On Implementing Security at the Transport Layer

Swaminathan Pichumani
Juniper Networks
spichumani@juniper.net

Sneha Kumar Kasera
School of Computing, University of Utah
kasera@cs.utah.edu

Abstract— **We design a framework that implements security at the TCP layer to meet the necessity for a practical and truly end-to-end security solution. We call our framework TCPsec. TCPsec is a security extension to TCP and implemented in the kernel. Applications may use TCPsec through regular TCP sockets by setting special socket options. TCPsec uses a Secure Socket Layer (SSL)-like handshake to set up a secure session. It is interoperable with Network Address Translators. We implement TCPsec in the FreeBSD 4.7 kernel and evaluate its performance. Our implementation and evaluation show that TCPsec incurs only a modest overhead as compared to TCP and performs competitively with SSL. We also provide a formal verification of our protocol state machine.**

I. Introduction

Due to the dependence of many important and sometimes critical applications on networked computers, network security has become an indispensable requirement. Today, a broad spectrum of security solutions is available for the Internet at different network protocol layers. Layering provides one of the primary dimensions used to classify protocols and defining their functionality and limitations. Security protocols can be designed to protect data at different layers of the protocol stack starting from the application layer to the link layer. Secure Socket Layer (SSL) [1] or Transport Layer Security(TLS) [2][1] and Internet Protocol Security (IPsec) [3] are amongst the most popular and widely-deployed security protocols. Both of these protocols have important inherent shortcomings due to their location in the network protocol stack. Although SSL is called a transport layer protocol, it actually runs as a user-level library above TCP. Since SSL operates at a layer above TCP, it cannot authenticate the TCP header. Unprotected TCP header opens room for several denial-of-service attacks due to the possibility of bogus TCP packet injection [4]. These include the data injection attack, the blind reset attack using the RST bit, and the blind reset attack using the SYN bit.

In order to understand the seriousness of the data injection attacks on SSL, we conduct experiments where an 802.11 (wireless LAN) web client communicates with a web server in the wired network. In our experiments, we use SSL (https) sessions between the client and the server. We construct an attacker program that runs on an attacker node in the same wireless LAN as the web client. This attacker program first sniffs TCP packets and then spoofs the client IP address and injects its own fake TCP packets with appropriate sequence numbers and checksums. These fake packets are accepted by the TCP layer but not at the SSL layer by the web server, causing the SSL layer to discard established connections. We run ten SSL sessions each for a specific file size. Figure 1 shows how the percent of successful attacks varies with the https file transfer size. We observe that for file sizes beyond 50 Kbytes, the chance of successfully attacking the https web sessions is very high. This is because for 50 Kbytes or larger file sizes, the attacker program has enough time to successfully create a fake TCP packet with a valid sequence number. Our experiments prove that it is very easy to attack SSL in wireless networks and raise a serious concern about its use[2].

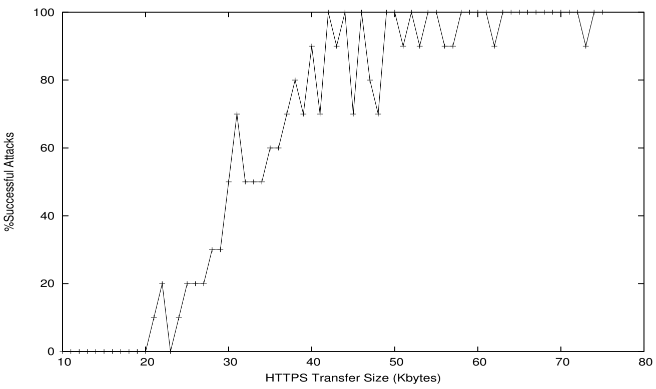


Fig. 1. Percent of successful attacks vs file size.

On the other hand, IPsec provides end-to-end security at the network layer allowing all applications running on top of IP to communicate securely. In the commonly deployed Encapsulating Security Payload (ESP) mode [5], IPsec encrypts and/or authenticates the entire transport layer payload. However, IPsec suffers from two limitations. First, IPsec exchanges a large number of messages for establishing a security association among two end-points (the numbers are provided in Section V-E). Hence it suffers from a high overhead for short to moderate-lived associations among end-points. IPsec is more suitable for long lived associations among two end-points (e.g., a secure virtual private network tunnel setup) so that the initial setup cost is amortized over a reasonably long period of communication and possibly over multiple TCP connections between the end-points. Second, its design has serious implications when firewalls are used to filter unwanted traffic between end-points. When the entire payload is only

[1]Except for a few differences, SSL and TLS protocols are very similar in their design and implementation. Hence, in our work we use SSL as the reference protocol.

[2]Additional link layer security could help thwart these attacks but then it must be implemented on every vulnerable link in the end-to-end path.

authenticated and not encrypted, firewalls can effectively perform their filtering functions but data privacy is compromised. When the entire payload is encrypted, the firewalls cannot view the transport protocol headers that it requires to perform its filtering functions. This problem has led to many organizations deploying split security using IPsec instead of complete end-to-end security. Split security is implemented by terminating IPsec connections at gateways or firewalls. the employee to secure all her transactions on the public Internet. We envision that in the future, users, especially wireless users will be able to request firewall services from service providers. The split security model where all the data is visible to the firewalls will be unacceptable to these users. It is possible to use unencrypted but authenticated fields in the IPsec headers [3] to implement some firewall rules but such solutions are very limited in scope. It is also possible to use SSL over split IPsec but that would be inefficient due to overheads of both these approaches. Recently, there have been a few proposals to make IPsec more *firewall-friendly* [6]–[8]. However, these approaches still suffer from the message complexity of IPsec.

