Digital Signatures for Flows and Multicasts*

Chung Kei Wong Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188

June 14, 1999 (revised)

Abstract

We present chaining techniques for signing/verifying multiple packets using a single signing/verification operation. We then present flow signing and verification procedures based upon a tree chaining technique. Since a single signing/verification operation is amortized over many packets, these procedures improve signing and verification rates by one to two orders of magnitude compared to the approach of signing/verifying packets individually. Our procedures do not depend upon reliable delivery of packets, provide delay-bounded signing, and are thus suitable for delay-sensitive flows and multicast applications. To further improve our procedures, we propose several extensions to the Feige-Fiat-Shamir digital signature scheme to substantially speed up both the signing and verification operations, as well as to allow "adjustable and incremental" verification. The extended scheme, called eFFS, is compared to four other digital signature schemes (RSA, DSA, ElGamal, Rabin). We compare their signing and verification times, as well as key and signature sizes. We observe that (i) eFFS is the fastest in signing (by a large margin over any of the other four schemes) and as fast as RSA in verification (tie for a close second behind Rabin), (ii) eFFS allows a tradeoff between memory and signing/verification time, and (iii) eFFS allows adjustable and incremental verification by receivers.

1. Introduction

Data confidentiality, authenticity, integrity, and non-repudiation are basic concerns of securing data delivery over an insecure network, such as the Internet. *Confidentiality* means that only authorized receivers will get the data; *authenticity*, an authorized receiver can verify the identity of the data's source; *integrity*, an authorized receiver can verify that received data have not been modified; *non-repudiation*, an authorized receiver can prove to a third party the identity of the data's source.[1]

Most investigations on securing data delivery over packet networks have focused on unicast delivery of data sent as independent packets. Exceptions include recent papers on scalable secure multicasting [1, 13, 20] and a flow-based approach to datagram security [14]. All of these papers are mainly concerned with data confidentiality.

In this paper, our concerns are data authenticity, integrity and non-repudiation for delay-sensitive packet flows, particularly flows to be delivered to large groups of receivers. For an individual message (packet), these concerns can be addressed by one of many available digital signature schemes [6, 15, 17, 19]. However, these schemes are not efficient enough for signing/verifying packets individually for delay-sensitive flows, such as packet video.

In the Internet, multicast has been used successfully to provide an efficient, best-effort delivery service to large groups [2]. Consider a packet flow multicasted to a group of receivers. A consequence of best-effort delivery is that many receivers will not receive all of the packets in the multicasted flow. Furthermore, many multicast applications allow receivers to have widely varying capabilities (e.g., to receive layered video and audio transmissions) or needs (e.g., to receive different stock quotes, news, etc.). Consequently, receivers get different subsequences of packets from the same multicasted flow.

1.1. Existing techniques for signing flows

Conceptually, a digital signature scheme is defined by functions for key generation, signing, and verification. The signer (sender) uses the key generation function to create a pair of keys, a signing key, k_s, and a verification key, k_v. The signer keeps the signing key private, and makes the verification key publicly known to all verifiers (receivers).[2]

To sign a message m using signing key k_s, the signer calls the signing function which returns the signature of message m. The signer then sends the signed message, con-

*Research sponsored in part by Texas Advanced Research Program grant no. 003658-063 and by NSA INFOSEC University Research Program grant no. MDA904-98-C-A901. An early version of this paper appears in *Proceedings IEEE ICNP '98*, Austin, TX, October 1998.

[1] In the balance of this paper, we use "receiver" to mean "authorized receiver" unless otherwise stated.

[2] The signing and verification keys are also referred to as private and public keys, respectively.

sisting of message m and its signature, to verifiers. Having received the signed message, a verifier calls the verification function with key k_v. If the verification function returns true, then the verifier concludes that the signer did sign the message and the message has not been altered. Moreover, the signer cannot deny having signed the message (non-repudiation).

In practice, a message digest function, such as MD5 [18], is first applied to the message to generate a fixed-size message digest which is independent of message size. Signing a message means signing the digest of the message. (MD5 message digests are 128 bits long.)

A *flow* is a sequence of packets characterized by some attribute [16, 21]. Packets in a flow may be obtained from segmenting the bit stream of digitized video, digitized audio, or a large file. They may also be related data items, such as stock quotes, news, etc., generated by the same source.

It is easy and efficient to sign an *all-or-nothing* flow, that is, a flow whose entire content is needed before any part of it can be used, e.g., a long file. In this case, the signer simply generates a message digest of the entire flow (file) and signs the message digest.

Most applications, however, create flows that are not all-or-nothing. That is, a receiver needs to verify individual packets (or, more generally, application data units) and use them before the entire flow is received. For these flows, a straightforward solution is to sign each packet individually and each packet is verified individually by receivers. This solution is called the *sign-each* approach.

The sign-each approach is computationally expensive. The signing rate and verification rate are at most $1/(T_d(l) + T_{sign})$ and $1/(T_d(l) + T_{verify})$ packets per second, respectively, where $T_d(l)$ is the time to compute the message digest of an l-byte packet, T_{sign} is signing time, and T_{verify} is verification time for the message digest of a packet. The signing and verification rates,[3] in packets per second, of two widely used digital signature schemes, RSA [19][4] and DSA [15], with 512-bit modulus and using 100% processor time of a Pentium II 300 MHz machine are shown below.

packet size	Signing rate		Verification rate	
(bytes)	RSA	DSA	RSA	DSA
512	78.8	176	2180	128
1024	78.7	175	1960	127
2048	78.0	172	1620	126

If a slower machine is used, or only a fraction of processor time is available for signing/verification (e.g., a receiver machine has only 20% processor time for verification because the other 80% is needed for receiving and processing packets), then the rates should be decreased proportionally.

[3]The signing and verification rates are rates for signing and verifying 128-bit MD5 message digests of packets.

[4]In this paper, we use e=3 in RSA to obtain its fastest verification time without affecting its signing time.

The signing rate is not important for a *non-real-time generated* flow, i.e., a flow whose entire content is known in advance (such as stored video). This is because packets in the flow can be signed in advance. For a real-time generated flow, however, the signing rate must be higher than the packet generation rate of the flow. Furthermore, for delay-sensitive flows, real-time generated or not, the verification rate is important. From the above table, we see that the signing and verification rates of the sign-each approach, using either RSA or DSA, are probably inadequate for many applications.

Two techniques were previously proposed for signing digital streams [7] which, at first glance, may be used for signing packet flows. To describe the technique in [7] for signing a non-real-time generated flow, consider a sequence of m packets. The sender first computes message digest D_m of packet m (the last packet) and concatenates packet $m-1$ and D_m to form augmented packet $m-1$. Then, for $i = 1, ..., m-2$, the sender computes message digest D_{m-i} of augmented packet $m-i$, and concatenates packet $m-i-1$ and D_{m-i} to form augmented packet $m-i-1$. Message digest D_1 of augmented packet 1 is computed and signed. In this manner, only one expensive signing/verification operation is needed for the sequence of m packets. However, a necessary condition for using the above technique is the following *get-all-before* requirement: To verify packet i in the sequence, a receiver must have received every packet from the beginning of the sequence.

For a real-time generated flow, a similar technique is suggested in [7] with the same get-all-before requirement. For a sequence of m packets, only one expensive signing/verification operation is needed, plus one inexpensive *one-time signature* signing/verification for each packet in the sequence. However, since one-time signatures and keys are very large, this technique has a large communication overhead (around 1000 bytes per packet) [9, 10].

