A Flow-Based Approach to Datagram Security

Suvo Mittra
Stanford University
suvo@cs.stanford.edu

Thomas Y.C. Woo
Bell Laboratories
woo@research.bell-labs.com

Abstract

Datagram services provide a simple, flexible, robust, and scalable communication abstraction; their usefulness has been well demonstrated by the success of IP, UDP, and RPC. Yet, the overwhelming majority of network security protocols that have been proposed are geared towards connection-oriented communications. The few that do cater to datagram communications tend to either rely on long term host-pair keying or impose a session-oriented (i.e., requiring connection setup) semantics.
Separately, the concept of flows has received a great deal of attention recently, especially in the context of routing and QoS. A flow characterizes a sequence of datagrams sharing some pre-defined attributes. In this paper, we advocate the use of flows as a basis for structuring secure datagram communications. We support this by proposing a novel protocol for datagram security based on flows. Our protocol achieves zero-message keying, thus preserving the connectionless nature of datagram, and makes use of soft state, thus providing the per-packet processing efficiency of session-oriented schemes. We have implemented an instantiation for IP in the 4.4BSD kernel, and we provide a description of our implementation along with performance results.

1 Introduction

A datagram service is one in which self-contained messages, or *datagrams*, are transmitted from one *principal*[1] to another, and whereby each datagram is transmitted and received atomically and independently, in isolation of others. No prior setup is needed between the source and destination principals, nor is there any required state maintained between the two.

[1]We use the term *principal* here to avoid referring to a specific protocol layer. In general, a principal can be a host, a process or a user. The term principal is commonly used in security literature.

Appears in *Proceedings of the ACM SIGCOMM '97*, September 14-18, 1997, Cannes, France.

Datagram services have been widely adopted. Their success has been mainly attributed to their simplicity, flexibility, robustness and scalability. For example, many of the most important networking protocols, such as IP [22], UDP [21], and RPC [6], make use of an underlying datagram service model.

Recently, much attention has been paid to securing network communications, especially those based on datagrams. This can be seen most apparently in the many proposals for IP security [4, 11, 18]. In addition, both IPv4 and IPv6 [8], now have built-in provisions for security in the form of an Authentication Header (AH) and an Encapsulating Security Payload Header (ESPH) [1, 2, 3].

Unfortunately, existing proposals are neither satisfactory nor completely address the problem of datagram security. For example:

- They tend to adopt a connection-based session model for adding security. That is, they require extraneous message exchange for setting up security associations and the creation of "hard" state for security processing in the two communicating principals. The key advantages of a connection-based model are its well-defined unit of protection based on a connection, and the possible efficiency gain from the use of "hard" state. We believe, however, that the sacrifice of datagram semantics (and its desirable features as a result) may be too expensive a price to pay. In addition, as we shall describe, it is not necessary for providing security.
- They focus only on specific protocol layers, e.g., IP, instead of presenting general solutions that apply across protocol layers or stacks. We believe that security solutions are subtle enough to get right even once, thus any solution that we design should be consistently applicable across protocol layers. In addition, there is a great deal of debate regarding the correct placement of security functions in a protocol stack. A solution committed to any particular layer or stack may become obsolete before it ever gains popularity.
- They concentrate mostly on specific mechanisms without reference to policy issues. Separation of mechanism and policy is good system practice; but

the design of certain mechanisms are heavily influenced by the types of policies that are to be enforced. We believe it is critical to make explicit and investigate the coupling between the two.

What is needed is a unifying abstraction that offers a unit of protection and efficiency similar to that of a connection-based model, is meaningful across different protocol layers, and is amenable to policy control. A particularly fitting candidate for this is the notion of flows.

Loosely speaking, a flow is a sequence of datagrams satisfying some pre-defined attributes. It characterizes communications that are between that of datagram and connection. That is, a flow is neither datagram nor connection — it has the flavor of both.

The flow notion can be applied across different protocol layers. At the application layer, datagrams belonging to the same application "conversation" constitute a flow. At the transport layer, datagrams in a connection can be considered a flow.

The boundary of a flow is dynamically adjustable, by varying the set of attributes of interest. A policy can be used to specify the set of attributes of interest and the corresponding security requirements.

In this paper, we propose a new protocol for datagram security, called the Flow-Based Security Protocol (FBS), based on the concept of flows. FBS makes use of zero-message keying and *soft* states. The former allows it to maintain datagram semantics, while the latter makes its efficiency comparable to a connection-based approach.

FBS is not defined for any specific protocol layer. It assumes only the availability of an underlying (insecure) datagram transport. As an example of its application, we have defined a mapping to IP and implemented the mapping in the 4.4BSD kernel.

The balance of this paper is organized as follows. In the next section, we give a brief overview of existing work. In Section 3, we present the requirements and constraints for the design of our protocol. In Section 4, we elaborate on the concept of the flow and its application to security. In Section 5, we describe our proposed FBS protocol. In Section 6, we discuss features and vulnerabilities of FBS against specific attacks. In Section 7, we present a mapping of the FBS protocol to the IP layer. We also describe our implementation of this mapping and the performance results. In Section 8, we conclude.

2 Existing Approaches

The challenge of providing secure datagram services is not new. Many approaches have been proposed. These approaches can be broadly classified into two basic paradigms, namely, *session-based* and *host-pair* keying.[2] The former has an explicit notion of session, and sets up explicit security association prior to data exchange. The latter, on the contrary, does not require explicit state setup nor security negotiation prior to data exchange.

[2]This classification is not intended to be comprehensive or exact. Many schemes have the flavor of both.

2.1 Session-based Keying

Session-based keying takes many forms, some rely on a third party such as a key distribution center (KDC) and others agree on a key between the two corresponding principals.

In a KDC-based approach, before a source sends a datagram, it contacts the KDC to request a session key and an authentication *ticket*. The ticket, encrypted with the destination's secret key, allows the destination (and only the destination) to authenticate and decrypt transmissions from the source. To send a datagram, the source encrypts the payload with the session key and sends the encrypted payload together with the ticket. The destination recovers the session key from the ticket, and uses it to decrypt the payload. Protocols that use this approach include Kerberos [25], Sun RPC [26] and DCE [24].

In session-based keying without a third party, a dynamic key exchange is performed between the source and destination principals. This establishes a shared secret, which can be used to derive a session key. The session key is stored as part of the security association, and is used in securing ensuing communications. Protocols supporting this method include Oakley [18] and Photuris [11].

The key advantage of session-based keying is the possible efficiency gain with explicit state setup, especially in case of long-lived communications. However, this is achieved at the expense of datagram semantics.

2.2 Host-Pair Keying

The basic idea behind host-pair keying is that each pair of hosts have an implicit key, called the *pair-based master key*, that can be computed only by that host pair. The implicit key exists a priori, thus allowing a message encrypted using this key to be sent without arranging anything in advance. An immediate consequence of this is that protocols based on host-pair keying support datagram semantics. That is, no extra message exchange is required to set up secure communication, nor is there a need for hard state to be maintained.

When using host-pair keying, the precise meaning of host must be properly understood. Otherwise, "unexpected" vulnerabilities can result. As an example, consider the application of host-pair keying to a network layer protocol (e.g., IP). A host here would mean a network layer entity, such as a host with an IP address. Under host-pair keying, all traffic from different connections and users would be encrypted by the same implicit key. In other words, security is only provided at the host level; this may or may not be what is intended. Some of the problems with IP level host-pair keying are discussed in [5]. In particular, the compromise of a master key exposes all traffic (past and future) between the two hosts.

Basic host-pair keying can suffer from a "cut-and-paste" attack. That is, the encrypted payload from one datagram can be cut and inserted into another datagram without being detected. A simple countermeasure is to extend host-pair keying with per-datagram keys. Instead of using the master key to directly encrypt data, the master key is used to encrypt a per-datagram key, which is used to actually encrypt the data. A subtle problem with this is that the per-datagram keys should be cryptographically random, so that the compromise of one datagram key should not reveal past or future keys. Crytographically secure random number generators such as the quadratic residue generator [7] can be a performance bottleneck.

3 Requirements and Constraints

The basic requirement of any security protocol is to provide a prescribed set of security properties. In this case, it is to enforce separation between datagrams belonging to different flows. That is, datagrams sent in a flow must only be "readable" by the intended recipient, and datagrams accepted in a flow must have been originated by the claimed source and have not been modified in transit. The precise "boundary" of a flow is defined by an application or user.

