Simple Key-Management for Internet Protocols (SKIP)

Ashar Aziz <ashar.aziz@Eng.Sun.COM>

Martin Patterson <martin.patterson@France.Sun.COM>

Abstract

We present a key-distribution scheme that is particularly well suited for use in conjunction with a session-less datagram protocol such as IP or IPv6. We describe how this scheme can be used for both IP unicast and multicast applications, providing an approach for scalable multicast key distribution. We also describe an implementation of this scheme on the Solaris operating system, and present some findings on the performance of encryption at the IP layer.

1 Introduction

The majority of the key distribution schemes in literature deal with key distribution on a session oriented basis. However, the commonly used network layer protocols like IP are session-less datagram oriented protocols. In order to use a session oriented key distribution scheme with IP, one has to create and implement a pseudo-session layer underneath IP for the establishment and updating of IP traffic encryption/authentication keys. While this is plausible, it would be far less cumbersome to integrate a scheme that did not entail the overhead of such a pseudo session layer.

SKIP was designed for datagram oriented protocols like IP. The distinguishing feature of SKIP is that it requires no prior communication in order to establish and change traffic encryption keys. Furthermore, instead of session oriented keys, SKIP uses packet oriented keys that are communicated in-line with the packets.

2 SKIP

SKIP utilizes the fact that if each principal[1] in the network has a certified Diffie-Hellman [1] public key, then each principal shares a mutually authenticated key which is computable based solely on knowledge of the other principal's certificate. This works as follows. A principal (say I) has a private key i and a public key $g^i \bmod p$. This public key is distributed in the form of a public key certificate [2]. Similarly, principal J has a private key j and a certified public key $g^j \bmod p$. The pair I and J share the mutually authenticated secret $g^{ij} \bmod p$. This shared secret is implicit. It does not need to be communicated explicitly to either principal. Each principal can compute this secret based on knowledge of the other principal's identity and public key certificate. The shared secret is computed using the well known Diffie-Hellman algorithm [1].

This mutually authenticated long-term secret is used to derive a key, which is denoted K_{ij}. This key is derived by taking the low-order key size bits of $g^{ij} \bmod p$. The key K_{ij} is used as a master or key-encrypting key in order to provide IP packet based encryption and authentication.

An individual IP packet is encrypted (or authenticated) using a randomly generated packet key denoted as K_p. The packet key is in turn encrypted using K_{ij}. Since K_{ij} can be cached for efficiency, it allows traffic (i.e. packet) keys to be modified very rapidly (if necessary even on a per packet basis) without incurring the computational overhead of a public key operation. Furthermore, since the keys are communicated in the packets themselves, there is no need to incur the overhead and complexity of a pseudo session layer underneath IP.

Figure 1 shows an encrypted IP packet, using the two step encryption procedure described above:

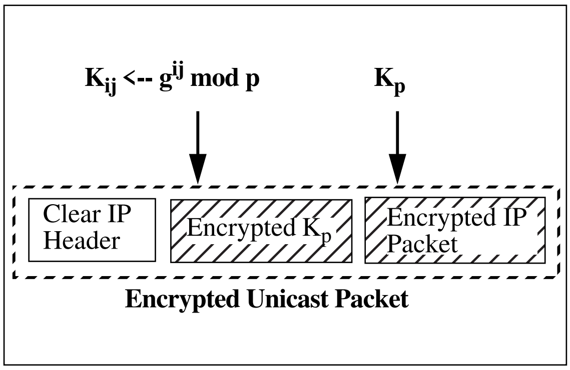

Figure 1 Encrypted Unicast Packet

When a node receives this encrypted packet, it will look up the sender's certificate. Using this and the receiving node's long-term private key, the receiving node can compute K_{ij}. Using K_{ij}, the receiving node can decrypt K_p, and hence decrypt the packet.

Although there is a packet key in each packet, it is not necessary to change the key in every packet. The keys can be changed as frequently as desired in line with key-management policies.

The description above has been provided using the number theoretic constructions of classic Diffie-Hellman, (namely exponentiation over a prime field). However, it can be generalized to any public key agreement algorithm. By public key agreement algo-

1. The term principal will normally refer to an IP node, in this paper.

rithm, we mean a cryptographic construction that allows one to compute a shared secret using another party's public value and one's own private value. Examples of cryptographic constructions which also provide the public key agreement property include constructions that employ elliptic curves over finite fields [3].

