Design and Implementation of IPsec on a Bare PC

N. Kazemi, A. L. Wijesinha, and R. Karne
Department of Computer and Information Sciences
Towson University
Towson, MD 21252
nkazem1@students.towson.edu, {awijesinha, rkarne}@towson.edu

***Abstract*—We consider IPsec as a means to provide security for a bare (OS-less) PC in a networked environment. We first present the underlying bare computing architecture that includes IPsec components. We then describe our design and implementation of IPsec with ESP and IKEv2 support for a bare PC. Bare PC optimizations that provide higher efficiency include application and protocol intertwining, and minimization of inter-layer communication and data copying. We also evaluate IPsec performance on a bare PC and compare it with a similar Linux system. Our experimental results indicate that 1) the cost of ESP processing is approximately 6% more for incoming versus outgoing packets; 2) an approximately linear relationship between ESP processing time and packet size exists for both tunnel and transport modes; 3) ICMP and HTTP response times with IPsec are less on a bare PC than on a Linux system.**

Keywords-IPsec; network security; bare machine computing; bare PC; application object; performance

I. INTRODUCTION

In bare machine computing [1], an application executes directly on the hardware without an Operating System (OS) or hard disk. The software is designed as an Application Object (AO) [2] whose executable code resides on a portable bootable device such as a USB memory stick. There are two main characteristics of bare machine computing that make it preferable to a conventional OS-based system in special cases. First, the absence of an OS and the streamlined design of an AO result in reduced overhead, which is useful for building low-cost high-performance systems. Second, security can be improved since attacks targeting OS vulnerabilities are eliminated and the smaller simpler AO code is easier to analyze and secure. This is useful for certain security applications where an OS is not needed and in fact may introduce hidden loopholes. Bare PC applications include Web servers [3], email servers [4], and VoIP clients [5].

Despite the inherent security advantages of bare computing, it is also necessary to protect AOs against typical security risks that are present in networked environments. IPsec, which is commonly used in VPNs to provide confidentiality and data integrity for IP traffic, is convenient for providing IP-level security for bare applications and enabling secure communication with conventional systems.

In this paper, we consider the design and implementation of IPsec on a bare PC and evaluate its performance. We also provide some performance comparison with applications that run on Linux. Due to the novel characteristics of bare machine computing and its unique architecture, the design and implementation of IPsec for an AO is different from a conventional implementation of IPsec in an OS-based system.

IPsec can be efficiently implemented on a bare PC by using optimizations not easily achieved in conventional systems. Examples are intertwining the application with network protocols within an AO, minimizing the cost of inter-layer communication, and eliminating data copying between application and network buffers.

The main contributions of this research are: 1) an implementation of IPsec for a bare (OS-less) PC; 2) experimental results related to the cost of IPsec processing on a bare PC; and 3) experimental results related to the increase in response time of applications when IPsec is present for a bare PC and for a conventional OS-based (Linux) PC. These results can serve as a baseline for designing future versions of IPsec for high-performance or high-security applications and security gateways running on bare machines.

The rest of this paper is organized as follows. In Section II, we provide a brief overview of bare machine computing, IPsec, and the Internet Key Exchange version 2.0 (IKEv2) protocol. We also review related work. In Section III, we describe the bare PC architecture relevant to IPsec. We introduce our design and implementation of IPsec on a bare PC in Section IV. In Section V, we present experimental results. Section VI contains the conclusion.

II. BACKGROUND

A. Bare Machine Computing

In bare machine computing, an AO, encapsulating one or more applications, contains only the minimal functionality required to run directly on the hardware without using an OS. In contrast to a conventional system where the OS manages computer resources on behalf of applications, bare machine computing uses direct interfaces, to the hardware, that allow the applications to optimize CPU and memory usage. In particular, the AO manages task scheduling, concurrency, and memory. While an OS performs functions to support a broad variety of applications, an AO is designed to serve the needs of a particular server or client application (or at most a few related applications). Thus, overheads associated with OS services are eliminated or minimized in bare machine computing.

The boot code for a bare PC application loads the self-contained AO into memory and initiates its execution. An AO uses a special bare machine API to the hardware [6] and includes its own device drivers, for example, to manage network or audio hardware interfaces, if needed by the application. Since there is no OS, code for a conventional application cannot be directly ported to a bare computer in view of system calls i.e., bare computing applications differ considerably from their OS-based counterparts.

In an OS-based system, the TCP/IP network protocol stack is viewed as a part of the OS, and conventional applications access network protocols via a socket interface that provides general operations needed for communication. In bare machine computing, an AO intertwines an application with the necessary protocols enabling optimizations with respect to task scheduling, data copying, and inter-layer communication. For example, communication between IPsec, IP, and UDP protocols in an AO is more efficient than in an OS-based system.