Apart from the above, we have the self-imposed requirement that the protocol should preserve datagram semantics. That is, it should not require state set up messages nor the maintenance of hard state in the two corresponding parties.

Note that the standard "features" of datagram communication, such as lack of sequencing and flow control, possibility of omission and duplication,[3] are not changed with the addition of the security protocol.

The main constraint in our protocol design is that it should be layer- and protocol stack-independent. That is, it should not assume that it will operate in a particular stack (e.g., TCP/IP) or a specific protocol layer (e.g., network layer). We accommodate this by first developing an abstract protocol, and then define mappings for different instantiations of the protocol. In this way, the system and implementation specific details are all encapsulated in the mappings only.

4 The Flow Concept

The concept of flow is not new. It has received much attention lately, especially in relationship to routing (e.g., in IPv6 [19]) and quality of service (e.g., RSVP [29]).

Broadly speaking, a *flow* is a sequence of datagrams satisfying some pre-defined attributes of interest [20]. This definition could be very general. But, typically, flows are used to refer to groups of datagrams that are to receive similar treatment in their network transport. For example, datagrams in an QoS flow are expected to enjoy similar quality of service.

Indeed, flows are a natural and flexible way to structure protocol data units at different layers of the protocol architecture. For example, at the application layer, application data with different semantics (e.g., video, audio, and whiteboard data) could be separated into their own flows. At the transport layer, data going from one transport end point (e.g., TCP/UDP ports) to another can be considered a flow.

[3]Except some malicious form of duplication — so called *replay attacks* (see Section 6).

The concept of flow subsumes both datagram and connection-oriented communications, thus sidestepping the debate between the two paradigms. From a security perspective, the model of datagram as a unit of protection is finer than necessary. This is because successive datagrams often belong to the same high-level communications and can be processed in a similar manner, making it inefficient to treat them as autonomous, independent units. The model of connection-oriented communication provides a well-defined unit of protection, i.e., all the datagrams in a connection should be protected the same way. Its main drawback is the fact that it is often encumbered with notions of connection setup and teardown, which are not necessarily required nor desired for security purposes. Flows offer the connectionless nature of datagram communication, while retaining the natural unit of protection of connection-oriented communication.

The key challenge in basing security on flows is flow identification. The ability to differentiate and isolate flows is as important as the actual security mechanism used to protect them. Depending on which layer the datagram service is implemented, this may or may not be easy. At the application layer, this can be easy if the flow semantics is built into an application. At the transport layer, it could be based on the available connection information. At the network layer, the requisite connection information may not be readily available and may need to be "deduced."

5 The FBS Protocol

In this section, we provide a detailed description of our proposed protocol. The discussion below is deliberately kept generic. Specifically, we avoid stipulating the use of specific cryptographic algorithms (e.g., encryption and hash functions) and the exact size of the security parameters (e.g., encryption key and confounder). The determination of these is often based on implementation choices, desired level of security, patent and export issues. As an example of a realization of our protocol, a mapping to IP is presented in Section 7.

5.1 Overview

The core of the FBS protocol consists of two mechanisms, namely, a *flow association* mechanism (FAM) and a *zero-message keying* mechanism. The former separates outgoing datagrams into flows, while the latter establishes the security parameters for a flow without incurring end-to-end message exchange.

These two mechanisms work tightly together. Specifically, the output of the flow association mechanism is an opaque flow identifier, called *security flow label* (*sfl* in short), which feeds into the zero-message keying mechanism to produce the per-flow key.

For generality, the FAM should be policy driven; this presents a significant design challenge. On one hand, the mechanism should be cleanly separated from policies. On

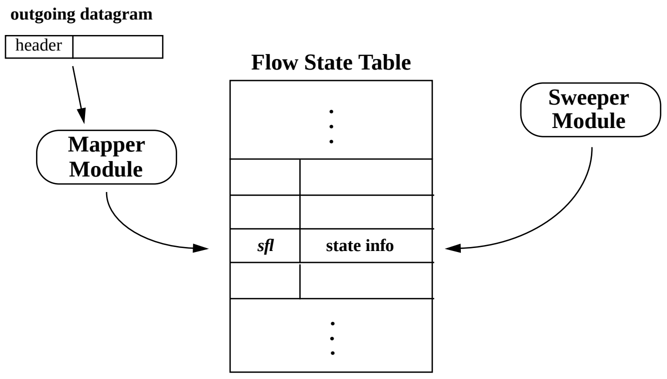


Figure 1: The Flow Association Mechanism

the other hand, it should be general enough to be able to support most policies, which could sometimes be protocol layer or operating system specific.

In our current design, policies are expressed by *policy modules* (e.g., the mapper and sweeper modules in Figure 1) which *plug* into the FAM, whose structure is shown in Figure 1. The design has three key elements:

- *Flow state table* — Each entry in this table stores information about an active flow. In particular, each entry contains the *sfl* for a flow and the necessary state information required for the operation of the mapper and sweeper modules below.
- *Mapper* module — This takes as input a set of attributes (e.g., destination principal address) of a datagram and possibly other system parameters (e.g., process id, time), and produces an index into the flow state table. If the indexed entry is "valid," the datagram is deemed to belong to the corresponding flow, and *sfl* contained therein is used as the security flow label for the datagram. Otherwise, a new flow with a new *sfl* is created, and installed into the entry. The validity of an entry is determined by inspecting its state information.
- *Sweeper* module — This is responsible for expiring flows. It removes flows that are no longer "active." It operates by scanning the entries in the flow state table and examining their state information.

In this approach, the desired security is encoded in the mapper and sweeper modules. Depending on the policy, a mapper, or a sweeper or both may be needed. Note also that although the flow association mechanism is stateful, the state is not distributed between the source and destination principals. Specifically, the source principal *actively* assigns the flow for a datagram, while the destination principal *passively* demultiplexes a datagram, based on its flow assignment, into the individual flows. No state synchronization is needed between the source and destination principals.

The zero-message keying mechanism is based on the basic Diffie-Hellman key exchange scheme [9]. It makes use of the Diffie-Hellman scheme to obtain a pair-based master key, and further derives from it a flow key. The derivation of the flow key is easy knowing the pair-based master key and the *sfl* for the flow. Knowledge of the flow key, however, does not allow one to recover the pair-based master key, or any other flow keys.

Thus, by including *sfl* in the datagram header, the correct destination principal can easily compute the flow key without incurring end-to-end message exchange. In addition, by caching the flow key as *soft state* at both ends, the computation of the flow key can be amortized over all datagrams in the flow (see Section 5.3).

5.2 Protocol Description

Notation and Basic Parameters

The protocol description makes use of a number of basic parameters. Their meaning and notation are explained here.

Let S and D denote, respectively, the source and destination principals. A minimal requirement of S and D is that they are uniquely addressable within the datagram services. In a specific realization of the protocol, the principals could be network interfaces on hosts, the hosts themselves, network protocol layers, applications, or end users.

Let s and d be the private values, as defined in the Diffie-Hellman key exchange scheme, of S and D respectively. The corresponding public values of S and D would then be $g^s \bmod p$ and $g^d \bmod p$ where g and p are respectively the (common and well-known) generators and prime modulus in the Diffie-Hellman key exchange scheme.

The confidentiality of the private values and the authenticity of the public values are assumed. However, the mechanisms for ensuring these are outside of the scope of the protocol specification. Typically, the private value is kept by a principal; while the public values are made available and authenticated via a distributed certification hierarchy (e.g., X.509 certificates [28]) or a secure DNS service [10].

We denote by

$$K_{S,D} = g^{sd} \bmod p$$

the implicit pair-based master key between S and D. Note that $K_{S,D}$ can be computed by either S or D (but no others) using their own private value and the other principal's public value.

