
struct if_msghdr {
    u_short ifm_msglen; /* to skip over non-understood messages */
    u_char ifm_version; /* future binary compatability */
    u_char ifm_type; /* message type */
    int ifm_addrs; /* like rtm_addrs */
    int ifm_flags; /* value of if_flags */
    u_short ifm_index; /* index for associated ifp */
    struct if_data ifm_data; /* statistics and other data about if */
};

/*
 * Message format for use in obtaining information about interface addresses
 * from sysctl and the routing socket.
 */
struct ifa_msghdr {
    u_short ifam_msglen; /* to skip over non-understood messages */
    u_char ifam_version; /* future binary compatability */
    u_char ifam_type; /* message type */
    int ifam_addrs; /* like rtm_addrs */
    int ifam_flags; /* value of ifa_flags */
    u_short ifam_index; /* index for associated ifp */
    int ifam_metric; /* value of ifa_metric */
};

/*
 * Interface request structure used for socket
 * ioctl's. All interface ioctl's must have parameter
 * definitions which begin with ifr_name. The
 * remainder may be interface specific.
 */
struct ifreq {
    char ifr_name[IFNAMSIZ]; /* if name, e.g. "en0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_metric;
        int ifru_mtu;
        caddr_t ifru_data;
    } ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr /* address */

```

```

#define      ifr_dstaddr  ifr_ifru.ifru_dstaddr      /* other end of p-to-p
link */
#define      ifr_broadaddr ifr_ifru.ifru_broadaddr  /* broadcast address */
#define      ifr_flags    ifr_ifru.ifru_flags /* flags */
#define      ifr_metric   ifr_ifru.ifru_metric /* metric */
#define      ifr_mtu      ifr_ifru.ifru_mtu /* mtu */
#define      ifr_media    ifr_ifru.ifru_metric /* media options
(overload) */
#define      ifr_data      ifr_ifru.ifru_data /* for use by interface */
};

struct ifaliasreq {
    char    ifra_name[IFNAMSIZ];          /* if name, e.g. "en0" */
    struct sockaddr ifra_addr;
    struct sockaddr ifra_dstaddr;
#define      ifra_broadaddr    ifra_dstaddr
    struct sockaddr ifra_mask;
};

struct ifmediareq {
    char    ifm_name[IFNAMSIZ];          /* if name, e.g. "en0" */
    int     ifm_current;                  /* current media options */
    int     ifm_mask;                     /* don't care mask */
    int     ifm_status;                   /* media status */
    int     ifm_active;                   /* active options */
    int     ifm_count;                   /* # entries in ifm_ulist
array */
    int     *ifm_ulist;                  /* media words */
};

/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
 * for machine (useful for programs which
 * must know all networks accessible).
 */
struct ifconf {
    int     ifc_len;                     /* size of associated buffer */
    union {
        caddr_t    ifcu_buf;
        struct ifreq *ifcu_req;
    } ifc_ifcu;
#define      ifc_buf      ifc_ifcu.ifcu_buf /* buffer address */
#define      ifc_req      ifc_ifcu.ifcu_req /* array of structures returned
*/
};

#include <net/if_arp.h>

#ifdef _KERNEL
#define      IFAFREE(ifa) \
    if ((ifa)->ifa_refcnt <= 0) \
        ifafree(ifa); \
    else \
        (ifa)->ifa_refcnt--;

struct ifnet_head ifnet;

void    ether_ifattach __P((struct ifnet *, u_int8_t *));
void    ether_input __P((struct ifnet *, struct ether_header *, struct mbuf *));
int     ether_output __P((struct ifnet *,
    struct mbuf *, struct sockaddr *, struct rentry *));
char    *ether_sprintf __P((u_char *));

```

```

void if_attach __P((struct ifnet *));
void if_down __P((struct ifnet *));
void if_qflush __P((struct ifqueue *));
void if_slowtimo __P((void *));
void if_up __P((struct ifnet *));
int ifconf __P((u_long, caddr_t));
void ifinit __P((void));
int ifioctl __P((struct socket *, u_long, caddr_t, struct proc *));
int ifpromisc __P((struct ifnet *, int));
struct ifnet *ifunit __P((char *));

struct ifaddr *ifa_ifwithaddr __P((struct sockaddr *));
struct ifaddr *ifa_ifwithaf __P((int));
struct ifaddr *ifa_ifwithdstaddr __P((struct sockaddr *));
struct ifaddr *ifa_ifwithnet __P((struct sockaddr *));
struct ifaddr *ifa_ifwithladdr __P((struct sockaddr *));
struct ifaddr *ifa_ifwithroute __P((int, struct sockaddr *,
                                   struct sockaddr *));
struct ifaddr *ifaof_ifpforaddr __P((struct sockaddr *, struct ifnet *));
void ifafree __P((struct ifaddr *));
void link_rtrequest __P((int, struct rentry *, struct sockaddr *));

int loioctl __P((struct ifnet *, u_long, caddr_t));
void loopattach __P((int));
int looutput __P((struct ifnet *,
                 struct mbuf *, struct sockaddr *, struct rentry *));
void lortrequest __P((int, struct rentry *, struct sockaddr *));
#endif /* _KERNEL */
#endif /* !_NET_IF_H */

```

C. IF_ATMSUBR.C