B. IPsec

IPsec [7] is a standard for protecting IP traffic and it has been widely used in implementing Virtual Private Networks (VPNs). However, any application running over IP may seamlessly benefit from IPsec's security services. IPsec defines two protocols: Authentication Header (AH) [8] and Encapsulating Security Payload (ESP) [9]. Most practical uses of IPsec are based on ESP, which provides data confidentiality, and optionally, data-origin authentication and data-integrity verification. It may also be used for replay-attack protection and traffic flow confidentiality. Communicating peers use ESP headers and ESP trailers that are placed inside IP datagrams to communicate ESP information.

IPsec supports two modes, transport and tunnel. In transport mode, IPsec protects upper layer protocol packets. In this mode, the endpoints of the IPsec connection are also the endpoints of communication. In tunnel mode (used in VPNs), an IP datagram is prefixed with an extra IP header and the entire IP datagram is protected. In this mode, one (or both endpoints) of an IPsec connection is usually a security gateway or router that implements IPsec. Therefore, in tunnel mode, these endpoints may or may not be the endpoints of communication.

In addition to ESP (or AH), the cryptographic algorithms, and a key management component, IPsec requires a Security Policy Database (SPD) and a Security Association Database (SAD). These databases define the security services that are afforded to network traffic, the IPsec connections between the two endpoints, and associated information for processing IP datagrams. The current bare PC implementation of IPsec includes minimal functionality for SPD and SAD, since it is only designed to support existing bare PC applications. However, it can be extended to support more complex IPsec requirements such as those for security gateways and VPNs.

C. The Internet Key Exchange Version 2.0 (IKEv2)

The key management component of IPsec used for managing Security Associations (SAs) is implemented via IKE, which exists in two versions IKEv1 [10] and IKEv2 [11]. Although many VPNs and OSs still support IKEv1, it has several shortcomings that are addressed in IKEv2. IPsec on a bare PC only supports IKEv2.

Using IKEv2, an initiator and a responder may exchange four IKE messages to establish SAs for an IPsec connection. The pair of IKE-SA-AUTH messages that follow the initial pair of IKE-SA-INIT messages are cryptographically protected. IKEv2 messages contain one or more payloads. For example, the IKE-SA-INIT messages shown in Fig. 1 contain Security Association (SA), Key Exchange (KE), and Nonce (N) payloads.

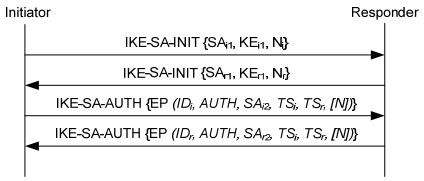

Figure 1. A sample IKEv2 message exchange to establish IKE SAs

IPsec endpoints perform a Diffie-Hellman (DH) exchange via IKE-SA-INIT messages to securely establish a shared secret, which is then used by a Pseudo-Random Function (PRF) to generate the required keys for cryptographic algorithms that are used by ESP and AH. The PRF and the cryptographic algorithms are negotiated using SA payloads. The exchange of IKE-SA-INIT and IKE-SA-AUTH messages results in the creation of four SAs: a pair of SAs for IKE traffic, and a pair of SAs (CHILD-SAs) for protecting IP traffic in each direction between initiator and responder.

D. Related Work

IPsec has been the subject of numerous studies. In [12], IPsec was implemented on several OSs including Linux and OpenBSD using lightweight kernel-user space communication, improved lookup speed and packet processing, and minimization of routing table lookups. The implementations used triple DES for encryption. Performance measurements including ICMP response times, and TCP/UDP throughput for various packet sizes were also provided. Early work such as [13] focused on the impact of hardware acceleration on performance by measuring throughput and transfer time. In [14], QoS support was added to an IPsec implementation on Linux. In [15], IPsec overhead with respect to ESP and AH was measured. In [16], ESP and IKEv1 performance was evaluated in a single client setting using MD5 for MAC calculations, and it was found that IKE overhead exceeded ESP overhead by a factor of 3. In [17], performance of IKEv1 with multiple clients was studied and caching techniques were shown to significantly reduce its overhead. It was also noted that IPsec ESP overhead significantly degraded HTTP performance of a VPN server in a multi-client environment compared to a native TCP/IP implementation.

In [18], a study of IPsec performance using ESP and AH in tunnel and transport mode determined that IPsec processing time increases linearly with packet size and that StrongS/WAN [19] performs better than native IPsec with Linux 2.6.9 for

large packets but the opposite is true for small packets. The authors indicated that integration of native IPsec with the kernel may have been the reason for its improved performance for small packets. This study included ESP with AES-128 encryption (no authentication) and AH with SHA-1.

Our study of IPsec differs from the preceding studies in that we consider the design and implementation of IPsec on a bare (OS-less) machine and evaluate its performance. We consider IKEv2 for key exchange and ESP with data confidentiality, data origin authentication, data integrity, and replay services.

