Experimenting with Server-Aided Signatures

Xuhua Ding, Daniele Mazzocchi and Gene Tsudik
Information and Computer Science Department
University of California, Irvine
{*xhding,dmazzocc,gts*}*@ics.uci.edu*

November 13, 2001

Abstract

This paper explores practical and conceptual implications of using Server-Aided Signatures (SAS). SAS is a signature method that relies on partially-trusted servers for generating public key signatures for regular users. Besides its two primary goals of 1) aiding small, resource-limited devices in computing heavyweight (normally expensive) digital signatures and 2) fast certificate revocation, SAS also offers signature causality and has some interesting features such as built-in attack detection for users and DoS resistance for servers.

1 Introduction

Digital signatures represent a basic building block for many secure applications. Their uses range from electronic commerce transactions to secure email, secure content (code, video, audio) distribution and other, more specialized applications such as document notarization. Traditionally, digital signatures are based on asymmetric (public key) cryptographic techniques which, at least in some settings, makes them computationally expensive.

While digital signatures are rapidly becoming ubiquitous, one of the major recent trends in computing has been towards so-called "smart" devices, such as PDAs, cell phones and palmtops. Although such devices come in many shapes and sizes and are used for a variety of purposes, they tend to have one feature in common: limited computational capabilities and equally limited power (as most operate on batteries). This makes them ill-suited for complex cryptographic computations such as large number arithmetic present in virtually all public key constructs.

Furthermore, in many envisaged setting, such as cell telephony and wireless web access, personal devices are in constant contact with a fixed, wired infrastructure. Consequently, access to more powerful (in terms of both CPU speed and not dependent on batteries) computing platforms is available to end-users.

At the same time, increased use of digital signatures accentuates the need for effective revocation methods. Revocation of cryptographic credentials and certificates has been an issue for a long time. However, only now the problem is becoming truly visible, e.g., the recent Verisign fiasco where a wrong certificate was issued (ostensibly to Microsoft) and its subsequent revocation was both slow and painful. Furthermore, current CRL-based revocation methods scale poorly and are not widely used in practice. For example, most current web browsers do not bother checking CRLs; only the upcoming Windows XP has some rudimentary CRL-checking facilities.

Effective revocation not only useful but vital in some organizational settings (e.g., government and military) where digital signatures are used on important electronic documents and in accessing critical resources. Consider a situation when a trusted user (Alice) does something that warrants immediate revocation of her security privileges. Alice might be fired, transferred or she may suspect that her private key has been compromised. Ideally – immediately following revocation – no one should be able to perform any cryptographic operations involving Alice's certificate, i.e., sign with her private key.

In addition, when a cryptographic certificate is revoked (or simply expires) digital signatures generated prior to revocation (or expiration) may need to remain valid. This is difficult to achieve with current revocation methods since CRLs (and similar methods like OCSP [1]) do not provide a secure means of distinguishing between pre- and post-revocation signature activity. The only way to do so is by using a secure timestamping service for all signatures. Although a secure timestamping service may provide a secure means of distinguishing between pre- and post-

revocation signature, it has not been widely adopted due to its prohibitive cost. Finally, we note that compromise of a private key can lead to an unlimited number of fraudulent signatures being generated and distributed by the adversary. As often happens in the event of compromise, contact with the revocation authority (CA) may not be immediate, e.g., in a sporadically connected wireless network. Therefore, it is important to find a way to limit potential damage.

In this paper we present a method, called Server-Aided Signatures (SAS), that is designed to addresses the aforementioned issues. Its goals are three-fold:

1. Assist small, limited-power devices in computing digital signatures
2. Provide fast revocation of signing capability
3. Limit damage from potential compromise

The rest of the paper is organized as follows. Next section provides a brief synopsis of our work and its contributions. Section 5 describes the SAS method in greater detail; it is followed by the security analysis in Section 6. Denial of service issues are addressed in Section 7. Then, implementation and performance measurements are discussed in Section 8. The paper concludes with the summary of benefits and drawbacks of SAS.

2 Synopsis

The signature method (SAS) discussed here is based largely on a weak non-repudiation technique due to Asokan et al. [2]. The most notable feature of the SAS method is its **on-line** nature. Specifically, each SAS signature is generated with the aid of a partially-trusted server called a SEM (short for **SE**curity **M**ediator). This feature can be viewed as a mixed blessing. Although it offers a number of benefits which are summarized below, the requirement for on-line help for each signature is clearly a burden. We discuss the drawbacks, both real and perceived, in Section 9.