```

/*      $NetBSD: if_atmsubr.c,v 1.12 1997/03/15 21:10:45 cgd Exp $      */

/*
 *
 * Copyright (c) 1996 Charles D. Cranor and Washington University.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *        This product includes software developed by Charles D. Cranor and
 *        Washington University.
 * 4. The name of the author may not be used to endorse or promote products
 *    derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 * IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT

```

```

* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
* DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
* THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
* THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/*
* if_atmsubr.c
*/

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/mbuf.h>
#include <sys/protosw.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/errno.h>
#include <sys/syslog.h>

#include <machine/cpu.h>

#include <net/if.h>
#include <net/netisr.h>
#include <net/route.h>
#include <net/if_dl.h>
#include <net/if_types.h>
#include <net/if_atm.h>
#include <net/ethertypes.h> /* XXX: for ETHERTYPE_* */

#include <netinet/in.h>
#include <netinet/if_atm.h>

#ifdef INET
#include <netinet/in_var.h>
#endif
#ifdef NATM
#include <netnatm/natm.h>
#include <netnatm/atm_auth.h>
#endif

#define senderr(e) { error = (e); goto bad;}

#define STARGATE /* Enable Stargate code */
#undef STARGATE_DEBUG /* Disable debug code */
#define MD5_METRIC /* Enable metrics */

#ifdef MD5_METRIC
#include <sys/time.h>
#define TRIAL 10000 /* 5000 sign and 5000 auth */
typedef struct timeval obs[3];
obs sample[TRIAL+1];
int framecount = 0;
extern long atm_frame_ctr;
#endif

/*
* atm_output: ATM output routine
* inputs:
* "ifp" = ATM interface to output to
* "m0" = the packet to output

```



```

*      "dst" = the sockaddr to send to (either IP addr, or raw VPI/VCI)
*      "rt0" = the route to use
*      returns: error code   [0 == ok]
*
*      note: special semantic: if (dst == NULL) then we assume "m" already
*              has an atm_pseudohdr on it and just send it directly.
*              [for native mode ATM output]   if dst is null, then
*              rt0 must also be NULL.
*/

int
atm_output(ifp, m0, dst, rt0)
    register struct ifnet *ifp;
    struct mbuf *m0;
    struct sockaddr *dst;
    struct rentry *rt0;
{
    u_int16_t etype = 0;                /* if using LLC/SNAP */
    int s, error = 0, sz;
    struct atm_pseudohdr atmdst, *ad;
    register struct mbuf *m = m0;
    register struct rentry *rt;
    struct atmllc *atmllc;
    u_int32_t atm_flags;

    if ((ifp->if_flags & (IFF_UP|IFF_RUNNING)) != (IFF_UP|IFF_RUNNING))
        senderr(ENETDOWN);
    ifp->if_lastchange = time;

    /*
     * check route
     */
    if ((rt = rt0) != NULL) {
        if ((rt->rt_flags & RTF_UP) == 0) { /* route went down! */
            if ((rt0 = rt = RTALLOC1(dst, 0)) != NULL)
                rt->rt_refcnt--;
            else
                senderr(EHOSTUNREACH);
        }

        if (rt->rt_flags & RTF_GATEWAY) {
            if (rt->rt_gwroute == 0)
                goto lookup;
            if ((rt = rt->rt_gwroute)->rt_flags & RTF_UP) == 0) {
                rtfree(rt); rt = rt0;
            }
            lookup: rt->rt_gwroute = RTALLOC1(rt->rt_gateway, 0);
            if ((rt = rt->rt_gwroute) == 0)
                senderr(EHOSTUNREACH);
        }
    }

    /* XXX: put RTF_REJECT code here if doing ATMARP */

}

/*
 * check for non-native ATM traffic   (dst != NULL)
 */
if (dst) {
    switch (dst->sa_family) {

```

```

#ifdef INET
    case AF_INET:
        if (!atmresolve(rt, m, dst, &atmdst)) {
            m = NULL;
            /* XXX: atmresolve already free'd it */
            senderr(EHOSTUNREACH);
            /* XXX: put ATMARP stuff here */
            /* XXX: watch who frees m on failure */
        }
        etype = htons(ETHERTYPE_IP);
        break;
#endif

    default:
#ifdef __NetBSD__
        || defined(__OpenBSD__)
        printf("%s: can't handle af%d\n", ifp->if_xname,
            dst->sa_family);
#elif defined(__FreeBSD__)
        || defined(__bsdi__)
        printf("%s%d: can't handle af%d\n", ifp->if_name,
            ifp->if_unit, dst->sa_family);
#endif
        senderr(EAFNOSUPPORT);
    }

    /*
     * must add atm_pseudohdr to data
     */
    sz = sizeof(atmdst);
    atm_flags = ATM_PH_FLAGS(&atmdst);
    if (atm_flags & ATM_PH_LLCSNAP) sz += 8; /* sizeof snap == 8 */
    M_PREPEND(m, sz, M_DONTWAIT);
    if (m == 0)
        senderr(ENOBUFS);
    ad = mtod(m, struct atm_pseudohdr *);
    *ad = atmdst;
    if (atm_flags & ATM_PH_LLCSNAP) {
        atmllc = (struct atmllc *) (ad + 1);
        bcopy(ATMLLC_HDR, atmllc->llchdr,
            sizeof(atmllc->llchdr));
        ATM_LLC_SETTYPE(atmllc, etype);
        /* note: already in network order */
    }
}