The get-all-before requirement of both techniques in [7] is too strong for practical Internet applications. Reliable packet delivery is not used by many applications for flows and multicasts. For example, reliable delivery is generally not used for video and audio flows due to the extra delays associated with retransmissions; either losses are tolerated or forward error correction techniques are used instead.

For large-scale multicast applications, reliable delivery of multicast packets is a difficult problem [5]. Moreover, even if reliable multicasting is available, receivers with different needs/capabilities may choose to get different subsequences of packets in a multicasted flow. In short, the get-all-before requirement is not satisfied.

1.2. Characteristics and requirements

We have observed various characteristics in the delivery of flows and multicasts by an unreliable packet network, such as the Internet. They are summarized below:

- Each packet in a flow may be used as soon as it is received.
- A receiver may get only a subsequence of the packets in a flow. Different receivers may get different subsequences.
- Delay sensitive flows require fast processing at receivers. Real-time generated flows require fast processing at senders as well.
- For a multicasted flow, many receivers are limited in resources (processing capacity, memory, communication bandwidth, etc.) compared to the sender, which is typically a dedicated server machine. In some environments, both senders and receivers may be limited in resources, e.g., mobile computers using wireless communications.
- Receivers may have widely different capabilities/resources. For example, receivers may be personal digital assistants, notebook computers, or desktop machines. Moreover, the resources available to a receiver for verifying signatures may vary over time.

Given the above characteristics, we design procedures for signing and verifying flows in Section 2 as well as a digital signature scheme in Section 3 to meet the following requirements:

- The signing procedure is efficient and, for real-time generated flows, delay bounded.
- The verification procedure is efficient (since many receivers have limited resources).
- Packets in a flow are *individually verifiable*.
- Packet signatures are small (i.e., small communication overhead).
- *Adjustable and incremental verification*: The verification operation is adjustable to the amount of resources a receiver has. It allows a receiver/verifier to verify a message at a lower security level using less resources, and later increase the security level by using more resources (e.g., if the message is important).

1.3. Contributions of this paper

In Section 2, we first describe and compare two chaining techniques (star and tree) for signing/verifying multiple packets using a single signing/verification operation (without the get-all-before requirement in [7]). We then present flow signing and verification procedures based upon the tree chaining technique. Since a single signing/verification operation is amortized over many packets, these procedures improve signing and verification rates by one to two orders of magnitude compared to the sign-each approach. The signing procedure also provides delay-bounded signing. Thus the procedures can be used for delay-sensitive flows.

Since signed packets in our procedures are individually verifiable, the procedures can be used to reduce the workload of *any machine that sends out a large number of signed packets to one or more destinations. There is no requirement that these packets belong to flows.* However, for packets that belong to a flow, the workload of the flow's receiver(s) is also reduced.

In Section 3, we turn our attention to improving the signing and verification operations in the procedures. Specifically, we present several extensions to the Feige-Fiat-Shamir digital signature scheme to speed up both signing and verification as well as to allow adjustable and incremental verification. In Section 4, the extended Feige-Fiat-Shamir (eFFS) scheme is compared to four well-known signature schemes [6, 15, 17, 19]. We compare their signing and verification times, as well as key and signature sizes. We observe that (i) eFFS is the fastest in signing (by a large margin over any of the other four schemes) and as fast as RSA in verification (tie for a close second behind Rabin), (ii) eFFS allows a tradeoff between memory and signing/verification time, and (iii) eFFS allows adjustable and incremental verification by receivers.

2. How to Sign a Flow

To digitally sign/verify delay-sensitive flows, the sign-each approach is computationally too expensive for many applications, particularly those applications that generate packet flows in real time.

As an alternative to the sign-each approach, we present two chaining techniques (star and tree) for providing authenticity to a group of packets, called a *block*, using a single signing operation. The basic idea is to compute a block digest which is signed. In order to make packets *individually verifiable*, each packet needs to carry its own authentication information consisting of the signed block digest (*block signature*) together with some chaining information as proof that the packet is in the block.

2.1. Star chaining

Consider m packets that constitute a block. In star chaining, the block digest is simply the message digest of the m packet digests (listed sequentially). Let $h(\cdot)$ denote the message digest function being used (e.g., MD5). Consider, for example, a block of eight packets with packet digests $D_1, \ldots, D_8$. The block digest is $D_{1-8} = h(D_1, \ldots, D_8)$, and the block signature, $sign(D_{1-8})$, is the block digest signed with some digital signature scheme (such as RSA, DSA or eFFS).

The relationship between the packet digests and the block digest can be represented by a one-level rooted tree, called an *authentication star*. Figure 1 illustrates an authentication star for eight packets, with packet digests at leaf nodes, and the block digest at the root.

For packets to be individually verifiable, each packet needs its own authentication information. Such authenti-

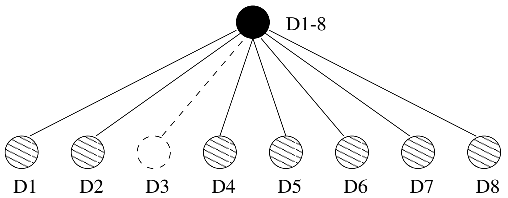


Figure 1. Star chaining technique.

cation information, called *packet signature*, consists of the block signature, the packet position in the block, and the digests of all other packets in the block. (We use the term *chaining overhead* to refer to all information in a packet signature except the block signature.)

Suppose the third packet in the above example is received. Its authenticity can be individually verified as follows. The verifier computes the digest D_3' of the packet received, and then the block digest $D_{1-8}' = h(D_1, D_2, D_3', D_4, \ldots, D_8)$, where $D_1, D_2, D_4, \ldots, D_8$ are carried in the packet signature. The verifier then calls the verification operation to verify D_{1-8}', i.e., to determine whether D_{1-8}' is equal to block digest D_{1-8} in block signature $sign(D_{1-8})$. The packet is verified if the verification operation returns true, i.e., $D_{1-8}' = D_{1-8}$.

Suppose the third packet is the first in the block to arrive and its authenticity has been verified. Afterwards, the verifier knows every node in the authentication star, i.e., all nodes in the authentication star are verified and can be cached. With caching, when another packet in the block arrives later, say the sixth packet, the verifier only needs to compute the digest D_6' of the packet received and compare it to the verified node D_6 in the authentication star. If they are equal, the packet is verified.

2.2. Tree chaining

Tree chaining subsumes star chaining as a special case. With tree chaining, the block digest is computed as the root node of an *authentication tree*.[5] Consider, for example, a block of eight packets with packet digests $D_1, \ldots, D_8$. The packet digests are the leaf nodes of a degree two (binary) authentication tree, with other nodes of the tree computed as message digests of their children, as shown in Figure 2. For example, the parent of the leaves D_1 and D_2 is $D_{12} = h(D_1, D_2)$ where $h(\cdot)$ is the message digest function being used. The root is the block digest, with the block signature being the signed block digest.

For a packet to be individually verifiable, each packet needs to carry its own authentication information (packet signature). In tree chaining, a packet signature consists of the block signature, the packet position in the block, and the siblings of each node in the packet's path to the root. (Again we use the term *chaining overhead* to denote all information in a packet signature except the block signature.)

[5]Tree chaining was first presented in [11]. Any rooted tree can be used as an authentication tree with packet digests at leaf nodes and the block digest at the root. In particular, there is no need to use a balanced tree.

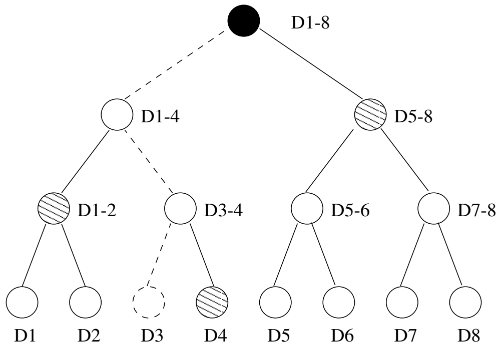


Figure 2. Tree chaining technique.