The flow key for a flow f, identified by security flow label *sfl*, going between S and D is denoted by K_f. It is defined to be

$$K_f = \mathcal{H}(\mathit{sfl} \mid K_{S,D} \mid S \mid D)$$

where $\mid$ is the concatenation operator and $\mathcal{H}$ a one-way cryptographic hash function. S and D are included to explicitly tie the flow key K_f to that of a flow between S and D; $K_{S,D}$ also implicitly provides a similar function. Candidates for $\mathcal{H}$ are MD5 [23], SHS [17] or even DES [15]. The derivation of K_f could potentially be highly expensive, especially if $K_{S,D}$ is not already known. In such a case, communication (hence roundtrip delay) to obtain the values for computing $K_{S,D}$ is required. Ideally, the computation of K_f should be done only once per flow,

Secure Flow Label	Confounder	Message Authentication Code	Timestamp

Figure 2: Security Flow Header Format

thus amortizing the cost over all datagrams in the flow. To accomplish this, we have introduced several levels of caching (see Section 5.3).

Security Flow Header

To maintain datagram semantics, our protocol requires a security flow header or FBS header to be added to each datagram. The fields in this header are shown in Figure 2. Most of these fields should be self-explanatory, we briefly explain them here:

- security flow label — *sfl* for the datagram.
- confounder — a *statistically* random value generated on a per-datagram basis for use in the computation of MAC (see below) and as the initialization vector (IV) for encryption. Typically, encryption is performed using a block cipher such as DES [15]. The confounder is used as the IV in the CBC, CFB, or OFB modes [16]. In case of ECB mode, the confounder is XOR'ed with every block of plaintext prior to encryption.

  A confounder helps to hide the presence of identical datagrams in the same flow, as the knowledge of such may be useful to an attacker. In general, the size of the confounder should match the block size of the encryption algorithm used.
- message authentication code (MAC) — a keyed message authentication code for the datagram. It should be keyed on the flow key and calculated over the confounder, timestamp and payload. The MAC serves two purposes: (1) it ensures the integrity of the datagram body and the other fields in the security flow header; (2) it provides a form of "flow" authentication, i.e., the datagram does belong to the flow indicated in the *sfl*.

  In our current design, the MAC is defined as:

  $$\mathcal{HMAC}(K_f \mid \text{confounder} \mid \text{timestamp} \mid \text{payload})$$

  where $\mathcal{HMAC}$ is some one-way cryptographic hash function. Note that the function $\mathcal{HMAC}$ need not be the same as the hash function $\mathcal{H}$ used in the computation of flow keys.
- timestamp — a time value for countering replay attacks.

The chosen size of these fields is generally a tradeoff between protocol overhead and security. An example set of choices is given in Section 7. The placement of these fields is an implementation choice.

For generality, the security flow header should also include an algorithm identification field, which specifies the cryptographic algorithms used (e.g., for MAC computation, encryption). It is straightforward, and we omit its description here.

Protocol Operation

The basic structure and operation of the FBS protocol is shown in Figure 3. As shown, the protocol consists of two communicating halves, namely, the send and the receive sides. The actions performed on the receive side is the "inverse" of that on the send side.

We describe the operation of the protocol below, and defer the discussion of implementation considerations to Section 5.3. In particular, the various caches as shown in Figure 3 are critical for protocol implementation, but strictly speaking, they are not part of the protocol.

The FBS protocol operation is shown in Figure 4. We first describe the convention. The operation of the underlying (insecure) datagram transport is abstracted in the two functions: Send() on the send side and Receive() on the receive side.

Each datagram P entering the FBS protocol layer is assumed to have a uniform structure. Specifically, it contains: (1) a header, denoted by P.header, which in turn includes fields indicating the source (P.source) and destination (P.destination) principals; (2) a body, denoted by P.body, that carries the higher-layer protocol payload; and (3) an FBS protocol header, denoted by P.FBSheader. From the point of view of Send() and Receive(), P.FBSheader may be treated as part of the datagram header or body, depending on the datagram transport format.

Send Processing. FBSSend() in Figure 4 shows the operation of sending a datagram P from S to D. The *secret* flag, if set, indicates that data confidentiality, i.e., encryption, is desired for the datagram body.[4]

The pseudo code should be fairly self-explanatory. The key steps are: (S1) classify the datagram into a flow using the flow association mechanism. To perform the classification, the flow association mechanism can inspect the whole packet[5] (the argument P) and other system parameters (denoted by "...''), such as time and process id. The precise input is determined by the policy module plug-ins; (S2–3) generate the flow key. Note that this is only a logical step. In a practical implementation, the flow key is computed once per flow, and cached in a so called *transmission flow key cache* (TFKC) (see Section 5.3); (S4–7) generate and insert the security flow header into the datagram; (S8–9) optionally encrypt the datagram body if data confidentiality is desired; and (S10) forward the resulting datagram to the lower layer for transport.

Receive Processing. FBSReceive() in Figure 4 shows the operation D undertakes to receive a datagram P from S. The key steps are: (R1) receive a datagram from the lower layer; (R2) retrieve the security flow header from the datagram; (R3–4) check the timestamp for freshness. The checking should be based on a sliding window centered on the current time; (R5–6) recover the flow key based on the *sfl*. Again, this is only a logical step. The

[4] In general, the value of *secret* is determined by the security flow policy for the datagram. We have omitted the details here.

[5] Though typically P.header should be sufficient.

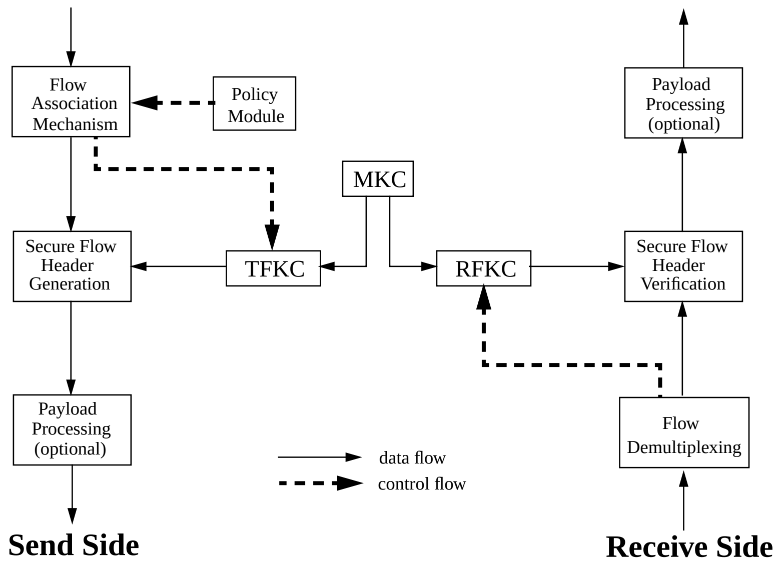

Flow Association Mechanism
Policy Module
Payload Processing (optional)
MKC
Secure Flow Header Generation
TFKC
RFKC
Secure Flow Header Verification
Payload Processing (optional)
Flow Demultiplexing
data flow
control flow
Send Side
Receive Side

Figure 3: FBS Protocol Architecture and Operation

```
        function FBSSend (Datagram $P$, boolean $secret$) {
(S1)        Flow $sfl$ = FAM ($P$, ...);
(S2)        Principal $D = P$.destination;
(S3)        $K_f = \mathcal{H}(sfl \mid K_{S,D} \mid S \mid D)$;
(S4)        Confounder $c$ = Random ();
(S5)        Timestamp $t$ = Time ();
(S6)        MAC $m = \mathcal{HMAC}(K_f \mid c \mid t \mid P$.body);
(S7)        $P$.FBSHeader = ($sfl$, $c$, $m$, $t$);
(S8)        if ($secret$)
(S9)            $P$.body = Encrypt ($K_f$, $c$, $P$.body);
(S10)       return Send ($P$);
        }
```

```
        function FBSReceive (boolean $secret$) {
(R1)        Datagram $P$ = Receive ();
(R2)        ($sfl$, $c$, $m$, $t$) = $P$.FBSHeader;
(R3)        if ($\neg$ Fresh ($t$))
(R4)            return error;
(R5)        Principal $S = P$.source;
(R6)        $K_f = \mathcal{H}(sfl \mid K_{S,D} \mid S \mid D)$;
(R7)        MAC $m' = \mathcal{HMAC}(K_f \mid c \mid t \mid P$.body);
(R8)        if ($m \neq m'$)
(R9)            return error;
(R10)       if ($secret$)
(R11)           $P$.body = Decrypt ($K_f$, $c$, $P$.body);
(R12)       return $P$;
        }
```

Figure 4: FBS Protocol Processing

actual flow key may be retrieved from a *receive flow key cache* (RFKC) (see Section 5.3); (R7–9) verify the MAC field; (R10-11) optionally decrypt the datagram body if data confidentiality was indicated; and (R12) forward the datagram to the upper layer for processing.