2.1 Zero-Message Master Key Update

In the preceding section we described how the nodes can compute one long-term key K_{ij}. Changing this key would require issuing a new certificate to one or the other principal. There are two desirable reasons for updating the master key. The first is that it minimizes the exposure of any given key-encrypting-key, making cryptanalysis more difficult. Second, updating the master key prevents the reuse of compromised traffic keys (K_p). Should a traffic key used for packet authentication ever be compromised (for whatever reason) then it cannot be used to send forged traffic, since the encryption of K_p under the current K_{ij} would not be known.

The master key is updated by sending in the packet a counter (say *n*) that only increments and is never decremented. The key K_{ij} becomes a function of this counter n as follows:

$$K_{ijn} = h(K_{ij}, n)$$

where *h* is a pseudo-random function such as MD5 [4]. A simple and stateless way of implementing *n* is to let *n* be equal to number of time-units since the more recent of I or J's certificate. For example, there can be a master key for a minute, an hour, a day, and so forth. In the absence of a clock, *n* can be constructed simply using a counter.

A second feature of the incrementing counter is that it prevents coarse-grain playback of traffic. Once the master keys have been updated, traffic encrypted or authenticated with the help of earlier master keys cannot be played back.

Once either principal obtains a new certificate, the counter is reset to zero. Even if the counter is updated once every second, a 25 bit counter suffices for the counter to not overflow in one year, which is a reasonable period to obtain new certificates.

2.2 SKIP for Multicast IP

The traditional approach to key distribution for multicast applications is to have a multicast Key Distribution Center (KDC) distribute the multicast traffic key to all authorized group members. This one shared key is then used by all groups members to encrypt multicast traffic.

There are at least two problems with this approach. The first is that key-change does not scale well to a large number of nodes. Key change policies need to be a function of the amount of data encrypted in any given key, and not just a function of time alone. This means that for high speed links, the keys need to be updated far more frequently than for slower speed links, for the same key-change policy. For a large number of nodes in a multicast group, it becomes difficult for a multicast KDC to be rapidly supplying updated keys to all group members.

The second problem with using the same key by all members of a multicast group is that it precludes the use of certain stream ciphers. This is because using the same key will result in the same key-stream to be used to encrypt different plaintexts. Since key-stream reuse is catastrophic to the security of a stream cipher it must always be avoided.

Fortunately, a simple modification of the basic SKIP algorithm described above solves both problems. Instead of distributing a traffic encryption key, a group owner distributes a Group Interchange Key (GIK). The GIK is then used as a key-encrypting-key similar to the way K_{ij} is used for unicast IP.

In order to send encrypted traffic to a multicast group, a member first requests the GIK from the group owner. The group owner's identity has to be known to the requesting node. This request is made using the unicast SKIP procedures described above, that include source origin authentication. Once the group owner determines that the requesting node is on the group's Access Control List (ACL), it will provide the GIK to the requesting node.

The requesting node will then encrypt the multicast traffic using randomly generated traffic keys (K_p). The traffic keys are in turn encrypted using the GIK, and sent in-line with the packet.

Figure 2 shows what an encrypted multicast packet looks like using the two step procedure described above:

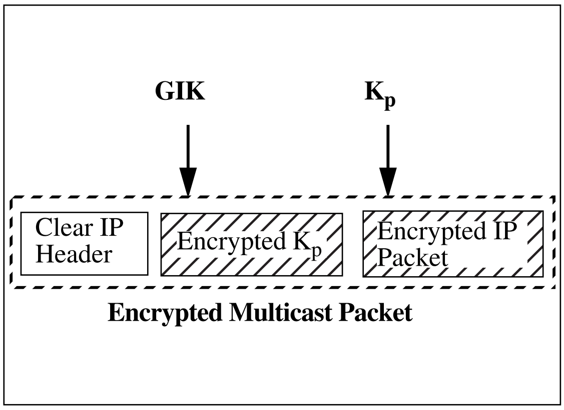

Figure 2 Encrypted Multicast Packet

This solves both the problems noted earlier. Changing the multicast traffic encryption key is simple. Each source of multicast traffic can do this by randomly generating a new traffic key (which is encrypted using the GIK) and communicating it in-line with the multicast data packets. Multicast traffic encryption keys can be updated extremely rapidly, even every packet if so desired, with no further communications overhead.