/*
 * Queue message on interface, and start output if interface
 * not yet active.
 */

s = splimp();
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);
    splx(s);
    senderr(ENOBUFS);
}
ifp->if_obytes += m->m_pkthdr.len;
IF_ENQUEUE(&ifp->if_snd, m);
if ((ifp->if_flags & IFF_OACTIVE) == 0)
    (*ifp->if_start)(ifp);
splx(s);
return (error);

bad:

```

```

        if (m)
            m_freem(m);
        return (error);
    }

/*
 * Process a received ATM packet;
 * the packet is in the mbuf chain m.
 */
void
atm_input(ifp, ah, m, rxhand)
    struct ifnet *ifp;
    register struct atm_pseudohdr *ah;
    struct mbuf *m;
    void *rxhand;
{
    register struct atm_pseudohdr *ah_new;
    register struct ifqueue *inq;
    u_int16_t etype = ETHERTYPE_IP; /* default */
    int s/*, index*/;

    if ((ifp->if_flags & IFF_UP) == 0) {
        m_freem(m);
        return;
    }
    ifp->if_lastchange = time;
    ifp->if_abytes += m->m_pkthdr.len;

    /******
     * Stargate modification code. Route all packets (except vpi 0 vci 32) to
     * the authentication function.
     */

    if ((ATM_PH_VCI(ah)) == 32) { /* must be a control cell! */
        printf("counter = %ld\n", atm_frame_ctr);
        atm_frame_ctr = 0; /* reset authenticated frame counter */
    }

#ifdef MD5_METRIC
    framecount = 0; /* reset frame counter for testing */
#endif

    }
    else {

#ifdef STARGATE_DEBUG
        ah_new = mtod(m, struct atm_pseudohdr *);
        printf("message length[in] = %d\n", m->m_pkthdr.len);
        printf("mbuf atm header = VPI %d VCI %d FLAGS 0x%x\n",
            ATM_PH_VPI(ah_new), ATM_PH_VCI(ah_new), ATM_PH_FLAGS(ah_new));
        printf("dump original mbuf contents\n");
        for(index=0; index < m->m_pkthdr.len; index++){
            printf("%02x", mtod(m, u_int8_t *)[index]);
        }; printf("\n");
#endif

        M_PREPEND(m, 4, M_DONTWAIT);
        ah_new = mtod(m, struct atm_pseudohdr *);
        ATM_PH_SETFLAGS(ah_new, 1);
        ATM_PH_SETVPI(ah_new, 0);
        if ((ATM_PH_VCI(ah)) == 33 || (ATM_PH_VCI(ah)) == 34) {

#ifdef MD5_METRIC
            microtime(&sample[framecount][1]);

```

```

#endif
    sign(m);
    if((ATM_PH_VCI(ah)) == 33){
        ATM_PH_SETVCI(ah_new,133);
    }
    else{
        ATM_PH_SETVCI(ah_new,134);
    }
}
else{
#ifdef MD5_METRIC
    microtime(&sample[framecount][1]);
#endif

    if(auth(m))
    {
        if((ATM_PH_VCI(ah)) == 133){
            ATM_PH_SETVCI(ah_new,33);
        }
        else{
            ATM_PH_SETVCI(ah_new,34);
        }
    }
    else
    {
        m_freem(m);
        return;
    }
}

#ifdef STARGATE_DEBUG
printf("message length[out] = %d\n", m->m_pkthdr.len);
printf("mbuf atm header = VPI %d VCI %d FLAGS 0x%x\n",
        ATM_PH_VPI(ah_new), ATM_PH_VCI(ah_new), ATM_PH_FLAGS(ah_new));
printf("dump modified mbuf contents\n");
for(index=0; index < m->m_pkthdr.len; index++){
    printf("%02x", mtod(m, u_int8_t *)[index]);
}; printf("\n");
#endif

#ifdef MD5_METRIC
    microtime(&sample[framecount][2]);
    framecount++;
    if(framecount >= TRIAL){
        long sum_sign = 0;
        long sum_auth = 0;
        int i;
        for(i = 0; i < TRIAL; i++){
            if((i % 2) == 0)
                sum_sign += (sample[i][2].tv_usec - sample[i][1].tv_usec);
            else
                sum_auth += (sample[i][2].tv_usec - sample[i][1].tv_usec);
        }
        printf("Average execution time(5000 trials): %lu usec(sign) and %lu
u sec(auth)\n", sum_sign/(TRIAL/2), sum_auth/(TRIAL/2));
        framecount = 0;
    }
#endif

