The ForwardDiffSig Scheme for Multicast Authentication

Diana Berbecaru, Luca Albertalli, and Antonio Lioy, *Member, IEEE*

***Abstract*—This paper describes ForwardDiffSig, an efficient scheme for multicast authentication with forward security. This scheme provides source authentication, data integrity, and non-repudiation since it is based on the use of asymmetric cryptography. At the same time, it offers also protection against key exposure as it exploits OptiSum, our optimized implementation of the ISum forward-secure signature scheme. A tradeoff exists in the used keys: Short keys provide speed at the signer, whereas long keys are preferable for long-term non-repudiation. Performance has been evaluated with a custom packet simulator and shows that, by grouping the packets, ForwardDiffSig is efficient in terms of speed even for long keys at the price of a significant signature overhead. Therefore, ForwardDiffSig is fast, exhibits low delay, and provides non-repudiation and protection against key exposure, but has a nonnegligible impact in applications with strict energy or bandwidth constraints.**

***Index Terms*—Computer network security, forward security, multicast authentication.**

I. Introduction

A WIDE range of group-oriented applications, like software distribution, stock quotes, or audio/video transmission, makes multicast authentication a hot topic nowadays. In multicast communication, one sender (or source) communicates with a set of receivers over untrusted channels. The main security requirements are data origin authentication and integrity, so that each receiver can easily detect fake and manipulated messages. In addition, other important requirements might be needed, depending on the application type and context: 1) real-time protection, that is the data needs to be processed with low delay (several milliseconds at most) as in the case of applications for timely control of power flow over physical power networks based on data from monitoring and control devices; 2) non-repudiation to allow a recipient proving the source of data to a third party; 3) resistance to packet loss; 4) efficiency so that the computational load and the communication overhead at the sender/receiver should be low; 5) scalability, as the authentication information should be independent of the number of receivers.

In the past, the computational power was seen as a very constrained and limiting resource, so researchers proposed various techniques to make multicast authentication computationally efficient, often dropping other requirements.

In TESLA [1], [2], computational efficiency comes at the price of not providing non-repudiation, the data must be buffered prior to verification, and the sender and receiver must be synchronized. TESLA is based on symmetric cryptography (it uses a Message Authentication Code, MAC) and exploits time as source of asymmetry (see Section II for a discussion of other symmetric-based schemes).

The tree-chaining technique [3] exploits asymmetric cryptography, but it does not individually sign each packet (asymmetric operations are much slower than the symmetric ones); rather, it groups packets to spread the cost of a digital signature over multiple packets. This scheme uses hash trees and requires data buffering at the signer and/or verifier.

The DiffSig scheme [4] also exploits tree chaining and proposes to use shorter asymmetric signing keys for packet-flow signing because the authenticity (and integrity) of the packets must be guaranteed only for a short time (like hours, days, or at most months) and not for years. In DiffSig, the receiver should receive and (process) the packets shortly after they have been signed, within a predefined buffering time, named "playback time." Any (signed) packet arriving after the "playback time" becomes intrinsically obsolete, even though the packet is cryptographically valid. The authors estimate also for how long a short key can be considered secure, but do not consider attacks originating from various types of adversaries (which could potentially employ high-cost fast machines to attack the keys) and do not measure the performance of the scheme on any particular platform.

Today, the scenario and challenges have changed somehow: the computational power has increased, the cryptographic operations and protocols perform quite well even on sensors or handheld devices [5], [6], cryptographic accelerators are available to speed up execution of cryptographic operations, and the network capacity has increased. On the other hand, we have to face more and more complex security attacks aiming to break keys and systems' security. Thus, we need continuous updates on the key lengths considered secure, and try to contain the damage after the security of a system is compromised. By following this logic, the security protocols (including the multicast authentication ones) should adopt mechanisms ensuring the security of the (past) data even if the security is compromised at some point, like the forward-secure signature (FSS) or encryption schemes.

In this paper, we propose a scheme for multicast authentication—named ForwardDiffSig—providing source authentication, integrity, and non-repudiation of the data stream, along with forward security. We also measure, through experiments, the cost of the various components of this technique taking into consideration that, even though the computational power and

Manuscript received July 28, 2008; revised May 25, 2009 and February 16, 2010; accepted April 27, 2010; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor D. Agrawal. Date of publication July 12, 2010; date of current version December 17, 2010.

D. Berbecaru and A. Lioy are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, 10129 Torino, Italy (e-mail: diana.berbecaru@polito.it; antonio.lioy@polito.it).

L. Albertalli was with the Dipartimento di Automatica e Informatica, Politecnico di Torino, 10129 Torino, Italy. He is now with Accenture, Milan 20154, Italy (e-mail: luca.albertalli@accenture.com).

Digital Object Identifier 10.1109/TNET.2010.2052927

the network capacity have increased, they can still be considered limited, especially on servers.

Our scheme uses the tree-chaining technique with asymmetric keys for flow signing along with our optimized version of the ISum FSS scheme to achieve non-repudiation (even in the presence of dishonest signers) and protection against key exposure. Our optimized implementation of ISum (named OptiSum) guarantees a constant time for the key update operation at the signer. ForwardDiffSig can be used also in "sign-each mode," when each packet is signed individually. For better efficiency, especially in the "sign-each mode," the schema can use short signing keys (as in DiffSig), but not shorter than the key length required to achieve a minimum security level. Our approach uses the security levels and key lengths defined by ECRYPT2 [7] on an annual basis, which are currently deemed acceptable for different applications. Simulations show that our scheme is efficient in terms of speed of the signed stream at the price of a rather large size of the signature attached to each packet and the need to loosely synchronize sender and receivers. Due to the communication efficiency issue, our scheme is more suitable for applications running on devices with no energy or bandwidth constraints, whereas in resource-constrained networks, such as sensor networks, the MAC-based schemes or the hybrid offline/online solutions might be preferred.

The rest of the paper is organized as follows. Related works are in Section II, the underlying network and security models are in Section III, while in Section IV we evaluate the ISum scheme and our optimization OptiSum. Our ForwardDiffSig scheme is presented in Section V, and Section VI describes its implementation and usage modes. Section VII demonstrates, through simulations, the efficiency of our scheme in its three modes. Finally, we discuss the pros and cons of our scheme in Section VIII and draw conclusions in Section IX.

II. Related Works

Multicast authentication schemes. Multicast authentication protocols can be divided in two classes: MAC-based protocols (exploiting symmetric cryptography) and digital signature-based protocols (exploiting asymmetric cryptography).

In the simplest symmetric-based solution, the sender (S) generates a secret key and shares it with all the receivers (R). The secret key is used to create MAC values on the original packets. Since the MAC values are rather small and can be created and verified quickly, the protocol has good performance, but source authentication and non-repudiation are not guaranteed because any receiver owning the secret key could also create a valid data stream. Alternatively, S might share a distinct secret key K_i with each of the N receivers, and send, for each packet P, the proof $\mathrm{MMAC} = \langle H(K_1, P) || \dots || H(K_N, P) \rangle$ [8]. In this construction, no coalition of users can create a message/MMAC pair that will fool a user outside the coalition (since they do not have the outsider's MAC key), but the MMAC length grows linearly with the number of receivers. In [9], S shares a different set of keys with each recipient. S knows a set of N keys, and $R = \{K_1 \dots K_N\}$ and calculates N MACs for each packet, that is $\langle \mathrm{MAC}(K_1, P), \dots \mathrm{MAC}(K_N, P) \rangle$. Each recipient knows only a shared subset of keys $R_u \subset R$, where every key K_i is included in R_u with probability $1/(w+1)$, independently for every i and u. Each recipient calculates $|R_u|$ MAC values for each packet: If all the calculated MAC values match the transmitted ones, then the packet is authentic. This technique is applicable only for small multicast groups, i.e., the subset R_u must be chosen to ensure (with high probability) that no coalition up to w malicious users have knowledge of all the keys known by another user within the multicast group.

The multicast authentication schemes based on asymmetric mechanisms provide packet integrity, source authentication, and non-repudiation, but are computationally expensive (due to digital signature creation) and could therefore introduce delays at the signer or verifier. To mitigate this problem, all these schemes use some form of signature amortization technique. For example, in [10]–[12], the packets are chained to spread the signature creation cost over many packets, whereas in [3], the packets are grouped in blocks, which are digitally signed. These schemes support also various types of redundancy at signature generation in order to resist to packet loss; that is, even though some packets (less than a certain threshold) are lost, the received packets still contain the authentication information required to verify the packets. Other hash-chaining techniques are computationally efficient but are not robust to packet loss, such as [13]. The hybrid offline/online solution [14] avoids the speed problem associated with the use of digital signature algorithms: Offline computation is used to create buffers of one-time short key pairs and to sign the public components with a long key, while online computation is used to create a signature of the message using a one-time private key from the buffer of keys. This scheme achieves good speed (reaching 500–1000 signatures/s), whereas its main disadvantage is the signature overhead (about 1 kB/packet), which could be reduced to less than 300 bytes with the help of other cryptographic techniques.

Erasure code algorithms might be also used [15], [16] to achieve signature amortization. In this case, a single signature calculated over n packets can be split, so that if m out of the n packets are received, then the receiver can always reconstruct the signature and authenticate each of the m received packets. However, such algorithms are not resistant to packet insertion. Another solution proposed to mitigate the computational cost of signature creation is to use fast signature techniques such as the optimized version of the Feige–Fiat–Shamir scheme [3] or the one-way functions without trapdoors [17]. TESLA [2] features high computational efficiency and low communication overhead, but it requires packet buffering either at the sender or at the receivers. TV-HORS [18] provides fast signing/verification and buffering-free data processing, as well as tolerance to packet loss and strong robustness against malicious attacks, but it requires very loose synchronization among S and R, it uses a relatively large key size, and does not provide forward security.

Forward-secure signature schemes. These schemes offer a solution to key exposure attacks: Once a private key is exposed, not only all the future digital signatures created with the compromised key are possibly fake, but also all the past signatures, since there is no way to tell whether a signature has been generated before or after the key compromise event (unless a separate time-stamping mechanism is used). In a FSS scheme, the time is divided into T time periods, and the private key is "updated"; that is, a different asymmetric private key is used by the signer in each time period. Each private key is used to sign messages in the current time period, whereas the public key (of the FSS scheme) is constant for all periods. To verify a forward-secure

signature applied on a message, a verifier has to check both the validity of the signature and the time period. When passing to a new time period, the signer deletes the signing key used in the previous time period. The signature scheme is forward-secure because it is impossible for an adversary to forge the signature for a past period, even if he obtains the secret key for the current period.