III. ARCHITECTURE

The bare PC architecture to support a typical AO that includes IPsec is shown in Fig. 2. The block titled "IPsec Boundary" contains components that relate to IPsec processing. The architecture is flexible i.e., depending on the encapsulating AO and the context in which IPsec is applied, some elements of this architecture can be eliminated or new elements can be added. For example, an IPsec gateway may not include a TCP object unless it needs a Web interface for administrative purposes.

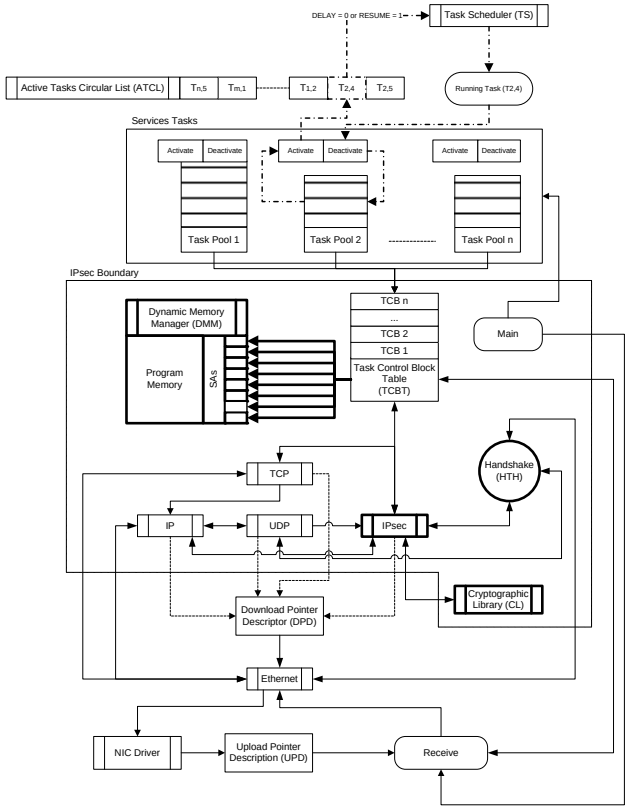

Figure 2. Bare PC architecture with IPsec components

A. Tasks

When a bare PC is booted, the Main task starts. This task runs whenever no other task is running, and it can activate other tasks as needed. For example, the Main task starts the Receive task and a series of tasks for other services.

The Task Scheduler (TS) is a multiplexor mechanism that switches between the Main, the Receive, and other tasks in the Active Tasks Circular List (ATCL). It selects a single task for execution, giving higher priority to the Receive and Main tasks. A task runs as a single thread of execution until it requires an event to continue, in which case it is suspended or delayed. The tasks in the ATCL are eligible for execution if 1) their wait time elapses; or 2) a resume event occurs. Selection of an active task is done in a FIFO manner.

Each task pool is associated with a service (function) that an AO offers. When a service request arrives, an entry from the appropriate task pool is assigned to the request i.e., an available task is activated. The activated task is handled by a corresponding task handler, which provides the logic that an executing task follows until it is completed and returned to the corresponding task pool. For example, when a new IPsec connection request arrives, a task from the handshake task pool is assigned to it. This task follows the execution logic in the Handshake Task Handler (HTH) until the connection is established or timed out.

B. Transition Control Block Table

The Transition Control Block (TCB) table is a data structure that maintains task-related information. An array of fixed (but configurable) size is used to implement the TCB table. Each element of this array represents a task that is in one of the three states: idle, pending execution, or executing. Two circular lists (not shown in Fig. 2) that correspond to the first two states provide pointers to elements of the TCB table. This design offers fast access to retrieve tasks.

C. Upload and Download Pointer Descriptors

The Upload Pointer Descriptor (UPD) and Download Pointer Descriptor (DPD) are buffers that store incoming and outgoing network frames, respectively. They are implemented as circular lists. The device driver for the NIC stores the incoming frames in the UPD and retrieves outgoing frames from the DPD for transmission on the link.

When an Ethernet frame arrives on the link, the Receive task handles it by requesting a new TCB table entry or updating an existing one. This entry is then forwarded to the Ethernet object and packet processing follows the usual path through upper layers except that information may also be conveniently exchanged between layers if needed. Such processing may result in the activation of more tasks according to the type of the packet and its processing needs. Throughout the lifetime of the packet, the incoming frame stays in the UPD while it is being processed so that the frame is easily accessible to any upper layer protocols. The Receive task frees up the UPD entry when packet processing has been completed. Outgoing frames are processed in a similar manner except that they follow a downward path through the upper layers.

D. Network and Security Services

The Ethernet, IP, TCP, and UDP objects provide customary network-related services. If the bare PC were a router/security gateway/firewall, the IP object would include additional functionality, and if NAT were also used, it would manage a

NAT table and make the necessary changes to packet headers. IPsec interacts with IP when handling ESP packets and with UDP during the key exchange phase. The key exchange is handled by the HTH. In a bare PC, this communication is very efficient since all protocols can communicate directly with IPsec as needed.

The Cryptographic Library (CL) provides cryptographic operations. It includes encryption (and decryption) algorithms and hash functions.

