RC 19981 (03/16/95)
Computer Science 21 pages

IBM Research Report

Secure Transport Protocols for High-Speed Networks

E. Basturk, M. Bellare, C.-S. Chow, R. Guérin

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Secure Transport Protocols for High-Speed Networks

Erol Basturk[*] Mihir Bellare[†] Chee-Seng Chow[‡] Roch Guérin[§]

Advanced Networking Laboratory
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA.

January 31, 1994

Abstract

We argue that the increasingly widespread use of networks is leading to new security concerns such as *secure delivery* which are most naturally associated with the transport level and best addressed by the design of *secure transport protocols.* We present general methods for such a design which are directed at optimizing performance on a *high speed* network. Our work reflects the experience gained in designing and implementing a particular secure transport protocol.

[*] e-mail: basturk@watson.ibm.com.
[†] e-mail: mihir@watson.ibm.com.
[‡] email: cschow@watson.ibm.com.
[§] email: guerin@watson.ibm.com.

1 Introduction

As we entrust more and more of our communication to networks, the concern with security increases. This paper begins by arguing that the *nature* of the security guarantees which users will need and expect is changing. In particular, new requirements, like *secure delivery*, are emerging. In contrast to traditional security features which tend to be purely application level, the new ones are most naturally identified with the transport level. We suggest the design of *secure transport protocols* as the most effective way of filling all needs, and present a general method to transform a given transport protocol into a secure one. The design puts particular emphasis on maintaining the level of performance of the original transport protocol, particularly in a high speed environment. Our work reflects the experience gained in the design and implementation of a specific secure high-speed transport protocol.

1.1 Security requirements and their placement

Two parties, denoted A and B, communicate over a network. We will naturally view the parties as applications who use a common transport protocol. An adversary E monitors the parties' communications, and is even capable of injecting or removing messages from the lines. Nonetheless, the parties wish to be able to communicate "securely." Their advantage over E is to possess a random, shared secret key a (eg. a DES key) which E doesn't know.[1,2]

Our discussion of security features which follows takes the perspective of a designer and architect of a secure communications medium. It reflects the need to add security functionality in a way which is consistent with, and best matches, the existing communication structure. To this end, we distinguish security features according to the layer of the communication structure to which they are most naturally associated. We found that the traditional features are mostly application level, while the new ones we deem necessary are transport level.

Application level security features: Integrity and confidentiality. *Integrity* and *confidentiality* are the two traditional, and important, security features. The first means that it isn't possible for E to modify the contents of transmitted data. The second means that E cannot figure out the content of the transmission but the receiver can recover it. Note they are requirements

[1] How they come into possession of such a key is not our concern here. Rather, the problem is how to use the key to communicate securely.

[2] For simplicity we stick to this symmetric case. Our methods can be modified to work in the public key scenario as well.

on the *data* transmitted by the application. This is the reason we view them as application level security features, a view which is reflected in the traditional method of providing these features, which we discuss next.

APPLICATION LEVEL SECURITY. Application level security features are easily (and usually) provided by *application level security.* Here the sender applies appropriate cryptographic functions to his data *before* passing it to the transport for conveyance. This is the approach of Kerberos [5] and KryptoKnight [3]. Integrity is usually guaranteed by *signing* the data before passing it to the transport level. Confidentiality is usually guaranteed by having A *encrypt* the data before passing it to the transport level, and having B *decrypt* the ciphertext which is received by his transport module and passed up to him.

TRANSPORT LEVEL SECURITY FEATURES: SECURE DELIVERY. Secure delivery, roughly speaking, is a guarantee to the sender A that his message was received by B and not intercepted and destroyed by the adversary. It is essentially an extension of the standard transport level *reliability* feature to the more adverserial setting we are discussing, which is why it seems most natural to view it as a transport level feature.

Note that the presence of the application level security features does not necessarily imply the security of delivery. The adversary can easily make a sending application believe that whatever it transmitted has been received by the receiving application, even if the data are signed or encrypted by the upper layer. How? Suppose A sends some data to B which E would prefer never reaches B (we will see later examples of why E might like to do this). E stops the data ¿from reaching B (remember, all communication goes through her and she has the power to do this). Then, she sends a transport level acknowledgement of received data to the sending port. This acknowledgement is a transport level flow. Not being generated by an application, it is not signed. The transport protocol at the sending end will believe the acknowledgment, and tell the sender that its data has been received![3]

Is the lack of secure delivery a serious security problem? Yes. Here are two examples that help see why. The first is the example that lead us to pinpoint the problem. It is the problem of securing control flows in a network. We have a number of (trusted) network *control points* linked by a spanning tree. At any time, one control point is the root of the tree. Suppose A wishes to pass

[3] An implicit assumption is that the application is not performing its own acknowledgments and retransmission, but, more naturally, relying on the transport for these features. Indeed, asking the application to duplicate these transport functions is usually poor design in that it belies the layered communication structure, and would also adversely affect performance.

its root position to B. It sends a (application level signed) messages saying so, and, assuming the message arrived, relinquishes the root position itself. Without secure delivery, we don't know B got the message, and now the tree has no root.