Any FSS scheme consists of four operations: a key generation algorithm *KeyGenerate*, a secret key evolution algorithm *Update*, a signature generation algorithm *Sign*, and a signature verification algorithm *Verify*. The *theoretic* FSS schemes [19]–[21] are proved to be secure in the random-oracle model and are not considered in our approach. The *generic* provably secure FSS schemes—such as the Bellare–Miner tree, the "product" construction, the "Sum" and "iterated sum" (ISum) constructions, and the MMM tree [22]—use any arbitrary base signature scheme (e.g., RSA [23]), and thus their security is provable as long as the underlying standard signature scheme is secure. Typically, in a FSS scheme, the parameter T must be fixed in advance and passed in input to *KeyGenerate*, and it influences also the signature size and the verification time. In the MMM tree scheme [22], the operations do not depend on the total number of time periods (T), but on another security parameter of the scheme (l), which must be super-logarithmic in T in order to avoid exhaustive search attacks. So far, the FSS-related research work dealt mostly with the specification and evaluation of some FSS schemes, including the MMM tree and ISum schemes [22], [24], [25], whereas their use in practical scenarios is still an open topic. In our work, we considered the ISum scheme, we evaluated its parameters, and we proposed an optimization of the ISum scheme.

III. System Model

This section describes the model underlying this paper: first the network and security models, and then the actions of the adversary model.

Network Model. We consider a multicast group involving one sender (S) and a potentially large number of receivers (R). Each message is delivered from S to each R through a lossy and insecure network. The intermediate nodes in the network only forward the packets and do not provide any security guarantee (such as integrity and authenticity checks). These nodes may also be malicious and drop or modify packets originating from S or even inject fake packets. We consider applications in which each generated message is unknown to S until it is ready to send. In addition, we consider that the sender's application might need to send packets at slow rates (no delay constraint at R), or at higher rates (fast flows need to be supported). The packet sending rate might be constant or dynamic, but in case of dynamic flow, we consider an upper time delay (τ) at S. We assume existence of a trusted key server from which any R can securely obtain the public key of S, in particular the public key of the FSS scheme required to verify the signature on the packets. We also assume that the signer or the verifier can buffer data and that there exists a secured time synchronization system, which enables R to get loosely synchronized with S.

Security model. In our scheme, we aim to provide data source authentication and integrity for the packets sent by S, so that R can verify that the received data originates from S and was not altered on the way. In particular, if the received packet was injected or modified by an attacker, R can recognize and discard it; if the packet is really sent by S and arrives intact, it can be authenticated by R. In addition, we ask for forward security (even in the presence of dishonest signers), i.e., even if the signing key of S is compromised, the past signatures still remain valid.

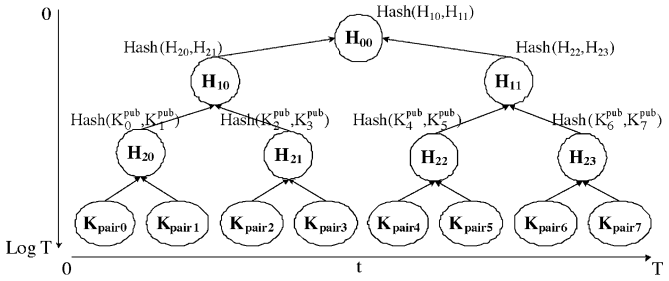

Fig. 1. Structure of an ISum (hash) tree. Each leaf contains a base signature key pair $(\mathcal{K}_i^{\mathrm{pri}}, \mathcal{K}_i^{\mathrm{pub}})$, associated to a time period $t \in \{0, 1 \ldots T\}$, and the root is the public key $(\mathrm{K}_{\mathrm{pub}})$ of the ISum scheme.

Attacker model. The goal of the attacker is to inject a malicious message and persuade R that the message was sent by S, or to modify the packet sent by S without being detected. We assume that the attacker has full control over the network: It can selectively eavesdrop, capture, drop, resend, delay, and alter arbitrary packets. The computational power of the attacker might be limited, but it is not necessarily bounded to that of S or R, i.e., the adversary can use more powerful devices and a larger storage than S and R.

IV. ISum and OptiSum Forward Signature Schemes

A. ISum Scheme

The Sum scheme [22] joins two traditional signature schemes T_1 and T_2 using the Sum $(\oplus)$ composition method to obtain a FSS scheme with $T = T_1 + T_2$ number of periods. The public key of the new scheme is the result of applying a collision-resistant hash function on the public keys of the two schemes. To save space, the keying material of the T_2 scheme might not be actually stored at S while T_1 is still in use [22]; instead, S stores the random seed used by the pseudorandom number generator for the generation of the keys of the T_2 scheme. When the time comes to switch to T_2, its keys are generated on the fly from the random seed. Note also that in almost all the FSS schemes, the security of the storage used for the keying material (in our case, the random seed) is crucial. The signer is expected to promptly detect key (storage) compromise by using intrusion detection methods and to announce the key revocation in real time.

The Iterated Sum (ISum) construction recursively uses the Sum method to obtain an FSS scheme, starting from a traditional signature scheme, like RSA, DSA [26], or ECDSA [27]. The result of Sum composition in ISum is a binary Merkle hash tree [28]—called *ISum tree*—with a key pair of a traditional signature scheme (e.g., RSA) in every leaf, and the hash of the two child values in every internal node (Fig. 1). The hash associated to the root node—denoted $\mathrm{K}_{\mathrm{pub}}$—is the public key of the ISum scheme, which is valid in all time periods T.

A signature $\sigma_{Ij}(\mathcal{M})$ calculated with ISum on a message $\mathcal{M}$ in the time interval j $(0 \leq j < T)$ is composed of the following: the signature $\sigma'(\mathcal{K}_j^{\mathrm{pri}}, \mathcal{M})$ calculated with the base signature algorithm and the private key K_j^{priv} for the time period j

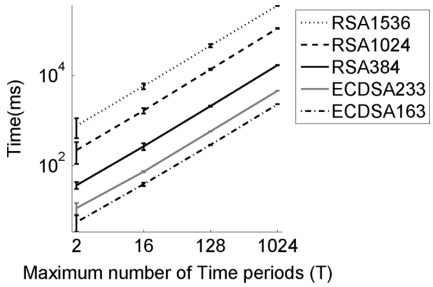

Fig. 2. ISum key generation time based on the parameter T, RSA and ECDSA algorithms, and various key lengths.

(denoted *leaf*$_j$), the time index j, the public key K_j^{pub} for the time period j, the public key stored in the sibling of *leaf*$_j$, the hash of the parent node, and the sibling of the parent node [22], [29]. Thus, two hash values are composed in the signature for each level up to the root node. Considering the ISum tree shown in Fig. 1 and RSA as the base signature scheme, $\mathcal{K}_2^{\text{pri}}$ and $\mathcal{K}_2^{\text{pub}}$ are respectively the RSA private and public keys in K_2^{pair}, and the ISum signature at $t = 2$ is

$$\sigma_{I2}(\mathcal{M}) = \left\langle \sigma'\left(\mathcal{K}_2^{\text{pri}}, \mathcal{M}\right), 2, \mathcal{K}_2^{\text{pub}}, \mathcal{K}_3^{\text{pub}}, \mathcal{H}_{20}, \mathcal{H}_{21}, \mathcal{H}_{10}, \mathcal{H}_{11} \right\rangle. \tag{1}$$

To verify the signature $\sigma_{Ij}(\mathcal{M})$, the verifier checks the signature σ' on $\mathcal{M}$, and for each level in the tree, it compares the hash in the signature $\sigma_{Ij}(\mathcal{M})$ with the hash of the two concatenated sibling values. For example, to verify $\sigma_{I2}(\mathcal{M})$, the verifier executes the following four tests:

$$\begin{aligned} \mathcal{V}_{\mathrm{K}_{\text{pub}}}(\mathcal{M}, \sigma_{I2}) &\stackrel{?}{=} \mathcal{V}_{\mathcal{K}_2^{\text{pub}}}(\mathcal{M}, \sigma') \\ h\left(\mathcal{K}_2^{\text{pub}} \| \mathcal{K}_3^{\text{pub}}\right) &\stackrel{?}{=} \mathcal{H}_{21} \\ h(\mathcal{H}_{20} \| \mathcal{H}_{21}) &\stackrel{?}{=} \mathcal{H}_{10} \\ h(\mathcal{H}_{10} \| \mathcal{H}_{11}) &\stackrel{?}{=} \mathcal{H}_{00}. \end{aligned} \tag{2}$$

B. Experimental ISum Evaluation

To evaluate ISum, we measured several parameters with the libfss [30] implementation: signature size and time for key updating and signature creation and verification. The experiments were run on a PC Intel Core2 Duo 2.13 GHz, 2 GB RAM, running Debian 4.1 and OpenSSL 0.9.8 g [31]. During the tests, all unnecessary services were switched off. We used RSA and ECDSA signatures with short and long keys, hash was calculated with SHA-1 [32], and different values for T were used (2, 16, 128, 1024).

Key generation. *KeyGenerate* is composed basically of two tasks: the generation of all key pairs to obtain the leaves of the ISum tree, and the calculation of all hash values (i.e., $d - 1$ for a tree of depth d) to obtain the root value. Consequently, the time required for key generation grows linearly with T (Fig. 2), reaching 10 s for 1024-bit RSA keys and $T = 128$. Note that this operation impacts on another time-consuming operation, the *Update* one.

Signing and verification times. The computational cost of an ISum signature is almost equal to the one of its traditional base signature: As it is evident from (1), the difference is small (being due to the additional hash calculations) and does not depend on T. The time required to verify an ISum signature is proportional to $\log(T)$, and the slope is proportional to the time required for a hash calculation [see (2)]. That is, $\text{cost}(\mathcal{V}_{\mathrm{K}_{\text{pub}}}(\mathcal{M}, \sigma_{\text{ISum\_RSA}})) = \text{cost}(\mathcal{V}_{\mathcal{K}_{\text{RSA}}^{\text{pub}}}(\mathcal{M}, \sigma'))$ $+ \text{cost}\ (\mathcal{H}) \cdot \log_2 T$. Nevertheless, since hash operations are much faster than the RSA ones, the time required for ISum signature verification is almost equal to that for RSA signature verification.