    atm_output(ifp, m, NULL, NULL);
    return;
}
/*

```

```

* End Stargate modification code
*****/

    if (rxhand) {
#ifdef NATM
        struct natmpcb *npcb = rxhand;
        s = splimp();
        npcb->npcb_inq++;
        splx(s);
        schednetisr(NETISR_NATM);
        inq = &natmintrq;
        m->m_pkthdr.rcvif = rxhand; /* XXX: overload */
#else
        printf("atm_input: NATM detected but not configured in kernel\n");
        m_freem(m);
        return;
#endif
    } else {
        /*
         * handle LLC/SNAP header, if present
         */
        if (ATM_PH_FLAGS(ah) & ATM_PH_LLCSNAP) {
            struct atmlhc *alc;
            if (m->m_len < sizeof(*alc) && (m = m_pullup(m, sizeof(*alc))) == 0)
                return; /* failed */
            alc = mtod(m, struct atmlhc *);
            if (bcmp(alc, ATMLLC_HDR, 6)) {
#ifdef __NetBSD__ || defined(__OpenBSD__)
                printf("%s: recv'd invalid LLC/SNAP frame [vp=%d,vc=%d]\n",
                    ifp->if_name, ATM_PH_VPI(ah), ATM_PH_VCI(ah));
            #elif defined(__FreeBSD__) || defined(__bsdi__)
                printf("%s%d: recv'd invalid LLC/SNAP frame [vp=%d,vc=%d]\n",
                    ifp->if_name, ifp->if_unit, ATM_PH_VPI(ah), ATM_PH_VCI(ah));
            #endif
            m_freem(m);
            return;
        }
        etype = ATM_LLC_TYPE(alc);
        m_adj(m, sizeof(*alc));
    }

    switch (etype) {
#ifdef INET
        case ETHERTYPE_IP:
            schednetisr(NETISR_IP);
            inq = &ipintrq;
            break;
#endif
        default:
            m_freem(m);
            return;
    }

    s = splimp();
    if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
    } else
        IF_ENQUEUE(inq, m);
    splx(s);
}

```

```

/*
 * Perform common duties while attaching to interface list
 */
void
atm_ifattach(ifp)
    register struct ifnet *ifp;
{
    register struct ifaddr *ifa;
    register struct sockaddr_dl *sdl;

    ifp->if_type = IFT_ATM;
    ifp->if_addrlen = 0;
    ifp->if_hdrlen = 0;
    ifp->if_mtu = ATMMTU;
    ifp->if_output = atm_output;

#ifdef __NetBSD__ || defined(__OpenBSD__)
    for (ifa = ifp->if_addrlist.tqh_first; ifa != 0;
         ifa = ifa->ifa_list.tqe_next)
#elif defined(__FreeBSD__) && ((__FreeBSD__ > 2) || defined(_NET_IF_VAR_H))
/*
 * for FreeBSD-3.0. 3.0-SNAP-970124 still sets -D__FreeBSD__=2!
 * XXX -- for now, use newly-introduced "net/if_var.h" as an identifier.
 * need a better way to identify 3.0. -- kjc
 */
    for (ifa = ifp->if_addrhead.tqh_first; ifa;
         ifa = ifa->ifa_link.tqe_next)
#elif defined(__FreeBSD__) || defined(__bsdi__)
    for (ifa = ifp->if_addrlist; ifa; ifa = ifa->ifa_next)
#endif
        if ((sdl = (struct sockaddr_dl *)ifa->ifa_addr) &&
            sdl->sdl_family == AF_LINK) {
            sdl->sdl_type = IFT_ATM;
            sdl->sdl_alen = ifp->if_addrlen;
#ifdef notyet /* if using ATMARP, store hardware address using the next line */
            bcopy(ifp->hw_addr, LLADDR(sdl), ifp->if_addrlen);
#endif
            break;
        }
}

```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. ATM AUTHENTICATION CODE

A. *ATM_AUTH.H*

```
#ifdef _KERNEL
    int sign(struct mbuf *);
    int auth(struct mbuf *);
#endif
```

B. *ATM_AUTH.C*

```
/* ATM authentication code
 * sign() and auth()
 */
#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/mbuf.h>
#include <sys/md5.h>
#include <netnatm/atm_auth.h>

/***** Define the Authentication Trailer (AT) 28 bytes *****/
#define MAC_LEN      16      /* message digest, 128 bit */
#define VO_LEN       2       /* version/options */
#define SEQ_LEN      4       /* sequence number */
#define KI_LEN       4       /* key index, 32 bit */
#define KEY_LEN      16      /* key, 128 bit */
#define KEYTABLE_LEN 20      /* ki and key combined in table */
#define AT_LEN       (VO_LEN + SEQ_LEN + KI_LEN + MAC_LEN)
#define TPART_LEN    10 /* (AT_LEN - (KI_LEN + KEY_LEN)) 6 bytes */

#if 0
    #define AUTH_DEBUG 1 /* uncomment for debug messages */
#endif

/* Keytable entry is ki(4 bytes) + key(16 bytes) */

typedef union _keytable {
    char index[20];
    struct keyinfo {
        u_int32_t ki;
        char key[16];
    } keyinfo;
} Keytable;