To verify a packet individually, a verifier needs to verify its path to the root. Consider, for example, the dashed path in Figure 2 for the third packet. Each node in the path needs to be verified. A verifier computes the digest D_3' of the received packet, and then each of its ancestors in the tree. That is, $D_{3-4}' = h(D_3', D_4)$, $D_{1-4}' = h(D_{1-2}, D_{3-4}')$, and $D_{1-8}' = h(D_{1-4}', D_{5-8})$, where D_4, D_{1-2} and D_{5-8} are carried in the packet signature. The verifier then calls the verification operation to determine whether D_{1-8}' is equal to block digest D_{1-8} in block signature $sign(D_{1-8})$. The packet is verified if the verification operation returns true, i.e., $D_{1-8}' = D_{1-8}$.

Suppose the third packet is the first in the block to arrive. After verifying it, the verifier knows the following nodes[6] in the authentication tree: $D_3, D_4, D_{1-2}, D_{3-4}, D_{1-4}, D_{5-8}$ and the block digest D_{1-8}. These are verified nodes which can be cached. By caching verified nodes, the verifier only needs to compute each node in the authentication tree at most once.

For example, after verifying the third packet, to verify the sixth packet which arrives later, the verifier computes the digest of the packet received, D_6', its parent $D_{5-6}' = h(D_5, D_6')$, and its grandparent $D_{5-8}' = h(D_{5-6}', D_{7-8})$. If D_{5-8}' is equal to the cached node D_{5-8}, the sixth packet is verified.

2.3. Comparison of chaining techniques

We performed experiments on a Pentium II 300 MHz machine running Linux, and compared star and tree chaining. We used MD5 as the message digest function [18] for generating 128-bit message digests. Figure 3 shows the MD5 computation time versus input size. We observe that the MD5 computation time can be regarded as a linear function in input size (for a large input, i.e., 1024 bytes or more).

For each chaining technique, an authentication tree is first built for a block of packets,[7] i.e., each node is computed as the message digest of its children. The time to build an

[6]Some are carried in the packet signature and the others have been computed.

[7]We will use "tree" instead of "tree/star" since star chaining is a special case of tree chaining.

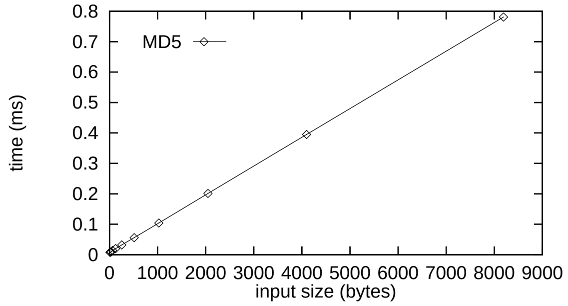


Figure 3. MD5 computation time (ms).

authentication tree (excluding time to compute packet digests for leaf nodes) is called the *tree build time*. The block signature is then obtained by signing the block digest at the root. After that, the packet signature of each packet is built from the authentication tree and the block signature. The time to build a packet signature is called *packet signature build time*. The *chaining time* for a block at a signer is the sum of tree build time and packet signature build time for all packets in the block.[8] Figure 4(a) shows the chaining time for a block of packets at a signer.

Consider the total signing time for all packets in a block which is the block's chaining time plus the signing time of the block digest. The block digest signing time is 12.7 ms using 512-bit RSA and 5.6 ms using 512-bit DSA. For a block of 16 packets, from Figure 4(a), the chaining time is 0.21 ms for a degree two authentication tree. The total signing time is $0.21 + 12.7 = 12.9$ ms using 512-bit RSA. Thus the average signing time for one packet is $12.9/16 = 0.81$ ms, which is less than 1/15 of the block digest signing time using 512-bit RSA.

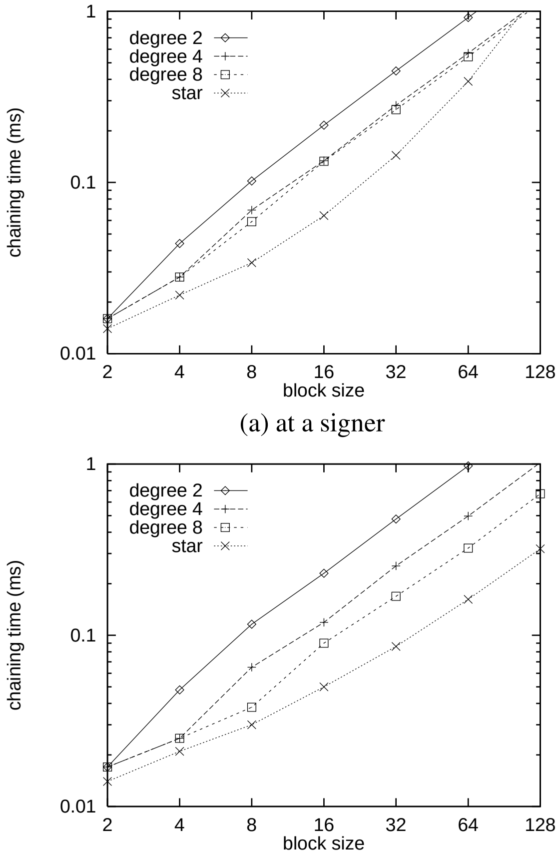
To verify packets in a block, an authentication tree is built from packet signatures as packets arrive. The *chaining time* for a block at a verifier is the sum of tree build time and time to verify chaining information in the packet signature of every packet in the block.[9] Figure 4(b) shows the chaining time for a block of packets at a verifier *with caching* of verified nodes.

Consider the total verification time for all packets in a block which is the block's chaining time plus the verification time of the block signature. The signature verification time is 0.40 ms using 512-bit RSA and 7.6 ms using 512-bit DSA. For a block of 16 packets, from Figure 4(b), the chaining time is 0.24 ms for a degree two authentication tree. The total verification time is $0.24 + 0.40 = 0.64$ ms using 512-bit RSA. Thus the average verification time for one packet is $0.64/16 = 0.04$ ms, which is 1/10 of the signature verification time using 512-bit RSA.

From Figure 4(a), note that for any block size smaller than or equal to 64 packets, star chaining takes less time

[8]Note that chaining time does not include time to compute packet digests for leaf nodes and time to sign the block digest.

[9]Note that chaining time does not include time to compute packet digests for leaf nodes and time to verify the block signature.



(b) at a verifier

Figure 4. Chaining time (ms) for a block at a signer and at a verifier (with caching of verified nodes).

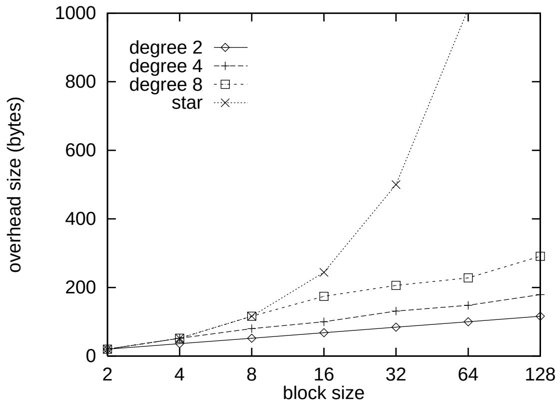


Figure 5. Average chaining overhead size (bytes) per packet.

at a signer than tree chaining (degrees two to eight). However, for a larger block size, star chaining takes more time at a signer than tree chaining, because the chaining time for a star is $O(m^2)$ and the chaining time for a tree is $O(m \log(m))$ where m denotes block size.