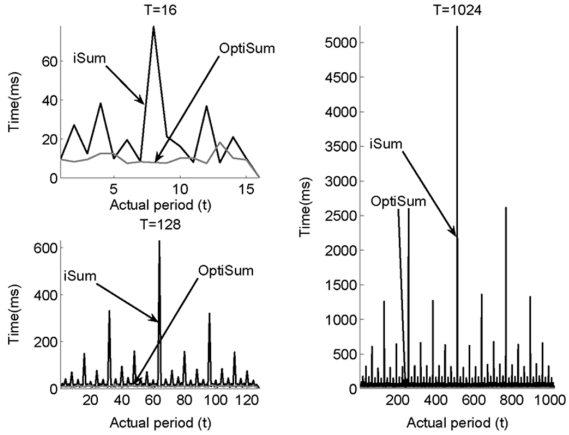

Fig. 3. Performance of key update operation in ISum and OptiSum for various values of T.

Update time. In ISum, the time required for an *Update* is highly variable in T (the higher T is, the higher is the standard deviation) and presents a precise geometrical structure (Fig. 3), which is due to the number of keys to be generated when passing from one period to the next one. In practice, one key needs to be generated when passing from $t = 0$ to $t = 1$ (K_1 in Fig. 5), two keys (K_2, K_3) when passing from $t = 1$ to $t = 2$, four keys $(K_4, \ldots K_7)$ when passing from $t = 2$ to $t = 3$, and so on. Thus, an *Update* operation requires a time proportional to the number of keys to be generated: The more keys to be generated, the higher the peak (Fig. 3), where the highest peak depends on the depth d of the ISum tree.

ISum signature size. From (1), we calculate the ISum signature size σ_{Ij}, where $L(x)$ is the size of x

$$\begin{aligned} L(\sigma_{Ij}(\mathcal{M})) = L\left(\sigma'\left(\mathcal{K}_j^{\text{pri}}, \mathcal{M}\right)\right) + 2 \\ \cdot L(h) \cdot (\log_2(\mathrm{T}) - 1) + 2 \cdot L\left(\mathcal{K}_j^{\text{pub}}\right). \end{aligned} \tag{3}$$

Note that $L(\sigma_{Ij})$ depends on T because the higher T is, the deeper is the associated ISum tree, and thus the longer the authentication path (Fig. 4) becomes.

C. Proposed OptiSum Scheme

To optimize the *Update* operation, we proposed in [33] a double cache technique that can be easily applied to ISum to obtain a more efficient scheme called *OptiSum*. In the current work, we have measured the efficiency of OptiSum through experiments and illustrate its benefits compared to ISum.

In OptiSum, the signer uses basically two caches: One cache stores the "future" private keys or key pairs associated to the leaves of the ISum tree of depth d, and the other one stores the hash values associated to the internal nodes of the ISum tree. OptiSum performs better in terms of time required for the key update operation (Fig. 3) and the signature size (Fig. 4), but it

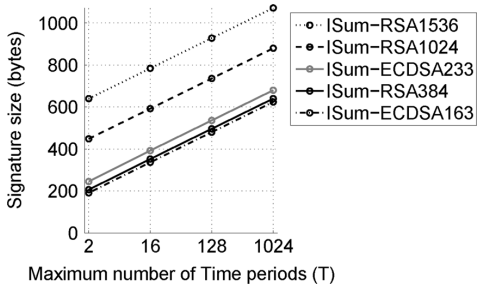

Fig. 4. ISum signature size calculated on 20 bytes of data, for various key update periods (T) and key lengths.

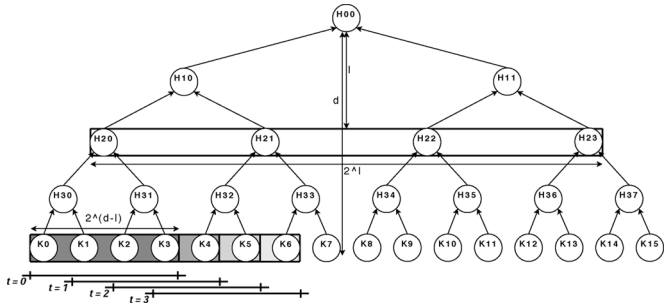

Fig. 5. Double cache updating technique applied to ISum. The position of the cache of hashes is not optimized; the figure just illustrates the content of the two caches.

does not impact on the performance of key generation. Unlike ISum, the key generation in OptiSum is made in reverse order, from $t = T$ to $t = 0$, so at the end of *KeyGenerate* the cache of hashes will contain all the hash values of the ISum tree at level l, whereas the cache of keys will contain all the keys for the first 2^{d-l} time periods.

Key update optimization. The cornerstone of OptiSum is calculating only one key when passing from $t-1$ to t, namely the key corresponding to the $(t+2^{d-l}-1)$-th time period. This key is inserted in the cache of keys when the old key is destroyed.[1] For example, in Fig. 5, K_4 is generated and inserted in the cache of keys when passing from $t = 0$ to $t = 1$, whereas K_0 is deleted from this cache because it is not needed anymore. Thus, the cache of keys moves forward with one position, and the first key on the left is the new private key of OptiSum. Since the data contained in the cache of keys is only related to future periods, its only requirement is the ability to destroy old keys, as in any FSS scheme. The memory occupied by the two caches is

$$\text{space} = 2 \cdot \sqrt{\mathrm{T}} \cdot \sqrt{L(\text{key})} \cdot \sqrt{L(h)} \tag{4}$$

where $L(h)$ is the hash size and $L(\text{key})$ is the key size (of a private key or of a key pair). If l is rounded to the next integer, the additional memory required is less than 6%. See [33] for a demonstration of these formulas. The ISum tree traversal requires a minimum cost because the hashes in the authentication path can be easily calculated from the cache of keys and the cache of hashes, so no other key (re)generation is required. From the hashes in the cache, we can build a subtree from the root node down to the level of the cache of hashes; next, from the cache of keys, we can build the subtree from the level of the cache of hashes to the active key.

[1]No key generation occurs if $t > T - 2^{d-l}$.

OptiSum signature size. The signature size in (3) can be reduced in two steps. First, we notice that some hashes associated to the parent nodes in the ISum tree could be omitted since they can be calculated from the child nodes [e.g., $\mathcal{H}_{21}$ and $\mathcal{H}_{10}$ in (1)]. Second, since a public key is larger than a hash, it is preferable to include in the signature the hash of the public key instead of the key itself associated to the leaf. Thus, instead of the signature in (1), we could use the OptiSum signature at $t = 2$, denoted σ_{O2}

$$\sigma_{O2}(\mathcal{M}) = \left\langle \sigma'\left(\mathcal{K}_2^{\text{pri}}, \mathcal{M}\right), 2, \mathcal{K}_2^{\text{pub}}, h\left(\mathcal{K}_3^{\text{pub}}\right), \mathcal{H}_{20}, \mathcal{H}_{11} \right\rangle. \tag{5}$$

Thus, the OptiSum signature size is smaller than the ISum signature size, and it can be calculated as

$$L(\sigma_{Oj}(\mathcal{M})) = L(\sigma'(\mathcal{K}_i^{\text{pri}}, \mathcal{M}) + L(h) \cdot \log_2(\mathrm{T}) + L(\mathcal{K}_j^{\text{pub}}). \tag{6}$$

V. ForwardDiffSig Scheme

In ForwardDiffSig, we employ the tree-chaining technique [3] for signature amortization, along with OptiSum to ensure packet authentication, integrity, and forward security of the signed flow. The tree-chaining technique consists in grouping several packets in a block and constructing for each block a Merkle Hash Tree (MHT), which is called *packet tree*. The MHT construction phase is also called *tree setup*. For example, considering the packets $p_1 \ldots p_8$, the packet digests $H_1 \ldots H_8$ are assigned to the MHT leaves, whereas the internal nodes are computed as digests of their children. The packet tree root is the block digest, and the *block signature* (BS) is the block digest digitally signed with a traditional signature algorithm. For a packet to be individually verifiable, it needs to carry its own *authentication data (or path)*, called *packet signature*. A packet signature consists of the block signature, the packet position in the block, and the siblings of each node (at every level in the tree) on the path from the packet to the root. To verify the packets in a block, the verifier builds an MHT from the packet signatures as packets arrive. At this point, we have to address several issues in our scheme. 1) In which way does the block size affects the signing rate? Wong and Lam [3] considered rather small blocks (composed of two up to 128 packets), whereas we should consider also bigger blocks, which could be processed efficiently via space/time-efficient MHT traversal algorithms proposed recently. 2) The root of the packet tree is signed with OptiSum, whose signature cost (in terms of time) is almost equal to the one of a traditional signature. If we assume to use key lengths normally in use nowadays (e.g., 1024-bit RSA keys), which packet rate can be obtained with the tree chaining technique? 3) If we want to achieve even higher speed at the signer, one solution would be to use shorter keys, as proposed in DiffSig. In this case, what is the smallest secure key size to be used, and for how long we can use it? In other words, what is the key size offering minimum security level for an application using OptiSum and for how long can we use it? Certain types of applications might have lower security requirements, whereas other security-sensitive applications might demand higher protection. Our responses to these issues are given in the following sections.

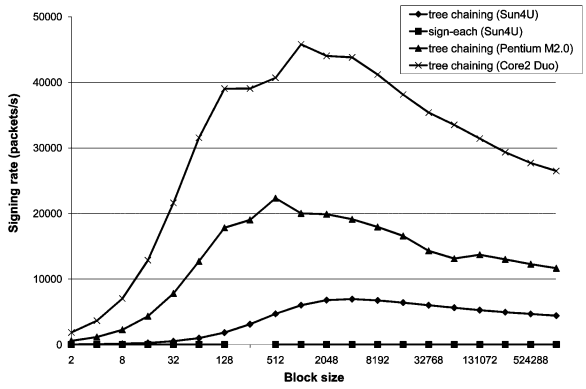

Fig. 6. Signing rate in tree chaining and sign-each modes with 1024-bit RSA key.

A. Estimating the Signing Packet Rate

In a previous work [34], we measured the time required to sign a packet flow with the tree-chaining technique on different platforms and by varying the number of packets placed in the packet tree leaves. To setup and traverse the packet tree, we used the Szydlo's algorithm [35], which is more time-efficient than the algorithm proposed by the same author in [36], but less efficient than the algorithm in [37].