Informally, a SAS signature is computed as follows (see also Figure 1):

- First, a prospective signer (Alice) contacts her SEM and provides the data to be signed as well as a one-time *ticket*.
- SEM checks Alice's revocation status and, if not revoked, computes a half-signature over the data as well as other parameters (including the one-time ticket). SEM then returns the results to Alice.
- Alice verifies SEM's half-signature and produces her own half-signature. Put together, the two respective half-signatures constitute a regular, full SAS signature. This signature is accompanied by SEM's and Alice's certificates.

The two half-signatures are inter-dependent and each is worthless in and of itself. This is despite the SEM's half-signature being a traditional digital signature: in the context of SAS, a traditional signature computed by a SEM is not, by itself, a SAS signature. The half-signature computed by a user (Alice, in our example) is actually a one-time signature [3].

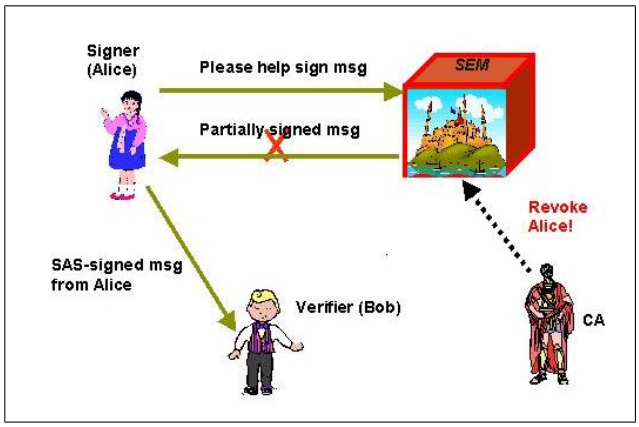

Figure 1: SAS architecture

Verifying a SAS signature is easy: verifier (Bob) obtains the signature and verifies the two halves along with the two accompanying certificates.

The main idea is that a SEM, albeit only partially trusted, is more secure, and much more capable (in terms of CPU and power consumption) than an average user. It can therefore serve a multitude of users. Also, because of its "superior" status, SEM is much less likely to be revoked or compromised. Since a signer (Alice) is assumed to have much less computing power then a SEM, the latter performs the bulk of the computation, whereas, Alice does comparatively little work. In the event that Alice's certificate is revoked, the SEM simply refuses to perform any further signatures on Alice's behalf. (See Figure 1.) Thus, revocation is both implicit and fast. However, this does not obviate the need for Certificate Revocation Lists (CRLs) since Alice's certificate may be revoked after some fraudulent signatures have been already generated. A CRL may still be necessary to convey to all verifiers the exact time of revocation and hence to sort out pre- and post-revocation signatures.

The general system model of SAS is a good fit for many mobile settings. For example, as mentioned in

Section 1, cell phones are only usable when in touch, via a nearby base station, with a fixed infrastructure. Each phone-call requires communication with the infrastructure. This communication can be overloaded to piggyback SAS protocol messages.

3 Related Work

The SAS method is based on a weak non-repudiation technique proposed by Asokan et al. in [2]. In very general terms, SAS is an instantiation of a mediated cryptographic service. Recent work by Boneh et al. [4] on mediated RSA (mRSA) is another example of mediated cryptography. mRSA provides fast revocation of both signing and decryption capability. However, the computation load on the client end is increased in mRSA, which is something that SAS aims to minimize.

In [5] Reiter and McKenzie propose a the same additive splitting technique to improve the security for portable devices where the private-key operations are password-protected. Recently, they also proposed another scheme for the more challenging problem of mediated (2-party) DSA signatures [6]. Ganesan[7] also exploited (earlier, in 1996) the same idea for improving Kerberos security as part of the Yaksha system.

Another way to look at SAS is as an instantiation of "hybrid" multi-signatures [8]. Viewed more broadly, the SAS method can be included in the more general framework of threshold cryptography[9] and secure multi-party computation[10].

There is also much related work on the topic of certificate revocation; including CRLs, Δ-CRLs, CRTs, 2-3 lists and skip-lists. This is reviewed in more detail in Appendix B.

4 Background

In this section we go over some preliminaries necessary for the remainder of the paper.

4.1 Hash Functions

Informally, a **one-way** function $f()$ is a function such that, given an input string x it is easy to compute $f(x)$, whereas, given a randomly chosen y, it is computationally infeasible to find an x such that $f(x) = y$. A one-way **hash** function $h()$ is a one-way function that operates on arbitrary-length inputs to produce a fixed length digest. If $y = h(x)$, y is commonly referred to as the *hash of* x and x is referred to as the *pre-image* of y. A one-way hash function $h()$ is said to be **collision-resistant** if it is computationally hard to find any two distinct input strings x, x' such that $h(x) = h(x')$.

Several secure and efficient collision-resistant one-way hash functions have been proposed, e.g., SHA or MD5 [11]. In the rest of the paper, $h()$ denotes a collision-resistant one-way hash function.