Since each source of encrypted traffic generates random traffic keys, all sources of encrypted traffic naturally use different traffic keys. This allows any kind of stream cipher to be used for multicast traffic encryption purposes.

It is important to allow stream ciphers to be used in conjunction with IP multicast. A common use of IP multicast is conferencing, especially video-conferencing. Since video is a demanding application from the perspective of speed and throughput, it is important to allow the use of ciphers that are efficient in software. Some of the most widely used and efficient in software ciphers are stream ciphers (for example, RC4 from RSA Data Security Inc.). Due to key-stream reuse issues, it is not possible to use a stream cipher like RC4 for IP multicast, if all members of the multicast group use the same traffic encryption key.

3 SKIP architecture for Solaris 2

The SKIP key management scheme presented above has been implemented on a number of operating systems, including Sun Microsystems' Solaris 2 Unix. In this section we describe the architecture of Solaris SKIP, showing how it integrates with existing OS systems such as the networking drivers[2].

Figure 3 below shows the major components:

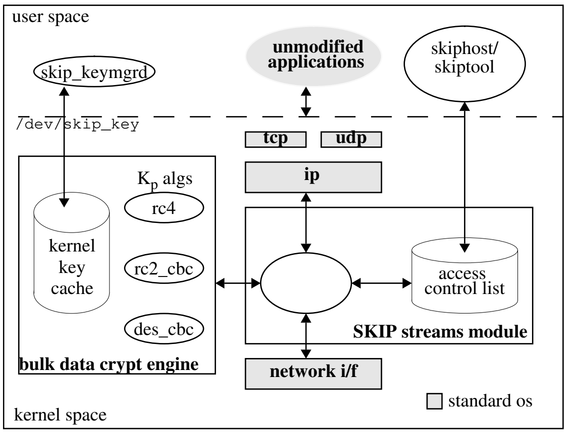

Figure 3 Solaris SKIP components

The three main components are:

1. The SKIP key manager daemon *skip_keymgrd* which maintains a database of certificate information about peers. The key manager is responsible for calculating and caching the Diffie-Hellman long term secret used to derive K_{ij}, the master or key-encrypting key for each peer. The key manager also works in conjunction with the SKIP kernel to generate a pseudo-random K_p, the ephemeral packet encryption key.
2. The SKIP bulk data crypt engine. In conjunction with the user space key manager, this offers bulk data encryption and decryption services to kernel clients. Within the SKIP engine are a number of generic cryptographic modules which implement algorithms such as DES in Cipher Block Chaining (CBC) mode, RC2 CBC and RC4 from RSA Data Security Inc. These kernel modules are used by the SKIP bulk data crypt engine to perform packet encryption. Though implemented in software, we modeled their interface so that they could be easily replaced by a hardware implementation. Each module provides Open, Encrypt, Decrypt, GetOutputLen and Close entry points. The Open entry point accepts a key and an optional initialization vector. Open returns an opaque cipher state which is passed as a parameter to all subsequent cryptographic operations using the same key. Closing a module ends the use of a particular key.
3. The SKIP streams module, a client of the bulk data crypt engine. This is inserted between the Solaris TCP/IP stack and the network interface. Using an access control list based on the peer's IP address, packets are either passed in the clear, dropped or sent to the bulk data crypt engine for encryption/decryption. The access control list is maintained by the user-level tools *skiphost* (command line interface) and *skiptool* (graphical user interface).

3.1 Method of Operation

Solaris SKIP operates entirely transparently requiring no modifications either to existing applications or OS software. During the system boot process, the SKIP streams module is inserted between the IP layer and the network interface using standard streams operations (unplumb network interface, push SKIP, replumb network interface). In this way, the SKIP module intercepts all packets entering and leaving the IP layer. If a packet is to be encrypted, the SKIP module will call the bulk data crypt engine's entry point. This function accepts a buffer containing the plaintext packet, a buffer to hold the encrypted packet and the SKIP parameters K_{ij} algorithm (key encryption), K_p algorithm (packet encryption) and peer certificate id required to encrypt the packet.