As shown in Figure 4(b), star chaining takes less time at a verifier than tree chaining for all block sizes.

For each chaining technique, a packet signature has two parts, the block signature and the chaining overhead. In general, if a tree is not balanced and full, the chaining overhead sizes of different packets are different. Figure 5 shows the average chaining overhead size per packet. The size of the block signature is not included in Figure 5 since it depends on which signature scheme is used (e.g., the block signature is 64 bytes for 512-bit RSA, and 40 bytes for 512-bit DSA).

From Figure 5, note that the chaining overhead of star chaining is much greater than tree chaining for block sizes larger than eight. If a small communication overhead is important, packet signature sizes should be reduced. We recommend the use of degree two tree chaining which requires the smallest chaining overhead. (From Figure 4, a degree two tree has a slightly higher chaining time than the alternatives, but the difference is insignificant because chaining time is much smaller than signing/verification time of the block digest. See Figures 7 and 8 in Section 2.4.)

2.4. Flow signing and verification procedures

A flow is signed by partitioning it into blocks of packets, with each block signed using tree chaining. For a non-real-time generated flow, blocks are of the same size m, chosen to be a power of the authentication tree degree d. For a real-time generated flow, the packet generation rate is time-varying for many applications, such as compressed video and voice-activated audio. For these applications, partitioning the flow into fixed size blocks may lead to an unpredictable (perhaps unbounded) signing delay. Instead, the flow is partitioned by fixed time periods, and packets generated in the same time period are grouped into a block (see Figure 6).

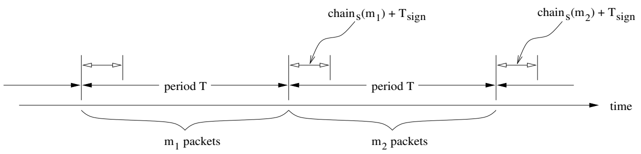


Figure 6. Signing a real-time generated flow.

For both real-time and non-real-time generated flows, the flow verification procedure is the same. For the first received packet in a block, i.e., the block signature carried in the packet signature is new to a verifier, the verifier computes the packet digest, and every ancestor of the packet digest.[10] For the computed block digest (the root of authentication tree), the verifier calls the verification operation to verify that it is equal to the block digest in the block signature. If so verified, then all computed nodes and their children are verified and cached.

For a packet that is not the first received packet in a block, the verifier computes the packet digest. If the packet digest has been cached and the cached value is equal to the computed packet digest, then the packet is verified. Otherwise, the verifier computes every non-cached ancestor of the packet digest. For the highest non-cached node computed, the verifier then computes its parent. If the computed parent and its cached value are equal, the packet is verified and all computed nodes and their children are verified and cached.

We implemented the flow signing and verification procedures (see Appendix) and performed experiments on a Pentium II 300 MHz machine running Linux. We used MD5 as the message digest function, and experimented with both 512-bit RSA and 512-bit DSA as the signature scheme for block signatures.

Figure 7 and Figure 8 show, respectively, the flow signing and verification rates for 1024-byte packets.[11] Note that tree and star chaining are one to two orders of magnitude more efficient than the sign-each approach. The flow signing and verification rates increase with block size. However, the rates vary only slightly with the chaining technique used and with the tree degree in tree chaining. Since degree two tree chaining has the lowest chaining overhead (packet signature size), we recommend the use of degree two tree chaining.

Figure 9 and Figure 10 show, respectively, the flow signing and verification rates for packets of size 512, 1024, or 2048 bytes. We used degree two tree chaining. From the figures, observe that the flow signing and verification rates decrease as the packet size increases. It is because more time is needed to compute the message digest of a larger packet. The decrease is more pronounced when the block size used is large, since more time is used to compute packet digests for a large block than a small block. Observe also that the flow signing and verification rates increase with block size and the increase is greater for a smaller packet size.

2.5. Bounded delay signing

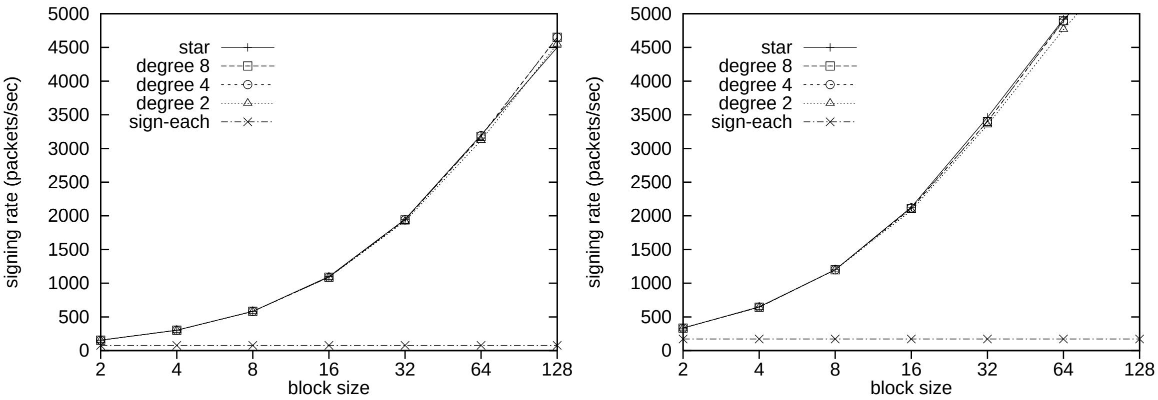
Consider Figure 6. Assume that, in period T, at most m packets are generated and their packet digests computed. The time for signing a block of m packets is $chain_s(m) + T_{sign}$ where $chain_s(m)$ is the chaining time for a block of m packets at a signer, and T_{sign} is the block digest signing time. Therefore, the delay of any packet within the block is at most $D_s = T + chain_s(m) + T_{sign}$.

Table 1 shows the delay upper bound D_s for period $T =$ 50 ms. Note that the upper bound is fairly insensitive to block size since the block's chaining time is much smaller than the block digest signing time.

For a given application with a specified upper bound, D_s, for signing a real-time generated flow at a known packet rate, we can work backwards and derive an appropriate value for the parameter T needed for the signing procedure of a real-time generated flow. Observe, from Figure 6, that

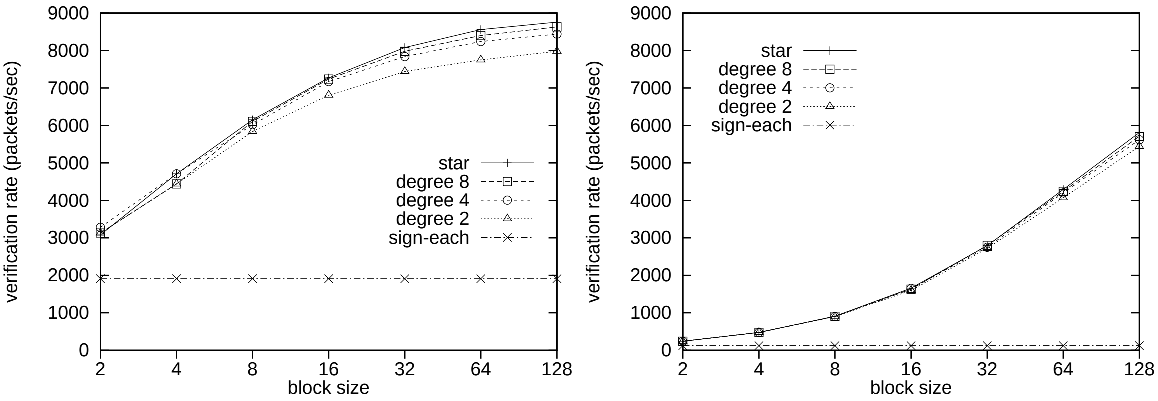
[10] An ancestor node is computed as the message digest of its children which are either computed or carried in the packet signature.