A collision-resistant one-way hash function can be recursively applied to an input string. The notation $h^i(x)$ is the result of applying $h()$ i times starting with the input x, that is:

$$\underbrace{h^i(x) = h(h(\ldots h(h(x))\ldots))}_{i \text{ times}}$$

Recursive application results in a *hash-chain* generated from the original input:

$$x = h^0(x), h^1(x), \ \ldots, \ h^n(x)$$

Hash chains have been widely used since early 1980-s starting with the well-known Lamport's method [12].

4.2 Model and Notation

We distinguish among 3 types of entities:

- *Regular Users* – entities who generate and verify SAS signatures.
- *Security Mediators (SEMs)* – partially-trusted entities assisting regular users in generating SAS signatures.
- *Certification Authorities (CAs)* – trusted off-line entities that issue certificates and link the identities of regular users with SEMs.

SEMs and CAs are verifiable third parties from the users' point of view.

All participants agree on a collision-resistant one-way hash function family $\mathcal{H}$ and a digital signature scheme. In SAS, the latter is fixed to be the RSA scheme [13]. Furthermore, each signer (Alice) selects a "personalized" hash function $h_A() \in \mathcal{H}$. In essence, $h_A()$ can be thought of as a keyed hash (e.g., [14]) with a known key set to the identity of the signer. When applied recursively, we also include the index of the hash function link in each computation, i.e., $h_A^i(x)$ can be thought of as a keyed hash where the known key is the concatenation of the signer's identity (Alice) and the index of the link, i.

In order to minimize computation overhead for regular users, $h()$ must be efficient and the digital signature scheme must be efficient for verifiers. (This is because, as will be seen below, verification is done

by regular users, whereas, signing is done by much more powerful SEMs.) SHA and MD5 are reasonable choices for the former, while RSA [13] satisfies the efficient verification requirement when used with a small exponent such as 3, 17 or 65,537.

4.3 Communication Channel

We assume that the communication channel between each user and a SEM is **reliable** (but neither private nor authentic). Reliability of the channel implies that the underlying communication system provides sufficient error handling to detect, with overwhelming probability, all corrupted packets. One way to achieve this is by having each protocol packet accompanied by its hash. Furthermore, timeouts and retransmissions are likewise handled by the communication system with the assumption that a packet eventually gets through.

We note that, even if the user is disconnected from the network[1] after sending a signature request to its SEM and before receiving a reply, the user will eventually obtain the correct reply (if the request ever reached the SEM) whenever the communication channel is re-established. Specifically, as described in the next section, a SEM always replies with the last signature it computed for a given user.

5 SAS Description

We now turn to the detailed protocol description.

5.1 Setup

To become a SAS signer, Alice first generates a secret quantity SK_A^0 randomly chosen from the range of $h_A()$. Starting with this value, Alice computes a hash-chain:

$$\{\ SK_A^0, SK_A^1, \ldots SK_A^{n-1}, SK_A^n\ \} \text{ where}$$

$$SK_A^j = h_A^j(SK_A^0) = h_A(SK_A^{j-1}) \text{ for } 1 \leq\ j \leq\ n$$

The last value, SK_A^n, is referred to as Alice's *SAS root key*. It subsequently enables Alice to produce $(n-1)$ SAS signatures.

Each SEM is assumed to have a secret/public RSA key-pair (SK_{sem}, PK_{sem}) of sufficient length. (We use the notation $[x]^{sem}$ to denote SEM's signature on string x). Each CA also has its own key-pair much like any traditional CA. In addition to its usual role of issuing and revoking certificates a CA also maintains a mapping between users and SEMs that serve them. This relationship is many to one, i.e., a SEM serves a multitude of users. Exactly how many depends on many factors, such as: SEM's hardware platform, average user signature request frequency, network characteristics, etc. We expect the number and placement of SEMs in an organizational network to closely resemble that of OCSP Validation Agents (VAs) [1].

In order to obtain a SAS certificate $Cert_A$, Alice composes a certificate request and submits it to the CA via some (usually off-line) channel. Alice's SAS certificate has, for the most part, the same format as any other public key certificate; it includes values such as the holder's distinguished name, organizational data, expiration/validity dates, serial number, public token key, and so forth. Additionally, a SAS certificate contains two other fields:

1. Maximum number of signatures n that the enclosed public key can be used to generate, and
2. Distinguished name and certificate serial number of the SEM serving the certificate holder.

Once issued, Alice's SAS certificate $Cert_A$ can be made publicly available via a directory service such as LDAP [15].

5.2 SAS Signature Protocol

The protocol proceeds as follows. (In the initial protocol run the signature counter $i = n-1$; it is decremented after each run. This counter is maintained by both SEM and Alice.)