Observations. The following observations can be made about the FBS protocol:

- Flows are unidirectional. This is a direct result of the asymmetric roles played by the source and destination of a datagram. The source principal dictates the flow assignment of a datagram, while the destination principal "accepts" it. Thus, a duplex protocol will have two flows, one in each direction.
- It requires no hard state in either side for its operation, thus maintaining datagram semantics. Each datagram is self-contained, and can be processed independently. As we will show in the next subsection, key caching can be used to speed up protocol processing, but the contents of the cache represent only *soft* state.
- The flow key is used to directly encrypt data. With use, an encryption key will "wear out" and should be changed. The lifetime of an encryption key depends on the encryption algorithm, the length of time it has been used, and the amount of data that has been encrypted with it. With FBS, rekeying can be easily accomplished via the FAM by changing the *sfl*. Rekeying decisions, though, are made by policy modules.

5.3 Implementation Considerations

Generating the Security Flow Label

The FAM, and in turn the policy modules, is responsible for "starting" new flows. Generating the value of *sfl* itself is not difficult. The essential requirement is that the same value of *sfl* not be assigned to two different flows. This can be done by simply keeping a large (at least 64-bit) counter that serves as the value of the next *sfl* and incrementing the counter each time an *sfl* is allocated. The initial value of the counter should be randomized to prevent attackers who try to exploit reuse of *sfl* values by continuously reseting the protocol subsystem (e.g., by bringing the machine down). It is assumed that the pair-based master key will be changed (e.g., by changing the private value of a principal) before this counter wraps around. Note that *sfl* need not be random, because it is fed into a one-way, pseudorandom hash function.

Generating the Timestamp

The main purpose of the timestamp is to ensure that datagrams from an earlier flow can not be replayed and accepted. The value of the timestamp should be based on real time. For example, it could be encoded as the number of minutes elapsed since some fixed time in the past. The use of minute resolution is sufficient as the timestamp is only intended as a coarse protection against replays.

The use of timestamps also implies loose time synchronization is needed across machines. The precise synchronization requirement depends on the nature of the freshness check and the level of acceptable risk.

Generating the Confounder

Although a confounder is required on a per-datagram basis, its generation is not computationally expensive. This is because a confounder needs only be statistically random, as opposed to cryptographically random. For example, the confounder can be generated using the highly efficient linear congruential generators [12]. Of course, the seed for the generator must be randomized in each initialization of FBS.

Computing the MAC

The size of MAC is dependent on the algorithm used. For example, MD5 produces 128-bit hashes while SHS produces 160-bit hashes. To reduce header overhead, it is possible though, with reduced security, to use only part of these hashes as the MAC.

The MAC computation is an expensive operation. It requires touching all the data in the datagram. An efficient implementation should try to combine all such data touching operation into a single pass. For example, if data confidentiality is desired, then the MAC computation and encryption should be rolled into one loop. To take this one step further, an in-kernel implementation of FBS can combine this with other data touching operations such as checksumming or user space-kernel crossing data transfer.

Key Caching

Besides the MAC computation and the optional encryption, the major protocol overhead of FBS are: (1) computation of the flow keys; (2) computation of pair-based master keys; and (3) fetching of the public values of the correspondent principal. Ideally, (1) should be performed once per flow, while (2) and (3) should be performed once per correspondent principal. Key caching can be used to approximate this, hence amortizing the cost over all datagrams. Indeed, many levels of key cache can be used, we describe them below (see Figure 5[6]).

[6]The figure is drawn for an in-kernel implementation of FBS. The structure for a user-level implementation is similar, except for the user space-kernel boundary.

Public values cache (PVC). This is a cache of the public value certificates of the correspondent principals. Caching of public value certificates, instead of the public values themselves, is preferred because the former need not be secure; a certificate can be verified each time it is used.

In case of a cache miss, the public value certificate must be fetched from some certificate authority on the network. The fetch request should not and need not be secure. It should not be secure because this will otherwise create a circularity problem, i.e., generating new fetch requests that themselves need to be secured. It need not be secure because the certificates are to be verified on receipt. The *secure flow bypass* in Figure 5 allows such requests to bypass FBS. An alternative is to "pin" certain certificates in the cache upon initialization.

PVC cache misses are served by the *master key daemon* (MKD) and are extremely expensive. It incurs at the minimum a round trip communication delay. Therefore, the minimum size of PVC should be at least the average number of correspondent principals that a principal can concurrently communicate with.

Master key cache (MKC). This is a cache of pair-based master keys, indexed by names of principals. These master keys are computed using entries in the PVC and installed by the MKD.

Pair-based master key computation is fairly expensive, because it involves modular exponentiation.

Transmission flow key cache (TFKC). This a cache of transmission flow keys indexed by a combination of *sfl*, D and S.[7] With TFKC, a flow key is computed once and installed in the TFKC at the start of a new flow. Subsequent datagrams in the same flow can be processed by the stored flow key, as long as it remains in the cache. Figure 6 shows the modification of the FBS send side operation with the use of TFKC. Specifically, the code in Figure 6 replaces line (S3) in FBSSend() in Figure 4. The expression $e \in C$ and $C[e]$ denote respectively the operation to verify the existence of entry mapped to e in cache C and the operation to retrieve the entry mapped to e in cache C. Upcall() is an OS primitive that allows kernel functions to directly call a user-level function.

A TFKC cache miss is not as expensive as an MKC cache miss. Specifically, a flow key can be recomputed from the *sfl* and pair-based master key (from MKC). To minimize cache miss, the TFKC cache size should be at least the average number of active flows.

Receive flow key cache (RFKC). This is a cache of receive flow keys indexed by a hash of *sfl*, S and D. It can be understood as the analogue of the TFKC on the receive side. Thus, its structure and use is very similar to TFKC. We omit the details here.

With proper caching, the overhead of the FBS protocol can be reduced to the bare minimum, i.e., only MAC

[7]The inclusion of S is for "multi-homed" principals, i.e., principals with more than one addresses.

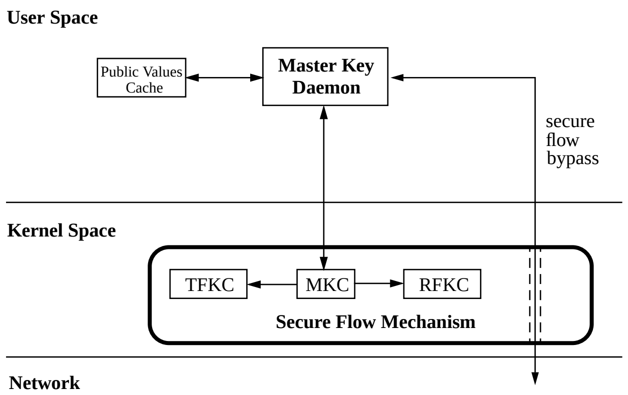


Figure 5: Key Cache

```
if (($sfl$,$D$,$S$) $\in$ TFKC) {
    $K_f$ = TFKC[($sfl$,$D$,$S$)].key;
} else {
    if ($D \in$ MKC) {
        $K_{S,D}$ = MKC[$D$].key;
    } else {
        $K_{S,D}$ = Upcall (MKDaemon, $D$);
        MKC[$D$].key = $K_{S,D}$
    }
    $K_f = \mathcal{H}(sfl \mid K_{S,D} \mid S \mid D)$;
    TFKC[($sfl$,$D$,$S$)].key = $K_f$;
}
```

Figure 6: FBS Send Processing using Cache

computation and encryption. Thus, it is critical to understand the factors affecting the cache performance.

Cache misses can be divided into three types: compulsory (cold), capacity, and collision misses. Cold misses can not be avoided, they are necessary for the initialization of the cache entries. Capacity misses can be curtailed by making the cache size at least the average number of simultaneously active cache entries. A cache entry is considered *active* if it may be referenced again. This is generally not a problem because the number of flows into a given principal is not very large compared to the amount of memory available to store the flow information in a modern kernel.