Suppose B is a broker, and A and (the adversary) E both want to buy a certain stock. E prevents A's purchase requests from reaching B so that she can buy the stock herself.

Actually secure delivery is only one example of a transport level security feature that is desirable. In Appendix A we will discuss things more formally and point to a mathematical notion of security which we believe captures the actual requirement, and includes secure delivery, integrity (and, optionally, confidentiality) as special cases. Roughly, we will ask that either the flows between the parties maintain, in the presence of E, the "relative ordering" as well as content that they would have if E were not present, or the parties detect the presence of E.

Transport level security. In this design, the application passes its key a to the transport module, and leaves to the latter the task of applying any cryptographic functions or taking other security steps. The advantage is that the transport module can compute a signature which covers not only the data but also (relevant parts of) the transport header which it creates, particularly the byte sequence numbers. It can also sign the acknowledgments messages which exist only at the transport level. As we will see, these possibilities are key to the provision of secure delivery.

1.2 The design of efficient and secure transport protocols

We specify a method to transform any transport protocol into a "secure" version, equipped not only with the new security features discussed above, but providing, at least as options, the standard application level security features. Below we discuss the elements of the solution followed by some of its features.

Elements of the solution. The design will be specified in full in the body of the paper, but here is a brief sketch. The secure transport protocol has two phases, the *initialization* phase and the *steady state*. In the former, an *entity authentication and key exchange* protocol is executed. This authenticates the parties and yields them a *session key* α. In the steady state, if the integrity option is selected, both data integrity and secure delivery are provided by attaching to each packet a signature, computed under key α, and taken over some portions of the header and the data payload. Note that acknowledgments, which have no data, are also signed. If confidentiality is desired, the data is encrypted, also under α. See the rest of the paper for more details on which parts of the header must be signed and which left in the clear; why the design guarantees secure delivery; and also for explanations on why a session key is needed when the parties already share the long lived key a.

COMPATIBILITY. Usually, applications are written on top of an existing transport protocol before any security is included in it. We want to assure *compatibility* of a secure version of this protocol with the regular non-secure version. The secure transport protocol then becomes a superset of the regular transport protocol. Applications using the regular transport protocol should not have to be modified at all.

PERFORMANCE. We take into account various *efficiency* requirements for the secure version of a transport protocol. Certain functions should be designed so that they can readily be off-loaded to hardware without impacting the original flow and processing of messages. Specifically, encryption and decryption may initially be performed in software, but, as speed increases, dedicated hardware components will almost certainly be required. For example, software encryption and decryption for secure video conferences may be feasible at low resolution. However, the provision of high resolution secure video conferences is likely to require hardware support. Our protocol should be able to operate in both environments. There are many other design aspects that need to reflect this requirement. For example, it may be worthwhile to factor into the design the eventual parallelization of certain processes, to define "intelligent" packet format for faster parsing, to allow rapid context retrieval based on simple connection-based information – all these things should be considered in order to provide greater efficiency.
In addition, the design should seek to minimize the performance degradation resulting from the provision of security. Obviously, performance degradations cannot be avoided altogether as security typically involves additional flows and/or processing. The goal, though, is to limit such degradations by restricting the number of new flows needed to support security functions, and by remaining as true as possible to the design philosophy of the original transport protocol. For example, latency can be reduced if the same flows used to authenticate the two parties wishing to communicate, also serve as the vehicle for distributing the session key they will use. Similarly, further reduction can be achieved by including user data in the initial authentication flows, so that under the optimistic (but generally true) assumption that authentication will succeed, data transmission is not unnecessarily delayed. Whenever possible, security segments are "piggybacked" on, or combined with, existing flows.

PROOFS. We mentioned above that we have a mathematical definition of the security properties our protocol should satisfy. Correspondingly, we have proofs that if certain cryptographic functions, such as the DES, are suitably hard, then our design meets the definition.

Implementation. Section 3 briefly describes an implementation in which we added security features to an existing high-speed transport protocol. The implementation embodies the design principles outlined above as well as functions and techniques discussed in the rest of the paper.

2 Design of a secure transport protocol

An understanding of our design choices is facilitated by first explaining the performance issues which influence them. Following this explanation we specify the user interface of the secure transport protocol and a high level sketch of the design. Then we go deeper into the components.

2.1 Performance Issues

While previous sections motivated the need for security at the transport layer, they also mentioned a number of performance considerations that are of importance when designing an efficient secure transport protocol. Here we review these performance issues and describe how they influence the design. Details specific to individual components are provided in subsequent sections.