In this paper, we design a framework that efficiently implements security at the TCP layer to meet the necessity for a practical, and truly end-to-end security solution. *Our framework, that we call TCPsec, provides a light-weight and firewall-friendly alternative to SSL and IPsec.* There are several advantages of building security at the TCP layer. First, TCP already has a handshake mechanism for connection setup. We extend this handshake mechanism to allow for key exchange and security parameter negotiation similar to the one implemented in SSL. Unlike IPsec [9] no separate key exchange is needed. Second, when an application chooses to use TCPsec security, the TCP headers/payload could be flexibly encrypted on a per connection basis. Third and importantly, security at TCP leaves the control with end-points only and thus provides end-to-end security. Applications may use TCPsec through regular TCP sockets by setting special socket options. TCPsec is backward compatible with TCP. In the absence of TCPsec, regular TCP is used for communication. TCPsec is also interoperable with Network Address Translators (NATs) [10]. The availability of TCPsec at the peer node and the presence of NATs are detected during the handshake.

The use of TCPsec will also require both application and kernel-level changes. In order to address this concern, we explore two approaches - one that uses application layer proxies to avoid any changes in the applications and another that uses a kernel sandboxing framework, Self-spreading Transport (STP) [11], to ease kernel upgrading. The STP framework allows us to ship, compile, and load TCPsec at a remote host. We implement TCPsec in the FreeBSD 4.7 kernel and evaluate its performance. Our implementation and evaluation show that TCPsec incurs only a modest overhead as compared to TCP and performs competitively with SSL.

The rest of the paper is organized as follows: We describe the design of our TCPsec framework in Section II. In Section III, we provide some implementation details of TCPsec in FreeBSD. We discuss the important deployment problems associated with TCPsec and explain our solutions in Section IV. In Section V, we evaluate our TCPsec implementation. In Section VI, we formally verify the correctness of the enhancements made to the TCP protocol. In Section VII, we survey the related work and contrast it with TCPsec. Finally, we conclude the paper in Section VIII.

II. Protocol Design

This section outlines the key design features of TCPsec protocol. The primary factors that influence the design of TCPsec are:
Security and Efficiency: The primary goal of TCPsec is to provide security to application data in an efficient manner with low overheads.
Use of well-established security design principles: We aim to use well-established security design principles from the widely deployed protocols such as SSL and IPsec.
Deployment issues: TCPsec's design is also influenced by the requirement to support easy deployment of the protocol and interoperability with regular TCP.

TCPsec provides the three basic security functions: authentication, message integrity, and confidentiality. Like most other security protocols [1], [3], it implements these functions in two different phases: the handshake phase during which the nodes authenticate the identity of each other and establish a secure session by exchanging parameters, and the data exchange phase during which the application data is securely transmitted using the keys negotiated in the handshake phase. The secure session ends when the TCP connection is closed. We now describe these two phases.

A. Handshake phase

We extend TCP's three-way handshake to negotiate and carry the security parameters. The options field in the TCP header provides the first choice for transmitting the security parameters. However, TCP restricts the maximum size of an option to 40 bytes, while the parameters exchanged during the handshake, especially the certificates, can be much larger than the size allowed by TCP options. Hence, we introduce TCPsec header and TCPsec options fields to carry the TCPsec parameters. An entire TCPsec packet is shown in Figure 4. The TCPsec options are similar in format to the TCP options except for the maximum length allowed in each option. The TCPsec options are preceeded by a TCPsec header. The TCPsec header includes the *data_offset*, which gives the offset to user data from the TCPsec header in 32-bit words, and the *frag_id*, which gives the sequence number of the current handshake fragment, as explained in Section II-A.1. The sequence of TCPsec messages, along with their parameters are explained below.
(1) In the first SYN packet to the server, the client informs the availability of TCPsec by setting a special SEC flag in the TCP header[3]. The TCPsec options sent in the SYN message are: (i)

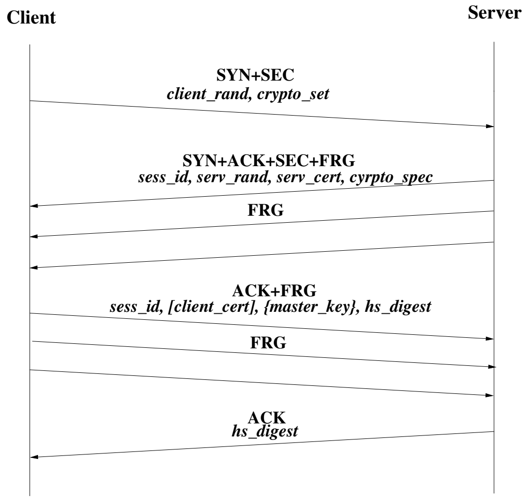
[3]When either of the two sides does not wish to use TCPsec, the handshake falls back to that of regular TCP. The details are provided in [12].

client random number, used in the generation of session keys, (ii) *crypto_set*, which consists of crypto algorithms supported by the client.
(2) In its reply, the SYN+ACK message, the server includes the following parameters: (i) *session_id* to uniquely identify a session, (ii) *server_certificate* to prove its identity to the client, (iii) *server_random number* used in the key generation, and (iv) *crypto_spec*, which contains the set of crypto algorithms chosen by the server from the crypto set provided by the client.
(3) On receiving the SYN+ACK message, the client authenticates the server using the server's certificate. The client generates a master key for the new session and encrypts it using the server's public key. It sends the ACK message with the following parameters: (i) *session_id*, (ii) an optional *client_certificate* to authenticate itself to the server, (iii) encrypted *master_key* and (iv) handshake message digest. The handshake message digest is a keyed digest of all the messages received and sent by the client including the current ACK message. This digest is sent to the server, which in turn verifies the received digest by comparing it with its own digest of all the handshake messages sent and received.
(4) The fourth and additional message added to the TCP handshake is an ACK message from the server to acknowledge the receipt of the parameters sent by the client to the server in the previous message. In addition, this message also carries a handshake digest from the server to the client. This digest prevents any impersonation of the server, since the right digest can be generated only by the server.