Care, however, must be taken to not delete active flow information because of collision misses. Collision misses can be avoided by increasing the associativity of the cache, by using a better replacement policy, or by indexing the cache with a better hash function that distributes entries uniformly. Because it is imperative that these caches are implemented in software and because they must exhibit quick access time, the associativity of the caches can not be too great. In turn, low degrees of associativity reduce the influence of the replacement policy. Therefore, we must look to better hash functions.

Simple hash functions, such as modulo and XOR'ing, are fast but have the disadvantage that they provide little randomness unless the input to the hash function is already random. The input for all our cache could be highly correlated, e.g., local network addresses and sequential *sfl*s. Therefore, the hash function for these caches must randomize the input to a number whose modulo can then be used to index the cache. An example of such a hash function is CRC-32. Using such a hash function and a reasonable size direct-mapped cache, we can reduce cache lookup time to $O(1)$ time in most cases.

6 Analysis

This section examines the features and vulnerabilities of the FBS protocol with respect to a number of common attacks. The discussion is by no means exhaustive; it is intended to highlight some of the design rationale behind FBS.

6.1 Perfect Forward Secrecy and Back Traffic Protection

A protocol has *perfect forward secrecy* if the compromise of any long-term keying material does not compromise future session keys or traffic. *Back traffic protection* is a similar idea but it relates to the compromise of past session keys or traffic. These protections are typically accomplished by not using any long-term keys to encrypt session keys or traffic, but using them only for authentication.

However, it can be seen that no zero-message keying protocols can provide these protections. This is because in such a protocol, the sender must generate the session key and make it known to the receiver while keeping it private from other parties. There is no other secure channel, though, between the sender and the receiver except the one protected by the long-term key.

FBS does, however, provide better protection than a scheme based on host-pair keying. Under host-pair keying, easy access to the master key is available as it is used to directly encrypt the traffic. Under FBS, the master key is never used for encryption, and breaking a flow key does not help in recovering the master key nor compromising other flow keys.

6.2 Replay Attacks

Replay attacks can be used to trick a receiver into accepting the same data twice. It can also be used to compromise the confidentiality of data (see Section 7.1).

FBS uses a window-based timestamp scheme to counter replay attacks. The key advantage of such a scheme is that it does not require maintenance of hard state information. The drawback is the need for loose time synchronization.

Many other security protocols use a nonce-based scheme to protect against replay. This, however, requires extra communication for the initial nonce agreement as well as hard state information, thereby violating datagram semantics.

In any case, the replay protection afforded by a datagram security protocol can not be perfect. If an attacker

```
struct FSTEntry {
  boolean valid;     /* valid flag              */
  byte    proto-num; /* protocol number         */
  uint64  sfl;       /* security flow label     */
  uint32  saddr;     /* source ip address       */
  uint16  sport;     /* source port #           */
  uint32  daddr;     /* destination ip address  */
  uint16  dport;     /* destination port #      */
  uint32  last;      /* last packet arrival time */
} FST[FSTSIZE];
```

```
sweeper ()
{
    i = 0;
    do {
        if (FST[i].valid &&
            (curtime - FST[i].last) > THRESHOLD)
            FST[i].valid = FALSE;
        i = (i + 1) mod FSTSIZE;
    } while (i != 0);
}
```

```
mapper (saddr, sport, daddr, dport, proto-num)
{
    i = CRC-32 (saddr, sport, daddr, dport,
                proto-num) mod FSTSIZE;
    e = FST[i];
    if (e.valid                    &&
        (e.proto-num == proto-num) &&
        (e.saddr     == saddr)     &&
        (e.sport     == sport)     &&
        (e.daddr     == daddr)     &&
        (e.dport     == dport))
        return e.sfl;
    e.sfl       = sfl++;
    e.proto-num = proto-num;
    e.saddr     = saddr; e.sport = sport;
    e.daddr     = daddr; e.dport = dport;
    e.valid     = TRUE;
    FST[i]      = e;
    return e.sfl;
}
```

Figure 7: Security Flow Policy Modules

is able to replay a datagram within the allowable "freshness" window, the attack will succeed. For wide-area networks, the "freshness" window may be large (on the order of minutes) to account for transmission delays and unsynchronized machines.

Ultimately, complete replay protection can only be achieved in high-layer protocols. Fortunately, most clients of datagram services already have some form of replay protection in terms of built-in sequencing, for dealing with occasional omission and duplication of datagrams.

7 A Mapping to IP

There has been a lot of controversy regarding the appropriateness of IP security. A complete discussion of the pros and cons of this debate is (fortunately) beyond the scope of this paper. Whatever one's view is on IP security, the mapping in this section is useful for understanding FBS. In particular, (1) it provides a concrete demonstration of how FBS can be applied in practice; (2) it presents a specific security policy that could be of general interest; (3) it fills in some implementation details not covered earlier in the generic description. For concreteness, we have implemented our IP mapping as part of the IP code [27] in the 4.4BSD kernel [14].

7.1 Security Flow Policy

At the IP level, host/gateway to host/gateway security can be easily provided. This can be done by encrypting all datagrams going from one host/gateway to another.

A more ambitious goal of IP security, however, is to provide some form of "conversation"-level[8] security. This clearly can only be an approximation because IP does not have access to all the necessary system and protocol information for determining the extent of a conversation. Strictly speaking, this also violates layering, because it requires IP to look at information from higher layers.[9]

A flow is defined by its *sfl*. To approximate conversation-level security, we need to allocate *sfl*s such that the same label is not shared among different conversations. The challenge is how to detect when a conversation begins and ends.

In FBS, the extent of a conversation is defined by the FAM. A first approximation to a conversation can be obtained by considering the sequence of datagrams sharing the same 5-tuple of

$$< \text{protocol number}, \text{source ip address}, \text{source port number}, \\ \text{destination ip address}, \text{destination port number} >$$

This policy separates different "types" of application communication into individual flows, but fails to capture the time variant nature (hence ownership changes) of such communications. Specifically, datagrams sent on the same host-port pairs belonging to different incarnations of high-level communications are classified into the same flow.

To tighten this policy, some element of time should be introduced. One possibility is encoded in the flow state table (FST) definition, mapper and sweeper modules in Figure 7. This is also the security flow policy we have used in our implementation.[10] We emphasize that these definition and modules are intended as illustrations, and express a specific example policy that we believe captures our notion of conversation.

The definition and modules in Figure 7 should be fairly self-explanatory. In words, they encode the following security flow policy:

[8]Conversation here means a sequence of datagrams belonging to the same high-level communication. For example, a TCP connection is typically a conversation, and so is a sequence of UDP datagrams in a whiteboard session.

[9]This is not uncommon in security. For example, it is routinely practiced in packet-level firewalls. Also, the implementation of TCP/IP in 4.4BSD already performs this sort of layer violation for the sake of efficiency.

[10]These definition and modules do not handle raw IP (including ICMP and IGMP) packets. For brevity, we do not discuss the issue of raw IP in this paper. Essentially, raw IP can be considered as host-level flows.

a secure flow is defined as a sequence of datagrams of the same transport layer protocol (mostly either TCP or UDP) going from a port on a host to another port on another (not necessarily distinct) host such that the datagrams do not arrive more than `THRESHOLD` apart[11]

This policy nicely separates most of the interactive (e.g., TELNET, X-window) and sustained or periodic data transfer (e.g., FTP, NFS) conversation into individual flows.

A flow under this example policy is orthogonal to a connection. Specifically, a connection may be broken up into multiple flows, and a flow may span multiple connections.

An example of the former is a long TELNET session with large quiet periods. Interestingly, the partitioning of a long duration conversation into multiple flows is better from a security perspective.

An example of the latter occurs when a process quickly begins (within a time of `THRESHOLD`) using a port that was just freed up by another process. The FAM would fail to detect the change of conversation and continue to use the same *sfl*. This can potentially pose a security problem. Specifically, an attacker can recover the encrypted data sent in a flow by (1) recording the datagrams in the flow; (2) reallocating the same port used for the flow right after the original destination principal exited; (3) replaying the recorded datagrams to itself at this port. FBS would gladly decrypt the datagrams and hand them to the attacker if they are still "fresh." One way to counter this problem is to impose a wait of `THRESHOLD` on port reallocation. This fix is, strictly speaking, outside the scope of FBS, as it requires changes in the networking code outside of FBS (e.g., the `in_pcballoc` function in 4.4BSD TCP/IP implementation).