[11] Verification rates were computed assuming no packet loss.



(a) using 512-bit RSA (b) using 512-bit DSA

Figure 7. Flow signing rate (packets/sec) for 1024-byte packets.



(a) using 512-bit RSA (b) using 512-bit DSA

Figure 8. Flow verification rate (packets/sec) for 1024-byte packets.

	number of packets generated in period T						
	2	4	8	16	32	64	128
tree deg 2	62.9	62.9	62.9	63.1	63.3	63.8	64.9
tree deg 4	62.8	62.9	62.9	63.0	63.2	63.5	64.2
tree deg 8	62.8	62.9	62.9	63.0	63.2	63.5	64.2

Table 1. Signing delay bound (ms) for period T = 50 ms using 512-bit RSA.

T must be larger than $T_{sign} + chain_s(m)$, and D_s must be larger than $2(T_{sign} + chain_s(m))$.

2.6. Selecting a digital signature scheme

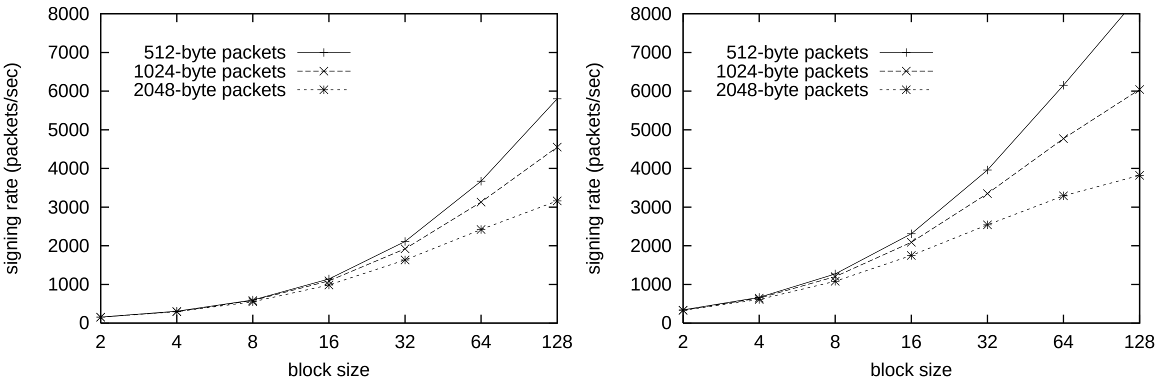
For non-real-time generated flows, signing efficiency is not critical. Thus a signature scheme with an efficient verification operation, such as RSA, can be used in the flow signing and verification procedures. For real-time generated flows, however, it is critical that both signing and verification are highly efficient. Furthermore, in choosing a digital signature scheme, we must also consider machine capabilities (sender and receiver), as well as the fraction of processor time available for signing and verification.

Using 100% processor time of a Pentium II 300 MHz machine, the flow signing and verification rates for 1024-byte packets, degree two tree chaining, and block size sixteen are shown below.

	signing rate	verification rate
512-bit RSA	1090 packets/sec	7030 packets/sec
512-bit DSA	2140 packets/sec	1660 packets/sec

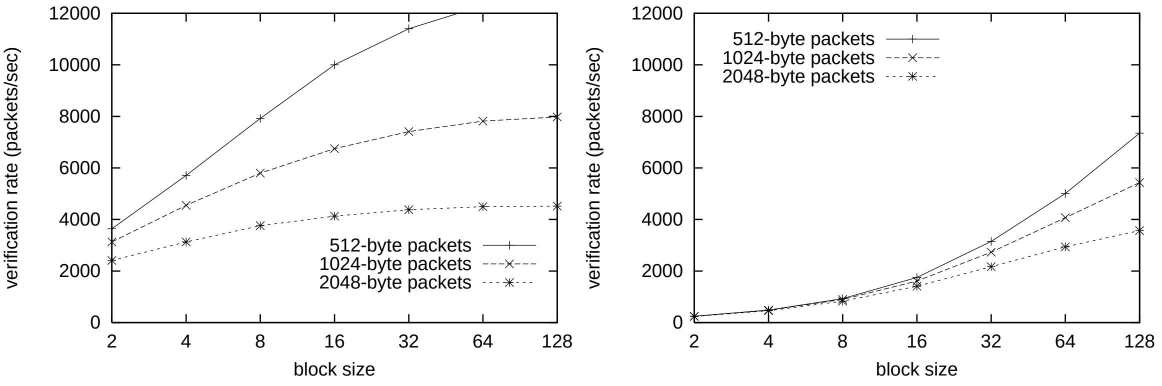
Note that using DSA, the flow verification rate is smaller than the flow signing rate. This is undesirable because receivers/verifiers are generally less powerful than the signer/sender, e.g., the receivers may be personal digital assistants or low-end notebook computers. Using RSA, the flow signing rate may not be high enough for some applications. Although we can increase the flow signing and verification rates by using a longer period or a larger block size, neither option is desirable. A larger block size increases the chaining overhead (packet signature size). A longer period increases the delay for signing real-time generated flows.

To obtain a signature scheme better than RSA and DSA for signing/verifying flows, we propose several extensions to the Feige-Fiat-Shamir (FFS) signature scheme. The ex-



(a) using 512-bit RSA (b) using 512-bit DSA

Figure 9. Flow signing rate (packets/sec) for degree two tree chaining.



(a) using 512-bit RSA (b) using 512-bit DSA

Figure 10. Flow verification rate (packets/sec) for degree two tree chaining.

tended scheme, called eFFS, is presented in the next section. The eFFS scheme has a very efficient signing operation (much more efficient than those of RSA and DSA) and a verification operation as efficient as that of RSA. A performance comparison of eFFS with five other signature schemes (including FFS, RSA and DSA) is given in Section 4.

3. The eFFS Signature Scheme

In Section 3.1, we first describe the basic Feige-Fiat-Shamir (FFS) signature scheme [3, 4]. The eFFS signature scheme is derived from FFS with two kinds of extensions. Three extensions to speed up the signing and verification operations of FFS are presented in Section 3.2. An extension to provide *adjustable* and *incremental* signature verification is presented in Section 3.3.

3.1. Feige-Fiat-Shamir signature scheme

In the basic FFS signature scheme with parameter (k, t) [3, 4], each signer chooses two large primes p and q, and computes modulus $n = pq$. Then, the signer chooses k integers $v_1, \ldots, v_k$ (or k integers $s_1, \ldots, s_k$), and computes $s_1, \ldots, s_k$ (or $v_1, \ldots, v_k$) by $s_i^2 = v_i^{-1} \bmod n$. The signing key is $\{s_1, \ldots, s_k, n\}$ and the verification key is $\{v_1, \ldots, v_k, n\}$.

To sign message m, the signer does the following steps: (1) choose t random integers, $r_1, \ldots, r_t$, between 1 and n, and compute $x_i = r_i^2 \bmod n$ for $i = 1, \ldots, t$; (2) calculate the message digest $h(m, x_1, \ldots, x_t)$ where the message digest function $h(\cdot)$ is public knowledge and the message digest is at least $k \times t$ bits long; let $\{b_{ij}\}$ be the first $k \times t$ bits of the message digest where $i = 1, \ldots, t$, and $j = 1, \ldots, k$; (3) compute $y_i = r_i \times (s_1^{b_{i1}} \times \ldots \times s_k^{b_{ik}}) \bmod n$ for $i = 1, \ldots, t$. The signature of message m consists of $\{y_i\}$ for $i = 1, \ldots, t$ and $\{b_{ij}\}$ for $i = 1, \ldots, t$ and $j = 1, \ldots, k$.