The provision of security is bound to somewhat degrade performance, but the goal is to keep this impact at a minimum. A first generic step towards minimizing the perceived performance impact of security is to provide a range of security "options" to users. Each option corresponds to a different trade-off between the level of security it affords and its cost in terms of performance. For example, users with different requirements may want to choose between encryption algorithms of different complexity and strength. It is, therefore, important to incorporate the flexibility of such choices when designing a secure transport protocol. Accordingly, our design provides a wide range of security options which are further detailed later.

There are, however, performance issues that are common to any security option, and their impact should also be minimized. Performance degradations caused by the provision of security have two major causes: additional flows and increased processing. Additional flows are typically required for authentication purposes during the initial phases of a connection. Increased processing is the result of the computational requirements of security functions such as signature generation and verification, and encryption/decryption. We now review the design implications of both, and identify the components they impact most.

Additional Flows. As mentioned above, security typically requires additional flows during the initial (setup) phase of a connection. This translates in an increased overhead and a greater latency before data transmissions can start. This can be particularly significant for short connections. There are, however, several rules that can/should be followed to limit this.

A first and obvious one is to restrict the number of additional flows needed, i.e., provide as many security functions as possible in each flow. For example (see below), key distribution can be carried out jointly with the first authentication flow rather than through a separate one. Another simple rule is to combine security flows, whenever possible, with existing protocol flows. For example, initial authentication flows can be piggybacked onto the connection establishment procedures (if authentication fails, the connection is aborted). This allows the parallelization of authentication and connection handshakes and limits, therefore, any increase in latency. Finally, a last step to minimize latency is to allow data transmission to start even before the authentication (or connection establishment, for that matter) phase has completed. This can be achieved by allowing the inclusion of user data in the initial authentication/connection flows (again, the data are discarded if the authentication fails).

To summarize, the three design rules that should be followed in order to limit the performance impact of security flows are as follows:

(1) Minimize the number of additional flows;

(2) Piggyback security flows on existing protocol flows;

(3) Allow early data transmission (during initialization phase).

The design we describe below follows these design rules.

Increased Processing. The other aspect of security that has the potential to degrade performance is the additional processing required by security functions. This is most significant for long connections as it applies to all steady state transmissions. There are, however, again some rules that can be followed to limit this impact.

The main one is that all computations should be performed in the most timely fashion possible. In other words, while an increase in the amount of computation is unavoidable, it is important to ensure that nothing slows down or delays this additional processing. In all instances we would like all computations to be carried out "on-the-fly" as data is being transmitted or received. This minimizes latency.

When applied to security functions such as signature generation and verification or encryption/decryption, the requirement of on-the-fly processing has several implications. For example, we would like decryption of a received packet to proceed independently of that of any other packet (note that this also allows parallelization) as packets may be received out of order.

Computation of cryptographic functions can be greatly speeded up by use of dedicated hardware. Our design does not require such hardware to be available, but adopts various design principles that would allow this hardware, if available, to be exploited in the best possible way. On-the-fly processing is important in this regard as it facilitates the eventual off-loading to the hardware.

Both signature computation and encryption/decryption rely on the use of secret information (key) shared by the two communicating parties. Retrieval of this information is, therefore, necessary for such processing. While one can assume its immediate availability when transmitting data, it must typically be retrieved upon receipt of a packet and this context retrieval step can introduce latency. It is, therefore, useful to design security functions, in particular signature computation, which allow processing to start immediately upon arrival of data, without waiting for the associated context retrieval to complete.

To summarize, our requirement for timely and on-the-fly security computations translates into three specific design rules:

(1) Signature computation starts before the retrieval of connection context;

(2) Packets are independently encrypted;

(3) Hardware utilization is facilitated wherever possible.

2.2 The user interface

Startup information. Each potential user (application) A of the secure transport protocol has a table T_A whose entries are pairs: $T_A = (A_1, a_1), \ldots, (A_N, a_N)$ for some N. Each A_i is the identity of another user, and a_i is the key shared by A and A_i (but not known to any other user or the adversary). Henceforth this key will be referred to as the *long lived* key (LL-key).

Connection initiator interface. Now suppose A wishes to initiate a connection with B, with whom it shares LL-key a. It interfaces with a local piece of software to which it passes the identity B and the key a. It can also control the nature and strength of the security to be provided by specifying some options which will be discussed below. This software creates for her an entity we call her *secure transport machine* and denote by M_A. Once initialized, A can send (and receive) data via the secure transport machine just as it would via the original (insecure) version. That is, when it wants to send a message m, it passes it to the secure transport machine, and the latter is responsible for relaying the message, applying any necessary cryptographic functions to it, taking care of retransmission, and in general providing all transport facilities.