To encrypt, the SKIP engine does the following. If this is the first time the system is communicating with the peer, then the following actions take place:

1. An asynchronous message requesting a packet encryption key (K_p) is sent to the user space key manager.
2. If the key manager has not already calculated K_{ij}, it will do so and cache the result. It will then generate a pseudo-random K_p and encrypt K_p in K_{ij}. The result is returned to the SKIP engine.
3. While these calculations are taking place, the packet and any subsequent packets to the same peer are buffered for a short period of time. If the

2. The Solaris SKIP implementation is available (with export restriction) for non-commercial use at no cost from: *http://skip.incog.com*

key manager does not respond within that time (due to software failure or whatever) the storage is deleted.

4. When the key manager responds with the message containing K_p and encrypted K_p, the SKIP engine will call the appropriate cryptographic module's Open routine with K_p as a parameter, and then write a SKIP header containing encrypted K_p to the output buffer. The packet is then encrypted and written to the output buffer using the cryptographic module's Encrypt routine.

The information about the peer is cached in the SKIP engine. Subsequent packets to the same peer can be encrypted and sent without further assistance from the user space key manager.

SKIP counts the number of bytes encrypted using a given key and will request a new key once a configurable limit (by default 512 Kbytes) is exceeded. The SKIP engine will continue to use the old key for a short period of time while waiting for the new key after which it will stop and cease encrypting.

The cached information is automatically removed after a specified period of non-use. If there was an open cipher state associated with the cache entry, the appropriate cryptographic module's Close routine is called before deleting the entry.

Packets are decrypted in a similar fashion. The SKIP module accepts an encrypted packet and passes it to the engine. The first time a system receives a packet from a peer the following actions take place:

1. The SKIP protocol header is decoded, retrieving the encrypted K_p, the peer id and algorithm information.
2. An asynchronous message supplying the above information is sent to the user space key manager requesting the plaintext version of K_p.
3. If the key manager has not already calculated K_{ij}, it will do so and cache the result. The key manager then uses K_{ij} to decrypt the packet encryption key K_p and the result is returned to the SKIP engine.
4. While these calculations are taking place, the packet and any subsequent packets from the same peer with the same encrypted K_p and algorithm information are buffered for a short period of time. If the key manager does not respond within that time the storage is deleted.
5. When the key manager responds with the message containing the plaintext K_p, the SKIP engine will call the appropriate cryptographic module's Open routine with K_p as a parameter, and then decrypt the waiting packets using the cryptographic module's Decrypt routine.

The information about the peer is cached in the SKIP engine. Subsequent packets from the same peer with same encrypted K_p and algorithm information can be decrypted without further assistance from the user space key manager.

The cached information is automatically removed after a specified period of non-use. If there was an open cipher state associated with the cache entry, the appropriate cryptographic module's Close routine is called before deleting the entry.

When a sender changes encryption key, the old cached key entry is retained for a short period time. This has the beneficial effect of efficiently dealing with out-of-order IP packets around key-change boundaries which might otherwise cause unnecessary key manager requests.

All communications between the SKIP kernel and the key manager take place using a pseudo device driver.

3.2 Performance Characteristics

The SKIP kernel software is multithreaded to take advantage of the symmetric multiprocessing capabilities of Solaris 2. This allows several encrypt/decrypt operations to be scheduled in parallel for improved performance. Figure 4 below shows the performance of the SKIP bulk data crypt engine running on a multiprocessor Sun SPARCcenter 2000 under Solaris 2.4. The benchmark creates 6 SKIP kernel clients each of which encrypts and decrypts 1000 1Kbyte packets using RC4. We see that as we bring more processors online, we obtain improved throughput from SKIP.

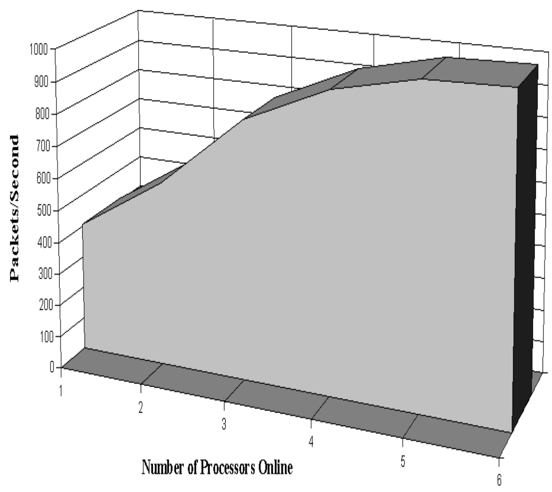

Figure 4 SKIP MT kernel performance on multiprocessor hardware.