7.2 Our Implementation

Crytographic Operations. The CryptoLib library [13] is used for all cryptographic operations. It includes most of the commonly used cryptographic primitives, e.g., Diffie-Hellman key exchange, DES, RSA, MD5, etc. The key reasons for its choice are its general availability and portability (Sparc and Intel, Solaris and NT).

When data confidentiality is desired, we use DES for encryption and MD5 for MAC computation.[12] Otherwise, keyed MD5 is used to compute the MAC. The performance numbers for CryptoLib on a Pentium 133 with 512kB L2 cache are: 549kB/s for DES in CBC mode and 7060kB/s for MD5.

FBS Header. For our implementation, the FBS header field sizes are as follow: *sfl* is 64 bits, confounder is 32 bits, timestamp is 32 bits, and MAC is 128 bits. For DES encryption, the confounder is first duplicated to provide a 64-bit quantity. The timestamp is encoded as the number of minutes since 00:00 GMT January 1, 1996 GMT. With 32 bits, the timestamp will not wrap around in the next 8000 years.

In our current implementation, the FBS header is placed in between the normal IPv4 header and the IP payload. This is unusual but can be understood as a short-cut form of IP encapsulation. An alternative is to implement it as an IP option, but the 40 byte maximum is fairly limiting.

Protocol Code. Our FBS implementation was done in FreeBSD 2.1.5, a PC-variant of 4.4BSD. The whole FBS implementation consists of a number of program and header files. Most of the FBS protocol processing code is contained within the file `ip_fbs.c`. The implementation mostly follows the pseudo-code presented in Sections 5.2, 5.3 and 7.1 with a few exceptions pointed out below.

On the send side, IP output processing as implemented in 4.4BSD can be thought of as having three logical parts. In the first part, it performs the bulk of the output processing, including options processing and route selection. In the second part, it fragments the packet, if necessary. In the last part, it sends the packet(s) out on the interface chosen in the first part. We modified the code to include a hook to our `FBSSend()` function between the first and second parts. This allows FBS send processing to be basically transparent to IP, while receiving the benefits of IP fragmentation and reassembly.

For efficiency reasons, we have combined the flow association mechanism and the flow key generation. More specifically, `FBSSend()` hashes on the 5-tuple as described in Section 7.1 and uses the result as an index into the TFKC. If the indexed entry is "active" (last use is less than `THRESHOLD` ago), it uses the stored flow key. Otherwise, it begins a new flow by assigning a new *sfl* and calculating the new flow key. In this way, the mapper module and the key cache lookup are combined (by combining the `FST` and the TFKC), thus saving an extra lookup. The job of the sweeper module also becomes implicit as it is absorbed into the mapping phase.

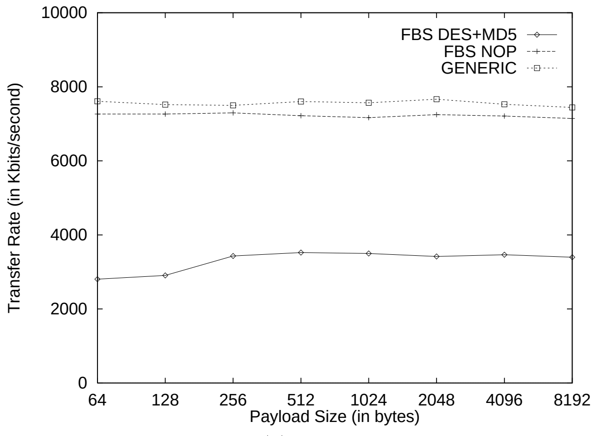
After creating the FBS header, `FBSSend()` inserts it between the IP and the transport layer headers. It fixes the IP header to account for the increase in the packet size. To IP, the FBS header is simply a part of the higher layer header.[13] A forwarding router also will not see anything "strange" about FBS processed IP packets.

Similar to the output processing, IP input processing in 4.4 BSD can also be thought of as having three logical parts. The first part performs the bulk of the input processing except for reassembly. Then, if the packet is not being forwarded, the second part reassembles the packet, if necessary. Finally, the third part dispatches it to the correct higher-layer protocol. The hook to our `FBSReceive()` is between the second and third parts. Thus, as in output processing, FBS receive processing is basically transparent to IP. The operation of `FBSReceive()` is similar to what was described in Section 5.2. It removes

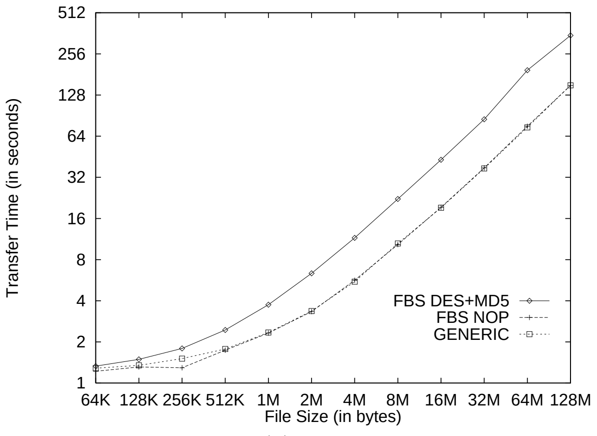
[11] A hash collision can prematurely terminate a flow. This does not affect security though. Also, almost no collision is observed with a reasonable `FSTSIZE`, e.g., 32 or above.

[12] For efficiency, DES could have been used for both encryption and MAC computation.

[13] This is as it should be given that FBS is an end-to-end protocol.

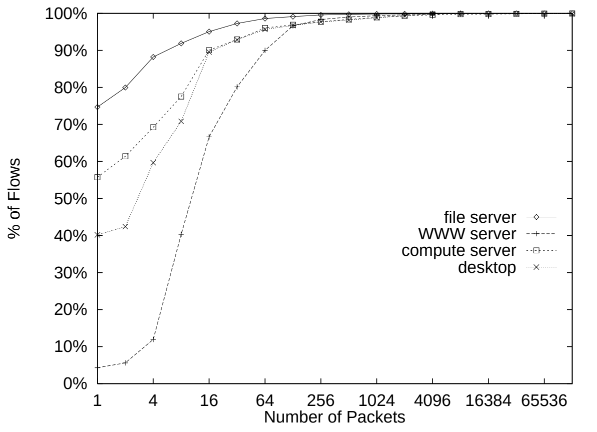


(a) ttcp

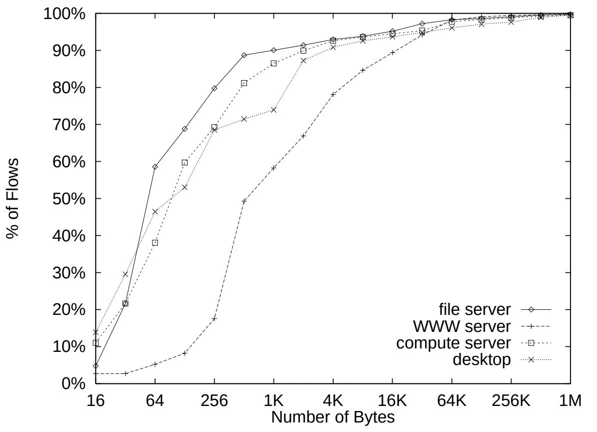


(b) rcp

Figure 8: FBS Performance Results



(a) Flow Size in Packets



(b) Flow Size in Bytes

Figure 9: Flow Size

the FBS header, performs processing on the header, and hands the packet back to IP for dispatch. We omit the details here.

Our implementation required only minor modifications to the rest of the 4.4BSD networking code. In particular, only three files had to be changed: `ip_input.c`, `ip_output.c`, and `tcp_output.c`. Files `ip_input.c` and `ip_output.c` each required two lines of changes to provide the hooks to our `FBSSend()` and `FBSReceive()` functions. `tcp_output.c` was changed because of a dependency unique to the BSD implementation. Specifically, `tcp_output()`, for the sake of performance, attempts to calculate exactly how much data it can place in a packet without triggering fragmentation. It then places exactly this much data in the packet and sets the DF (Don't Fragment) flag when filling in the IP header. This breaks when we insert our FBS header. We modified its calculation to include the FBS header size.