Responder interface. At the receiving end, party B has also initialized a secure transport machine M_B. This machine has access to the table T_B of identities and associated keys. An incoming packet identifies the identity of the (claimed) sender, and the receiving machine then uses the table to get the associated key. With this key it can now verify signatures or perform any other necessary cryptography.

SUMMARY. In summary, we view A as having a secure machine M_A with which it interfaces. Similarly, B interfaces with its secure machine M_B. The cross network communication is performed by the two machines, who send each other packets across the network and are responsible for the secure transport mechanism.

OPTIONS. The options amongst which the initiator can select are the following.

(1) *Authentication and session key exchange (AKE) protocol specification.* The application chooses one out of a set of four possible authentication and session key exchange (AKE) protocols to be executed at the beginning of the interaction. The choices are described in Section 2.4.

(2) *Integrity option.* If this option is selected then data integrity and secure delivery are provided. Otherwise they are not provided.

(3) *Confidentiality option.* If this option is selected then data confidentiality is provided by encryption. Otherwise, the data will be in the clear.

(4) *Basic cryptographic primitive specification.* The protocol lists a set of cryptographic primitives such as MD5 (Message Digest number 5), SHA (Secure Hash Algorithm), DES (Data Encryption Standard) and double DES, which it will use to provide both integrity and confidentiality. The application chooses which one to use.

As discussed in Section 2.1, the purpose of providing options is to allow the user to trade security for performance. We now see how this is reflected in the last three options listed above. The discussion on AKE selection is in Section 2.4.

An application which wants, say, only integrity, should not be unnecessarily burdened with the performance cost inherent in providing encryption. Similarly an application wanting only confidentiality should not be burdened with the cost of signing inherent in providing integrity. So integrity and confidentiality are better made (orthogonal) options.

Cryptographic primitives exhibit a trade-off between strength and speed: more secure primitives take more time to compute. An application which needs a high level of security must be willing to pay for it in performance, and can choose a primitive which is very strong but adds to the performance cost. An application which needs just a little security can specify a weaker but faster primitive and get better performance.

2.3 Overview of design

We divide the design into the following components:

(1) *Initialization: Authentication and session key exchange.* When A initializes M_A (by passing it B's identity and the shared key a) the machine initiates and executes an authentication and key exchange protocol with M_B. Specifically, the protocol used is the one chosen by the initiator in the options specified above. This protocol results in the distribution of a new (fresh) *session key* α while making sure that this key is authentic (ie. known only to M_A and M_B).

(2) *Steady state:* Initialization completed, M_A is ready to receive data from the application A and send it to B; similarly, M_B is ready to receive data from M_A and pass it to B.

  (2.1) *Integrity via signing.* If the integrity option is selected then data is *signed* by M_A under the session key α (a signature is also called a MAC, or message authentication code). The signature covers not only the data payload but also the transport header (as explained later). On receiving a signed packet, M_B uses α to verify the signature.

  (2.2) *Confidentiality via encryption.* If the confidentiality option is selected then the data payload is encrypted by M_A under the session key α. On receiving a packet, M_B uses α to decrypt the payload and then passes the plaintext to B.

The following sections explain the components indicated above.

2.4 Authentication and session key exchange

As discussed above, the secure session begins with an authentication and key exchange protocol which yields to the parties a session key α to be used (to sign and encrypt) in the rest of the session. The first thing that needs to be explained is: Since the parties already share a key (namely the LL-key a) why get another one? The use of a session key has several motivations. First, it helps avoid cross session attacks in which data from one session are replayed in another. Second, it minimizes exposure of the long lived key. It is a standard technique in cryptographic practice. Notice that only the transport machines M_A and M_B know about the session key. A and B never see it.

There are many AKE protocols reflecting, as usual, a trade-off between performance and security. Following our principle of allowing the application the option of choosing its level of trade-off we will specify several options of AKE protocols. To discuss them let's begin by looking at the issues. There are three basic kinds of authentication. *Initiator* authentication leaves the responder convinced of the identity of the initiator, but not necessarily vice versa. *Responder* authentication leaves the initiator convinced of the identity of the responder, but not necessarily vice versa. *Mutual* authentication leaves each party convinced of the identity of the other.

AKE protocols typically use either *timestamps* or *nonces* to ensure "freshness" of a party's flows. (A nonce as used here is a random number). Nonce based methods require more flows (or "passes" as they are called in this area) than timestamp ones and thus adversely affect performance. On the other hand, timestamps require some degree of clock synchronization, with decreasing security as one allows a greater margin of error in this synchronization.

Potentially we thus have six options: three kinds of authentication, and either timestamps or nonces in each case. When one examines the basic flow structure in each case, however (cf. [1]) it is easy to eliminate two of them.[4] This leaves the following options.