1) Fragmentation: Some of the TCPsec options, especially those containing the certificates, could make the TCP segment size exceed the maximum allowed to prevent IP fragmentation [13]. This introduces the necessity for fragmenting the handshake messages that carry certificates at the TCPsec layer. In order to facilitate reconstruction of fragments at the TCPsec receiver, we use two fields - one from the TCP header flags and another from the TCPsec header.
(1) *frag_id*, which is a part of the TCPsec header, is used to correctly order the received fragments. The *frag_id* indicates the sequence number (not the TCP sequence number) of a fragment, starting from 0 for the first fragment, 1 for the second fragment, and so on.
(2) In order to identify the last fragments of a particular handshake message, we introduce an additional TCP header flag called the *FRG* flag. The sender sets the *FRG* flag to indicate the presence of more fragments. The *FRG* is reset in the final fragment of a handshake message.

Buffering of fragments, for reconstruction of handshake messages, opens room for a potential denial-of-service attack: an attacker can send a large number of bogus fragments and easily overflow the victim's memory. In order to prevent this attack, we restrict the maximum number of fragments. An handshake flow with fragmentation is shown in Figure 2.

2) TCP Finite State Machine: We make two important changes to the TCP three-way handshake to set up a TCPsec session. First, we add an additional message to the handshake phase and second, we allow fragmentation of handshake messages. These changes made to the TCP handshake result in changes to the TCP Finite State Machine (FSM). The new TCP FSM is shown in Figure 3. The client and the server start



Client
Server
SYN+SEC
client_rand, crypto_set
SYN+ACK+SEC+FRG
sess_id, serv_rand, serv_cert, cyrpto_spec
FRG
ACK+FRG
sess_id, [client_cert], {master_key}, hs_digest
FRG
ACK
hs_digest

Fig. 2. TCPsec handshake messages.

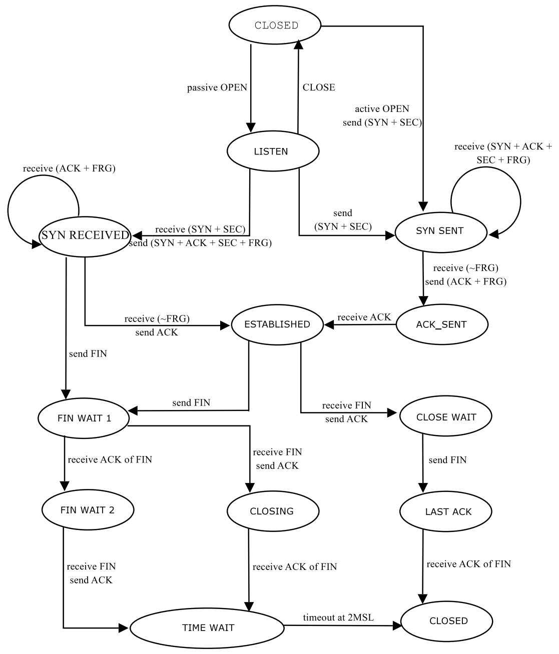


Fig. 3. Modified TCP finite state machine.

from the CLOSED state. The server goes to the LISTEN state on a passive open and starts listening for connection requests.
(1) On an active open, the client sends a SYN and enters the SYN_SENT state.
(2) The server (in the LISTEN state) on receiving the SYN, replies with the SYN+ACK and enters the SYN_RECEIVED state. The server's reply can potentially be fragmented across multiple segments depending on the total option size. The server sends all the fragments at once before entering the SYN_RECEIVED state.

(3) The client (in the SYN_SENT state) receives the fragments from the server and stays in the SYN_SENT state as long as there are more fragments to receive. On receiving the last fragment in the SYN_SENT state, the client sends an ACK. Similar to the SYN+ACK from the server, the client's ACK can be fragmented. The client sends all the fragments and moves to a new state called ACK_SENT. In the ACK_SENT state, the client waits for an acknowledgment from the server for the security parameters it sent in its ACK.
(4) The server receives the ACK fragments from the client in the SYN_RECEIVED state. On receiving the final fragment, the server responds with an ACK and moves to the ESTABLISHED state.
(5) The client receives the final ACK from the server and moves to the ESTABLISHED state.

This completes the handshake process in TCPsec and a secure session is setup. The connection teardown sequence remain the same except that these messages are cryptographically protected in TCPsec.

B. Data exchange phase

The user data is securely transmitted during the data exchange phase of a session. Integrity protection is provided by generating a keyed digest of the data using an agreed upon algorithm and confidentiality is provided by encrypting the data using a symmetric key algorithm. In order to provide encryption and message authentication, a set of session parameters are maintained. These session parameters are initialized during the handshake phase and used to secure data exchange.

Similar to SSL and IPsec, security associations in TCPsec are unidirectional. Hence, TCPsec uses separate read and write keys for encryption and message authentication operations in each direction of communication. A set of four keys is generated from the master key for each session. Two of the keys are used for encryption (read and write encryption keys) and two other keys are generated for message integrity (read and write authentication keys).

Encryption and integrity protection are independent operations and can be done in any order. However, [14] explains vulnerabilities associated with weak algorithms when encryption is performed after authentication. Hence in TCPsec, we generate the digest after the encryption process. This method also allows us to authenticate a message before decrypting it thereby avoiding the overhead of decrypting bogus messages.