Step 1. Alice starts by sending a request containing: $[\text{Alice}, m, i, SK_A^i]$ to its assigned SEM. If Alice does not wish to reveal the message to the SEM, m can be replaced with a suitable keyed (or, more accurately, randomized) hash such as the well-known HMAC [14]. (In that case, Alice would send $HMAC_r(m)$ where r is a one-time random value used a key in the HMAC computation.)

Alice may also (optionally) enclose her SAS certificate.

Step 2. Having received Alice's request, SEM obtains $Cert_A$ (either from the request or from local storage) and checks its status. If revoked, SEM replies with an error message and halts the protocol. Otherwise, SEM compares the signature index in the request to its own signature counter. In case of a mismatch, SEM replies to Alice with the

[1]This can happen if a wireless device, e.g., a cell phone, is momentarily out of range of any base station.

lowest-numbered half-signature produced in the last protocol run and aborts.

Next, SEM proceeds to verify the received public key (SK_A^i) based on Alice's SAS root key contained in the certificate. (If this is Alice's initial request, the signature counter is initialized to Having received Alice's request, SEM obtains $Cert_A$ (either from the request or from local storage) and checks its status. If revoked, SEM replies with an error message and halts the protocol. Otherwise, SEM compares the signature index in the request to its own signature counter. In case of a mismatch, SEM replies to Alice with the lowest-numbered half-signature produced in the last protocol run and aborts. Specifically, SEM checks that $h_A^{n-i}(SK_A^i) = SK_A^n$. In case of a mismatch, SEM replies to Alice with the last recorded half-signature and aborts the protocol.

Next, SEM signs the requested message with its private key to produce: $[Cert_A, m, i, SK_A^i]^{SEM}$. Other attributes may also be included in SEM's half-signature, e.g., a timestamp. SEM decrements Alice's signature counter, records the half-signature and returns the latter to Alice.

In the above, SEM assures that – for a given SAS certificate – exactly one signature is created for each $[i, SK_A^i]$ tuple. We refer to this property as the **SAS Invariant**.

Step 3. Alice (who is assumed to be in possession of SEM's certificate at all times) verifies SEM's half-signature, records it and decrements her signature counter. If SEM's half-signature fails verification or its attributes are wrong (e.g., it signs a different message than m or includes an incorrect signature counter $j \neq i$), Alice aborts the protocol and concludes that a hostile attack has occurred.[2] (See Section 7 below.)

Finally, Alice's SAS signature on message m has the following format:

$$SIG_i = [Cert_A, m, i, SK_A^i]^{SEM}, SK_A^{i-1}$$

The second part, namely SK_A^{i-1}, is Alice's half-signature. As mentioned earlier, it is actually a one-time signature: $h_A(SK_A^{i-1}) = SK_A^i$.

Note that Alice must use her one-time keys in strict sequence. In particular, Alice must not request a SEM half-signature using SK_A^{i-1} unless, in the last protocol run, she obtained SEM's half-signature containing SK_A^i.

[2]Our communication channel assumption rules out non-malicious packets errors.

5.3 SAS Signature Verification

SAS signature verification comes in two *flavors*: *light* and *full*. The particular choice depends on the verifier's trust model. Recall that the philosophy of SAS is based on much greater (yet not unconditional) trust placed in a SEM than in a regular user. If a verifier (Bob) fully subscribes to this, i.e., trusts a SEM more than Alice, he can chose light verification. Otherwise, if Bob is equally suspicious of SEMs as of ordinary users, he can choose full verification.

Light verification involves the following steps:

1. Obtain and verify[3] $Cert_{SEM}$
2. Verify SEM's RSA half-signature: $[Cert_A, m, i, SK_A^i]^{SEM}$
3. Verify Alice's half-signature: $h_A(SK_A^{i-1}) \stackrel{?}{=} SK_A^i$

Full verification requires, in addition:

4. Verify $Cert_A$
5. Check that $i < n$
6. Verify Alice's SAS root key: $h_A^{n-i}(SK_A^i) \stackrel{?}{=} SK_A^n$

Note that light verification does not involve checking Alice's SAS certificate. Although this may seem counter-intuitive, we claim that SAS signature format (actually SEM's half-signature) already includes $Cert_A$ as a signed attribute. Therefore, for a verifier who trusts the SEM, step 2 above implicitly verifies $Cert_A$.

It is easy to see that, owing to the trusted nature of a SEM and the **SAS Invariant**, light verification is usually sufficient. However, if a stronger property (such as non-repudiation) is desired, full verification may be used.

5.4 State and Registration

As follows from the protocol description above, both Alice and the SEM maintain state. Alice's SAS state amounts to the following:

$$Cert_A, Cert_{SEM}, SK_A^0, i, \{SIG_n, ..., SIG_{n-i-1}\}$$

[3]This may be done infrequently.