E. Memory Management

The Dynamic Memory Manager object allows an AO to directly manage program memory. It provides primitive services for dynamic memory allocation and also provides usage statistics. A buffer in RAM is assigned to each service (or application) that the AO provides. For example, if the AO includes Web server and IPsec services, one chunk of memory is allocated to each service by specifying a base memory address and a length. A memory allocation method is called to dynamically allocate memory to a program variable. Memory de-allocation is also explicitly requested in a similar manner. Memory allocation is based on a heuristic algorithm that combines the best-fit and first-fit algorithms for memory allocation and dynamically expands and shrinks unused memory blocks to minimize memory fragmentation.

IV. DESIGN AND IMPLEMENTATION

The design of IPsec for a bare PC is based on the preceding bare machine architecture and thus differs from a conventional OS-based design. In particular, it does not rely on any OS-based concepts, for example, kernel and user modes, or OS functions. An AO is self-sufficient and it handles the necessary task scheduling, memory management, and concurrency control.

A. Objects

The IPsec implementation on a bare PC supports: 1) set up and management of SAs; 2) IKEv2; 3) data protection using ESP; 4) transport and tunnel modes; 5) data confidentiality service using AES-CBC-128; 6) data integrity verification services using HMAC-SHA1-96; 7) key expansion function based on HMAC-SHA1; 8) authentication service using Pre-Shared Keys (PSKs); and 9) replay attack protection.

The main objects that support IPsec are: 1) IPsec; 2) Handshake; 3) BigInt; 4) SHA1; 5) HMAC-SHA1; and 6) AES. The last four objects are provided by the CL. Fig. 2 shows the IPsec-related objects in bold.

The IPsec object provides all IPsec-related operations including parsing and constructing IKE messages, generating key material, and processing ESP packets. Specific examples are methods to respectively: parse an incoming IKE-SA-INIT message; construct a response to an IKE-SA-INIT message; generate keys for cryptographic algorithms; verify authenticity of an incoming ESP packet; and construct an ESP packet. This object also includes a method that analyzes an incoming IKE-SA message, initializes an SA, and requests activation of a handshake task.

The handshake object contains the code for the HTH that implements the handshake logic for establishing SAs. The handshake object relies on the IPsec object for message processing.

The BigInt object provides arithmetic functions for working with big integers. For example, this object provides a method for modular exponentiation of two big integers. Using a given key, the HMAC-SHA-1 produces a 20-byte Message Authentication Code (MAC) based on a SHA-1 hash value produced by the SHA-1 object. The AES object provides encryption and decryption services based on the AES algorithm.

B. IKE-SA Management

The HTH implements a state machine, shown in Fig. 3, which handles two scenarios (one for an initiator *i* and one for a responder *r*), when establishing an IPsec connection.

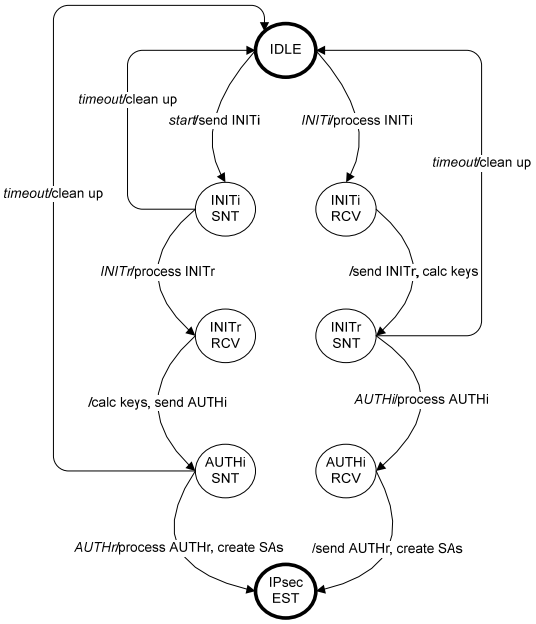

Figure 3. State transition diagram for HTH

Upon arrival of a UDP packet that contains an IKE-SA message, the UDP object forwards the message to the IPsec object, which performs the following operations to process the message:

1. For an IKE-SA-INIT message
   a. requests a new TCB table entry (activate a handshake task),
   b. requests creation of a new buffer for a new SA,
   c. updates the requested TCB table entry,
   d. and, starts the corresponding handshake task.
2. For other IKE-SA messages
   a. requests a lookup for an existing TCB table entry,
   b. updates the entry to resume the task,

c. and, resumes the corresponding handshake task.

In the bare PC IPsec implementation, outgoing messages are directly manipulated in the DPD. For example, in response to the initiator's IKE-SA-INIT message, the responder's HTH acquires a pointer to the buffer (a DPD entry) by sending a request to the Ethernet object. This pointer is passed to the IPsec object which constructs a response and passes the pointer and the length of the message to the UDP object. The latter adds a UDP header to the message and sends the updated pointer to the IP object. Similarly, the IP object adds an IP header to the UDP packet in the buffer, adjusts the pointer, and calls the Ethernet object for further processing.