The flow signing rate is given by the ratio of the total time required to setup and traverse the packet tree and the block size, that is the number of leaves of the packet tree. We performed tests on three distinct platforms: a Sun4U Enterprise 250 (2 UltraSPARC-II 400 MHz) with 512 MB of memory; an Intel Pentium M 2 GHz with 1 GB RAM, running Debian 4.0, OpenSSL ver. 0.9.8e and GCC ver. 4.1.2; and an Intel Core2 Duo 2.13 GHz with 2 GB RAM—identified respectively as Sun4U, Pentium M2.0, and Core2 Duo in Fig. 6. In the experiments, we used 1024-byte packets, SHA-512 to compute the hash values, and RSA with 1024-bit keys to digitally sign the root of the packet tree. A sign operation with a 1024-bit RSA key (measured with the *openssl speed* command) took 60 ms on Sun4U, 3.4 ms on Pentium M2.0, and 2.3 ms on Core2 Duo. A hash operation with SHA-512 algorithm operating on 1024-byte blocks took 0.445 ms on Sun4U, 0.025 ms on Pentium M2.0, and 0.012 ms on Core2 Duo.

Peak signing rate. As presented in [3], the flow signing rate increases with the block size, but this happens only for blocks of medium size. If the block size grows further (e.g., up to 1 048 576 packet digests), then the actual flow signing rate decreases due to the time required for packet tree setup and traversal. Intuitively, also the delay at the signer increases with the growth of the block size. On our test platforms, the peak signing rate is reached between 512 and 2048 packet digests per block (Fig. 6) and is obviously higher for the platforms with larger memory and faster CPU. In comparison, the flow signing rate (packet/s) reported in [3] with 1024-byte packets, a 512-bit RSA key, and MD5 [38] message digests on a Pentium II 300 MHz hardly reaches 5000 packets/s for a block composed of 128 packet digests, which is much lower than the rate obtained on two of our test platforms. In [34], we constructed a mathematical model to evaluate also the optimum depth of the MHT tree (based on the time required for hashing and signing operations) for obtaining the maximum signing packet rate.

DiffSig scheme: pros and cons. The aforementioned experiments show that a good signing rate can be obtained with 1024-bit RSA keys, currently considered secure even for long time protection. However, for better performance, the asymmetric key length could be shortened. This approach is taken by the DiffSig scheme that builds two types of signatures: the Expedited Processing (EP) signatures, which are created with short asymmetric keys on the data packets, and the Assured Processing (AP) signatures, which are created with long asymmetric keys to achieve non-repudiation of origin. The AP signatures are used also to sign the EP signing keys offline.

However, if the key used to create an AP signature is exposed in time period i, then all the AP and EP signatures created before and after time period i need to be considered compromised. Even though the signer might know which signatures were issued by him and which ones by an attacker (using the stolen key), there is no way for the receiver to distinguish among them. Moreover, in case of a dishonest signer, DiffSig cannot really achieve non-repudiation of origin: Such a signer may see an AP key compromise as a golden opportunity to repudiate at some point a previously signed flow. For example, he might even compromise his key on purpose (e.g., by anonymously posting his AP secret key on the Internet) and claiming to be victim of a computer break-in.

To prevent the actual key compromise and to make the "faked" key compromises less believable, a possible solution would be to use threshold cryptography [39]: The AP secret key might be distributed among n different computers, so that any t of them might construct valid AP signatures. In this case, breaking into a subset $t-1$ of individual computers would not affect the security of the AP key, but in case of the total exposure (e.g., simultaneous compromise of all the computers holding key shares), the past signatures must be considered invalid anyway. A radically different approach to protect the keys is to securely erase them: A securely erased key cannot be stolen. Thus, the AP signatures generated with such a key cannot be repudiated by a fallacious claim of exposure (assuming that the key was not exposed prior to the erasure). An EP packet signature, which was generated before the corresponding EP secret key was erased, should also remain trustworthy forever even after the signer's computer is compromised. To determine which is the shortest key length that can be employed securely by our scheme, we refer to the ECRYPT2 yearly report on algorithms and key sizes.

B. Recommended Asymmetric Key Sizes

ECRYPT2 recommends the minimum key sizes for the symmetric keys (Table I) to protect against different attackers, where the minimum security provides protection for a few months. Given any desired (symmetric key) security level, ECRYPT2 proposes to convert the symmetric key into an equivalent asymmetric key size using the data in Table II. The same report also presents equivalent symmetric key sizes for today's commonly used RSA and DLOG[2] key sizes (Table III). From these tables, we deduce that we can use a 640-bit RSA key to protect from "Hacker" attacks employing normal PC and no additional hardware, and a 768-bit RSA key or a 816-bit RSA key to protect from "hacker" attacks employing "malware" hardware. A 512-bit RSA key length is considered insecure, so we will not

[2]Schemes based on the discrete logarithm problem.

TABLE I
ECRYPT: MINIMUM SYMMETRIC KEY SIZES FOR VARIOUS ATTACKERS (2008–2009)

attacker	*budget*	*hardware*	*min security*
"Hacker"	0	PC	53 bits
	$<$ \$400	PC(s)/FPGA	58 bits
	0	"malware"	62 bits
Small organization	\$10k	PC(s)/FPGA	64 bits
Medium organization	\$300k	FPGA/ASIC	68 bits
Large organization	\$10M	FPGA/ASIC	78 bits
Intelligence agency	\$300M	ASIC	84 bits

TABLE II
ECRYPT: KEY-SIZE EQUIVALENCE (2008–2009)

security (*bits*)	*RSA*	*DLOG*		*EC*
		field size	*subfield*	
48	480	480	96	96
56	640	640	112	112
64	816	816	128	128
80	1248	1248	160	160
112	2432	2432	224	224
128	3248	3248	256	256
160	5312	5312	320	320
192	7936	7936	384	384
256	15424	15424	512	512

TABLE III
EFFECTIVE KEY-SIZE OF COMMONLY USED RSA/DLOG KEYS

RSA/DLOG key (bits)	security (bits)
512	50
768	62
1024	73
1536	89
2048	103

use such a key length (or shorter keys) in our scheme. We note also that organizations like IETF or NIST also released recommendations [40], [41] containing key-sizes equivalence, so a user of our scheme could use any of these reports to determine the minimum secure key length. IETF also issued the maximum length of time that selected RSA modulus sizes should be used [42] in accordance to recent results [43], [44] on key sizes: For RSA, a 768-bit key should be used for at maximum one week, whereas a 1024-bit key for at most one year.

C. ForwardDiffSig Architecture and Signature Formats

In ForwardDiffSig, the sender splits the packet flow in several containers (Fig. 7), each composed of blocks of packets. For simplicity, we assume that the lifetime of a container is equal to one time period in the ISum tree created for EP signatures.

The dimension of the block is not fixed, but it depends on the characteristics of the flow to be signed, like the speed of the input data flow and the maximum delay imposed at the receiver. In practice, we considered and evaluated three modes tolerant to packets loss, which are detailed in Section VI:

- ForwardDiffSig sign-each, which accommodates multicast flows at medium transfer rates;
- ForwardDiffSig with fixed delay, appropriate for flows at constant or dynamic packet rates but having fixed delay requirement;
- ForwardDiffSig continuous flow, appropriate for flows at constant and rather high packet rates.

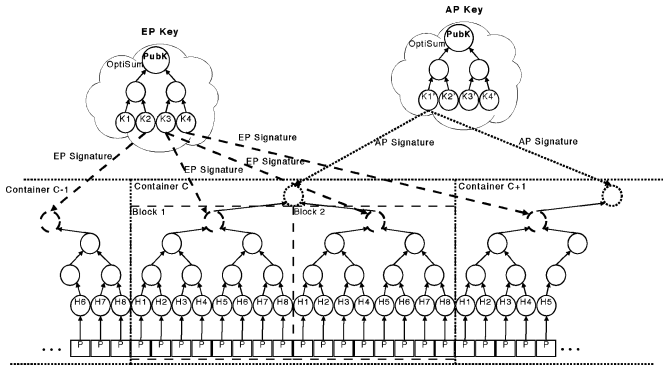

Fig. 7. ForwardDiffSig architecture. The keys $\mathrm{K}1, \ldots, \mathrm{K}4$ are used to create the EP OptiSum signatures $\sigma_{\mathrm{EP}_1}, \ldots, \sigma_{\mathrm{EP}_4}$, whereas the keys $\mathrm{K}1', \ldots, \mathrm{K}4'$ are used to create the AP OptiSum signatures $\sigma_{\mathrm{AP}_1}, \ldots, \sigma_{\mathrm{AP}_4}$.

In the "sign-each" mode, the block is composed of only one packet and the OptiSum signature is calculated/verified individually on each packet, whereas in the other two modes, the signature is amortized over several packets using the tree-chaining technique.

All three modes start with an initial synchronization protocol, where each receiver compares its local time to that of the sender and registers the difference. We remark that a rough upper bound T_{skew} on the clock difference between the sender and the receiver is sufficient. In this phase, the sender calculates also the signing parameters (e.g., as shown in Section VII-D) and publishes the *PubK* keys in the "EP key" and "AP key" clouds shown in Fig. 7. *EP OptiSum signatures*. Since these signatures are possibly created using short keys, they do not provide long-term non-repudiation. We denote by σ_{EP_i} the signature performed with the OptiSum scheme on one block of packets using the EP key pair corresponding to the ith time period

$$\mathrm{EPKey}_i = \left\{K_i^{\mathrm{pri}}, K_i^{\mathrm{pub}}\right\} \quad \text{where } 1 \le i \le T.$$

The packet format with the EP OptiSum signature changes with the mode. In the "sign-each" case, the packet format is

$$P_j = \langle M_j, \sigma_{\mathrm{EP}_i}(M_j)\rangle.$$

The other cases use signature amortization, and the format is

$$P_j = \langle M_j, \sigma_{\mathrm{EP}_i}(R_k), AuthPath_{M_j}\rangle$$

where M_j is the packet payload to be authenticated, $AuthPath_{M_j}$ is the authentication path of M_j in the packet tree, and $R_k (1 \le k \le d)$ is the root of the packet tree. By using (2), each receiver can verify the σ_{EP_i} signature and the *PubK* in the "EP key" cloud. Note that the signature σ_{EP_i} contains the indication of the period $i (0 \le i \le T)$ during which the private key (and implicitly the OptiSum signature) is valid. As explained in Section IV, the EP key K_i^{pri} is deleted by the signer when passing to a new time period.

AP OptiSum signatures. The AP keys are used to create AP OptiSum signatures for non-repudiation (as in DiffSig scheme). Since this task is not time-sensitive, the sender does not need to transmit the AP OptiSum signatures along with every packet. We denote by σ_{AP_j} the signature performed with OptiSum for

the cth container with the AP key pair corresponding to the jth time period

$$\text{APKey}_j = \{K_j^{\text{pri}}, K_j^{\text{pub}}\} \quad \text{where } 1 \le j \le T'.$$

Note that T' is the total number of periods for the AP cloud that in general will differ from T.