The first three values are self-explanatory. The fourth is Alice's current signature counter, (i), and the rest is the list of previously generated signatures for the same $Cert_A$. The state kept by the SEM (for each user) is similar:

$$Cert_A, i, \{SIG_n, ..., SIG_{n-i-1}\}$$

The amount of state might seem excessive at first, especially considering that some users might be on small limited-storage devices. There are some optimizations, however. First, we note that Alice can periodically off-load her prior signatures to some other storage (e.g., to a workstation or a PC when the PDA is charging). Also, it is possible to drastically reduce state maintenance for both users and SEMs if successive signatures are accumulated. For example, each SEM's half-signature can additionally contain the hash of the last prior SAS signature. This optimization results in storage requirements comparable to those of a traditional signature scheme.

Registration in SAS can be done either off- or on-line. In the off-line case, SEM obtains Alice's SAS certificate via manual (local or remote) installation by an administrator or by fetching it from the directory service. To register on-line, Alice simply includes her SAS certificate as an optional field in the initial SAS signature request to the SEM. Before processing the request as described above, the SEM checks if the same certificate is already stored. If not, it installs in the certificate database and creates a new user entry. (See Figure 2.)

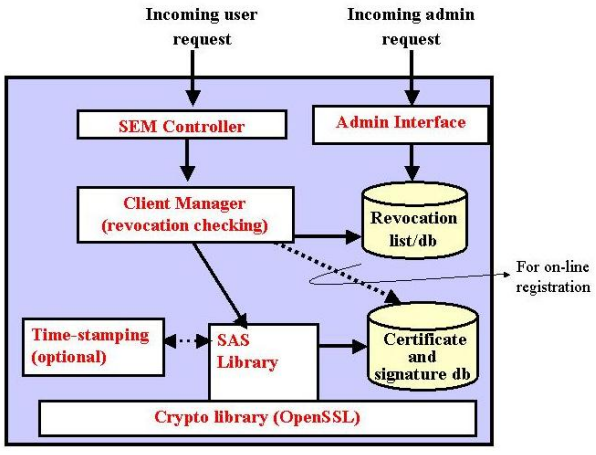

Figure 2: SEM architecture

6 Analysis

We now consider the efficiency and security aspects of the SAS signature method.

6.1 Efficiency

The cost of our signature protocol can be broken up as follows:

1. Network overhead: round-trip delay between Alice and SEM
2. SEM computation: signature computation plus other overhead (including hash verification of user's one-time public key, database processing, etc.)
3. User computation: verification of the SEM half-signature and other (commitment to storage) overhead.

Clearly, (1) and (3) are extra steps as compared with a traditional signature method. The extra cost of light signature verification (referring to the steps in the previous section) is only in Step 3 which consists of a single hash operation. Full verification costs an additional certificate validation (Step 4) as well as $(n - i)$ hash operations in Step 5.

6.2 Security Analysis

We claim that the SAS signature method achieves the same security level as a traditional digital signature scheme if SAS signature and verification protocols are executed correctly. Due to space limitations, we only present an informal security analysis.

To forge a SAS signature, an adversary can attempt to:

TYPE 1: forge a SEM's half-signature (i.e., an RSA signature) or

TYPE 2: find a quantity SK_A^* such that $H(SK_A^*) = SK_A^i$. Recall that SK_A^i is included in SEM's half-signature.

Clearly, a TYPE 1 attack is an attack on the underlying signature scheme, i.e., RSA, and, as such, is not specific to the SAS method. Therefore, we only consider TYPE 2 attacks. However, finding SK_A^* implies a successful attack on either the collision-resistance or the one-wayness property of the underlying hash function $h_A()$. Even we were to allow the possibility of the adversary mounting a successful TYPE 2 attack, the scheme remains secure if *full*

verification is used. (Recall that full verification includes not only checking $H(SK_A^*) \stackrel{?}{=} SK_A^i$ but also $h_A^{n-i}(SK_A^i) \stackrel{?}{=} SK_A^n$.)

We observe that, in any practical digital signature scheme, a collision-resistant one-way hash function is first applied to the message in order to produce a fixed-length digest which is then signed. Hence, a successful TYPE 2 attack on a SAS signature is, at the same time, an attack on the digital signature scheme.

6.3 Disputes

In case of a dispute between a signer (Alice) and a verifier (Bob), the latter submits the disputed SAS signature to an unbiased arbitrator who starts by verifying the following:

- Alice's and SEM's certificates are valid and certified by a CA.
- SEM's half-signature is valid.
- Alice's one-time key is a hash pre-image of the value in SEM's half-signature.
- The SAS root key in $Cert_A$ can be derived from the one-time public key by repeated hashing.