(1) *Initiator authentication with time stamp, one pass.* Initiator authenticates himself to responder in a single flow. Useful for secure datagrams.

(2) *Responder authentication with nonces, two pass.* Responder authenticates himself to initiator. Protocol has the following basic form: initiator issues a nonce serving as a challenge, and responder answers. Two flows.

(3) *Mutual authentication with time stamps, two passes.* The above protocols are combined. The initiator authenticates himself with a timestamped flow while simultaneously issuing a challenge, and the responder answers.

(4) *Mutual authentication with nonces, three passes.* Each party authenticates the other with nonces. Three flows.

Our protocols are inspired by the AKE protocols of [1] but have been modified to increase efficiency. We briefly review the performance considerations which motivated these modifications, and identify the resulting design guidelines.

[4] An initiator nonce based protocol would need three flows within which we can do mutual nonce based anyway; similarly a responder time stamped protocol would need two flows within which we can do responder nonce based anyway.

A first step towards minimizing the overhead imposed by security on connection setup is to embed session key distribution in the *first* authentication flow. That is, the receipt of the first flow suffices to recover the session key. Moreover, use of the long-lived key is eliminated after the first flow and only the session key is used. If hardware is being used for signing then this minimizes the key switching time for the initiator, since he can switch to the session key as soon as the first flow is dispatched.

User data are included in the authentication flows to reduce latency. These user data are then retained by the receiver and delivered only after the successful completion of the authentication process. In case the authentication fails, all previously received data are discarded, and the upper layer is notified of the authentication failure. Piggybacking control (connection) information in the initial authentication flows can also help lower the overhead imposed by security on connection setup.

Error rates are low in high-speed networks and authentication should succeed most of the time. Transmitting user data during the authentication process can, therefore, reduce latency without impacting security. More generally, the protocol will allow transmission of user data not only in the authentication flows themselves, but also in any "initial transmission window" which may be allocated to the transmitter.[5]

Note that data packets in the authentication flows are signed using the *session* key which is another reason to switch as soon as possible to using it instead of the long-lived key.

The AKE protocols use signatures, and some of the choices in their design reflect performance issues arising from signature consideration. These issues are a part of the next section.

2.5 Integrity and signatures

Signatures are used in the AKE protocols and also, if the integrity option is selected, in the steady state. In the latter, their use will guarantee a broad set of properties including protection of the data against modification, secure delivery, and protection against replay attacks. Let's begin by describing the mechanisms and see why they yield the above mentioned properties in the steady state. Then we will look at design choices which are motivated by performance issues and which apply to the use of signatures in both the AKE protocols and the steady state.

We denote by MAC_α a message authentication code under α. It has, roughly speaking, the following property. If an adversary sees a sequence $(m_1, \mathrm{MAC}_\alpha(m_1)), \ldots, (m_k, \mathrm{MAC}_\alpha(m_k))$ of messages

[5] An initial transmission window may be allocated by the transport protocol to minimize connection setup latency.

together with their MACs (or signatures), then it is infeasible for this adversary to produce a pair of the form $(m, \mathrm{MAC}_\alpha(m))$ in which $m \notin \{m_1, \ldots, m_k\}$.

In the steady state (integrity option assumed selected) the sender attaches to each packet p the value $\mathrm{MAC}_\alpha(\bar{p})$ where $\bar{p}$ is the subset of p which includes the transport header, the full data payload, and the trailer of the packet p. We will see that of particular importance is the fact that the transport header includes the sequence number of the packet which is thereby covered by the MAC. We assume that all packets in each direction, even those which are purely transport level acknowledgements (ACKs), can be differentiated. A simple solution is to include in each packet a large enough (to avoid wrap-around) *signature* counter. We also assume that the transport header contains information which identifies the sender so that the identity of the sender is covered by the MAC as well. On receiving p, τ the receiver checks that $\tau = \mathrm{MAC}_\alpha(\bar{p})$ and rejects the packet unless this is true.

The signature prevents the adversary from making the receiver accept packets which were not in fact produced by the sender. The signature by itself does not, however, prevent a *replay attack*. In this attack the adversary simply resends a packet which was previously sent by the sender, so that its signature is legitimate and will be accepted by the receiver.[6] This attack fails because the signature also covers the (signature) sequence number of the packet, and the receiver does not accept packets out of order. Indeed, as part of the normal transport level checking, out of order packets will be discarded.

The manner in which security against replay is guaranteed illustrates well an advantage of security at the transport level. Sequencing is a standard and already present transport function. We simply exploit its presence. Had we wanted to prevent replay at the application level we would had to implement sequencing with transport level counters and provide for our own checking and retransmission, thereby not only making implementation significantly harder but duplicating the transport functionality and degrading performance.