To verify the signature of message m, a verifier computes $z_i = y_i^2 \times (v_1^{b_{i1}} \times \ldots \times v_k^{b_{ik}}) \bmod n$ for $i = 1, \ldots, t$. The signature is valid if and only if the first $k \times t$ bits of $h(m, z_1, \ldots, z_t)$ are equal to the $\{b_{ij}\}$ received. (It can be shown that z_i computed by the verifier is equal to x_i at the signer.)

The security level of FFS(k, t) depends on the following: (1) the size of modulus n (i.e., the size of the primes p and q), and (2) the value of product kt. A system with a larger modulus is more secure, and a system with a larger kt product is more secure. If two systems have the same modulus and same kt product (but different k and t values), then their security levels are about the same.

	$t=1$		$t=2$		$t=4$	
	key	sig	key	sig	key	sig
$kt=64$	4160	72	2112	136	1088	264
$kt=128$	8256	80	4160	144	2112	272

Table 2. FFS signing/verification key and signature sizes (bytes) with 512-bit modulus.

Assuming $|v_i| = |n|$ and $|s_i| = |n|$, where $|x|$ denotes the size of x in bits, the signing/verification key size is $(k+1)\times|n|$ bits, and the signature size is $t\times|n|+k\times t$ bits. The signing/verification key size only depends on k, but the signature size is proportional to t. Thus, for a fixed kt product, we can reduce the signature size by using a smaller t (and a larger k). For $t=1$, the signature size is minimized, but the signing/verification key size is maximized. Table 2 shows the signing/verification key size and signature size of FFS with 512-bit modulus.

3.2. Extensions to speed up FFS

Small verification key (small v-key)

In FFS, the sizes of signing key components $\{s_i\}$ affect the signing time, and the sizes of verification key components $\{v_i\}$ affect the verification time. An improvement suggested in [12] is to use the first k prime numbers as verification key components $\{v_i\}$. However, since not every prime number p satisfies the condition that there exists an integer s such that $s^2 = p^{-1} \bmod n$, we propose to use the first k prime numbers that satisfy this condition as verification key components.[12] This extension reduces both the verification time and the verification key size.

Chinese remainder theorem (crt)

The signing operation in FFS involves the computing of $y_i = r_i \times (s_1^{b_{i1}} \times \ldots \times s_k^{b_{ik}}) \bmod n$. For $n = pq$, from the Chinese Remainder Theorem, a signer can compute y_i from a_i and b_i using the following formula: $y_i = ((a_i - b_i)\times q \times q_p^{-1} + b_i) \bmod n$ where $q_p^{-1} = q^{-1} \bmod p$, $a_i = r_i \times (s_1^{b_{i1}} \times \ldots \times s_k^{b_{ik}}) \bmod p$ and $b_i = r_i \times (s_1^{b_{i1}} \times \ldots \times s_k^{b_{ik}}) \bmod q$. Thus, instead of computing y_i directly with multiplication operations in mod n, a signer first computes a_i and b_i with multiplication operations in, respectively, mod p and mod q. Then y_i is computed from a_i and b_i. Since multiplication operations in mod p and mod q are more efficient than in mod n, the signing time is decreased.

This Chinese Remainder Theorem improvement can only be used by a signer because knowledge of the factors of modulus n is required. A few hundred bytes of additional memory are needed for storing a few large integers (for 512-bit modulus).

Precomputation (precomp)

A signer can further speed up the signing operation by using more memory. To illustrate the basic idea of this improvement, consider the signing operation with $k = 4$. To sign a message, a signer computes $y_i = r_i \times (s_1^{b_{i1}} \times \ldots \times s_4^{b_{i4}}) \bmod n$, for $i = 1, \ldots, t$. Since $s_1, \ldots, s_4$ do not change from message to message, and $b_{i1}, \ldots, b_{i4}$ are either one or zero, the signer can precompute and store the product (mod n) of every non-empty subset of $\{s_1, \ldots, s_4\}$. Let $S_{b_1\ldots b_4}$ denote the precomputed product $s_1^{b_1} \times \ldots \times s_4^{b_4} \bmod n$. Then, to sign a message, the signer simply computes y_i by $r_i \times S_{b_{i1}\ldots b_{i4}} \bmod n$.

For large k, it is not practical to precompute the product (mod n) of every non-empty subset of $\{s_1, \ldots, s_k\}$. Instead, the signer partitions $\{s_1, \ldots, s_k\}$ into smaller sets and precomputes each of them. If each smaller set contains four s_i, then it is a 4-bit precomputation. Similarly, if each smaller set contains eight s_i, then it is an 8-bit precomputation. For 4-bit precomputation with $k = 128$ and 512-bit modulus, a signer needs to store $128/4 \times (2^4 - 1) = 480$ products. That is, additional memory of 480×512 bits or 31 kilobytes is required. The additional memory required by 8-bit, 12-bit, and 16-bit precomputation are 261 kilobytes, 2.88 megabytes, and 33.6 megabytes, respectively.

Although a similar precomputation can be used in verification, it is not effective with the small v-key extension. This is because when small primes are used as public key components, their products can be computed very efficiently.

Performance comparison

We implemented the three speedup extensions using the large integer arithmetic routines from CryptoLib [8]. Table 3 and Table 4 show the times for signing and verifying (with 512-bit modulus) 128-bit message digests using different speedup extensions for different values of (k, t).[13] The results were obtained on a Pentium II 300 MHz machine running Linux. Note that, for a fixed kt product, the signing/verification time is smaller when t is smaller.

In the experiments to be reported in the balance of this paper, we used *8-bit precomp + crt + small v-key* for eFFS signing, and *small v-key only* for eFFS verification.

3.3. Adjustable and incremental verification

In multicast or group communications, receivers typically have different amounts of resources, and the resources available to a receiver for verification vary over time. It is thus desirable to have an adjustable and incremental signature verification operation. With this extension, a signature

[12] In practice, for k up to 128, the verification key components $\{v_i\}$ are less than 2^{16}, and each component can be stored in 16 bits.

[13] For basic FFS, we specified signing key components $\{s_i\}$. Verification key components $\{v_i\}$ were chosen by CryptoLib.

	eFFS parameter (k,t)					
	$(32,1)$	$(32,2)$	$(64,1)$	$(32,4)$	$(64,2)$	$(128,1)$
basic FFS	3.95	7.87	7.21	15.62	14.35	13.72
small v-key	3.95	7.84	7.21	15.63	14.36	13.72
crt + small v-key	3.13	6.20	5.35	12.44	10.63	9.78
4-bit precomp + crt + small v-key	1.95	3.84	2.99	7.61	5.92	5.08
8-bit precomp + crt + small v-key	1.47	2.87	2.02	5.67	3.98	3.14

Table 3. eFFS signing time (ms) with 512-bit modulus.

	eFFS parameter (k,t)					
	$(32,1)$	$(32,2)$	$(64,1)$	$(32,4)$	$(64,2)$	$(128,1)$
basic FFS	3.65	7.07	7.12	14.01	13.63	13.44
small v-key	0.33	0.62	0.43	1.21	0.81	0.65
4-bit precomp + small v-key	0.32	0.60	0.41	1.16	0.76	0.59
8-bit precomp + small v-key	0.32	0.59	0.40	1.14	0.74	0.57

Table 4. eFFS verification time (ms) with 512-bit modulus.

	kt product		
	$kt = 32$	$kt = 64$	$kt = 128$
1-level signature	1.47	2.02	3.14
2-level signature		2.87	3.98
4-level signature			5.67

Table 5. eFFS t-level signature signing times (ms).