/* The allocation of memory is done at the callee of getkey or
   verikey */

Keytable *key_table;

long atm_frame_ctr = 0;

static Keytable * getkey(void);
static Keytable * verikey(u_int32_t);
```



```

/*****
* sign(struct mbuf *m)
* Used to generate and append a MAC trailer
* Uses: getkey() (temporarily implemented below)
*       MD5Init(), MD5Update(), MD5Final() (included with NetBSD)
*****/

int sign(m)
    register struct mbuf *m;
{
    u_int32_t ki;
    u_int16_t vo;    /* version/options field */
    u_int32_t seq;    /* sequence number */
    unsigned char procbuf[5200]; /* work buffer for md5 computation */
    unsigned char digest[ MAC_LEN ];
    unsigned char tpart[ TPART_LEN ];
    unsigned char at[ AT_LEN ];
    MD5_CTX context;
    struct mbuf *m_new;
    int off, msg_len;

    union {
        u_int8_t c[4];
        u_int32_t l;
    } l_util;

    union {
        u_int8_t c[2];
        u_int16_t s;
    } s_util;

#ifdef AUTH_DEBUG
    { int idx;
      printf("debug[sign]: dump untouched mbuf contents\n");
      for(idx=0; idx<m->m_pkthdr.len; idx++){ /*dump data*/
          printf("%02x", mtod(m, u_int8_t *)[idx]);
      }; printf("\n");
    }
#endif
    vo = 0; /* To be implemented in future */
    s_util.s = vo;
    tpart[0] = s_util.c[0];
    tpart[1] = s_util.c[1];

    seq = 0; /* To be implemented in future */
    l_util.l = seq;
    tpart[2] = l_util.c[0];
    tpart[3] = l_util.c[1];
    tpart[4] = l_util.c[2];
    tpart[5] = l_util.c[3];

    key_table = getkey();
    ki = key_table->keyinfo.ki;
    l_util.l = ki;
    tpart[6] = l_util.c[0];
    tpart[7] = l_util.c[1];
    tpart[8] = l_util.c[2];
    tpart[9] = l_util.c[3];

    /* Get message length */
    msg_len = m->m_pkthdr.len;

    /* Copy mbuf data into procbuf */

```

```

    off = 0;
    for( m_new=m; m_new; m_new=m_new->m_next) {
        bcopy(m_new->m_data, procbuf + off, m_new->m_len);
        off += m_new->m_len;
    }

    /* Append tpart (including ki) to msg payload in procbuf */
    bcopy(tpart, procbuf + msg_len, TPART_LEN);

    /* Append key to procbuff */
    bcopy(key_table->keyinfo.key , procbuf+msg_len+TPART_LEN, KEY_LEN);

#ifdef AUTH_DEBUG
{ int idx;
  printf("debug[sign]: dump procbuf contents\n");
  for(idx=0; idx<msg_len+TPART_LEN+KEY_LEN; idx++){
      printf("%02x", procbuf[idx]);
  } printf("\n");
}
#endif

    /* Run MD5 on procbuf */
    MD5Init(&context);

    /* Exclude the psuedo-header "procbuf + 4" */
    MD5Update(&context, procbuf + 4, (msg_len - 4)+TPART_LEN+KEY_LEN);

    MD5Final(digest, &context);

#ifdef AUTH_DEBUG
{ int idx;
  printf("debug[sign]: message digest is\n");
  for(idx=0; idx<16; idx++){
      printf("%02x", digest[idx]);
  } printf("\n");
}
#endif

    /* compose auth trailer (at) - tpart followed by mac */
    bcopy(tpart, at, TPART_LEN);
    bcopy(digest, at+TPART_LEN, MAC_LEN);

#ifdef AUTH_DEBUG
{ int idx;
  printf("debug[sign]: auth trailer (at) contents\n");
  for(idx=0; idx<26; idx++){
      printf("%02x", at[idx]);
  } printf("\n");
}
#endif

    /* append at to mbuf */
    m_copyback(m, msg_len, AT_LEN, at);

#ifdef AUTH_DEBUG
{ int idx;
  printf("debug[sign]: dump signed mbuf contents\n");
  for(idx=0; idx<m->m_pkthdr.len; idx++){ /*dump data*/
      printf("%02x", mtod(m, u_int8_t *)[idx]);
  }; printf("\n");
}
#endif

    /* printf("Packet signed!\n"); */

```

```

    /* free key_table memory */
    free(key_table, M_TEMP);
    return(1); /* success */
}

/*****
 * auth(struct mbuf *m)
 * Used on incoming frames to authenticate frames based on an
 * MD5 computed message digest.
 * Uses: verikey() (temporarily implemented below)
 *       MD5Init(), MD5Update(), MD5Final() (included with NetBSD)
 *****/

int auth(m)
    register struct mbuf *m;
{
    u_int16_t vo;          /* version/options field */
    u_int32_t seq;         /* sequence number */
    u_int32_t ki;         /* Key Index */
    unsigned char procbuf[5200]; /* work buffer for md5 computation */
    unsigned char mac0[ MAC_LEN ]; /* MAC carried by frame */
    unsigned char digest[ MAC_LEN ];
    MD5_CTX context;
    struct mbuf *m_new;
    int off, ki_index, mac_index, msg_len;

#ifdef AUTH_DEBUG
    { int idx;
      printf("debug[auth]: dump untouched mbuf contents\n");
      for(idx=0; idx<m->m_pkthdr.len; idx++){ /*dump data*/
          printf("%02x", mtod(m, u_int8_t *)[idx]);
      }; printf("\n");
    }
#endif
    vo = 0;
    seq = 0;
    m_new = m;