In TCPsec, we allow a flexible encryption of the TCP segment (see Figure 4). With our flexible encryption scheme, although the TCP payload (user data) is always encrypted, an application can decide to encrypt only a part of the TCP header. It is mandatory that the port numbers of the TCP header are left unencrypted for identifying the TCP data structures and the destination application. Other than the port numbers, an application can choose to encrypt any part of the TCP header. By default, the header is left unencrypted and the payload alone is encrypted. The other details of the data handshake phase including the choice of explicit versus explicit initialization vectors, and the placement of

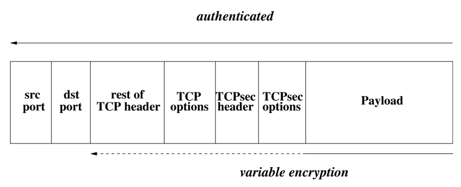


Fig. 4. TCPsec packet format with the TCPsec header and options.

the message authentication code are described in the detailed version of this paper [12].

C. NAT interoperability

Network Address Translation (NAT) devices are widely used today as an external interface to private Intranets, including enterprise networks and home networks, to save IP addresses. To provide NAT interoperability, we include a new option in the TCPsec handshake called the *NAT* option. In the first SYN message of TCPsec, the client sends the *NAT* option containing the port numbers and the source IP address it uses for the connection as part of the TCPsec options. In the presence of a NAT along the path to the server, the port numbers and the IP address are modified. The TCPsec server can verify this by comparing the values it receives in the packet with those in the option. When a NAT is detected, the server includes the *NAT* option in its reply with the source address and port number it uses. After the exchange of *NAT* option by the client and the server, the port numbers and the IP address received in the options field replace the values present in the TCP header for message digest calculation and verification. This scheme is similar to the process described in [15] for IPsec interoperability with NATs.

III. IMPLEMENTATION

We implement the TCPsec protocol in the FreeBSD kernel version 4.7. Most of our code is a part of TCP's input and output routines. For the cryptographic functionality, we port an existing crypto-library [16] to the kernel. This crypto code forms a separate kernel module that must be loaded in the kernel when TCPsec is enabled.

A. Socket library

We enhance the socket library to allow applications to enable TCPsec and configure security policies on a per-socket basis. Applications can control the parameters through the use of the *setsockopt()* function call. Table I outlines the different socket options available for applications. However,

Socket Option	Purpose	Default value
TCPSEC	TCPsec socket	NOT_ENABLED
ENC_ALGO	encryption algorithm	3DES
MD_ALGO	message digest algorithm	MD5
ACCPT_TCP	accept non-TCPsec connections (server)	DONT_ACCPT

TABLE I
NEW SOCKET OPTIONS TO SET TCPSEC PARAMETERS.

an application's choice can be overridden by enabling global security policies. Global security policies are set by a system administrator. The security policy defines the choices of security parameters such as encryption and authentication algorithms and enforces all applications to use them. When the global parameters are set, individual choices of applications are ignored.

B. Kernel data structures

The TCPsec session parameters are stored in a data structure called the *TCPsec control block*, which can be accessed from the TCP control block of a TCP connection. A TCPsec control block is allocated for each secure connection. However, to prevent the SYN flood attacks on the server side, as in the case of the TCP control block, the TCPsec control block is allocated only after the client's acknowledgment (third message in the handshake phase) is received at the server. In FreeBSD, the *syncache* [17] data structure is used to store information about initiated connection requests until a control block is allocated. The syncache is implemented as a hash table and a maximum limit is imposed on the total memory that can be used by the syncache. In TCPsec, we use the syncache to also store the security parameters exchanged before the TCPsec control block is created for the connection. This increases the size of the syncache data structure and hence the maximum size allocated for the syncache must be increased to support the same number of simultaneous connection attempts.

C. Handshake Digest

One of the security parameters exchanged during the handshake is the digest of all handshake messages as discussed in Section II-A. This digest is computed over all the handshake messages and the authentication keys. However, the authentication keys are not available until the master secret is exchanged. In order to avoid the overhead of saving the fragments exchanged during the handshake, which in turn increases the risk of a denial-of-service flooding attack, we compute the digest incrementally similar to SSL. The handshake digest is calculated as:

$$MD5(MD5(handshakemessages) + auth\_key + const)$$

where, $const$ = 0x1122 for server and $const$ = 0x2233 for client. These constants are the same as those used in SSL's handshake digest calculation. When a fragment is received or sent, the digest's context is updated. After the keys are generated, the final handshake digest is generated as an MD5 digest of the authentication key concatenated to the digest of the messages. This method is a modified version of the HMAC [18] digest calculation.

IV. Deployment

Ease of deployment significantly influences the success of a protocol. While TCPsec overcomes the drawbacks of SSL and IPsec, it has its associated deployment difficulties that discourage the deployment of the framework. In this section, we discuss these issues and briefly describe our solutions to make TCPsec deployment feasible.

TCPsec's primary deployment problems are two fold. First, applications must be changed to enable TCPsec. These changes involve setting the appropriate socket options defined in Section III (unless system wide security is enforced for all applications). Second, kernel upgrade and recompilation are required at both the communicating nodes since TCPsec is part of the kernel. In contrast, SSL being an application layer protocol, does not require kernel upgrade and IPsec does not require application changes. In order to support TCPsec deployment, we implement the following two solutions.

A. Application Proxies

To allow existing applications to use TCPsec without requiring modifications, we develop application proxies, one each for the client and the server. These proxies run as daemons co-located on the same machine with the corresponding server or client application[4]. A proxy communicates with the unchanged application using TCP. However, the client and server proxies communicate with each other using TCPsec. We derive the use of application proxies from S-Tunnel [19], which uses proxies to allow applications to use SSL without requiring any modification in the applications themselves. The details related to the use of application level proxies for deploying TCPsec are available from [12].

B. STP Sandboxing Mechanism

Since TCPsec is a part of the kernel, deploying TCPsec requires kernel changes and recompilation. Like most other TCP versions, TCPsec must be deployed at both the communicating nodes to provide security. To simplify the kernel deployment of TCPsec, we use the Self-spreading Transport Protocol [11] sandboxing mechanism.