The AP OptiSum signing procedure works as follows. The sender generates

$$\sigma_{\text{AP}_j} = \text{OptiSum}(K_j^{\text{pri}}, H(H(\sigma_{\text{EP}_1}(R_1))\| \cdots \|H(\sigma_{\text{EP}_T}(R_d))))$$

and sends to the receivers the string

$$H(\sigma_{\text{EP}_1}(R_1))\|H(\sigma_{\text{EP}_2}(R_2))\| \ldots \|H(\sigma_{\text{EP}_T}(R_d))\|\sigma_{\text{AP}_j}.$$

Each receiver can verify the σ_{AP_i} signature using (2) and *PubK* in the "AP key" cloud. Also, the AEP key K_j^{pri} is deleted when passing to a new time period j.

D. Choosing the Signing Parameters

In ForwardDiffSig, for the minimum security level, the RSA keys used to create the EP OptiSum signatures are rather short, typically between 640 and 816 bits. This scheme could be subverted if an attacker manages to discover the private key used to create the signature $\sigma_{\text{EP}_i}(M_j)$ (in the "sign-each" case) or the signature $\sigma_{\text{EP}_i}(R_k)$ (where R_k is the root of the kth packet tree in the "fixed delay" and "continuous flow" cases) before the receiver gets the packet P_j and the associated OptiSum signature, since in this case the attacker could change the packet payload and the attached signature. To prevent this attack, the receiver has to check the following security condition on each packet it receives, and he must drop the packet if the condition does not hold.

> *Security condition.* A packet P_j arrived *safely* if the receiver can unambiguously decide—based on its synchronized time, T_{skew}, and considering an attacker profile (e.g., the ones shown in Table I)—that the time elapsed between packet sending and receiving is less than the time required to compromise via cryptanalysis attack the private key K_i used to create the OptiSum signature contained in P_j.

We introduce the following notations, which will be further used to express and evaluate the security condition.

T_{safe} is the time interval for which a private key is considered secure against cryptanalysis attacks. For example, [42] estimates that, for RSA, T_{safe} is one week for a 768-bit key and one year for a 1024-bit key. To remain on the safe side (due to the continuous progress in hardware speed and cryptanalysis techniques), we suggest to use values that are smaller by one or two orders of magnitude, such as T_{safe} equal to 1 h for a 768-bit key. In general, if P_{limit} is the acceptable probability of failure, p is the probability of a successful attack on the key (e.g., factorize N, where N is the modulus of an RSA key) in T_{safe}, and T is the number of keys used to sign a stream, our scheme can be considered secure if $T \cdot p < P_{\text{limit}}$. Every time a signing key is updated, the adversary stops trying to attack the previous one, and starts trying to attack (i.e., factorize N in case of an RSA key) the new one. Since each attack on each key is independent from the previous one, the probability that at least one works is the sum of probabilities that each one works separately. Calculating the probability of successful attacks on various types of keys is out of scope of the current paper. We considered the RSA key lengths and T_{safe} values based on the data reported in Section V-B.

T_{update} is the time interval for which a private key (associated to a leaf in the ISum tree) is used for signing operations. At expiration of T_{update}, the key is updated with the Update algorithm of OptiSum.

T_{skew} is the maximum tolerable synchronization error admitted by the signer, that is the upper bound on clock difference between signer's clock and receiver's clock. T_{delay} is the maximum tolerable network delay.

At session setup, the sender announces the values T_0, T_{update}, and T_{safe}, where T_0 is the starting time interval of the first time interval for which a private key is going to be used. These announcements are signed by the sender. In all three cases, the receiver verifies the security condition as follows: Each receiver knows the values of T_0, T_{update}, and T_{skew}, where T_{skew} is obtained from the initial synchronization protocol. If the receiver receives the packet at time t_R and i is the index of the time period interval $(0 \le i < T)$, then the receiver checks the following security condition:

$$t_R < T_{\text{delay}} + T_0 + i \cdot T_{\text{update}} + T_{\text{skew}} < T_0 + i \cdot T_{\text{safe}}. \quad (7)$$

Based on the security level considered adequate, the sender chooses the key length and he calculates T_{update}. In addition, he needs also to determine the maximum tolerable synchronization uncertainty T_{skew} and the maximum tolerable network delay T_{delay}. If $T_i = T_0 + i \cdot T_{\text{update}}$ and $\Delta_{\max} = T_{\text{skew}} + T_{\text{delay}}$, then the last packet of the ith time period is sent at $T_i + T_{\text{update}}$, and it will be received at $T_i + \Delta_{\max} + T_{\text{update}}$. For $\mathrm{i} = 1$, we can further rewrite (7) as

$$T_{\text{safe}} > T_{\text{skew}} + T_{\text{delay}} + T_{\text{update}}. \quad (8)$$

The sender must check (8) to choose T_{update} based on T_{safe} and $\Delta_{\max}$. Furthermore, he has to consider also the total number of periods T because two important parameters depend on it: the size of the EP OptiSum signature applied to the packet, and the total time for which the public key of the OptiSum scheme will be valid. If T_{tot} is the interval time for which the public key (K_{pub}) of OptiSum is valid, then $T_{\text{tot}} = T \cdot T_{\text{update}}$, whereas the signature size is given by (6).

If a signer chooses T small to obtain a short signature size, then T_{update} is large, and consequently the private key used to sign the packets has to be long. As said, the actual key length depends ultimately on the security level considered acceptable for the application. The use of shorter keys is however a solution for increasing the speed. If T is enlarged, then the key length may be shortened, but the key length depends on the security level required, and very short keys (smaller than the one defined for the minimum security level) cannot be used. An example of parameter choice at the signer side based on application requirements is given in Section VII-D.

VI. ForwardDiffSig Implementation Details

The signing process in ForwardDiffSig is organized in three steps: 1) generation of the ISum trees to obtain the public keys

for the OptiSum scheme, both the EP keys and the AP keys; 2) signature creation; 3) key evolution. The first phase can be done offline and does not impact on the signing speed, whereas the other two phases need to be executed in real time. Note that the signing phase is executed whenever a packet (or a block of packets) needs to be signed, whereas the key update phase takes place at the expiration of each T_{update} time interval. Before calculating an OptiSum signature, the signer must check whether the signing key is still valid in the current time period. If this is not the case, he must call the key update procedure. The drawback in this case is that the key update task could block the signing process (even if just for a short amount of time). Consequently, in all three modes we assume that key updating is performed in parallel with the signing task, and the signing key is shared between the signing and key evolution tasks.

If the signer caches all the keys generated with *KeyGenerate*, then the key update time would be irrelevant because an *Update* operation would simply consist in deleting an old key and changing the reference to the new key. However, in this case the signer must have significant memory (proportionally to T) to store all the relevant keys. By using OptiSum, the signer reduces the memory occupied at the signer (proportional to $\sqrt{T}$) [33], and the key updating time is constant and equal to the time required to generate a new key.

A. ForwardDiffSig "Sign-Each" Mode

In this case, the block is composed of one packet, which is signed individually using an EP OptiSum key. Thus, no packet tree is actually constructed by the signer, and the performance is constrained by the dimension of the buffer storing the packets to be signed.

B. ForwardDiffSig "Fixed Delay" Mode

This case is particularly useful when the size of the data flow and/or the transfer rate are not known in advance. When the flow size is known in advance, the blocks can have the same size m, whereas for a real-time unpredictable flow, the packet generation time is time-varying for many applications. Partitioning the flow into fixed-size blocks for these applications may lead to an unpredictable (perhaps unbounded) signing delay [3]. Consequently, in this case, the flow is partitioned at fixed time periods τ, and the packets generated in one period are grouped into a block, which is signed and sent to the verifier.

To implement this mode, we exploit a stack to construct a packet tree as they arrive (actually a forest of packet trees) and a timer to count the fixed interval τ. The *Constructor* task is in charge of the packet tree's construction and implements Algorithm 1. The *TreeSigner* task implements Algorithm 2, and runs in parallel with the *Constructor*. At the expiration of the time period τ, *TreeSigner* extracts the oldest complete packet tree from the stack, signs its root with the OptiSum scheme and the key K_i corresponding to that time period, and then traverses the tree to construct the corresponding authentication path for each packet. Finally, *TreeSigner* sends P_j to the verifier, where each P_j is composed of the packet itself M_j, the authentication path $AuthPath_{M_j}$, and the signature $\sigma_{\text{EP}_i}(R_k)$. Note that, together with the constructed packet tree, the *Constructor* also saves the hashes corresponding to the internal nodes of the packet tree, so the time required for its traversal can be ignored.

Algorithm 1 Packet tree construction

Require: *pack* is the input packet
Require: *stack* is the used stack
new *node*
node.hash $\Leftarrow \mathcal{H}$ (*pack*)// the hash of the packet
node.time $\Leftarrow$ *t*// store the time of reception
node.level $\Leftarrow$ 0 level of the node
node.packet $\Leftarrow$ *pack*
stack $\Leftarrow \mathcal{PUSH}$ (node)
while $\mathcal{TOP}$(*stack*).*level*
$= \mathcal{TOP}$(*stack* $-\mathcal{TOP}$(*stack*)).*level* **do**
right $\Leftarrow \mathcal{POP}$ (*stack*)
left $\Leftarrow \mathcal{POP}$ (*stack*)
node.left $\Leftarrow$ *left*
node.right $\Leftarrow$*right*
node.hash $\Leftarrow \mathcal{H}$ (*left, right*)
node.time $\Leftarrow$ *left.t*// reception time of the oldest packet
node.level $\Leftarrow$ *left.level* $+1$// level of the node
stack $\Leftarrow \mathcal{PUSH}$ (*node*)
end while

Algorithm 2 Packet tree signing and traversal

Require: *stack*
Require: τ is the maximum admitted delay
loop
wait τ
node $\Leftarrow \mathcal{BOTTOM}$ (*stack*)
$\mathcal{SIGN}_{\text{OptiSum}}$ (*node.hash*, K_i)
traverse the tree rooted in *node* and for each leaf j
send packet P_j
$\mathcal{DELETE}$ (*stack, node*)
end loop

The number of packets in the block (equal to the number of leaves in the packet tree) is given by $N_{\text{leaf}} = R_P \cdot \tau$, where R_P (rate of packets) is the average number of packets in the unit of time arriving at the *Constructor*, and τ is the fixed delay, which is also the time elapsed between two extractions of the packet trees from the stack.