C. ESP Packets Processing

The bare PC IPsec implementation uses the ESP protocol to protect data packets. IPsec protection is enabled by setting a global flag, ACTIVE_IPSEC. Fig. 4 shows the inter-layer communication for processing data packets.

Incoming ESP packets are encapsulated in IP datagrams. The IP object recognizes such datagrams by inspecting the protocol ID field in the datagram's header and removing the IP header from the datagram before forwarding the encapsulated ESP packet to the IPsec object. To process an incoming ESP packet, the IPsec object:

1. determines the security parameters of the packet by performing an SA lookup,
2. ensures that the packet has not been replayed,
3. checks packet integrity,
4. decrypts the packet and verifies success,
5. and, removes the ESP-related data and passes the encapsulated plaintext content directly to the object corresponding to the Next Protocol field in the ESP packet.

If any of the above operations fails, the packet is dropped.

For outgoing packets, when IPsec is enabled, the IP, TCP, UDP, or ICMP object forwards the packet directly to the IPsec object for processing. To process an outgoing packet the IPsec object:

1. determines if the packet is subject to IPsec protection
2. performs the following operations when an SA exists:
   a. pads the packet (if needed),
   b. adds the ESP header to the packet,
   c. encrypts the packet,
   d. adds the ESP trailer to the packet,
   e. and, if the SA is for tunnel mode, adds an inner IP header to the packet;
3. and, passes the packet to the IP object.

For an SA in tunnel mode, the bare PC may be a client that is tunneling its own packets, or a router/security gateway that is tunneling a received IP datagram.

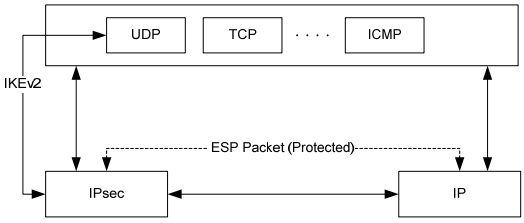

Figure 4. IPsec inter-layer communication

D. Bare PC Optimizations

In a bare PC, IPsec is integrated within the implementation of an AO such as a Web or email server, a VoIP or email client, or a security gateway. However, since there is no OS, all IPsec processing is self-contained and can be optimized for a given AO. Furthermore, since the AO intertwines IPsec and other protocols, inter-layer communications are very efficient. For example, a security application (or other layers) may directly access the Ethernet buffer for sending IP datagrams.

As shown in Fig. 2 and Fig. 4, the bare PC architecture allows all protocol layers in an AO to directly access and manipulate IPsec header information as needed. This is in contrast to standard OS-based implementations of IPsec, in which IPsec is implemented in the OS kernel and key exchange and management is controlled from user-space. In such systems, IPsec may be integrated within the IP code, exist as a separate layer between IP and the link layer (bump-in-the-stack), or be incorporated in a hardware device between a router and the network (bump-in-the-wire). In a bare PC, IPsec and other protocol objects are part of the same AO. Thus, IPsec-related processing is more efficient (since no system calls are needed and IP as well as higher layer protocols communicate directly with IPsec).

IPsec on a bare PC also benefits from the elimination of data copying from application to Ethernet buffers. As noted before, the AO has direct access to the Ethernet buffer and data in a packet is not duplicated. Thus, IPsec objects can manipulate headers and access the data for authentication and encryption with minimum delay. As discussed in IKE-SA Management section, the key exchange message processing is also streamlined. Furthermore, the intertwining of protocols within the bare PC architecture enables information to be stored during SA establishment and recovered during subsequent SA lookup with less overhead than in an OS-based system.

V. RESULTS

This section contains a description of experiments conducted using IPsec on a bare PC and the results obtained. Results for OS-based systems (with unessential services disabled) are included when possible as a means for comparison and also for identifying areas of improvement in future bare PC implementations of IPsec.

A. Experimental Setup

The small test LAN used for experiments consists of five PCs connected by a 100 Mbps Cisco FastHub 400 series as shown in Fig. 5. The hardware specifications for the PCs are given in Table 1.

A and B are PCs running a Web server AO that supports IPsec. C and D are PCs that run StrongS/WAN 4.2.12 on Linux (Fedora 10 Kernel Version 2.6.27.5-117-fc10.i686 Kernel Version: 2.6.27.5-117-fc10.i686 and Ubuntu 8.1 Kernel Version 2.6.27-7-server, respectively), and E is a PC running Wireshark Version 1.2.1 [20] on Windows XP to monitor network traffic.