Secure delivery is guaranteed because the ACKs include sequence numbers and are signed. Note that the signature also provides error detection, so that no CRC or error detection code is required. Replaying of packets from one session into another is not possible because each session has a different session key.

We now turn to design issues motivated by performance. Let us recall some of the issues identified

[6] To see the importance of preventing such attacks, take the following example. The adversary collects one genuine ACK. Now, when the sender sends any flow, she can prevent secure delivery. She kills the sender's flow but replays to him the ACK she has from before. The sender accepts the ACK and believes his data has reached the receiver.

in Section 2.1. It is essential that the steady state overhead involved in computing and checking signatures and counters be kept to a minimum. Ideally, we want signatures to be computed and verified on-the-fly; i.e., MAC generation (verification) should be carried out as data transmissions (receptions) proceeds rather than before (after) these operations have completed. The goal is to avoid slowing down the regular flow of data, and allow the possible off-load of signature computations and verifications to dedicated hardware. The ability to achieve such on-the-fly processing depends on two major components. The first is the latency in retrieving the information needed to perform the necessary computations. The second is the nature of the computations themselves. The first component is most relevant at the receiver, while the second is equally important at both the transmitter and the receiver.

The main source of latency when receiving a new packet is the time needed to recover the associated connection context. The context contains information such as the shared session key, the counter value of the last accepted packet, the type of cryptographic function used, etc., which are central to the verification of a packet integrity. Some delay is obviously unavoidable when retrieving this information, but it should be made as small as possible and its impact on the packet verification process should be kept to a minimum. A first step in achieving this is to allow context retrieval to be performed as quickly as possible by specifying early on in the packet all the required information. While rapid context retrieval certainly helps it does not eliminate all initial latency, and it is therefore still important to allow computations to start even before the full context has been retrieved. Finally, latency can also be kept small by ensuring that computations are of minimal complexity while meeting our security requirements.

These different requirements have been incorporated in our design, where the goal of minimal latency is reflected in the following two rules.

First, all data used to generate the signature is present in the packet. Let's illustrate with an example from the AKE protocols. In such challenge response protocols, it is common that one party (say B) proves his identity by signing a challenge R_A sent by A. In such a case, rather than returning $< R_A, \sigma >$ with $\sigma = \mathrm{MAC}_\alpha(R_A)$, party B could return just the short tag σ; after all, A already knows R_A (it is in A's context) so why waste bandwidth sending it? The reason for preferring to send the full $< R_A, \sigma >$ is that signature verification can begin without waiting for the outcome of context retrieval. On the other hand, the cost of resending R_A is minimal since bandwidth is not that expensive at these speeds. In general, this translates into the following packet format: a signed packet will be of the form $< D, \sigma >$ where σ is a signature of packet D under key α.

Note that we cannot prevent context retrieval of R_A in the above example since A must definitely check that what is received is the signature of the right challenge (namely the challenge A just sent). But putting the information in the flow decouples the signature checking from this context retrieval.

Second, MAC computations are kept simple and can start prior to context retrieval. In particular, we rely on MAC implementations of the form $\mathrm{MAC}_\alpha(m) = f_\alpha(h(m))$, where h is a (hash) function depending neither on α nor on the cryptographic primitive chosen in the initial options. The use of such an implementation will enhance performance in our setting as MAC computation does not have to wait for α and the cryptographic function to be retrieved from the context; we can begin computing $h(m)$ straight away. By the time we need to compute f_α the key α and cryptographic function will have been retrieved. In addition, since h can be chosen to be a standard hash function and only f_{alpha} needs to involve more computationally intensive cryptographic functions, the overall cost of signature generation/verification can be kept low.

The authentication protocols and packet formats of our implementation guarantee minimal latency by following the above two rules.

2.6 Confidentiality via encryption

Confidentiality means user data should not be readable by the adversary. The secure transport protocol will encrypt user data, under the session key, and use the ciphertexts as the actual payload. As encryption is done prior to packetization, the signature (if present) of an outgoing packet is computed *after* data encryption and the signature of an incoming packet is computed *before* decryption. This complies with the requirement of a signature computed "on the fly". If the signature checking at reception fails, it means the the packet is corrupted, and the packet is simply dropped: decryption is avoided.

Note that only user data are encrypted. One might, in principle, consider encrypting also certain parts of the transport headers. (We cannot, of course, encrypt the *entire* transport header: at a minimum the connection identifier must be in the clear to allow the receiver to know under which key it should decrypt!) Although encryption of parts of the transport headers might be argued to provide a higher degree of privacy, we believe the gain is marginal compared to the loss in performance. Encryption is a costly operation, and keeping it to a minimum is required. This also translates into a number of additional design rules.

In our design, we don't want to reencrypt the data each time it must be retransmitted. Therefore, we make sure the ciphertexts (as opposed to the plaintexts) are available for retransmission. This is achieved by requiring that a ciphertext $E(x)$ of plaintext x is of the same length of x, and simply replacing the input plaintext buffer with the corresponding ciphertext.