    /* Get message length */
    msg_len = m->m_pkthdr.len; /* includes authentication trailer!!! */

#ifdef AUTH_DEBUG
    printf("debug[auth]: msg_len = %d\n", msg_len);
#endif

    /* Copy mbuf (including tpart and mac) to procbuf */
    off = 0;
    for( m_new=m; m_new; m_new=m_new->m_next) {
        bcopy(m_new->m_data, procbuf + off, m_new->m_len);
        off += m_new->m_len;
    }

    /* Extract the 4 byte key index */
    ki_index = (msg_len - MAC_LEN) - KI_LEN;
    ki = *(u_int32_t *) (procbuf + ki_index);

#ifdef AUTH_DEBUG
    printf("debug[auth]: extracted ki = %d\n", ki);
#endif

    /* Extract message digest (MAC) */
    mac_index = msg_len - MAC_LEN;
    bcopy (procbuf + mac_index, mac0, MAC_LEN);

```

```

#ifdef AUTH_DEBUG
{ int idx;
  printf("debug[auth]: dump extracted mac \n");
  for(idx=0; idx<MAC_LEN; idx++){ /*dump data*/
    printf("%02x", mac0[idx]);
  }; printf("\n");
}
#endif

/* Use ki to get key */
key_table = verikey(ki); /* includes 4 byte ki and 16 byte key */

#ifdef AUTH_DEBUG
{ int idx;
  printf("debug[auth]: key is \n");
  for(idx=0; idx<KEY_LEN; idx++){ /*dump data*/
    printf("%02x", key_table->keyinfo.key[idx]);
  }; printf("\n");
}
#endif

/* Overwrite mac field in procbuf with key */
bcopy(key_table->keyinfo.key , procbuf+(msg_len-MAC_LEN), KEY_LEN);

#ifdef AUTH_DEBUG
{ int idx;
  printf("debug[auth]: dump procbuf contents\n");
  for(idx=0; idx<msg_len; idx++){
    printf("%02x", procbuf[idx]);
  } printf("\n");
}
#endif

/* Run MD5 on procbuf */
MD5Init(&context);

/* Exclude psuedo-header "procbuf + 4" */
MD5Update(&context, procbuf + 4, msg_len - 4);

MD5Final(digest, &context);

#ifdef AUTH_DEBUG
{ int idx;
  printf("debug[auth]: message digest is\n");
  for(idx=0; idx<16; idx++){
    printf("%02x", digest[idx]);
  } printf("\n");
}
#endif

/* Compare computed digest with recieved MAC */
if(bcmp(digest, mac0, MAC_LEN) != 0) {
  printf("Packet does not authenticate correctly!\n");
  free(key_table, M_TEMP);
  return(0); /* failure */
} else {
  /*(printf("Packet authenticated!\n");*/
  atm_frame_ctr++;
  /* Remove AT from mbuf */
  m_adj(m, -(AT_LEN));
  free(key_table, M_TEMP);
  return(1); /* success */
}

```

```

    }
} /* end auth(m) */

/*****getkey()*****/
static Keytable *getkey()
{
    int i;

    key_table = (Keytable *) malloc(sizeof(u_int8_t)*20, M_TEMP, M_NOWAIT);
    key_table->keyinfo.ki = 101;
    for(i=0; i<16; i++){
        key_table->keyinfo.key[i] = 37;
    }
    /* should return array with ki and key */
    return (Keytable *) (key_table);
}
/*****/

/*****verikey()*****/
static Keytable *verikey(ki)
    u_int32_t ki;
{
    int i;
    if(ki==101){
        key_table = (Keytable *) malloc(sizeof(u_int8_t)*20, M_TEMP, M_NOWAIT);
        for(i=0; i<16; i++){
            key_table->keyinfo.key[i] = 37;
        }
        return (Keytable *) (key_table); /* success */
    }
    else
        return(NULL); /* failure */
}
/*****/

```

APPENDIX D. KEY MANAGEMENT CODE

A. KDC.JAVA

```
//-----
// Filename: KDC.java
// Date:      June 1999
// Compiler:  JDK 1.2
//-----

import java.io.*;
import java.net.*;
import java.util.Random;
import java.util.Date;

/**
 * this class creates a KDC object.  This object
 * acts as a client in the networking sense for Stargate
 * objects.  The KDC is responsible for generating
 * the random KI's and authentication keys, as well
 * as the masks for updating tables with new values.
 * There are helper methods to get information about
 * the KI and keys.  The KDC will set up a time for
 * sending masks to Stargate devices and will indicate
 * to the Stargate device when new tables should be used.
 * @author Katrina Hensley
 */

public class KDC implements Serializable{

    /**
     * MAXNUM is the number of objects to be created
     * in the key table
     */
    private static final int MAXNUM = 720;

    /**
     * current_time used to hold the current time;
     */
    private static long current_time;

    /**
     * sendWaitTime used to determine when to send
     * the next data message to Stargate.  It is
     * dependent on the (cryptoperiod * # of keys-1) - startWaitTime
     * -1 from # of keys is used to add in some slack
     */
    private static long sendWaitTime;