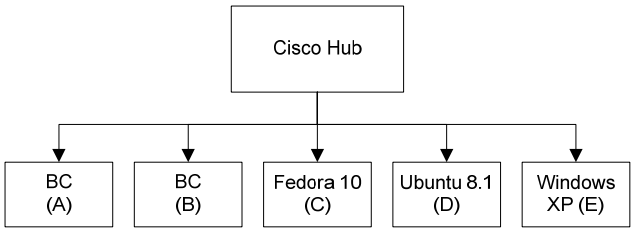

Figure 5. Test LAN

TABLE I. HARDWARE SPECIFICATIONS FOR THE PCS IN THE TEST LAN

Computer	Processor	RAM	NIC
A	2.4 MHz	512 MB	3COM 905CX 10/100
B	2.4 MHz	512 MB	3COM 905CX 10/100
C	3.2 MHz	1024 MB	Intel PRO/1000 MT
D	2.4 MHz	512 MB	3COM 905CX 10/100
E	2.4 MHz	1024 MB	Intel PRO/1000 MT

B. Performance Metrics

To measure the IPsec overhead on a bare PC, we use these metrics: 1) internal (processing) times to measure the computing overhead for ESP processing; 2) ping response times to measure the delay in receiving a response when packets are protected by ESP; 3) HTTP response and connection times to measure the impact of ESP protection on Web traffic; 4) IKEv2 message processing time to measure the impacts of selecting different modular exponential groups (ModP) for DH exchange.

C. Internal ESP Timing

To determine the ESP packet processing time, we modified the AO on machine A (bare PC). The modification included creation of checkpoints for capturing processor ticks. The processor ticks corresponded to the various ESP operations associated with incoming and outgoing packets. These operations are shown in Table 2.

To conduct the experiment, we sent ICMP packets, with packet (payload) sizes varying from 64 to 1024 bytes in 64-byte increments, from machine C (Linux Fedora) to machine A (bare PC). The experiment was repeated 5 times for each data size and the reported results are averages (the variances were negligible and are omitted). The time to process ICMP packets were not included in these internal timings.

TABLE II. OPERATIONS ASSOCIATED WITH PROCESSING OF INCOMING AND OUTGOING ESP PACKETS

Incoming Traffic	
Service	**Operation**
IPsec	SA lookup Anti-replay window Data integrity verification Packet decryption Decryption verification
Data integrity	MAC calculation MAC verification
Data confidentiality	Packet decryption Decryption verification
Outgoing Traffic	
Service	**Operation**
IPsec	SA lookup Padding Encryption MAC calculation
Data confidentiality	Data encryption
Data integrity	MAC calculation

Fig. 6 shows internal processing times for incoming ESP packets with transport mode and with tunnel mode including MAC calculation and verification (HMAC-SHA-1), data decryption and verification (AES-CBC-128), and total ESP processing.

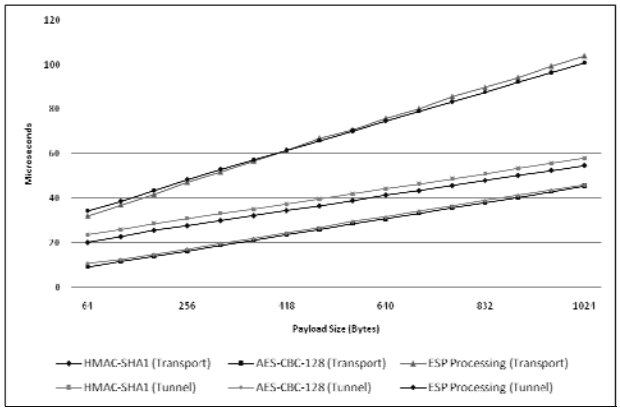

Figure 6. ESP internal timing for incoming packets

As expected, an approximately linear relationship between packet size and processing time is seen for all three measures (regardless of whether transport or tunnel mode is used). Data integrity verification takes more time than decryption but the difference is about the same for all packet sizes. The difference between total ESP processing time and the time for decryption and integrity verification is primarily due to overhead associated with replay protection and decryption verification, which does not vary significantly with packet size. The difference in time for integrity verification between tunnel and transport mode is essentially constant for all packet sizes, with tunnel mode being more expensive as expected. The same is true for decryption. However, the results for ESP packet processing show that as packet sizes increase, the difference between tunnel mode and transport mode becomes less and

transport mode becomes slightly more expensive than tunnel mode for packet sizes greater than about 640 bytes. While it is clear that tunneling overhead is dominated by processing overhead for larger packets, we were not able to determine why transport mode is eventually more expensive.

Fig. 7 shows the processing times for outgoing ESP packets. The correlation between processing time and packet size for MAC calculation (HMAC-SHA-1), data encryption (AES-CBC-128), and all ESP operations with tunnel and transport mode are linear (similar to the case of incoming packets) with tunnel mode being slightly more costly. The processing times for outgoing ESP packets is approximately 6% less than those for incoming packets in either mode. This difference is due to the additional processing for replay protection and integrity and decryption verification on incoming packets.