C. ForwardDiffSig "Continuous Flow" Mode

When the flow to be signed is generated continuously and at a rather high constant bitrate, we can eliminate the fixed delay τ. As above, the *Constructor* task (run by the signer) constructs the packet tree as packets arrive and passes the tree to the *TreeSigner* task. Since the timer is eliminated, *TreeSigner* runs "continuously," which means that as soon as it finishes to sign the current block, it gets from the *Constructor* the tree containing the packets that got accumulated on the stack while *TreeSigner* was busy signing and traversing the current block. Thus, in this mode the number of packets in the block (equal to the number of leaves in the tree) is

$$N_{\text{leaf}} = R_P \cdot T = R_P \cdot (T_{\text{sign}} + N_{\text{leaf}} \cdot \log_2 N_{\text{leaf}} \cdot T_{\text{trav}}) \quad (9)$$

TABLE IV
RSA SIGN/VERIFY MEASURED ON TEST MACHINE WITH OPENSSL INTERNAL BENCHMARK

key length	*sign*	*verify*	*sign/s*	*verify/s*
512 bits	0.278 ms	0.019 ms	3595.1	53789.9
1024 bits	1.036 ms	0.049 ms	965.1	20340.8
2048 bits	5.707 ms	0.155 ms	175.2	6458.2
4096 bits	36.703 ms	0.530 ms	27.2	1886.8

TABLE V
PERFORMANCE OF HASHING ALGORITHMS (IN kB/s) MEASURED WITH OPENSSL INTERNAL BENCHMARK

	block size (bytes)			
	16	64	256	1024
md5	27398	84692	203374	311176
sha1	27436	78363	169927	238837

where R_P and T_{sign} have the same meaning as above, and T_{trav} is the time required to traverse the packet tree, which in our case is equal to the time required to copy the hashes from the stored tree to the authentication path. If T_{trav} is negligible with respect to T_{sign}, then we can approximate N_{leaf} as $N_{\text{leaf}} \cong R_P \cdot T_{\text{sign}}$.

For example, if we consider that, at $T = 0$, only one packet is generated and needs to be signed, and the time required to construct and traverse the packet tree can be ignored, then the number of packets that get accumulated on the stack and are going to be signed at $T = 1$ depends on T_{sign} and the input rate R_P. If T_{sign} is constant for a certain key length, then if R_P grows, the packet tree grows also, which means more packets are grouped together, resulting ultimately in a bigger rate of the output signed flow. Nevertheless, when the packet tree gets bigger, the time spent to construct and traverse the tree cannot be ignored any more. Through measurements, we show (in Section VII) that at some point the output rate remains constant even though the input rate R_P further increases. This behavior is due to the time needed to calculate the internal hash values of the packet tree.

VII. EXPERIMENTAL RESULTS

To test the performance of ForwardDiffSig, we implemented a packet simulator to generate, sign, and verify the packet flow. For the experiments, we used the same computer already described in Section IV-B. On the test machine, we measured the time required for raw RSA sign, verify, and MD5/SHA-1 hash operations with the *openssl speed* utility of the OpenSSL library. Tables IV and V report the results.

A. Packet Simulator

Our simulator is implemented in C++ and is based on two abstract classes, *Source* and *Sink* (Fig. 8). These classes are used as base points for the creation and destruction of data packets, while the abstract classes *Filter* and *LossyFilter* model any operation performed on the packets. The class *LossyFilter* has two outputs: one for the packets correctly processed, and one for the packets considered lost. The packets transmitted in the system are represented by the class *Packet*, which contains both the packet payload to be signed (whose size is set at its creation) and the EP OptiSum signature applied on the packet itself, represented by the class *OptiSumSignature*.

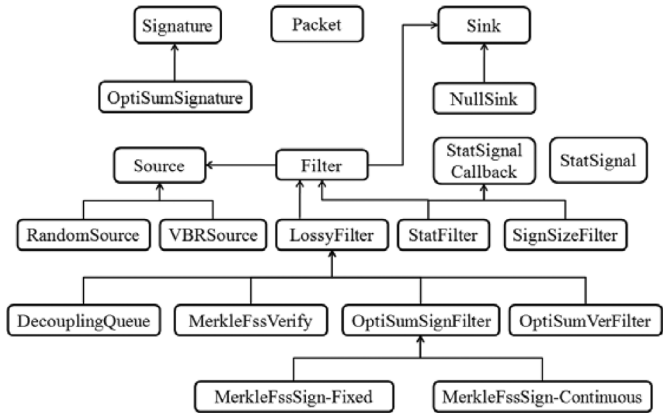

Fig. 8. Classes used by the packet simulator.

This structure allows to perform various processing and complex simulations on the packets. Each filter is responsible for one or more processing operations on the packets; a filter can simply implement a queue, or it can create or verify signatures. The filters can also be used to calculate statistics on the packets, such as the mean bit rate or the dimension of the signatures of a certain packet flow. All the filters computing statistical data inherit their interface from *StatSignalCallback*, which is the auxiliary interface of *StatSignal* used to print out statistics at regular intervals.

For packet generation, we derived the classes *RandomSource* and *VBRSource* from the *Source* abstract class. These classes exploit a pseudorandom generator, which generates byte sequences whose size is uniformly distributed among 64 and 1518 bytes, corresponding to the minimum and maximum Ethernet frame sizes. The first class generates a packet flow at a constant bit rate, whereas the second one generates a flow of variable bit rate, with a variability of $\pm 70\%$ with respect to the mean bit rate (passed as a parameter) and a random number of variations per second. In this way, we were able to generate an (input) packet flow at high rates, characteristic of a GigaBit Ethernet interface.

The only implementation of the abstract class *Sink* is *NullSink* that simply discards the received packets, as needed in the case of simulating lost packets. The class *StatFilter* is used to generate and print the statistical data calculated on the packets traversing the filter, e.g., the block size, the packet size, the output bit rate, the mean and maximum delay, calculated as the time elapsed from when the packet was generated until it was signed (at the signer) or from when it was received until it was verified. The class *SignSizeFilter* records the size of the OptiSum signature. The class *OptiSumSignFilter* implements the OptiSum signing, verifying and key update operations, as presented in Section IV-C.

B. Evaluation of ForwardDiffSig Sign Each

Sign. In this case, the simulator generates random packets with the class *RandomSource*. The packets are inserted into a buffer (*DecouplingQueue*), which is accessed by the class *OptiSumSignFilter* implementing the signing algorithm. The class *StatFilter* is used to perform measurements on the packets (including the lost packets), whereas the class *NullSink* is used to

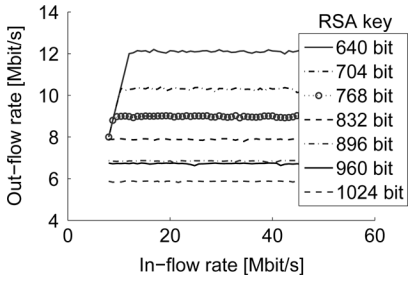

Fig. 9. Signer speed in "sign-each" mode.

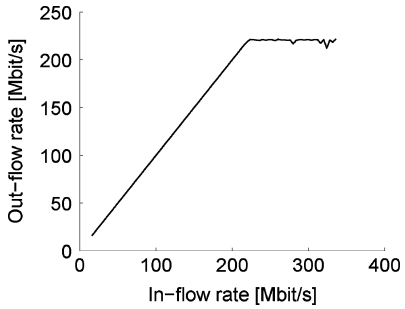

Fig. 10. Verifier speed in "sign-each" mode for a 640-bit RSA key.

destroy the packets. If the buffer is full because the signing algorithm is not able to consume (i.e., sign) all the packets produced by the generator, then the packets are lost. We've tested the *OptiSumSignFilter* filter with RSA keys of various sizes in the range 640–1024 bits, whereas the speed of the input flow varied from 5 to 50 Mb/s. The signing key used to create EP OptiSum signatures was updated every 5 s, for a total of 4096 time periods, which means the OptiSum public key was valid for about 5 h. The input source flow is composed of packets whose size is uniformly distributed between 64 and 1518 bytes, and the average packet size is 791 bytes. The results obtained for the signing operation are shown in Fig. 9. Note that for all RSA key lengths smaller than 896 bits, the signed packet flow grows proportionally with the input flow, up to a saturation point, corresponding to a specific input flow bit rate for each key size.

Verify. The class *OptiSumVerFilter* verifies the signatures (created with *OptiSumSignFilter*) and measures the speed (verification rate) of the output flow. Since it is impossible to generate signatures at a high rate to saturate the verifier (as RSA verification is much quicker than signature generation), we chose to pregenerate the OptiSum signatures offline and save them in memory. The class *SignedSource* generates signed packets, which are saved in a buffer (*DecouplingQueue*). From this buffer, the packets are retrieved and verified with the filter *OptiSumVerFilter*. At the end of the verification process, one of the following cases is encountered: 1) the packets are authentic; 2) the packets are lost; or 3) the packets fail the verification (not authentic). Fig. 10 displays the speed of the output flow for a 640-bit RSA key.

C. Evaluation of ForwardDiffSig With Signature Amortization

Sign. The classes *MerkleFssSign-Fixed* and *MerkleFssSign-Continuous* (further simply referred as MerkleFssSign) inherit from the class *OptiSumSignFilter* the signing and key updating features. In addition, these classes implement the stack (detailed in Section VI-B) to construct the packet trees from the incoming

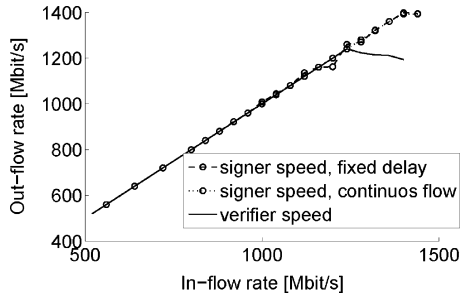

Fig. 11. Signer and verifier speed in "fixed delay" and "continuous flow" modes for a 640-bit RSA key. Similar behavior was obtained for shorter keys.

packets. *MerkleFssSign* uses an internal buffer to temporarily store the input packets for the construction of the packet trees. If the packets are inserted in the buffer at a higher rate than the rate at which they are consumed for the tree construction, then the buffer gets saturated and the packets are discarded. In this case, measurements are performed on the lost packets as well. When the buffer is not saturated, the packet tree is constructed and the root of the packet tree is signed with *MerkleFssSign*. The output of this class is composed of the packet payload, the packet signature and the corresponding authentication path. On this output, we measured the speed of the output flow and the signature size. Finally, the packets get destroyed with the class *NullSink*.