As might be expected, the choice of packet encryption algorithm has a profound effect on the network throughput. Figure 5, below shows a comparison of common bulk encryption algorithms implemented in software.

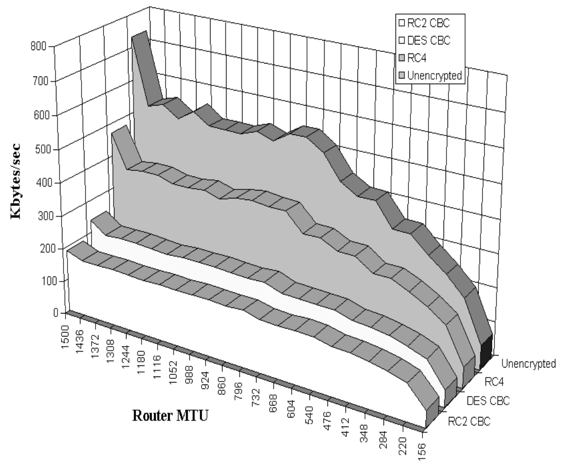

Figure 5 TCP/IP throughput as a function of encryption algorithm[3]

We see that using RC4, a fast stream cipher, we obtained about 65% of the throughput of an unencrypted channel. By comparison, DES and RC2 in cipher block chaining mode are clearly slower in this software implementation.

4 IP Fragmentation/Reassembly

One fundamental problem with IP security protocols is that the encrypted datagram is larger than the original due to the overhead of the IPSP header information plus any padding which might occur when using a block cipher. In the case of SKIP, this overhead is typically 20 (new IP header) + 28 (SKIP ESP header) bytes. In the case where we wish to encrypt a maximum transmission unit-sized IP packet, the resulting datagram would exceed the network interface's MTU. To avoid this problem the SKIP module inserted between TCP/IP and the network interface artificially reduces the interface's advertised MTU. By doing this, encrypted IP datagrams can be transmitted without further fragmentation by the local host.

4.1 IPSP and Path MTU Discovery

Even if we prevent fragmentation by the local host, a SKIP packet in transit may encounter a router with a smaller MTU. Two possible courses of action are available. Either we allow the SKIP packet to be fragmented by the router, or else we use Path MTU Discovery [5] to derive the largest datagram size which can be passed without fragmentation.

It is generally held that fragmentation is to be avoided for efficiency reasons [6], but in the case of any encapsulating security protocol this may not always be true. If we consider a case where there is a router with a small MTU in the path, a system using SKIP would emit many smaller packets (sized to fit the router's MTU) instead of a single larger packet which would be fragmented. In a SKIP system, this results in the transmission of many smaller SKIP packets (each with their own SKIP header and its corresponding overhead) instead of one larger (albeit fragmented) packet with exactly one SKIP header. Hence at smaller path MTU sizes, use of Path MTU Discovery may not be efficient. The larger the size of the SKIP header, the more pronounced this effect is likely to be.

To examine this effect further we measured TCP/IP throughput with and without Path MTU Discovery over a range of path MTU settings. The results are shown below in Figure 6.

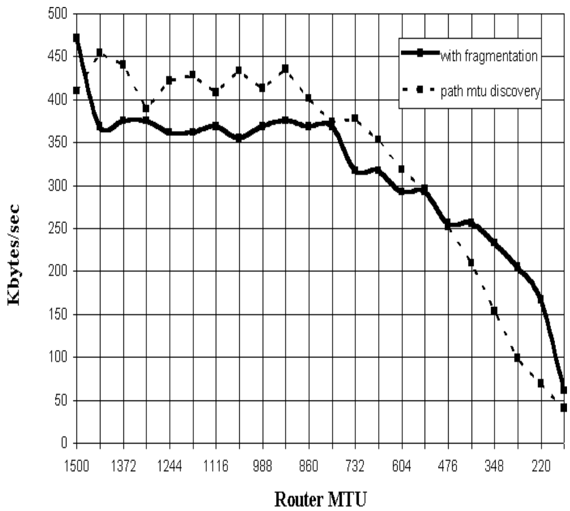

Figure 6 SKIP/RC4 Encrypted TCP/IP throughput with and without Path MTU Discovery[4]

3. TCP/IP throughput was measured using *ttcp* between a pair of 70Mhz Sun SPARCstation 5s running Solaris 2.4. A third SPARCstation 5 equipped with a Quad Ethernet board was used as a router. Each data point is the mean of 10 runs. The key size used was 40 bits, except for DES CBC which used 56 bits.