Next, we also want to minimize the processing associated with decryption so that it can always proceed with minimum latency, even in cases or errors and/or packet losses. Minimizing the associated processing means, among other things, that we would like to decrypt each packet only once. However, the ability to ensure this strongly depends on the retransmission policy used by the transport protocol. In particular, while selective retransmission readily allows only one decryption per packet, this is not necessarily true with go-back-n. In order to ensure, that under a go-back-n retransmission scheme decryption is performed only once for each packet, we must either stop decrypting as soon as a gap is detected (as it triggers a retransmission from the sender), or continue decrypting subsequent received packets but keep track of the associated spans. The first alternative introduces significant additional latency in the decryption process, while the second essentially amounts to to implementing selective retransmission. Because of the inherent inefficiency of go-back-n during decryption, we recommend that selective retransmission be used when confidentiality is requested.

The requirement that decryption be able to proceed with minimum latency even in case of packet losses or errors, also introduces requirements on the encryption scheme itself. In general, an encryption scheme is said to have a *history of length* i if the ability to decrypt a ciphertext block is conditional on already being in possession of the decryptions of the i previous ciphertext blocks. Schemes with large history will not be acceptable, because the loss of a single block affects the ability to decrypt i blocks. This effectively prevents the decryption of several subsequent blocks, and translates into increased latency. The most commonly used encryption scheme, namely cipher block chaining, has a history of two; although this is low enough to be tolerable, we prefer to avoid it. Instead, we have designed encryption schemes with history 0, so that the decryption of any packet can proceed independently of the reception of others. In addition, the scheme is such that both the encryption and the decryption procedures are parallelizable. This is in line with our design requirement to maintain the ability to use hardware support.

Another important design consideration in selecting an encryption/decryption scheme is its impact on error propagation. Lines usually have some degree of noise, so that a bit can be flipped now and then. Encryption, however, can effectively amplify the channel noise. Suppose a bit is flipped in $X = E(x)$, the encryption of x, and the receiver gets X'. In many encryption schemes, the decryption $D(X')$ differs from x in more than one bit, i.e., a single bit error has propagated to multiple bits. For example, in cipher block chaining it could differ in up to half the bits. This can translate in significant performance degradations for certain applications, whose data is either robust to limited errors (redundant information) or protected by some form of error correcting code. (Note that we assume here that data integrity is not selected as it would immediately result in the discarding of any packet in error, irrespective of the number of errors).

To avoid this problem, we use encryption schemes in which bit errors don't propagate. Specifically, our encryptions have the property that $d_H(E(x), E(y)) = d_H(x, y)$ for all x, y, where d_H is the Hamming distance. So error correcting on the plaintext is not affected. One might argue that the problem could have been solved in other ways. For example, put the ciphertext in error correcting form. This is bad for performance as well as inflexible. First, an application should have its choice of error correcting code and not have this dictated by the transport. Second, applications might be error correcting plaintext already, so that coding would be done twice.

To conclude, we stress that the extra features of our encryption schemes described above are obtained without any loss of security. Our encryption is DES based, and secure as long as DES behaves like a pseudorandom function. It even has security properties lacking in CBC-DES.

3 Implementation Overview

In this last section we briefly outline the structure of our implementation of a secure transport protocol based on a version of an existing, high-speed optimistic transport protocol similar to XTP [6, 4]. The implementation is in C and has been developed on an RS6000 workstation. All the features introduced in the previous sections (authentication, confidentiality, integrity, choice of the level of security) have been incorporated and tested.

An application which wants to use the transport protocol has to link with the transport machine object code (see Section 2.2). The code basically consists of an interface with the application and of two threads, one for transmission and one for reception of packets. The interface requests and responses are communicated to the threads through message queues.

To add security to the non-secure version of the protocol, four parts of the protocol have to be modified: the interface, the connection resources (contexts), the state machine and the format of the packets.

The application communicates with the RTP machine through the interface by using structures called *verbs* which contain data specific to particular actions of the protocol: OPEN, CONNECT, LISTEN, SEND, RECEIVE, DISCONNECT, CLOSE. Typically, OPEN allocates local resources, CONNECT allows the setting of the address to be called, SEND gives some buffers of data to be sent on a specific connection, etc. These verbs can be chained together and passed in one call to the RTP machine, which will then process them. For example, OPEN and the chained verbs CONNECT-SEND allows to specify a recipient and the content of a message. The transport machine will then packetize the message and send it over the network to the specified recipient. Once the message has been delivered, the application is notified. The connection can then be closed and resources freed by issuing a CLOSE verb. Verbs, besides their chaining capability, have the advantage that they can easily be extended.