We used the SHA-1 algorithm for packet tree construction, OptiSum was configured with $T = 4096$, the (RSA) key was updated every 5 s, and the time interval used to collect the packets in "fixed delay" mode was set to $\tau = 20$ ms. Fig. 11 illustrates the performance of our ForwardDiffSig scheme in terms of speed of the output signed stream. The signer speed measured is very good, reaching 1400 Mb/s for a single flow on our test machine. However, as mentioned, the output signed flow saturates at some point and remains constant independently of the input flow rate. We observed the saturation point for various key lengths, which makes us believe the saturation point depends on the time required to construct the packet tree, rather than on the length of the asymmetric signing key. Practically, for very high input flows, the signer cannot construct (on our machine) the packet tree (via hash operations) quick enough to consume the buffer, so it signs only part of the input flow and discards the rest of the packets. However, in the "continuous flow" mode, the longer the signing key and the longer T_{sign} is, thus the longer the signature size (since the packet tree depth grows).

Fig. 12 illustrates the OptiSum signature size versus the input flow bitrate. The signer throughput is in Fig. 13: for a constant input flow, the actual amount of data (payload) sent over the channel is about 65% in "continuous flow"'mode and 62% in "fixed delay" mode. The throughput is the ratio of the data payload and the total size of the packet containing the data payload and the signature.

The "continuous mode" is technically the same as the "fixed delay" mode, except that the signer does not wait for a fixed time to sign and send the packets, but a new cycle starts as soon as the signing of the previous block finishes. The "continuous mode" heavily loads the CPU and does not guarantee predetermined delays, but on average it offers lower latency at the signer (Fig. 14), and smaller packet signatures (Fig. 12). If in the "fixed delay" mode the input flow is provided at a variable bit rate,

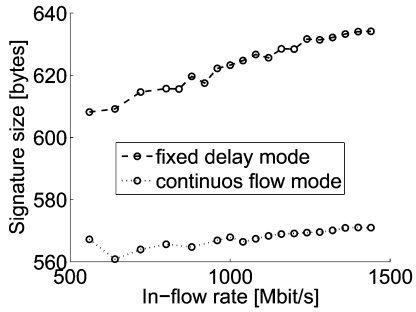

Fig. 12. Signature size in "fixed delay" and "continuous flow" modes (640-bit RSA key, $T = 4096$).

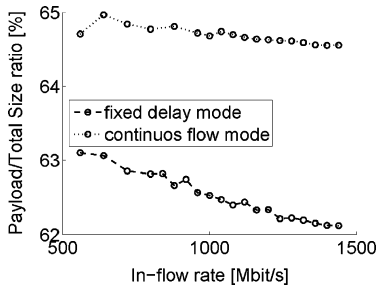

Fig. 13. Signer throughput in "fixed delay" and "continuous flow" modes (640-bit RSA key, $T = 4096$).

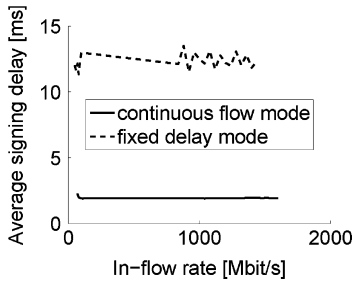

Fig. 14. Average delay in "fixed delay" and "continuous flow" modes for a constant input flow and a 640-bit RSA key.

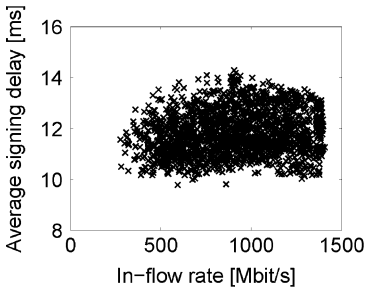

Fig. 15. Average delay in "fixed delay" mode for a variable input flow and a 640-bit RSA key.

then the average delay changes significantly. The experiment setup is similar as before, except the packet flow generator generates a flow at variable bit rate, with 70% variation from the central value. The cloud shown in Fig. 15 indicates that there is no correlation between the bit rate of the input flow and the delay measured, so in this case we can only say that the maximum delay does not exceed the maximum preconfigured delay (i.e., $\tau = 20$ ms).

Verify. The *OptiSumVerFilter* class implements the verification algorithm proposed in [3] to verify the packets both in "fixed delay" and "continuous flow" modes. This algorithm requires the verifier to cache entire packet trees on its side, in order to reuse them in a second moment and thus to optimize the verification time. To simulate the verifier functionality, we attached the input of the *OptiSumVerFilter* class directly to the output of the *MerkleFssSign* filter, without interposing any buffer between them. Even though the above configuration does not reflect a real case (where data buffering and additional network delay need to be taken into account), it is definitely suitable for our purpose: measuring the performance of the verification algorithm. However, in this case, there is a strong dependence of the verifier on the signer: When the signer saturates, the verifier does as well. We remind that a verify operation is composed of several hash calculations, whose number is proportional to the packet tree depth; see (2). When the input flow rate is high, the packet tree is quite deep because the block is composed of many packets. At some point, the verifier "saturates" for the same reason for which the signer "saturates," that is because the processor of the testing machine cannot calculate all the necessary hashes in less time than the time at which the signed packets are delivered to it. In this case, some of the packets remain unverified and will be discarded. This behavior can be observed in Fig. 11.

In our configuration, the verifier caches only one packet tree at a time, and it maintains for that tree only $2 \cdot h$ hashes (where h is the depth of the tree), i.e., the hashes required to authenticate the last packet received. This caching technique works out only if the packets arrive in the correct order or some other mechanism is used to reorder the packets correctly. Otherwise, other caching techniques could be used at the verifier, e.g., saving more packet trees and more nodes for each tree. The experimental speed obtained for the signed flow verification is very good, reaching 1200 Mb/s.

D. Selection of the Signature Parameters

Suppose ForwardDiffSig is used for secure distribution of a 1200 MB critical software update, which must be provided to several computers connected via 1-Mb/s network links. We assume that a low security level should be sufficient for such a transmission. Based on these requirements, the sender calculates the required ForwardDiffSig parameters for the sign-each mode with the SHA-1 hash algorithm. Since the distribution would take 9600 s, the validity of the OptiSum public key should be the same. For the network delay, we assume $T_{\text{delay}} = 2$ s. The signer could choose a 816-bit RSA key (low security level), which is considered to resist to cryptanalysis attacks for some hours (T_{safe}) assuming a "small organization" adversary (Table I). Thus, if T_{skew} is neglected, then the signer can choose $T_{\text{update}} = 80$ s, which is definitely smaller than T_{safe}. We calculate $T = T_{\text{tot}}/T_{\text{update}} = \frac{9600}{80} = 120$, so we choose $T = 128$, which is the closest multiple of 2. We recalculate $T_{\text{update}} = \frac{9600}{128} = 75$ s, which is conforming to the security condition (8). The signature overhead—computed with (6)—is about 344 bytes.

Once the key length has been chosen, we can check the bandwidth constraint. Assuming that the sender's machine has the same configuration as the one used in our tests, and assuming that the "sign-each" approach is used, we note from Fig. 9 that

the speed of the output flow is more than 8 Mb/s, and therefore in this specific case it is not a problem. If a higher value would be needed, then the signer could choose the "continuous flow" mode of the scheme. In this case, as shown by Figs. 11–13, the speed increase will be achieved at the expense of a longer signature (due to the presence of the authentication path inside the transmission).

VIII. Pros and Cons of ForwardDiffSig

According to [45], the tree-chaining technique is essentially good in client–server scenarios where latency is not an issue and the server is dedicated to serving a small number of multicast flows. Thus, our scheme inherits from the tree-chaining technique advantages like resistance to collusion attacks and packet loss and a rather good efficiency in speed when serving a single multicast flow. However, if a server having enough cycles to perform 10 public key operations/s were required to send 50 different flows, each with a packet rate of only one packet/s (e.g., serving multiple handheld devices) for a total of 50 packets/s throughput, it will be unable to meet its requirements unless the same signing key and authentication tree data structure is shared across different flows or an average delay of 5 s is imposed on each flow. Such a delay is almost always unacceptable. In our scheme, the server could perform better, but further tests have to be done to evaluate the performance in case of multiple flows.

It is interesting to note the gain at the verifier: For high-rate flows (and thus big blocks), the time required to verify a packet does not seriously decrease when using shorter keys because the time for hash calculation remains the same and this impacts each packet verified (2). On the contrary, for slow flows (e.g., blocks composed of one or two packets), the number of hash operations is very small, and thus, when using shorter keys, the time required to verify a packet could decrease even by 50% when compared to the time with long keys. Furthermore, by reducing the signature verification time, the resistance of the scheme to flooding attacks is increased as well [46].

What if we would use ECDSA? ECDSA is faster than RSA for sign operations, but it is slower for signature verification. In addition, the size of an ECDSA signature is shorter (considering the same security level) than a RSA one, e.g., a 156 bytes signature is obtained for a 1248-bit RSA key, whereas a 40 bytes signature is obtained for a 160-bit ECDSA key. The size of the base signature influences heavily the size of an OptiSum signature: When T is not very large (e.g., in case of $T = 4096$), more than 50% of the OptiSum signature size depends on the base signature size. For example, in Fig. 4, the ISum signature size calculated with ECDSA is three times smaller than the one calculated with RSA (for the same value of T). In the "sign-each" mode, accommodating slow transfer rates, it might be convenient to use ECDSA rather than RSA as base signature to obtain a shorter packet signature and a faster signer. If no support for ECDSA is available, the user might adopt RSA with a shorter key, but it needs to consider also the security level required and verify the security condition. In the "fixed-delay" and "continuous mode," we observed that a shorter RSA key does not improve the signer speed, but good signer speed is obtained even for longer RSA keys, as explained in Section VII-C. Thus, in this case, if we use ECDSA, we would obtain a good speed at the signer (even for long ECDSA keys), a shorter packet signature (than the one shown in Fig. 12 based on RSA keys), short delay at the signer, but a slower verifier speed and also a longer delay at the verifier.

The signature overhead, which is due to the size of the authentication information in the packet tree and also the size of the OptiSum signature, might thus be a problem in environments with strict energy or bandwidth constraints. As noted in [45], during burst periods, the packet overhead will be higher, and this can cause additional undesirable side effects, such as increased packet loss due to fragmentation, precisely at times when the traffic volume is large. Second, in our scheme the signer must know or approximate the duration of the streaming, so that he can choose the T parameter of the FSS scheme. This limitation is due to the ISum design, and one possibility would be to use an FSS scheme that is size-independent of T, such as MMM, obtaining however an even longer signature. Third, our scheme requires a secure clock synchronization between the source and the recipients, which may not be always feasible in a large multicast group. Moreover, as noted in [47] for TESLA, all secure clock servers become potential targets for adversaries who wish to defeat the authentication scheme.