To	level 1	level 2
From level 0	0.42	0.81
From level 1		0.40

(a) 2-level signature

To	level 1	level 2	level 3	level 4
From level 0	0.34	0.63	0.93	1.22
From level 1		0.30	0.60	0.89
From level 2			0.30	0.60
From level 3				0.31

(b) 4-level signature

Table 6. eFFS incremental verification times (ms) for kt = 128.

can be verified at different security levels. An adjustable verification allows a receiver to verify a message at a lower security level using less resources. An incremental verification allows a receiver to verify a message at a lower security level first, and later increase the security level by using more resources (e.g., if the message is important).

Since the security level of a signature scheme depends on its parameters, e.g., the modulus size, an obvious approach to provide adjustable and incremental verification is to use multiple keys (with different modulus sizes) to generate multiple signatures for different security levels. To verify at a lower security level, the verification key with a shorter modulus size is used to verify the corresponding signature. This approach is simple but very inefficient. In the following, we design an extension to FFS that provides adjustable and incremental verification efficiently.

Our extension to provide adjustable and incremental verification is to use t greater than one, and to include $\{x_i\}$ for $i = 2, \ldots, t$ in signatures. This is called a *t-level signature*.[14] This extension is as secure as the original scheme because $x_i = y_i^2 \times (v_1^{b_{i1}} \times \ldots \times v_k^{b_{ik}}) \bmod n$ for $i = 2, \ldots, t$ can be computed easily from the original signature, which consists of $\{b_{ij}\}$ and $\{y_i\}$, together with the verification key $\{v_1, \ldots, v_k, n\}$ which is publicly known.

To verify a t-level signature of message m at security level l of t (where $l \leq t$), a verifier does the following: (1) compute $z_i = y_i^2 \times (v_1^{b_{i1}} \times \ldots \times v_k^{b_{ik}}) \bmod n$ for $i = 1, \ldots, l$, and (2) verify that $z_2, \ldots, z_l$ are equal to $x_2, \ldots, x_l$ respectively, and the first $k \times t$ bits of $h(m, z_1, x_2, \ldots, x_t)$ are equal to the $\{b_{ij}\}$ received.

To increase the verification security level from l_1 to l_2, a verifier does the following: (1) compute $z_i = y_i^2 \times (v_1^{b_{i1}} \times \ldots \times v_k^{b_{ik}}) \bmod n$ for $i = l_1 + 1, \ldots, l_2$, and (2) verify that $z_{l_1+1}, \ldots, z_{l_2}$ are equal to $x_{l_1+1}, \ldots, x_{l_2}$ respectively.

The size of a t-level signature is $kt + (2t - 1) \times |n|$ bits. For 512-bit modulus and product $kt = 128$, a 1-level signature is 80 bytes and a 2-level signature is 208 bytes.

Table 5 shows different t-level signature signing times. For the same kt product, the signing time increases as the t value increases. However, the signing time is still smaller than using multiple keys to implement different security levels. For example, the 2-level signature signing time, which is 3.98 ms for $kt = 128$, is smaller than the time to sign two (original 1-level) signatures, one for $(k,t) = (64,1)$ and the other for $(k,t) = (128,1)$, which is $2.02 + 3.14 = 5.16$ ms.

Table 6 shows the (incremental) verification times from one level to a higher level for a 2-level signature and a 4-level signature with $kt = 128$. In particular, for a 2-level

[14]Note that the original (1-level) signature does not provide adjustable and incremental verification.

signature, a verifier can first verify a message at level 1 of 2 using 0.42 ms processor time, and later increase to level 2 (of 2) by using 0.40 ms additional processor time.

4. Comparison with other Signature Schemes

In this section, we compare eFFS(128,1) to FFS(128,1) as well as four other signature schemes available from CryptoLib [8], namely: DSA [15], ElGamal [6], RSA [19], and Rabin [17]. We compare their key and signature sizes, and signing and verification times. Then, we compare their signing and verification rates for 1024-byte packets when each is used as the signature scheme in our flow signing and verification procedures presented in Section 2. Experiments were performed on a Pentium II 300 MHz machine running Linux. Four different modulus sizes, 384, 512, 768, and 1024 bits, were used in the comparison. (Note that it is difficult to compare the security levels of different signature schemes even if they use the same modulus size.)

4.1. Key and signature sizes

	modulus size (bits)			
	384	512	768	1024
RSA	96	128	192	256
Rabin	96	128	192	256
DSA	136	168	232	296
ElGamal	144	192	288	384
FFS(128,1)	6192	8256	12384	16512
eFFS(128,1)	6192	8256	12384	16512

(a) Signing key size (bytes)

RSA	48	64	96	128
Rabin	48	64	96	128
DSA	164	212	308	404
ElGamal	144	192	288	384
FFS(128,1)	6192	8256	12384	16512
eFFS(128,1)	304	320	352	384

(b) Verification key size (bytes)

RSA	48	64	96	128
Rabin	48	64	96	128
DSA	40	40	40	40
ElGamal	96	128	192	256
FFS(128,1)	64	80	112	144
eFFS(128,1)	64	80	112	144

(c) Signature size (bytes)

Table 7. Signing key, verification key, and signature sizes (bytes) of different signature schemes.

Table 7 shows the signing/verification key and signature sizes. The signing keys are from 96 to 384 bytes in all schemes except FFS and eFFS whose signing keys are much larger, from 6,192 to 16,512 bytes. Note that a signing key is private to a signer. We do not expect the relatively large eFFS signing keys to pose a problem for sources/signers of packets.[15]

In RSA and Rabin, verification keys are from 48 to 128 bytes. In DSA, ElGamal, and eFFS, verification keys are slightly larger, from 144 to 404 bytes. Even for receivers with limited resources, we believe that a verification key as large as 400 bytes would not pose a problem. (Note that without the small v-key extension, FFS verification keys are as large as signing keys.)

The signature of DSA is the smallest and is 40 bytes for all modulus sizes. For all of the other schemes, the signatures are larger and about the same size, 48 to 256 bytes. In particular, the signature sizes of eFFS and the popular RSA are about the same.

4.2. Signing and verification times

Table 8 shows the signing and verification times for a 16-byte message (digest). DSA and ElGamal have been designed to achieve efficient signing (e.g., for use in smartcard applications), and RSA and Rabin have been designed to achieve efficient verification. From Table 8, note that the signing operations of DSA and ElGamal, with times from 3.9 to 18.9 ms, are much more efficient than those of RSA and Rabin, with times from 6.2 to 95.9 ms. On the other hand, the verification operations of RSA and Rabin, with times from 0.14 to 1.14 ms, are much more efficient than those of DSA and ElGamal, with times from 5.1 to 350.3 ms.

Note that the signing and verification operations of FFS are both inefficient. However, eFFS has a signing operation even more efficient than those of DSA and ElGamal, and a verification operation as efficient as that of RSA. This combination of the most efficient signing and highly efficient verification makes eFFS the best choice for most applications.