This is essentially the full SAS signature verification as described earlier. If any of the above steps fails, the arbitrator rules in Alice's favor. Otherwise, Bob wins the dispute.

Assuming the above procedure succeeds, Alice is asked to produce a different SAS signature with the same one-time key (i.e., same one-time signature). If Alice can come up with such a signature (meaning that the message signed is different from the one in the disputed signature), the arbitrator concludes that Alice's SEM cheated or was compromised. This conclusion is based on the apparent violation of the **SAS Invariant**. If Alice fails to produce a different signature, the arbitrator concludes that Alice attempted to cheat.

7 Denial of Service

The SAS signature protocol, unlike traditional signature schemes, involves multiple parties and communication. It is therefore subject to Denial of Service (DoS) attacks. Since we assume that the communication channel is reliable (cf. Section 4.3), only hostile DoS attacks are of interest. Also, our channel assumption states that all messages eventually get through; thus, attacks on the communication media are ruled out.

There are two types of DoS attacks: user attacks and SEM attacks. The purpose of a user attack is to deny service to a particular user whereas the purpose of a SEM attack is to deny service to all users served by a SEM. User attacks can be further divided into request and reply attacks. Request attacks involves modifying (or injecting) a user's signature request and a reply attack – modifying a SEM's reply.

7.1 User Attacks

Suppose that an adversary (Eve) intercepts the signature request and mounts a request attack. In this case, SEM receives a request that is perfectly legitimate (well-formed) from its point of view. It proceeds to sign it and send the signed reply back to Alice. Clearly, Alice discards the reply because it contains a signature for a different message. If Eve prevents the reply from reaching Alice, she gains no advantage since, as explained above, forging a signature requires Eve to come up with a one-time public key which she cannot do without breaking the hash function. Even if the reply does not arrive immediately, according to our communication assumption, it eventually reaches Alice who promptly detects an attack.

A slight variation on the above occurs when Eve has in her possession the last SAS signature generated by Alice. In this case, Eve can contact Alice's SEM with a well-formed request and without Alice's knowledge, i.e., Alice is off-line. However, this attack results in the same outcome as the above. This is because, eventually, Alice requests a new signature and SEM replies with the last (signed) reply. Alice, once again, detects an attack.

We note that these attacks can be prevented: one way to do so is for Alice not to reveal her i-th signature until $(i-1)$-st signature is computed. In other words, every other signature would be used strictly for this purpose. Then, if we suppose that Alice-SEM communication is private, revealing SIG_i to Bob (or Eve) is safe since a successful request to Alice's SEM would require knowledge of SK_{i-1} which Alice does not reveal until the next signature is requested. Yet another solution is to use a second, different hash chain for the sole purpose to authenticate Alice's requests to the SEM.

All in all, request attacks, while possible, are detected by the SAS signature protocol due to its "fail-stop" property: any manipulation of the signature request is detected by the user who can then invalidate its own certificate.

User reply attacks are comparatively less effective. If Eve modifies SEM's reply, short of forging an RSA signature, Alice detects that the reply is not what she expected and continues re-transmitting her signature request.

7.2 SEM Attacks

By virtue of serving a multitude of regular users, a SEM is a natural DoS attack target. This is not unique to SAS. For instance, it is easy to mount an effective DoS attack against an OCSP [1] (or even worse, a TSP [16]) server. It suffices for the adversary to flood the victim server with well-formed requests, i.e., requests for which the server is "authoritative" in OCSP. Since the server must digitally sign all replies, it will slowly grind to a halt.

In SAS, it is appreciably more difficult for the adversary to launch this type of an attack. The stateful nature of the SEM requires each signature request to be well-formed: it must contain the expected value of the current one-time public-key, i.e., the pre-image of the previously used public-key. All other requests are promptly discarded.

Therefore, in order to force the SEM to perform any heavy-weight tasks (of which signing is really the only one), the adversary must mount simultaneous user request attacks on as many users as possible thus hoping to flood the SEM. However, even if this were possible, the attack would quickly subside since the SEM will only perform a single signature operation per user before demanding to see a pre-image (next one-time public key). As we already established, finding the pre-image of the last signed one-time public key is computationally infeasible.

7.3 Loss of State

As SAS requires a non-trivial amount of state to be maintained by both users and SEMs, we need to consider the potential disaster scenarios that result in a loss of state.

Suppose that Alice looses all records of her prior signatures along with the signature counter. We further assume that she still has possession of her SAS certificate and the secret hash chain seed. Since these two values are fairly long-term, it is reasonable for Alice to store them in more permanent storage. Because of the "amnesia", Alice will attempt to obtain the initial signature from the SEM. Since SEM has retained all relevant state, it will reply with the last half-signature (including SEM's signature counter) generated for Alice's SAS certificate. Once she verifies the reply, Alice will realize her loss of state and resort to off-line means. However, if a malicious SEM is aware of Alice's loss of state, it can use this to its advantage by forging with impunity Alice's signatures.