    /**
     * startWaitTime is added to the current time to set
     * the sendWaitTime variable.  For testing the offset
     * is set for 20 minutes or 1200000 milliseconds
     */
    private static final long startWaitTime = (1000 * 60 * 20);
}
```

```

/**
 * KEY_LIFETIME is the cryptoperiod for each key
 * the period for testing is 2 min
 */
private static final long KEY_LIFETIME = (1000 * 60 * 2);

/**
 * MAXINT is used in creating the random #'s
 */
private static final int MAXINT = 2147483647;

/**
 * type will be used by the Stargate to determine
 * what information has been sent by the KDC.
 * 0 = table
 * 1 = mask
 */
private static String type = "0";

/**
 * data member for the 4-byte Key Index values
 */
private String keyIndex = " ";

/**
 * data member for the 16-byte key values
 */
private String keyData = " ";

//*****
// CLASS METHODS
//*****

/**
 * toString () to print key obj
 * @return String
 */
public String toString(){
    String retval= " ";

    //output format-> KI KEY
    retval = keyIndex + keyData;

    return(retval);
} //end toString

/**
 * get_table creates a table with KI and keys
 * this should only be called when Stargates
 * first come on line and then call get_mask
 * to formulate updated tables.
 * @return String used for the key table
 */
private String get_table() {

```

```

//create the table
String table = new String();

Random randNumGen = new Random(); //uses current time

//float value for new KI data member
int kiValue;

//long value for Keys data member
long keyValue1;

//long integer value for 2nd part of Keys data member
long keyValue2;

//create the max number Keys
for(int ix=0; ix < MAXNUM; ix++) {

    //the random seed value based on computer time
    kiValue = randNumGen.nextInt(MAXINT);
    keyValue1 = Math.abs(randNumGen.nextLong());
    keyValue2 = Math.abs(randNumGen.nextLong());
    keyIndex = String.valueOf(kiValue);
    keyData =
        String.valueOf(keyValue1) + String.valueOf(keyValue2);
    table = table.concat(keyIndex + keyData + " ");

}

return(table);
} //end get_table

/**
 * get_mask is used to get a "mask" used to
 * update existing key tables.
 * @return String
 */
private String get_mask(){

    String retVal = new String();

    Random randNumGen = new Random();
    Random randNumGen2 = new Random();

    retVal = retVal.concat(
        String.valueOf(randNumGen.nextInt(MAXINT)) +
        String.valueOf(Math.abs(randNumGen2.nextLong())));

    return(retVal);
}

/**
 * main function sets up connection with Stargate
 * and sends data.
 * @return void
 */
public static void main (String[] args){

    //variables

```



```

String host = args[0]; //provided host to connect to
int port = 8205;
Socket KDCsocket;
OutputStream outstream; //for sending info to Stargate
boolean goFlag = true; //set initial start flag to true
KDC testKDC = new KDC();
String tableToSend = new String();
String maskToSend = " ";
String startTime = " "; //tells when to begin use of table/mask
String message = " "; //holds the total data sent to Stargate

//check for argument
if (args.length < 1){
    System.err.println("Argument needed: Hostname");
    System.exit(0);
}

//set time for next send
//which is 2 min * #keys - 20 min
sendWaitTime = ((KEY_LIFETIME * (MAXNUM-1)) - startWaitTime);

//neverending loop to keep KDC running
while(true){

    //second while loop to process on initial startup
    while(goFlag == true){

        //initialize the network
        try {

            //get Stargate's ip address
            InetAddress Stargate = InetAddress.getByName(host);

            //connect to port on Stargate
            KDCsocket = new Socket(Stargate,port);

            //setup table
            tableToSend = testKDC.get_table();

            //set time variables
            current_time = System.currentTimeMillis();

            //output date of next time to send to the screen
            Date testTime = new Date(sendWaitTime + current_time);
            System.out.println
                ("Outputting when is next xmit " + testTime);

            //set time to start using table but since it
            //is the 1st table, start immediately
            startTime =
                String.valueOf(current_time);

            //setup the message
            message = type + tableToSend + startTime;
            System.out.println(message);

            //send the info to Stargate in the form of
            //[type | data | start_time]
            System.out.println("Table Sent...");

            outstream = KDCsocket.getOutputStream();
            PrintStream ss = new PrintStream(outstream);

```

```

        ss.println(message);

        //close socket
        KDCsocket.close();
        goFlag = false;
    }catch(Exception e){

        System.out.println("IOError...." + e.toString());
    }

}

}

//end inner while

//-----Start process to send mask-----//
try {

    //now wait for the next send time
    Thread.sleep(sendWaitTime);

    //then open socket & send mask

    //get Stargate's ip address
    InetAddress Stargate = InetAddress.getByName(host);

    //connect to port on Stargate
    KDCsocket = new Socket(Stargate,port);

    //clear the message
    message = " ";

    //set type to represent a mask
    type = "1";

    //set time variables
    current_time = System.currentTimeMillis();

    //feedback
    Date testTime = new Date(sendWaitTime + current_time);
    System.out.println
        ("Outputting when next xmit " + testTime);