7.3 Experimental Results

We have performed two types of experiments. The first type measures the raw performance of FBS. The use of any security protocol will incur a performance penalty. This provides an idea on the cost of using FBS. The second type collects data on flow characteristics in a typical server-based campus LAN environment. This is useful for understanding the suitability and general dynamics of a flow-based approach to datagram security.

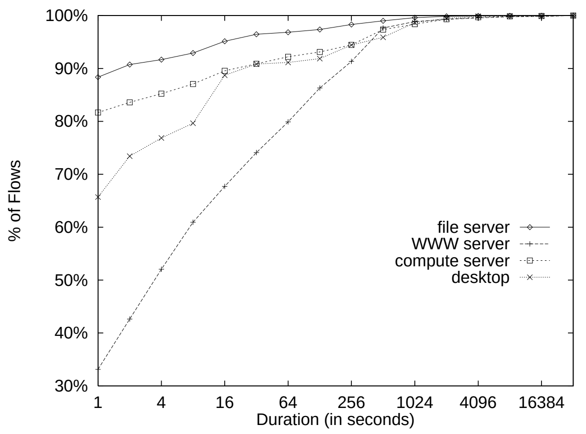
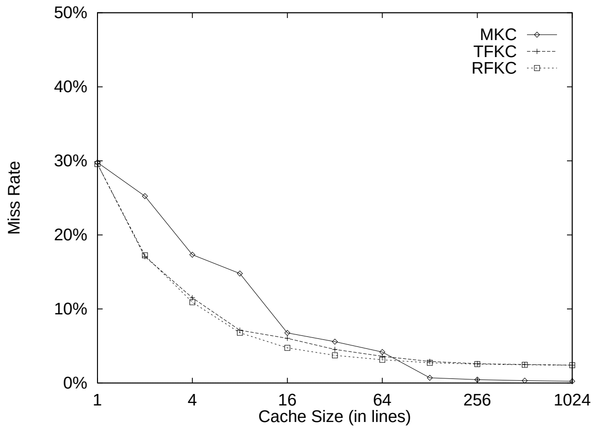


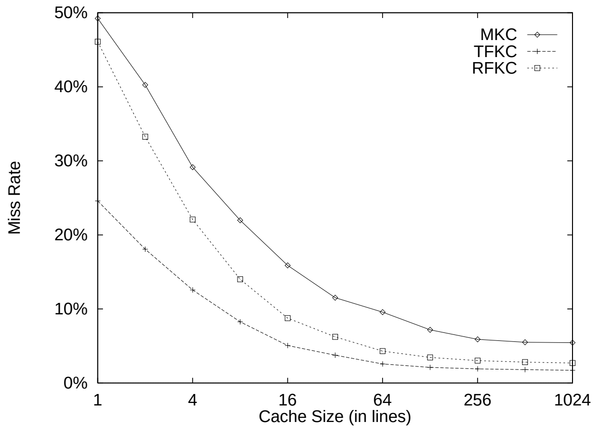
Figure 10: Flow Duration

Setup. Our experimental setup consists of a number of Pentium 133s with 512 L2 cache running FreeBSD 2.1.5. For timing measurements, these machines are put on a dedicated 10M Ethernet segment. We measure throughput using both `ttcp` and regular `rcp`.

For flow measurements, we use the Pentium 133s as network sniffers (using `tcpdump`) on our workgroup wide



(a) Cache Miss in a File Server



(b) Cache Miss in a Web Server

Figure 11: Cache Miss Results

LAN, which has a number of file and compute servers in addition to individual users' desktops. Separately, we also collected packet-level traces for a lightly hit (about 10,000 hits per day) WWW server. The collected traces are fed into a number of flow simulation programs to generate the final flow characteristics. These characteristics provide an idea on the dynamics of flows had every machine on the LAN implemented FBS and the security flow policy defined in Section 7.1.

Results. The timing results are shown in Figure 8. FBS DES+MD5, FBS NOP, and GENERIC represent respectively FBS with data confidentiality and MAC computation, FBS with "nullified" encryption and MAC computation (i.e., both encryption and MAC returns immediately), and regular 4.4BSD IP. As can be seen, FBS incurs very little overhead outside of the cryptographic operations, as it should be. A heavy penalty (7,700kb/s reduced to 3,400kb/s) is paid though when cryptographic operations are included. The extent of the penalty is mostly a function of the quality of the crypto implementation and how it is integrated with the networking code.

To evaluate the effectiveness of our flow-based approach for IP security, we consider the following aspects: (1) Is our approach suitable for IP security? That is, does our formulation of flow-based security match well with IP? (2) Is our approach feasible for IP security? That is, can it be applied to IP and achieve reasonable performance? (3) How should flows be defined for IP security? This may be too general to have a single answer. A more concrete question would be what does the policy proposed in Section 7.1 capture in our environment?

Due to space limitation, we can not present all the details of our experimental results. Instead, we will highlight only a few key observations. We emphasize that this represents only a preliminary study of flow characteristics. We also caution that the flow characteristics are very much dependent on the type of traffic and network environment.

To answer (1), consider Figures 9(a)–(b) and 10. We observe that the majority of flows are short, consist of few packets and transfer only a small amount of data. This strongly argues for the benefits of maintaining datagram semantics. The graphs also show that there are a few long-lived flows (e.g., for NFS) that carry the bulk of the traffic. Our approach also correctly captures and handles these long-lived flows with minimal overhead.

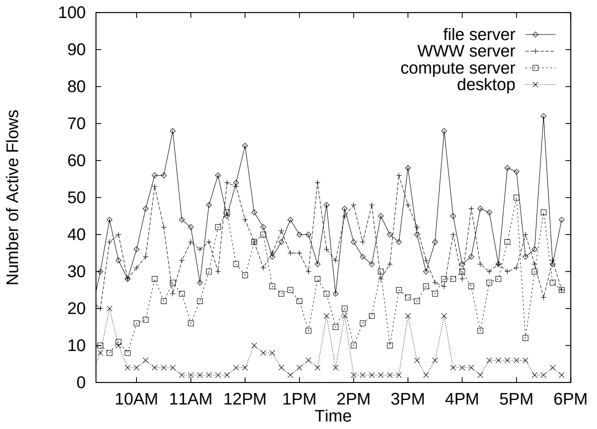


Figure 12: Number of Active Flows

To answer (2), consider Figures 11(a)–(b). The cache miss rate drops off sharply even with reasonably small cache sizes. This could indicate a packet train nature of datagrams in a flow. Figure 12 shows that the number of simultaneous active flows in a host are not exceedingly high, and can be easily handled by a modern operating system kernel.

To answer (3), consider Figure 13. As `THRESHOLD` increases from 300s to 600s, it shows the expected increase in the number of active flows, as flows are taking longer to expire. Interestingly though, the policy becomes relatively insensitive to the `THRESHOLD` value when it gets higher than 900s. Similarly, Figure 14 shows that the number of repeated flows, i.e., different flows with the same 5-tuple as defined in Section 7.1, drops off quickly as `THRESHOLD` increases. One way to interpret this is that `THRESHOLD` values of 300s or 600s provide good differentiation between flows, while maintaining reasonable stability in the flow dynamics (e.g., number of active flows).