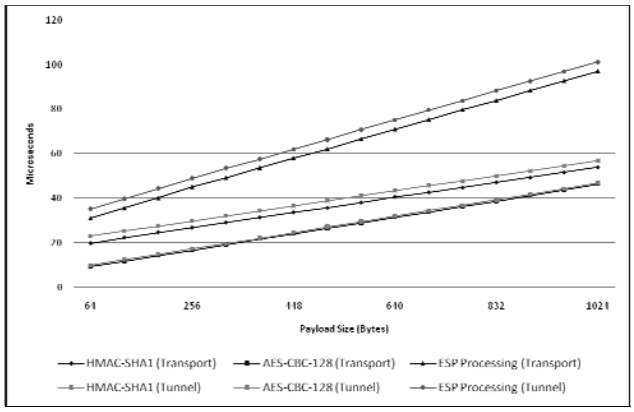

Figure 7. ESP processing time for outgoing packets

D. ICMP Response Time

To measure average ICMP response times, 100 ICMP (ping) packets of sizes varying from 64 bytes to 1024 bytes in 64 byte increments were sent from a) machine B to machine A; b) machine C to machine A; and c) machine C to machine D. Average response times were measured in three cases: 1) without IPsec protection; 2) with IPsec in transport mode; 3) with IPsec in tunnel mode.

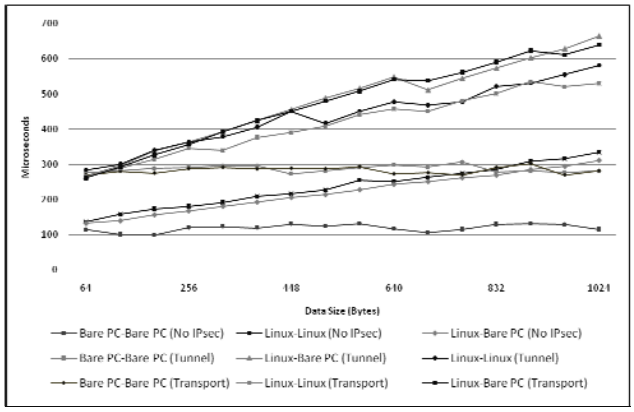

Figure 8. ICMP (ping) response time

The results shown in Fig. 8 indicate, as expected, that ICMP response times between bare PCs with or without IPsec are least and approximately constant for all packet sizes, while ICMP response times between Linux PCs are larger and increase with packet size. The difference between ICMP response time for transport and tunnel mode is negligible, although response time for transport mode is not always smaller. Surprisingly, highest ICMP response times are between a Linux PC and bare PC (again response time increases with packet size and there is a negligible difference between response time for tunnel and transport mode). In general, ICMP response times in all cases increased by over 100% with IPsec compared to no IPsec for both tunnel and transport mode. Again, it is not clear why the response time for certain packet sizes with transport mode is slightly greater than with tunnel mode.

E. HTTP Response and Connection Times

To measure the impact of ESP on HTTP traffic we used http-load [21] running on machine C to measure the response time and connection time for a bare PC Web server on machine A and an Apache 2.2 Web server on machine D when traffic was protected by ESP. The size of the static Web page used was 3.5 KB. Tunnel and transport mode were both tested since either could be used for secure communication between a bare PC Web server and a remote client (in tunnel mode, the bare PC server also functions as a security gateway). The results are shown in Fig. 9 and Fig. 10. Per the obtained results, with or without IPsec, the bare PC Web server's response time is better, and its connection time (which is controlled by the client rather than the server) is approximately the same as the connection time for the Apache Web server. The performance difference between transport and tunnel mode is negligible.

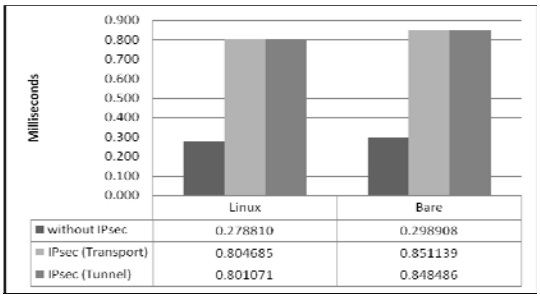

Figure 9. HTTP connection time

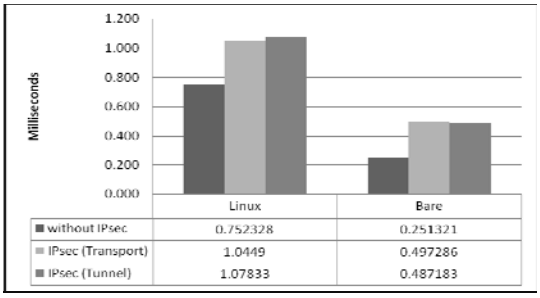

Figure 10. HTTP response time

F. *IKEv2 Message Processing Time*

We also conducted tests to compare processing time on a bare PC for the two types of IKEv2 messages. The results are shown in Fig. 11. While it can be seen that IKE-SA-INIT messages have larger processing times than IKE-SA-AUTH messages as expected, the difference between processing times for the two message types becomes significant as the cost of the DH computation increases with larger key sizes.