1) STP overview: STP provides a system in which the communicating end hosts use untrusted mobile code to remotely upgrade each other with the transport protocols that they use to communicate. With the use of STP framework, we can avoid kernel recompilation since modules can be dynamically loaded using STP's trusted compiler and loader. Figure 5 shows the important architectural components of STP. The central component of the framework is the STP sandbox, which provides a restricted and resource safe environment for mobile transport protocols. The sandbox implements a set of APIs, which provide transports constrained access to other layers in the system. Similarly the network layer and the socket layer interact with the transport protocols through the STP sandbox. Finally, the applications interact with the STP sandbox and the transports through existing socket interface and by setting special socket options. The policy manager daemon is used to resolve conflicts during connection establishment. These policy managers select a protocol to be used for a connection. When a new protocol code is shipped from one node to another, the trusted tool compiler generates a module on the remote node. This kernel module is loaded by a trusted loader.

[4]End-to-end security is compromised when the proxies are run on a different machine than the application, similar to the split IPsec model.

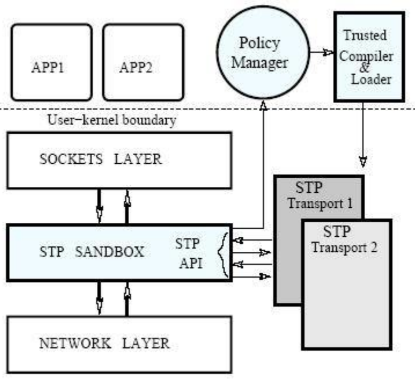


Fig. 5. Architecture of the STP framework.

2) STP handshake: STP enhances TCP's 3-way handshake to indicate each end-host's choice of the transport protocol in the TCP option. This handshake process also initiates out-of-band shipping of transport code when the selected transport is absent in a host. Consider the handshake scenario shown in Figure 6. Let node A use TCPsec protocol and node B use regular TCP. When a TCP connection is initiated by A

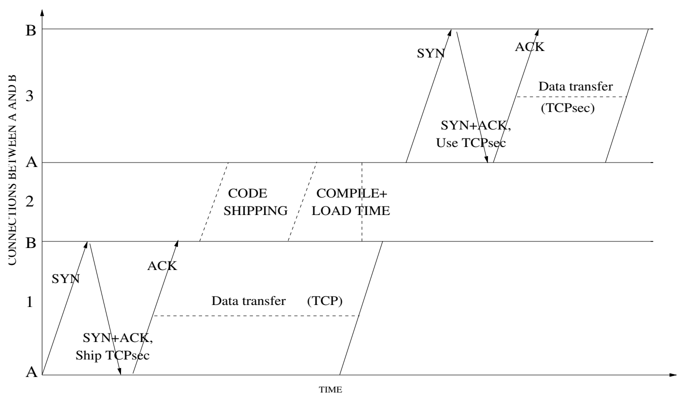


Fig. 6. An STP handshake example.

(connection 1), it also advertises its intent to use TCPsec as the transport protocol. Since node B does not have TCPsec, it replies back requesting node A to ship the code for TCPsec. The initiated connection falls back to the default TCP protocol since TCPsec is not available in node B. However, node A initiates an out-of-band shipping of TCPsec code to node B (connection 2). Using STP's trusted compiler and loader, TCPsec is deployed at node B. Once the deployment is complete, a future connection between node A and node B (connection 3) can use TCPsec.

3) TCPsec on the STP framework: We implement TCPsec on the STP framework. Our implementation involves porting TCPsec's hard-wired implementation from the kernel to an STP-based transport. STP transports fall into two distinct categories:

- *XTCP protocols:* Protocols in this category use TCP-compliant header and have TCP-like handshake.
- *NTCP protocols:* Protocols that differ from TCP in the transport protocol header and its handshake mechanism.

As explained in Section IV-B.2, in addition to the negotiation of protocol, the TCP connection parameters are exchanged during the STP three-way handshake. Hence, XTCP class of protocols can start data exchange right after the new protocol is attached. However, NTCP and other non-TCP compliant protocols must exchange their handshake parameters after the STP handshake. TCPsec, though an extension to TCP, differs significantly from TCP during the handshake phase. TCPsec requires additional security parameters to be exchanged to set up a secure session. Hence, TCPsec handshake options must be exchanged after the completion of STP handshake. This adds additional latency to the handshake phase. We call this the NTCP implementation of TCPsec transport.

In order to decrease the handshake latency, we also implement a different scheme by taking advantage of STP's handshake implementation. STP implementation sets the TCP transport as the default protocol for the XTCP domain. During the connection initiation, the domain is searched for the default protocol and used for STP's handshake. We extend this, by searching through the XTCP domain to find if the requested protocol has already been loaded. If the protocol is loaded, the selected protocol is attached and its handshake is used instead of the default protocol's (TCP's) handshake. Hence, TCPsec's handshake is overloaded to negotiate the protocol. We call this the XTCP implementation of TCPsec.

V. Evaluation

In this section, we present an evaluation of TCPsec. We start by evaluating the security aspects of TCPsec followed by its performance. Since TCPsec design is largely derived from SSL, we compare the security of TCPsec with that of SSL. For performance analysis, we first compare the performance of TCPsec with TCP to illustrate the overhead incurred in providing security. Next, we compare the performance of TCPsec with SSL to find how TCPsec competes with SSL. Last, we evaluate the TCPsec protocol to understand the overhead incurred when the STP framework is used for deployment. We conduct all our experiments on the Emulab [20] network testbed.