IX. Conclusion

Current multicast authentication schemes trade off several parameters like speed, latency, overhead, non-repudiation, forward security, and security level. Each parameter impacts on the others. We have proposed in this paper a new scheme for multicast authentication, named ForwardDiffSig, which allows both per-packet and per-block digital signing and verification. Even though the communication overhead in this scheme could be quite high, the scheme provides packet source authentication, integrity, and non-repudiation, along with protection from key exposure. Experimental measures prove that ForwardDiffSig exhibits high speed at the signer and verifier, low delay, and resistance to packet loss, thus making it suitable for use in multicast applications that employ an unreliable transport protocol. Possible future work could consist in extending ForwardDiffSig to support ECDSA, efficient signing of multiple flows, and employing more flexible FSS schemes, like the MMM one that is independent of the total number of update periods of the signing key.

Acknowledgment

The authors thank the anonymous reviewers for their valuable comments that helped in greatly improving this paper from its original version.

References

[1] A. Perrig, R. Canetti, D. Song, and J. Tygar, "Efficient and secure source authentication for multicast," in *Proc. NDSS*, 2001, pp. 35–46.

[2] A. Perrig, D. Song, R. Canetti, J. D. Tygar, and B. Briscoe, "Timed efficient stream loss-tolerant authentication (TESLA): Multicast source authentication transform introduction," RFC 4082, June 2005.

[3] C. K. Wong and S. S. Lam, "Digital signatures for flows and multicasts," *IEEE/ACM Trans. Netw.*, vol. 7, no. 4, pp. 502–513, Aug. 1999.

[4] N. Kang and C. Ruland, "DiffSig: Differentiated digital signature for real-time multicast packet flows," in *Proc. Trust Privacy Digital Business*, 2004, LNCS 3184, pp. 251–260.

[5] D. J. Malan, M. Welsh, and M. D. Smith, "Implementing public-key infrastructure for sensor networks," *ACM Trans. Sensor Netw.*, vol. 4, no. 4, pp. 1–23, 2008.

[6] D. Berbecaru, "On measuring SSL-based secure data transfer with handheld devices," in *Proc. 2nd Int. Symp. Wireless Commun. Syst.*, Sep. 2005, pp. 409–413.

[7] European Network of Excellence in Cryptology II, "ECRYPT2 yearly report on algorithms and keysizes (2008-2009)," Jul. 31, 2009 [Online]. Available: http://www.ecrypt.eu.org/documents/D.SPA.7.pdf
[8] D. Boneh, G. Durfee, and M. Franklin, "Lower bounds for multicast message authentication," in *Proc. Eurocrypt*, May 2001, LNCS 2045, pp. 437–452.
[9] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas, "Multicast security: A taxonomy and some efficient constructions," in *Proc. IEEE INFOCOM*, Mar. 1999, vol. 2, pp. 708–716.
[10] A. Perrig, J. Tygar, D. Song, and R. Canetti, "Efficient authentication and signing of multicast streams over lossy channels," in *Proc. IEEE Security Privacy*, 2000, pp. 56–63.
[11] P. Golle and N. Modadugu, "Authenticating streamed data in the presence of random packet loss," in *Proc. NDSS*, 2001, pp. 13–22.
[12] S. Miner and J. Staddon, "Graph-based authentication of digital streams," in *Proc. IEEE Security Privacy*, 2001, pp. 232–246.
[13] R. Gennaro and P. Rohatgi, "How to sign digital streams," in *Proc. Crypto*, 1997, LNCS 1294, pp. 180–197.
[14] P. Rohatgi, "A compact and fast hybrid signature scheme for multicast packet authentication," in *Proc. ACM CCS*, 1999, pp. 93–100.
[15] A. Pannetrat and R. Molva, "Efficient multicast packet authentication," in *Proc. NDSS*, 2003.
[16] J. M. Park, E. Chong, and H. Siegel, "Efficient multicast packet authentication using signature amortization," in *Proc. IEEE Security Privacy*, 2002, pp. 227–240.
[17] A. Perrig, "The BiBa one-time signature and broadcast authentication protocol," in *Proc. ACM CCS*, Nov. 2001, pp. 28–37.
[18] Q. Wang, H. Khurana, Y. Huang, and K. Nahrstedt, "Time valid one-time signature for time-critical multicast data authentication," in *Proc. IEEE INFOCOM*, 2009, pp. 1233–1241.
[19] M. Bellare and S. Miner, "A forward-secure digital signature scheme," in *Proc. Crypto*, 1999, LNCS 1666, pp. 431–448.
[20] M. Abdalla and L. Reyzin, "A new forward-secure digital signature scheme," in *Proc. Asiacrypt*, 2000, LNCS 1976, pp. 116–129.
[21] G. Itkis and L. Reyzin, "Forward-secure signatures with optimal signing and verifying," in *Proc. Crypto*, 2001, LNCS 2139, pp. 332–354.
[22] T. Malikin, D. Micciancio, and S. Miner, "Efficient generic forward-secure signatures with an unbounded number of time period," in *Proc. Eurocrypt*, 2002, LNCS 2332, pp. 400–417.
[23] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.
[24] X. Boyen, H. Shacham, E. Shen, and B. Waters, "Forward-secure signatures with untrusted update," in *Proc. ACM CCS*, Oct. 30–Nov. 3, 2006, pp. 191–200.
[25] B. Libert, J.-J. Quisquater, and M. Yung, "Forward-secure signatures in untrusted update environments: Efficient and generic constructions," in *Proc. ACM CCS*, Oct. 28–31, 2007, pp. 266–275.
[26] *Digital Signature Standard (DSS)*, NIST FIPS 186-3, 2006.
[27] "Public key cryptography for the financial services industry, the Elliptic Curve Digital Signature Algorithm (ECDSA) ANSI X9.62, 2005.
[28] R. Merkle, *Secrecy, Authentication and Public Key Systems*. Ann Arbor, MI: UMI Research Press, 1982, also appears as a Stratford University Ph.D. dissertation in 1979.
[29] E. Cronin, T. Malkin, S. Jamin, and P. McDaniel, "On the performance, feasibility and use of forward-secure signatures," in *Proc. ACM CCS*, 2003, pp. 131–144.
[30] E. Cronin, "Libfss 0.2.0 library" [Online]. Available: http://fuji.cis.upenn.edu/wiki/Main/Signatures
[31] "OpenSSL library" [Online]. Available: http://www.openssl.org
[32] *Secure Hash Standard (SHS)*, NIST FIPS 180-2, 2004.
[33] D. Berbecaru and L. Albertalli, "An optimized double cache technique for efficient use of forward-secure signature schemes," in *Proc. IEEE PDP*, Feb. 13–15, 2008, pp. 581–589.
[34] D. Berbecaru and L. Albertalli, "On the performance and use of a space-efficient merkle tree traversal algorithm in real-time applications for wireless and sensor networks," in *Proc. IEEE Int. Conf. Wireless Mobile Comput., Netw. Commun.*, 2008, pp. 234–240.
[35] M. Szydlo, "Merkle tree traversal in log space and time," 2003 [Online]. Available: http://www.szydlo.com/logspacetime03.pdf, preprint version.
[36] M. Szydlo, "Merkle tree traversal in log space and time," in *Proc. Eurocrypt*, 2004, LNCS 3027, pp. 541–554.
[37] M. Jakobsson, T. Leighton, S. Micali, and M. Szydlo, "Fractal Merkle tree representation and traversal," in *Proc. RSA Cryptographers Track, RSA Security Conf.*, 2003, pp. 314–326.
[38] R. Rivest, "The MD5 message digest algorithm," RFC 1321, Apr. 1992.
[39] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.
[40] H. Orman and P. Hoffman, "Determining strengths for public keys used for exchanging symmetric keys," RFC 3766, Apr. 2004.
[41] NIST, "TWIRL and RSA key size," RSA Laboratories, Tech. Rep., May 2006 [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/key management.html
[42] B. Weis, "The use of RSA/SHA-1 signatures within encapsulating security payload (ESP) and authentication header (AH)," RFC 4359, Jan. 2006.
[43] A. Shamir and E. Tromer, "Factoring large numbers with the TWIRL device," in *Proc. Crypto*, 2003, LNCS 2729, pp. 1–26.
[44] B. Kaliski, "TWIRL and RSA key size," RSA Laboratories, Tech. Rep., May 6, 2003 [Online]. Available: http://www.rsa.com/rsalabs/node.asp?id=2004
[45] M. J. Moyer, J. R. Rao, and P. Rohatgi, "A survey of security issues in multicast communications," *IEEE Netw.*, vol. 13, no. 6, pp. 12–23, Nov./Dec. 1999.
[46] A. Lysyanskaya, R. Tamassia, and N. Triandopoulos, "Multicast authentication in fully adversarial networks," in *Proc. IEEE Security Privacy*, May 9–12, 2004, pp. 241–253.
[47] A. Pannetrat and R. Molva, "Authenticating real time packet streams and multicasts," in *Proc. IEEE ISCC*, 2002, pp. 490–495.

Diana Berbecaru received the M.Sc. degree in computer science from University of Craiova, Craiova, Romania, in 1998, and the Ph.D. degree in computer engineering from Politecnico di Torino, Torino, Italy, in 2003.

She is a Research Assistant with Politecnico di Torino. Her research interests include public key infrastructures and network security protocols in wired and wireless networks.

Luca Albertalli received the M.Sc. degree (summa cum laude) in computer engineering from both Politecnico di Torino, Torino, Italy, and Politecnico di Milano, Milano, Italy, in 2008. He also holds the Alta Scuola Politecnica degree.

He is currently working for Accenture, a worldwide consulting and service company, where he is involved in network evolution strategy and network engineering projects.

Antonio Lioy (M'89) received the M.Sc. degree (summa cum laude) in electronic engineering and the Ph.D. degree in computer engineering from Politecnico di Torino, Torino, Italy, in 1982 and 1987, respectively.

He is a Full Professor with the Politecnico di Torino, where he leads the TORSEC research group active in information systems security. His research interests are in the fields of network security, PKI, and policy-based system protection.

Prof. Lioy is a Member of the IEEE Computer Society.