If Alice looses her entire storage, including the SAS certificate, the consequences are not particularly dire. The SEM will simply keep state of Alice's "orphan" certificate until it eventually expires.

Any loss of SEM's state is much more serious. Most importantly, if the SEM looses all state pertaining to Alice's SAS certificate, the **SAS Invariant** property can no longer be guaranteed. (Consider, for example, malicious Alice re-establishing state of her SAS certificate on the SEM and then obtaining n signatures with the same hash chain.)

7.4 SEM Compromise

SEM compromise is clearly the greatest risk in SAS. The adversary who gains control of a SEM can un-revoke or refuse to revoke SAS user certificates. Moreover, it becomes possible to produce fraudulent user signatures: since state is kept of all prior SAS signatures (corresponding to active SAS certificates), the adversary can sign on behalf of Alice for each (SK_A^i, SK_A^{i-1}) pair found in SEM's storage.

Nonetheless, a defrauded SEM user can still have recourse if she faithfully keeps state of all prior SAS signatures. Referring to the SAS dispute resolution procedure, when an arbitrator is presented with two distinct and verifiable SAS signatures for the same (SK_A^i, SK_A^{i-1}) pair, he concludes that the SEM has attempted to cheat.

7.5 Suicide in SAS

In order to provide rapid and effective response to potential attacks, SAS includes a way for the user to "self-revoke" a SAS certificate. This is easily obtained by placing a new value (X.509 extension) in the SAS certificate. This value, referred to as the "suicide hash", is the hash of a randomly selected secret quantity generated by Alice when composing her certificate request. To self-revoke the certificate, Alice simply communicates the corresponding suicide pre-image to the SEM and the CA. As a result, the former simply stops honoring any further signature requests (pertaining to Alice's certificate) while the latter places a reference to the said certificate on the next CRL.

A similar technique has been suggested (with the value revealed by the CA instead) by Micali [17] as part of a proposal for an efficient revocation scheme.

8 Implementation and Experiments

To better understand the implications of using SAS and to obtain valuable experimental and practical data, we implemented the SAS scheme, first as a limping proof-of-concept prototype and, later, as a fully functional and publicly available package.

The implementation, for the most part, follows the protocol as presented in Section 5. The SAS certificate issuance is done strictly off-line: all users obtain their SAS certificates from the CA as described in Section 5.1. The newly issued certificates are either transferred to SEM off-line or piggybacked onto each user's initial SAS signature request. We limit our implementation discussion owing to space limitations; further details, including the SAS signature and SAS certificate formats can be found in Appendix A.

8.1 SAS Application Example: Eudora Plug-in

To demonstrate the ease and utility of the SAS signatures, we developed a plug-in (on top of the SAS user library [18]) for the popular Eudora [19] mailer.

When composing email, the sender simply clicks on the plug-in button. When ready to send, the plug-in reads the user's SAS certificate and extracts the SEM's address. It then communicates with the SEM to obtain a SAS signature on the email message. The resulting signed email is verified automatically by the Eudora plug-in on the receiver's side. Even if the receiver does not use Eudora, the SAS-signed email can be verified by any S/MIME capable email client such as Netscape Messenger or Microsoft Outlook. The verification, however, requires the receiver (verifier) to install a stand-alone SAS email verifier program. This program is registered as the viewer for the new MIME type (``x.SAS-signature'').

Figure 3 shows a screen snapshot of the Eudora message composition window when the user is ready to send a signed email. It is essentially the same as the normal Eudora screen except for the small SAS button at the toolbar along the top of the window. Figure 4 depicts a screen snapshot of the Eudora mailer showing a SAS-signed email message being received. The user is presented with a signature icon on the message screen; clicking on it causes the mailer to invoke the plug-in's verification function the output of which is displayed in the Figure 5.

To conserve space we omit the depiction of a user trying to sign email with a revoked certificate. In this case, the plug-in displays an error message informing the user of his certificate's demise. Further details on the Eudora plug-in can be found in Appendix A.