A. Security analysis

TCPsec's security handshake is very similar to that of SSL. Therefore, TCPsec's security is comparable to that of SSL. However, we note the following differences between SSL and TCPsec. Unlike SSL, in TCPsec, the keyed message digest is generated over the encrypted data. As shown in [14], authentication after encryption provides a generically secure solution as compared to the encryption after authentication. SSL uses implicit sequence number to avoid replay attacks, whereas TCPsec uses explicit sequence numbers. Sequence numbers can be easily deduced by observing the data stream and hence exposing the sequence number does not introduce any security vulnerability. Hence, our use of explicit sequence numbers is secure. Finally, SSL uses implicit IVs (initialization vectors) derived from the cipher text of the previous message while TCPsec uses explicit IVs. As explained in [21], explicit

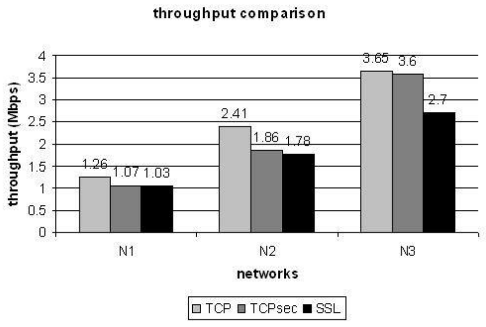


Fig. 7. Throughput Comparison: TCP vs TCPsec vs SSL.

IVs are secure as compared to implicit IVs. TLS version 1.1 [22] uses explicit IV to overcome the problems associated with implicit IV.

Though TCPsec is secure in its design, it introduces a security vulnerability due to the additional information stored in the syncache. Recall from Section III-B that the syncache data structure is used to store information about TCP connection requests at the server side until control blocks are allocated. As shown in Section III-B, memory allocated for the syncache is restricted to a maximum value to avoid the SYN flooding attack. Since the maximum size of the syncache is restricted, the increase in the syncache entry size does not cause a SYN flood attack. However, to support the same number of outstanding connection request, the memory allocated to the syncache must increase. Table II shows the number of outstanding connections allowed for the default memory limit of 15 KB and the memory bound required to support 500 simultaneous connections. Though the memory required by syncache in TCPsec is almost double the memory required in TCP, the actual value required to support large number of simultaneous connection attempts is still low (124 KB).

TCP # of conn	mem limit	TCPsec # of conn	mem limit
109	15 KB	60	15 KB
500	63.36 KB	500	123.5 KB

TABLE II

MAXIMUM # OF SIMULTANEOUS CONNECTION REQUESTS.

B. *TCPsec vs TCP*

In comparison to TCP, TCPsec incurs additional overhead due to (i) the processing associated with the cryptographic functions (ii) the extra protocol message during handshake, and (iii) increased segment size for carrying security information including TCPsec options, message digests, and IVs (although the increase in segment size is $\approx$ 60% for ACKs, it is less than 2% for 1500 byte packets). The first two factors increase TCPsec's handshake latency in comparison to TCP. Due to space limitations, numerical results showing this increase are presented in [12]. We evaluate the data throughput performance of TCPsec by considering networks of two FreeBSD machines separated by three different emulated links. The characteristics of these links are presented in Table III. To measure the throughput, a large file is transmitted

Link	Bandwidth	RTT	Loss
N1	2 Mbps	100 ms	0.5%
N2	5 Mbps	50 ms	0.5%
N3	25 Mbps	50 ms	0%

TABLE III

CHARACTERISTICS OF EMULATED LINKS.

from the server to the client and the time taken for the transfer is used in the throughput calculation. Figure 7 shows our throughput measurements. As expected, the throughput of TCPsec reduces, albeit by varying amounts, in all the three networks. However, this reduction is not drastic, and we believe that it could be traded for the security provided by TCPsec.

C. *TCPsec vs SSL*

TCPsec incurs obvious overheads when compared to TCP due to its additional security functions. For a fairer evaluation of TCPsec, we compare its performance with SSL. We derive most of TCPsec's design from SSL and hence we choose SSL as the reference protocol for performance comparisons. We first measure the handshake latency of both TCPsec and SSL on a network with two end hosts separated by a 5 Mbps bandwidth link with 100 ms RTT and 0.5% loss rate. We plot the results in Figure 8. In the handshake latency measurement, we measure the time taken to complete the entire negotiation to set up a secure session. In TCPsec this happens during the 4-way handshake whereas in SSL the time taken for session set up includes TCP's 3-way handshake and SSL's handshake as well. As expected, SSL's handshake is significantly more than TCPsec's handshake.

To compare the data throughput of TCPsec and SSL, we measure the data throughput of TCPsec and SSL on the three different networks N1, N2, and N3. We show the results in Figure 7. Since TCPsec and SSL involve the same set of cryptographic operations during data exchange, we do not expect their throughput to differ a lot. However, we find that the throughput of TCPsec is slightly higher than SSL in N1 and N2 and surprisingly, about 30% higher in N3. Although we believe that this throughput improvement is due to our efficient implementation of TCPsec, we need to investigate this result further.

D. *TCPsec with STP*

We calculate the throughput of TCPsec + STP on networks, N1, N2, and N3, and compare it against the throughput of a hard-wired native implementation of TCPsec. As seen from the Figure 9, the throughput of TCPsec transport on STP is roughly the same as throughput of the native implementation on all three network conditions. This indicates that the throughput of TCPsec is not significantly reduced by the overhead of the STP framework. Note that TCPsec throughput is same for both the XTCP and the NTCP implementations as they differ only in the handshake mechanism. Therefore we do not make any distinction between the two implementations in Figure 9.