    //set time to start using table
    startTime =
        String.valueOf(current_time + startWaitTime); //20 minutes

    Date testTime2 = new Date(current_time + startWaitTime);

    //get the mask
    maskToSend = testKDC.get_mask();

    //setup message
    message = type + "*" + maskToSend + "*" + startTime;

    //send it out
    System.out.println("the starting time is " + testTime2);
    System.out.println("SENDING MASK " + message);
    ostream = KDCsocket.getOutputStream();

    PrintStream ss = new PrintStream(ostream);

```


LIST OF REFERENCES

1. Cisco Systems Business Report, "Moving From TDM to Multiservice ATM," p. 17, 1999.
2. Celotek Corporation, "Frequently Asked Questions," <http://www.celotek.com/faqbody.com>, 1989.
3. Xie, G. G., Irvine, C., and Colwell C., "LLPA: A Protocol for High Speed Packet Authentication", Department of Computer Science, Naval Postgraduate School, May 1999.
4. Rivest, Ron, "The MD5 Message-Digest Algorithm," RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.
5. "Network Time Protocol," http://www.eecis.udel.edu/~ntp/ntp_spool/html/exec.htm, May 1998.
6. Stallings, W. Cryptography and Network Security: Principles and Practice. Upper Saddle River, Prentice Hall, Inc., 1999.
7. Levin, C., "Safety in Random Numbers," PC Magazine, v. 18, p. 30, 22 June 1999.
8. Wright, Gary R. and Stevens, W. Richard, TCP/IP Illustrated Volume 2 The Implementation, Addison-Wesley Longman, Inc. Reading, MA, 1995.
9. Touch, Joe, "Report on MD5 Performance," RFC 1810, Information Sciences Institute, University of Southern California, June 1995.

THIS PAGE INTENTIONALLY LEFT BLANK

BIBLIOGRAPHY

Chuang, Shaw-Cheng (1995). "Securing ATM Networks," Cambridge University ATM Documentation Collection 4 (The Green Book).

Denning, Dorothy E. (1999). Information Warfare and Security, ACM Press.

Flanagan, David. (1997). Java in a Nutshell, 2nd Edition, O'Reilly & Associates, Inc., Sebastopol, CA.

Gordon, Rob. (1998). Essential JNI Java Native Interface, Prentice Hall, Inc., Upper Saddle River, NJ.

Kelly, Al and Pohl, Ira (1998). A Book on C Programming in C, Forth Edition, Addison Wesley Longman Inc, Reading, MA.

Laurent, Maryline and Rolin, Pierre (1998). "ATM Security State of The Art," Proceedings of 1998 ATM Development, Rennes, France.

McDysan, David and Spohn, Darren (1999). ATM Theory and Applications. New York, McGraw Hill, Inc.

Sobell, Mark G. (1995). A Practical Guide to the UNIX System, Third Edition, Addison Wesley Longman, Inc., Reading, MA.

Sridharan, Prashant. (1997). Advanced Java Networking, Prentice Hall, Inc., Upper Saddle River, NJ.

Stevens, W. Richard (1994). TCP/IP Illustrated Volume 1 The Protocol, Addison-Wesley Longman, Inc., Reading, MA.

Stevens, W. Richard (1998). UNIX Network Programming Networking APIs: Sockets and XTI, Volume 1, Prentice Hall, Inc. Upper Saddle River, NJ.

Stevenson, David, Hillery, Nathan, and Byrd, Greg (1995). "Secure Communications in ATM Networks," Communications of the ACM, Vol. 38, No. 2.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
8725 John J. Kingman Road, Ste 0944
Fort Belvoir, VA 22060-6218

2. Dudley Knox Library 2
Naval Postgraduate School
411 Dyer Road
Monterey, California 93943-5101

3. Director, Training and Education 1
MCCDC, Code C46
1019 Elliot Road
Quantico, VA 22134-5027

4. Director, Marine Corps Research Center 2
MCCDC, Code C40RC
2040 Broadway Street
Quantico, VA 22134-5107

5. Director, Studies and Analysis Division 1
MCCDC, Code C45
3300 Russell Road
Quantico, VA 22134-5130

6. Marine Corps Representative 1
Naval Postgraduate School
Code 037, Bldg. 330, IN-116
555 Dyer Road
Monterey, CA 93940

7. Marine Corps Tactical Systems Support Activity 1
Technical Advisory Branch
Attn: Maj J. C. Cummiskey
Box 555171
Camp Pendleton, CA 92055-5080

8. Dr. Geoffrey G. Xie, CS/Xg 2
Naval Postgraduate School
Monterey, CA 93943

9. Professor Cynthia E. Irvine, CS/Ic 1
Naval Postgraduate School
Monterey, CA 93943

10. COL Timothy A. Fong 1
Deputy CEE for Information Assurance
Commander, Information Assurance Engineering Support Organization
Defense Information Systems Agency
5600 Columbia Pike
Falls Church, VA 22041
11. Captain Katrina Hensley 1
1321 S. 60th St.
Noble, OK 73068
12. Captain Fredrick Ludden 2
1211 40th St.
Parkersburg, WV 26101
13. Chairman, Code CS..... 1
Naval Postgraduate School
Monterey, CA 93943