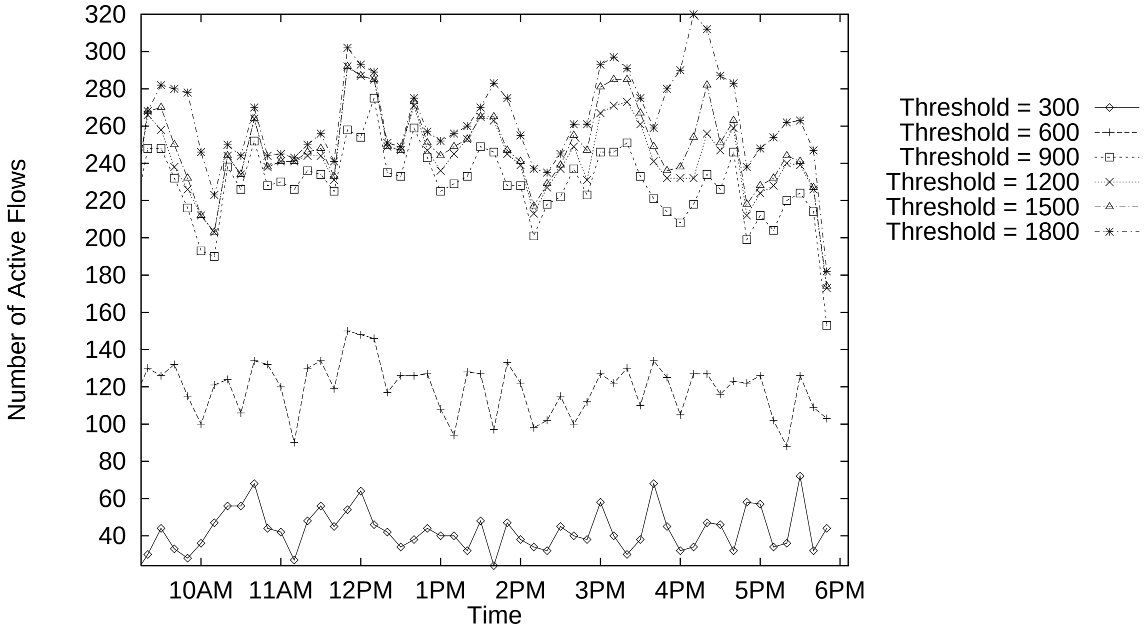


Figure 13: Active Flows for Different Thresholds

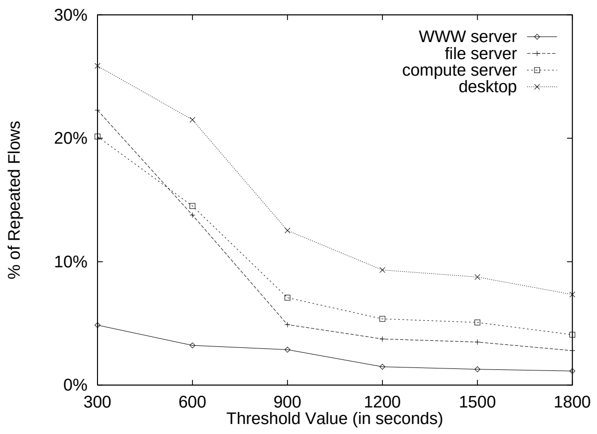


Figure 14: Repeated Flows

7.4 Comparison with IP Security Work

Strictly speaking, a comparison of FBS with IP security work is not appropriate. First, FBS is designed for general datagram security; it is not specifically targeted for IP. Second, the motivation for the design of FBS is different. FBS is concerned with how to structure secure communications as much as how the communications are secured. Most existing IP security work, on the other hand, focuses only on the protocol and mechanism aspects.

If one must compare, there are two key aspects of FBS: (1) the notion of flow-based security and FAM; and (2) the specific protocol and mechanism of FBS. (1) can be directly applied to most IP security work. It deals with policy issues and is complementary to protocol and mechanisms. Specifically, the FAM defines the unit of protection that should be enforced by the underlying IP security mechanism.

For (2), FBS makes use of zero-message keying and an explicit security flow label. SKIP [4] also provides zero-message keying based on Diffie-Hellman. The key advantage of FBS is that it provides security based on the unit of flows rather than hosts. This results in improved security because a compromised (flow) key only affects datagrams within that flow — it does not provide access to the master key which can be used to "unlock" all datagrams between a pair of hosts. FBS also provides better performance because key generation need only be done on a per-flow basis rather than a per-datagram basis.

8 Conclusion

The concept of flow offers a natural way to characterize network communications. On one hand, a flow recognizes the fact that individual datagrams may be correlated (e.g., belonging to the same high-level communication). On the other hand, a flow is not subject to the rigid boundary structure (delimited by explicit setup and teardown) of a connection.

The flexibility of flow makes it ideal as a base for defining security. Using flow as the unifying base, our FBS protocol is able to marry the benefits of a connectionless scheme (no extraneous message exchange and hard state) with the efficiency of a connection-oriented scheme.

Our implementation of the mapping of FBS to IP demonstrates both the feasibility and suitability of our flow-based approach for providing security to a datagram service such as IP.

In some cases, our notion of flow coincides with other notions of flow that have been proposed, e.g., QoS flows, while in other cases, it is orthogonal, e.g., flows for high-speed routing. A more thorough study of the relationship among these different notions of flow and the dynamics of flows for different traffic types and network environments is needed.

Acknowledgments

We would like to thank Craig Partridge and the anonymous referees for their very helpful comments. We are also grateful to the Distributed Systems Group at Stanford for providing us with access to their systems. Finally, we thank Scott Miller and the systems staff at Bell Laboratories for their help in setting up our testbed and collecting packet traces.

Suvo Mittra was partially supported by a National Defense Science and Engineering Graduate (NDSEG) Fellowship sponsored by the U.S. Air Force.

References

[1] R. Atkinson. *Security Architecture for the Internet Protocol.* RFC 1825, August 1995.

[2] R. Atkinson. *IP Authentication Header.* RFC 1826, August 1995.

[3] R. Atkinson. *IP Encapsulating Security Payload (ESP).* RFC 1827, August 1995.

[4] A. Aziz, T. Markson, and H. Prafullchandra. *Simple Key-Management For Internet Protocols (SKIP).* Internet Draft, August 14 1996.

[5] S. Bellovin. Problem areas for the IP security protocols. In *Proceedings of 6th USENIX Security Symposium*, San Jose, California, July 22–25 1996.

[6] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[7] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 5(2):364–383, 1986.

[8] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification.* RFC 1883, December 1995.

[9] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[10] D.E. Eastlake and C.W. Kaufman. *Domain Name System Security Extensions.* Internet Draft, August 5 1996.

[11] P. Karn and W.A. Simpson. *The Photuris Session Key Management Protocol.* Internet Draft, June 1996.

[12] D. Knuth. *The Art of Computer Programming: Volume 2, Seminumerical Algorithms.* Addison-Wesley Publishing Company, 2nd edition, 1981.

[13] J.B. Lacy, D.P. Mitchell, and W.M. Schell. CryptoLib: Cryptography in software. In *Proceedings of USENIX Unix Security Symposium IV*, pages 1–17, Santa Clara, California, October 4–6 1993.

[14] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley Publishing Company, 1996.

[15] National Bureau of Standards, U.S. Department of Commerce, Washingtion, D.C. *Data Encryption Standard.* FIPS Pub 46, January 15 1977.

[16] National Bureau of Standards, U.S. Department of Commerce, Washingtion, D.C. *DES Modes of Operations.* FIPS Pub 81, December 1980.

[17] National Institute of Standards, U.S. Department of Commerce, Washingtion, D.C. *Secure Hash Standard.* FIPS Pub 180, April 1993.

[18] H.K. Orman. *The OAKLEY Key Determination Protocol.* Internet Draft, May 1996.

[19] C. Partridge. *Using the Flow Label Field in IPv6.* RFC 1809, June 14 1995.

[20] L.L. Peterson and B.S. Davie. *Computer Networks — A Systems Approach.* Morgan Kaufmann Publishers, 1996.

[21] J. Postel. *User Datagram Protocol.* RFC 768, August 28 1980.

[22] J. Postel. *Internet Protocol: DARPA Internet Program Protocol Specification.* RFC 791, September 1981.

[23] R. Rivest. *The MD5 Message-Digest Algorithm.* RFC 1321, April 16 1992.

[24] W. Rosenberry, D. Kenny, and G. Fisher. *Understanding DCE.* O'Reilley & Associates, Inc., 1992.

[25] J.G. Steiner, C. Neuman, and J.I. Schiller. *Kerberos*: An authentication service for open network systems. In *Proceedings of USENIX Winter Conference*, pages 191–202, Dallas, TX, February 1988.

[26] Sun Microsystems, Inc. *Remote Procedure Call Protocol Specification Version 2.* RFC 1057, June 1988.

[27] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated, Volume 2 — The Implementation.* Addison-Wesley Publishing Company, 1995.

[28] CCITT Recommendation X.509 The Directory—Authentication framework, 1988. See also ISO/IEC 9594-8, 1989.

[29] L Zhang, S.E. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource ReSerVation Protocol. *IEEE Network Magazine*, 9(5), 1993.