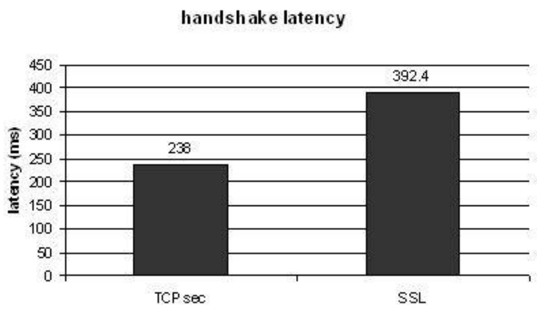


Fig. 8. Handshake Latency Comparison: TCPsec vs SSL.

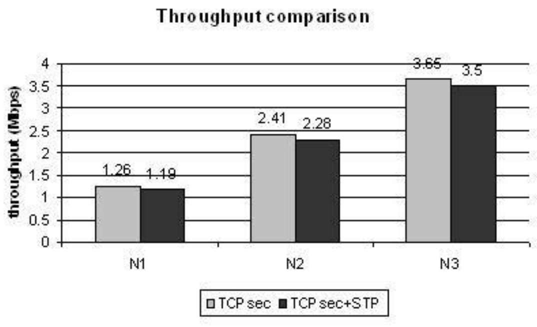


Fig. 9. Throughput Comparison: TCPsec transport in STP vs Native TCPsec.

E. TCPsec vs IPsec

IPsec uses the Internet Key Exchange (IKE) protocol for setting up a security association, and for exchanging security parameters between two end-points. IPsec uses other protocols (e.g., ESP [5]) for data transfer. IKE involves two phases [23]. In the first phase, 3 (aggressive mode) or 6 (main mode) messages, and in the second phase 3 additional messages are exchanged between the two end-points. Further, when TCP is used over IPsec, 3 more messages are exchanged for the TCP handshake. Hence, when used in a single connection application, IKE incurs a much higher message overhead (9 or 12 messages) and consequently, a much higher delay in comparison to TCPsec (4 messages). However, in scenarios where several application connections run over the same IPsec security association, the extra overhead of IKE gets amortized over these connections making IPsec more efficient for those scenarios. We have not yet measured the in-kernel data transfer performance of IPsec in the BSD kernel. However, we believe that the performance numbers will be very similar to those of TCPsec.

VI. Formal Verification

Formal verification methods allow us to prove the correctness of protocols and also identify the flaws in their design and implementation. We formally verify the correctness of our TCPsec protocol by building on an existing work [24], which verifies the behavior of TCP state machine. The TCPsec state machine verification consists of the following two steps.

The first step involves the construction a model of the TCPsec state machine using a formal language. We use the CTL specification language to model the TCPsec state machine. We modify the TCP model defined in [24] to include fragmentation of handshake messages and the fourth handshake message present in TCPsec. We define additional variables to model fragments that can be lost or arrive out of order and an additional state to include the fourth handshake message. To illustrate the modeling of state transitions, we explain the transition from SYN_SENT state to the ACK_SENT state. We model the rest of the transitions similarly. When the connection is in the SYN-SENT state, processing of an incoming segment consists of the following sub-steps.
1. If the ACK flag is set but the ACK is unacceptable, then the fragment is dropped and the state remains the same.
2. If the RST flag is set and the ACK is acceptable, the connection goes to the CLOSED state. 3. If the RST flag is set and the ACK is acceptable, then the segment is dropped and the state remains unchanged.
4. When all the fragments of the SYN+ACK are received (when the frg_ok is set) and the SYN+ACK flags are set together in any of these fragments, then the state changes to the ACK-SENT state.
5. When all the fragments of the SYN+ACK are received (when the frg_ok is set) and no fragment contains the SYN+ACK flags set, then the connection moves to the CLOSED state.
6. When a segment with SYN bits set is received, the state changes to the SYN-RECEIVED state.
7. When all the fragments are not received the state remains unchanged at SYN-SENT state.

The second step involves verifying the model using a model checker tool. In our work, we use the NuSMV [25] model checker to verify the state transitions in the TCPsec state machine. Each state transition is verified using two rules: The first rule verifies the liveness of the protocol[5] and the second rule verifies the specified state transition when the appropriate message is received[6]. We present the two rules corresponding to the transition from the LISTEN to the SYN_RECEIVED state below. Other rules are similar to those shown below.

```
SPEC EF(state = LISTEN & event = SEGMENT & !rst_flag
            & !ack_flag & syn_flag)
    SPEC AG((state = LISTEN & event = SEGMENT &
      !rst_flag & !ack_flag & syn_flag) -> AX(state =
                   SYN-RECEIVED))
```

The first rule states that sometime during the connection setup, the state will be LISTEN and a segment, with the RST and the ACK flags reset and the SYN flags set, is received. The second rule states that when the connection is in the LISTEN state and a segment with the RST and the ACK flags reset and the SYN flags set, is received then the state changes to SYN-RECEIVED state. Other transitions in the finite state machines

[5]This is specified as an EF(conditions) rule. The EF rule states that there exists some computational path leading to the 'conditions' mentioned.

[6]This is specified as an AG(conditions -> AX(S)) rule. The AG & AX rules state that all computational paths corresponding to the 'conditions' lead to the state 'S'.

have their corresponding rules.

We successfully verify all the transitions of the TCPsec state machine and conclude that the TCPsec modifications do not affect the correctness of the TCP finite state machine.

VII. Related Work

We present a brief overview of some of the existing efforts on implementing security at the transport layer [26]–[29] and contrast them with our work on TCPsec. While Security Protocol 4 [26] suggests a security framework model, it does not define the exact implementation for any particular transport protocol. [27] provides a very rudimentary implementation suggestion of SP4 at the TCP layer. This implementation suggests a separate key management protocol for setting up a cryptographic association. *SecureTCP* [28] is based on the idea of extending TCP to implement security. Among the existing work, secureTCP is closest to TCPsec. However, secureTCP provides only a basic framework for security. It fails to address many important issues including fragmentation of handshake messages to support exchange of large certificates, mutual authentication of communicating nodes, key generation for encryption and integrity protection, interfacing applications with secureTCP, and very importantly, deployability. TCPsec addresses all the drawbacks of secureTCP in our framework and additionally provides thorough evaluation, NAT interoperability, and an enhanced socket layer for applications to choose security on a per-socket basis.