4. Same hardware/software configuration as described in footnote 3.

With SKIP we see that Path MTU Discovery provides consistently lower throughput for path MTUs less than approximately 520 bytes. In comparison, unencrypted TCP/IP shows no such effect:

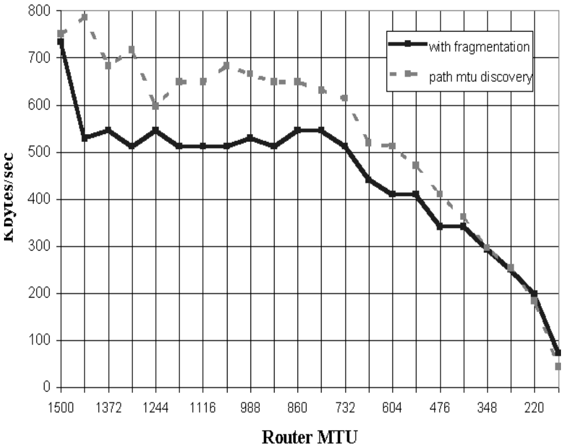

Figure 7 Unencrypted TCP/IP throughput with and without Path MTU Discovery[4]

For paths with small MTUs, turning off Path MTU Discovery when using SKIP may be helpful in improving throughput, otherwise turning on Path MTU Discovery is clearly beneficial.

One note about implementing Path MTU Discovery is that SKIP must intercept incoming ICMP "Fragmentation Needed but DF bit set" messages and reduce the proposed MTU value to account for the SKIP header overhead. The ICMP message checksum correspondingly needs to be recomputed.

References

[1] W. Diffie and M. Hellman, "New Directions in Cryptography," IEEE Trans. Info. Theory, IT-22, pp. 644-654, 1976.

[2] S. Kent, "Privacy Enhanced Mail: Part II: Certificate-Based Key Management", RFC 1422.

[3] Alfred Menezes, "Elliptic Curve Public Key Cryptosystems", Kluwer Academic Publishers, 1993.

[4] Ronald Rivest, "The MD5 Message-Digest Algorithm", RFC-1321, April 1992.

[5] Mogul, J., and S. Deering, S., "Path MTU Discovery", RFC 1191, DECWRL, Stanford University, November 1990.

[6] C. Kent and J. Mogul., "Fragmentation Considered Harmful", Proc. SIGCOMM '87 Workshop on Frontiers in Computer Communications Technology, August, 1987.

Acknowledgments

We would like to thank the following people:

Whitfield Diffie, for many helpful discussions on this subject.

Tom Markson, for implementing the key manager and remote certificate fetching.

Hemma Prafullchandra, for the CA implementation and for reviewing and providing technical and editorial critique.

Linda Cavanaugh, for providing editorial comments and formatting expertise.

Author Information

Ashar Aziz is a Senior Staff Engineer in Sun Microsystems Laboratories, Palo Alto, California. He designs and implements network protocols, operating systems and key-distribution protocols. His interests include issues related to high-speed networks, mobile networking and secure networking. He received a B.S. in Electrical Engineering from MIT in 1981 and received a M.S. in Electrical Engineering and Computer Science from the University of California, Berkeley, in 1985. He is a member of Tau Beta Pi, the National Engineering Honor Society, and a recipient of the UC Regents Fellowship. He holds three patents in the areas of mobile networking and key distribution protocols.

Martin Patterson is a Staff Engineer in Sun Microsystems Laboratories, Grenoble, France where he became a founding member in July 1990. He specializes in the design and implementation of operating system and networking software. He received a B.Sc. with First Class Honors from the Queen's University of Belfast in 1987. His interests include the implementation of high performance networking protocols, management systems and most recently network security systems. He previously worked in British Telecom as a Senior Software Engineer.