The interface to this protocol has been extended while remaining compatible with the previous non-secure version. Beyond simple compatibility advantages, this also greatly eases the work of "securing" existing applications built on top of this protocol. When security is provided, the CONNECT and LISTEN verbs have dedicated fields for security options, such as a long-lived key and bits to select the cryptographic function and whether integrity and/or confidentiality must be provided.

The protocol associates to each connection a resource called its *context*, which contains all the information necessary to provide the required transport services, such as states, pending data, timers, etc. Information relative to the security options and cryptographic keys is also stored in the context which, along with the RTP state machine, has also been expanded to include states relative to the authentication process.

A transport packet consists of a header followed by data, if any. The format of the packets has been designed to be extensible. Most transport control functions are present in optional fields called segments which are part of the transport header. A security segment has been added to include information needed for authentication and confidentiality. Following data, a trailer has been added that contains the packet signature.

A library of fast implementation of cryptographic functions has been used in order to implement the encryption algorithm and the signature scheme. Currently, MD5, SHA, DES, Double DES are the functions the upper layer can choose from.

To test the secure protocol, a secure file transfer application has been created from an existing non-secure version. The user only specifies once at the beginning of the transfer the type of security he wants to select for this transfer: integrity, confidentiality, etc. Then both the origin (client) and the destination (server) of the transfer have to agree on a long-lived key and a cryptographic function. The presence of an enemy has been simulated by performing different actions on transmitted packets, namely, bit flipping, packet delay, misordering or loss, and replay. As expected, the protocol ensured that under all possible (tested) attack scenarios, the presence of the enemy translated in either complete interruption of transmission (as if the link was cut...) or eventual successful transfer of the file. The protocol was, therefore, successful at preventing the enemy from taking any active action during the transfer.

4 Conclusions

We have presented the design of a secure transport protocol with the requirements of compatibility, efficiency and security strength. A brief description of an existing implementation of secure transport protocol running over a high-speed network has also been given.

References

[1] M. Bellare and P. Rogaway. Entity authentication and key distribution. *Advances in Cryptology – Crypto'93 Proceedings.*

[2] O. Goldreich, S. Goldwasser and S. Micali. How To Construct Random Functions. *Journal of the Association for Computing Machinery* **33**(4), 792-807 (October 1986).

[3] R. Molva, G. Tsudik, E. Van Herreweghen and S. Zatti. *Kryptoknight* authentication and key distribution system. ESORICS 92, Toulouse, France, November 1992.

[4] R. M. Sanders, A.C. Weaver. The Xpress Transfer Protocol (XTP): A Tutorial. *Computer Communication Review (USA)* Vol. 20, No. 5, Oct. 1990, pp 67-80.

[5] J. Steiner, C. Neuman, J. Schiller. Kerberos: An Authentication Service for Open Network Systems. *USENIX Winter Conference*, Dallas, Texas, February 1988.

[6] A.C. Weaver. The Xpress Transfer Protocol. *Proceedings International Workshop on Advanced Communications and Applications for High Speed Networks*, Munich, Germany 16-19 March 1992.

A Towards a formal definition of security

The introduction gave an intuitive indication of the security features we achieve. Informal security is, however, an approach often leading to incorrect and, therefore, insecure protocols. So it is important to be able to state precisely what is the problem being solved, and show that we do solve it. Accordingly, we are able to formally define the goal we achieve. This formal definition is in the model of [1]. Based on it, we are also able to prove that our protocols are secure as long as standard cryptographic primitives (such as DES) obey some standard well defined properties (eg. behaving like a "pseudorandom function" [2]). This section sketches the model and definition. In the formal model [1] all communication between A and B is via the adversary E. That is, any message is actually sent to the adversary. She decides whether she wants to relay it, modify it, destroy it, make copies of it, etc. At any time she may send party A (resp. B) a message purporting to be from B (resp. A). Moreover, she may start up multiple *sessions* between A and B, and try cross session attacks.

We consider two "experiments." The first imagines the adversary to be benign, only relaying messages like a good channel. The second considers an adversary misbehaving arbitrarily, but assumed to have neither the secret key a nor the power to break DES. Our definition says that a protocol is secure if, except with small probability, the protocol flows either "look the same" in the two experiments or the parties "detect" the adversary. More precisely, one of the following is true in the second experiment: (1) the parties reject the connection altogether (2) every flow sent (except perhaps the last) is faithfully relayed to the recipient, with relative ordering of flows maintained. Condition (2) simply states that the protocol is capable of recreating the appearance of a good channel despite the presence of the adversary. This statement can be formalized using the notion of "matching conversations" of [1]. (Of course, to preserve meaningfulness, we also add that when the adversary is benign, all flows are correctly delivered and the connection is not terminated; otherwise, our definition as given above could be satisfied by always having the parties reject the connection!)