In another related work, *TCP-MD5 option* [29], the authors provide a solution to protect the integrity of TCP data. However this scheme lacks important features such as key distribution mechanism, node authentication, and data confidentiality. TCPsec provides a much richer functionality covering all of key exchange, authentication, data confidentiality, and data integrity. Some variants of IPsec, including ML-IPsec [8], TF-ESP [6], and ESVP [7], that allow exposure of transport headers have also been proposed. However, these variants incur high message overhead and high setup delay compared to TCPsec when an IPsec security association is established for every application connection.

VIII. Conclusion

We have built a comprehensive security framework, TCPsec, at the TCP layer that provides a light-weight alternative to SSL and IPsec while overcoming their limitations. We have a prototype implementation of TCPsec in the FreeBSD kernel 4.7. We have developed a simple STunnel-like interface for applications to operate with TCPsec unchanged. To support easy upgrade of the kernel, we have implemented TCPsec in the STP framework, which provides a sandboxing mechanism to remotely upgrade transport protocols through code shipping. TCPsec is a working system and our evaluation shows that TCPsec provides an efficient security solution.

Acknowledgment

We thank Professor Jay Lepreau and other members of the Flux lab for helping us with the Emulab and STP software. We would also like to thank Sid Ramesh, Jun Cheol Park, Eun Yong Kang and Matt Probst for their effort in evaluating the https transfer attacks and the implementation of TCPsec proxies.

References

[1] A. Freier, P. Karlton, and P. Kocher, "The SSL Protocol Version 3.0," *IETF Internet Draft: draft-freier-ssl-version3-02.txt*, March 1996.
[2] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," *RFC 2246*, January 1999.
[3] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," *RFC 2401*, November 1998.
[4] M. Dalal, "Transmission Control Protocol security considerations," *IETF Internet Draft: draft-ietf-tcpm-tcpsecure-02.txt*, November 2004.
[5] S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)," *RFC 2406*, November 1998.
[6] S. Bellovin, "Transport-friendly esp (or layer violation for fun and profit)," ser. IETF-44 TF-ESP BOF, March 1999.
[7] S. Kasera, S. Mizikovsky, G. Sundaram, and T. Woo, "On securely enabling intermediary-based services and performance enhancements for wireless mobile users," in *ACM WiSe*, 2003.
[8] Y. Zhang and B. Singh, "A Multi-Layer IPsec Protocol," in *Proceedings of 9th USENIX Security Symposium*, August 2000.
[9] D. Harkins and D. Carrel, "The Internet Key Exchange (IKE)," *RFC 2409*, November 1998.
[10] K. Egevang and P. Francis, "The IP Network Address Translator (NAT)," *RFC 1631*, May 1994.
[11] P. Patel et al, "Upgrading Transport Protocol using Untrusted Mobile Code," in *Proceedings of SOSP*, Bolton Landing, NY, October 2003.
[12] S. Pichumani and S. Kasera, "Tcpsec: A Transport Layer Security Framework," University of Utah," Masters Thesis, 2006, www.cs.utah.edu/~swami/tcpsec_thesis.pdf.
[13] C. Kent and J. Mogul, "Fragmentation considered harmful," in *Proceedings of SIGCOMM*, August 1987.
[14] H. Krawczyk, "The Order of Encryption and Authentication for protecting communication (or: How secure is ssl?)," in *Proceedings of Advances in Cryptology*, Springer-Verlag Heidelberg, January 2001.
[15] A. Aboba and W. Dixon, "IPsec-Network Address Translation (NAT) Compatibility Requirements," *RFC 3715*, March 2004.
[16] "PeerSec Networks MatrixSSL," `http://www.matrixssl.org`.
[17] J. Lemon, "Resisting SYN flood DoS attacks with a SYN cache," in *Proceedings of BSDcon*, February 2002.
[18] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," *RFC 2104*, February 1997.
[19] M. Trojnara, "S-Tunnel," `http://www.stunnel.org`.
[20] B. White et al, "An Integrated Experimental Environment for Distributed Systems and Networks," in *Proceedings of OSDI*, December 2002.
[21] B. Moeller, "Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures," 2004, `http://www.openssl.org/~bodo/tls-cbc.txt`.
[22] T. Dierks and E. Rescorla, "The TLS Protocol Version 1.1," *IETF Internet Draft: draft-ietf-tls-rfc2246-bis-13.txt*, June 2005.
[23] C. Kaufman, R. Perlman, and M. Speciner, *Network Security: Private Communication in a Public World, 2nd Edition*. Prentice-Hall, 2002.
[24] J. Lu and Y. Zhu, "Automatic Verification of Transmission Control Protocol Using NuSMV," 1999, http://www.cs.toronto.edu/~chechik/courses99/csc2108/projects/8.pdf.
[25] F. G. A. Cimatti, E. Clarke and M. Roveri, "NuSMV: A new symbolic model verifier," in *Proceedings of CAV*, Trento, Italy, July 1999.
[26] C. Dinkel, N. Nazario, and R. Rosenthal, "A secure data network system transport protocol," *NISTIR 90-4228*, January 1998.
[27] L. Brown, "On Implementing Security Extensions to the TCP Transport Layer," Australian DFA, Technical Report CS8/92, 1992.
[28] T. Tsutsumi and S. Yamaguchi, "Secure TCP - Providing Security Functions in TCP Layer," in *Proc. of INET '95 Hypermedia*, 1995.
[29] A. Heffernan, "Protection of BGP Sessions via the TCP MD5 Signature Option," *RFC 2385*, 1998.