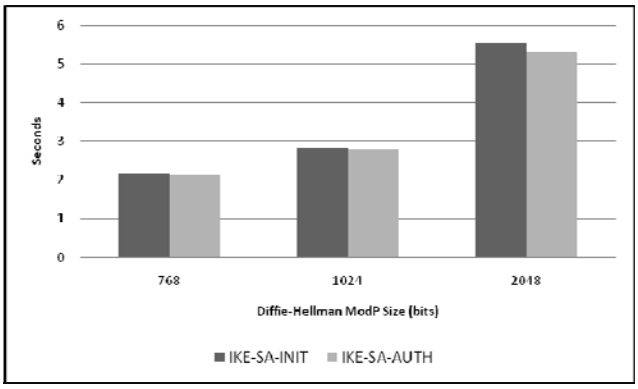

Figure 11. IKEv2 message processing time

VI. CONCLUSION

We described the design and implementation of IPsec on a bare PC with no OS. The bare PC architecture enables IPsec to be efficiently implemented by intertwining protocols and application, minimizing inter-layer communication cost, and eliminating data copying between application and network buffers. Our experimental results indicate that IPsec performance on a bare PC as measured by ESP processing time, ICMP (ping) response time, and HTTP response time for a Web server application is better than IPsec performance on Linux. Our results can serve as a baseline for designing future versions of IPsec for high-performance or high-security bare PC applications, and devices such as security gateways running on bare machines.

REFERENCES

[1] R. K. Karne, K. V. Jaganathan, and T. Ahmed, “DOSC: Dispersed Operating System Computing”, OOPSLA ’05, 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, Onward Track, ACM, San Diego, CA, October 2005, pp. 55-61.

[2] R. K. Karne, “Application-Oriented Object Architecture: A Revolutionary Approach,” 6th International Conference, HPC Asia 2002, Bangalore, Karnataka, India, December 2002.

[3] L. He, R. Karne, and A. Wijesinha, "The Design and Performance of a Bare PC Web Server", International Journal of Computers and Their Applications, vol. 15, no. 2, pp. 100 - 112, June 2008.

[4] G. Ford, R. Karne, A. Wijesinha, and P. Appiah-Kubi, “The Design and Implementation of a bare PC Email Server,” 33rd Annual IEEE International Computer Software and Applications Conference (CompSAC 2009), pp. 480-485, Seattle, WA, July 2009.

[5] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, “A Peer-to-Peer bare PC VoIP Application,” Proceedings of the IEEE Consumer and Communications and Networking Conference, IEEE Press, Las Vegas, NV, 2007.

[6] R. K. Karne, K. V. Jaganathan, and T. Ahmed, “How to Run C++ Applications on a Bare PC?” Proceedings of the 6th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, IEEE Computer Society, Washington DC, 2005, pp. 50-55.

[7] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," December 2005, RFC 4301.

[8] S. Kent, "IP Authentication Header," December 2005, RFC 4302.

[9] S. Kent, "IP Encapsulating Security Payload (ESP)," December 2005, RFC 4303.

[10] D. Harkins, D. Carrel, "Internet Key Exchange," 1998, RFC 2409.

[11] E. C. Kaufman, "Internet Key Exchange (IKEv2) Protocol," 2005, RFC 4306.

[12] A. D. Keromytis, J. Ioannidis, and J. M. Smith, “Implementing IPsec,” Proceedings of Global Internet (GlobeCom) 1997, pp. 1948-1952, November 1997.

[13] S. Miltchev, S. Ioannidis, and A. Keromytis, “A study of the relative costs of network security protocols,” USENIX, 2002.

[14] L. Volker, M. Scholler, and M. Zitterbart, "Introducing QoS mechanisms into the IPsec packet processing," 32nd IEEE Conference on Local Computer Networks (LCN 2007), pp. 360-367, 2007.

[15] G. C. Hadjichristophi, N. J. Davis IV, and S. F. Midkiff, “IPSec overhead in wireline and wireless networks for web and email applications,” 22nd IEEE International Performance, Computing, and Communications Conference, Phoenix, Arizona, April 2003.

[16] C. Shue, Y. Shin, M. Gupta, and J. Y. Choi, “Analysis of IPSec overheads for VPN servers,” IEEE ICNP’s NPSec Workshop, 2005.

[17] C. A. Shue, M. Gupta, and S. A. Myers, "IPSec: Performance Analysis and Enhancements," IEEE, Indianapolis, 2007, pp. 1527-1532.

[18] H. Niedermayer, A. Klenk, and G. Carle, “The Networking Perspective of Security Performance - a Measurement Study,” MMB 2006, pp. 119-136, Nurnberg,Germany, March 2006.

[19] A. Steffen. StrongSwan. [Online]. http://www.strongswan.org

[20] (2009, June) WIRESHARK. [Online]. http://www.wireshark.org

[21] ACME Laboratories, http-load. [Online]. http://acme.com/software/http_load.