4.3. Flow signing and verification rates

Table 9 shows the flow signing and verification rates of our flow signing and verification procedures (for 1024-byte packets, degree two tree chaining, block size sixteen, and 100% of processor time of a Pentium II 300 MHz machine). Both DSA and ElGamal have low flow verification rates, rendering them inappropriate for receivers with limited resources, such as personal digital assistants and low-end notebook computers. Both RSA and Rabin have low flow signing rates, rendering them inappropriate for real-time generated flows, such as live video/audio applications. By comparison, eFFS provides high flow signing rates suitable for real-time generated flows while its flow verification rates are also very high.

[15] Such signing keys are indeed too large for small devices, such as smartcards, but it is unlikely that these devices would be sources of packet flows or multicasts.

	modulus size (bits)					modulus size (bits)			
	384	512	768	1024		384	512	768	1024
RSA	6.2	12.7	36.2	79.4	RSA	0.26	0.40	0.70	1.1
Rabin	11.3	19.5	47.5	95.9	Rabin	0.14	0.20	0.38	0.56
DSA	3.9	5.6	10.2	16.3	DSA	5.1	7.6	14.7	24.2
ElGamal	5.1	6.8	12.3	18.9	ElGamal	24.4	51.9	157.5	350.3
FFS(128,1)	8.8	13.7	22.9	38.5	FFS(128,1)	8.5	13.4	22.1	37.3
eFFS(128,1)	2.3	3.1	5.2	8.2	eFFS(128,1)	0.53	0.65	0.82	1.1

(a) Signing time (ms) (b) Verification time (ms)

Table 8. Signing and verifying times (ms) of different signature schemes.

	modulus size (bits)					modulus size (bits)			
	384	512	768	1024		384	512	768	1024
RSA	1940	1090	413	193	RSA	7480	7030	6060	5290
Rabin	1200	739	321	163	Rabin	7960	7610	7010	6430
DSA	2760	2140	1320	874	DSA	2270	1660	949	609
ElGamal	2320	1850	1140	749	ElGamal	600	295	99	45
FFS(128,1)	1550	1070	624	395	FFS(128,1)	1590	1150	633	419
eFFS(128,1)	3920	3140	2160	1610	eFFS(128,1)	6640	6370	5760	5250

(a) Flow signing rate (packets/sec) (b) Flow verification rate (packets/sec)

Table 9. Flow signing and verification rates (packets/sec) for 1024-byte packets, degree two tree chaining, and block size sixteen.

5. Conclusions

We investigated the problem of signing/verifying delay-sensitive packet flows to provide data authenticity, integrity, and non-repudiation for Internet applications. We have designed flow signing and verification procedures, based upon a tree chaining technique, to meet the following requirements: (i) flow signing is efficient and, for real-time generated flows, delay-bounded, (ii) flow verification is efficient (for receivers with limited resources), (iii) packets in a flow are individually verifiable (for best-effort multicast delivery), (iv) packet signatures are small (for a small communication overhead), and (v) verification at a receiver is adjustable to different security levels and can be carried out incrementally (for receivers with limited resources).

We implemented our flow signing and verification procedures and performed experiments to compare different chaining techniques. From experimental results, we recommend the use of degree two (binary) tree chaining since it requires the smallest packet signature size (i.e., smallest communication overhead) while its signing and verification rates are comparable to the rates of other chaining techniques. Our flow signing and verification procedures are very efficient and achieve one to two orders of magnitude improvement compared to the sign-each approach.

Since signed packets in our procedures are individually verifiable, the procedures can be used to reduce the workload of any machine that sends out a large number of signed packets to one or more destinations. There is no requirement that these packets belong to flows. However, for packets that belong to a flow, the workload of the flow's receiver(s) is also reduced.

To further improve our procedures, we propose several extensions to the Feige-Fiat-Shamir digital signature scheme [3, 4] to speed up both the signing and verification operations, as well as to allow adjustable and incremental verification. The extended scheme, called eFFS, is compared to four other digital signature schemes, RSA [19], Rabin [17], DSA [15], and ElGamal [6], on the same computing platform (Pentium II 300 MHz machine running Linux).

The signing operation of eFFS is by far the most efficient of all the schemes compared. The verification operation of eFFS is as efficient as that of RSA (tie for a close second behind the verification operation of Rabin). In addition to efficient signing and verification, we have extended the eFFS scheme to allow a receiver to efficiently carry out adjustable and incremental verification. Such a capability is useful for large-scale multicast applications with a variety of receivers including some with limited resources.

References

[1] Tony Ballardie. *Scalable Multicast Key Distribution, RFC 1949*, May 1996.

[2] Stephen E. Deering. Multicast Routing in Internetworks and Extended LANs. In *Proceedings of ACM SIGCOMM '88*, August 1988.

[3] Uriel Feige, Amos Fiat, and Adi Shamir. Zero Knowledge Proofs of Identity. In *Proc. of the 19th Annual ACM Symposium on Theory of Computing*, 1987.

[4] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology — CRYPTO '86*, pages 186–194, 1987.

[5] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *Proceedings of ACM SIGCOMM '95*, 1995.

[6] T. El Gamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology — CRYPTO '84*. Springer-Verlag, 1985.

[7] Rosario Gennaro and Pankaj Rohatgi. How to Sign Digital Streams. In *Advances in Cryptology — CRYPTO '97*, 1997.

[8] J. B. Lacy, D. P. Mitchell, and W. M. Schell. CryptoLib: cryptography in software. In *Proceedings of USENIX: 4th UNIX Security Symposium*, October 1993.

[9] Leslie Lamport. Constructing digital signatures from a one-way function. Technical Report CSL 98, SRI Intl., 1979.

[10] Ralph C. Merkle. A Digital Signature based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, 1987.

[11] Ralph C. Merkle. A Certified Digital Signature. In *Advances in Cryptology — CRYPTO '89*, 1989.

[12] Silvio Micali and Adi Shamir. An Improvement on the Fiat-Shamir Identification and Signature Scheme. In *Advances in Cryptology — CRYPTO '88*, pages 244–247, 1990.

[13] Suvo Mittra. Iolus: A Framework for Scalable Secure Multicasting. In *Proceedings of ACM SIGCOMM '97*, 1997.

[14] Suvo Mittra and Thomas Y.C. Woo. A Flow-Based Approach to Datagram Security. In *Proceedings of ACM SIGCOMM '97*, 1997.

[15] National Institute of Standards and Technology. Digital Signature Standard. NIST FIPS PUB 86, U.S. Department of Commerce, May 1994.

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

[17] M.O. Rabin. Digitized signatures and public-key functions as intractible as factorization. Technical Report LCS/TR-212, MIT Laboratory for Computer Science, 1979.

[18] Ronald L. Rivest. *The MD5 Message Digest Algorithm, RFC 1321*, April 1992.

[19] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[20] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure Group Communications Using Key Graphs. In *Proceedings of ACM SIGCOMM '98,* Vancouver, B.C., September 1998.

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

Appendix. Flow verification procedure

```
procedure flowverify()
for each received packet
    if the block signature sign(root) in the packet signature is new then
        /* this is the first received packet in the block */
        compute the packet digest;
        compute each ancestor of the packet digest
            as the message digest of its children;
        let root' be the computed block digest;
        if (verify(root', sign(root)) = false) then
            the packet is not verified
        else
            the packet is verified;
            cache all computed nodes and their children as verified
        endif
    else    /* this is not the first received packet in the block */
        compute the packet digest;
        if (packet digest has been cached) then
            if (computed packet digest ≠ its cached value) then
                the packet is not verified
            else
                the packet is verified
            endif
        else
            compute all non-cached ancestors of the packet digest;
            let node be the highest node computed;
            compute the parent of node;
            if (computed parent ≠ its cached value) then
                the packet is not verified
            else
                the packet is verified;
                cache all computed nodes and their children as verified
            endif
        endif
    endif
endfor
```