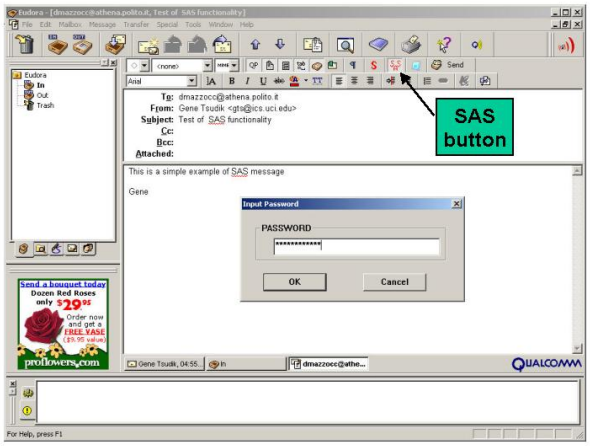

Figure 3: Snapshot of signer plug-in

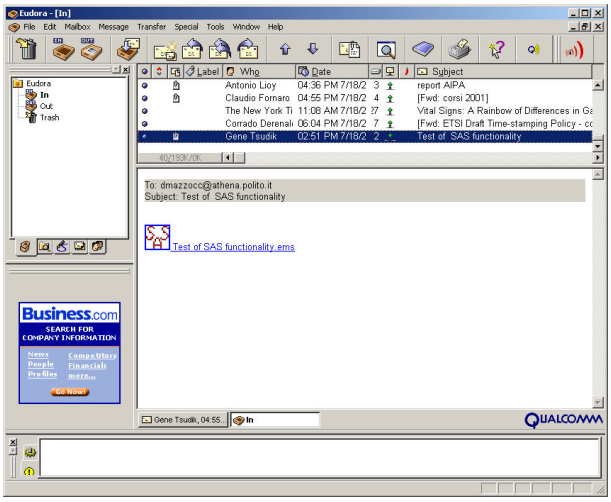

Figure 4: Verifier plug-in: signed email

8.2 Experimental Results

As emphasized in the introduction, one of the main goals of SAS is to off-load the bulk of signature computation from the weak user to the powerful SEM. To validate the goals and experiment with the SAS implementation, we ran a number of tests with various hardware platforms and different RSA key sizes.

All experiments were conducted over a 100 Mbit Ethernet LAN in a lab setting with little, if any, extraneous network traffic. All test machines ran Linux version 2.2 with all non-essential services turned off. The hardware platforms ranged from a measly 233-MHz PI (Pentium I) to a respectable 1.2-GHz PIV

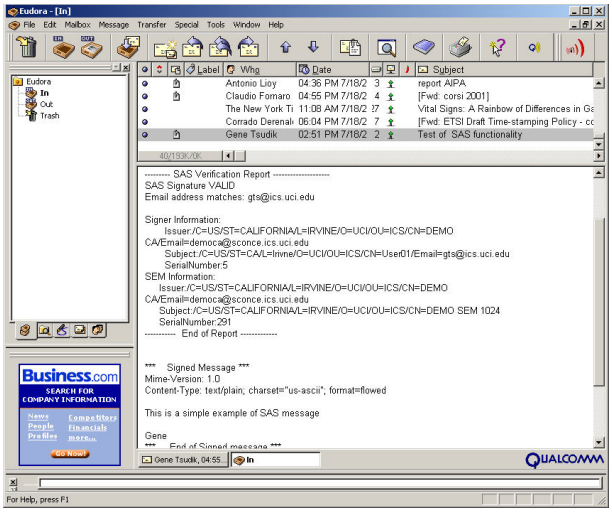

Figure 5: Verifier plug-in: verification

(Pentium IV). Note that we selected the lowest-end platform conservatively: only very high-end PDAs and palmtops approach 200-MHz processor speed; most are in the sub-100Mhz range. Our choice of the SEM platform is similarly conservative: a 933-MHz PIII. (At the time of this writing, 1.7-GHz platforms are available and affordable.)

Processor	**Key length (bits)**			
	1024	*2048*	*4096*	*8192*
PI-233 MHz	40.3	252.7	1741.7	12,490.0
PIII-500 MHz	14.6	85.6	562.8	3,873.3
PIII-700 MHz	9.2	55.7	377.8	2,617.5
PIII-933 MHz	7.3	43.9	294.7	2,052.0
PIV-1.2 GHz	9.3	58.7	401.2	2,835.0

Table 1: Plain RSA signature timings (ms)

First, we present in Table 8.2 plain RSA timings conducted with OpenSSL on the five hardware platforms. Table 8.2 illustrates the SAS timing measurements on the four user platforms with the SEM daemon running on a 933-MHz PIII. All SAS timings include network transmission time as well as SEM and user processing times. Finally, Table 8.2 shows the LAN round-trip communication delay between the user and the SEM, for different key sizes. The size of the signature request is determined by the digest size of the hash function, whereas, SEM's replies vary from roughly 164 bytes for 1024-bit RSA key to around 1,060 bytes for an 8K-bit RSA key.

We purposely used fairly conservative platforms for both the SEM and test users. The slowest user platform is a 233-MHz Pentium I laptop which is significantly faster than a typical PDA or a cell phone. The motivation was to show that, even a relatively fast user CPU, the speedup from SAS is appreciable. Clearly, a more realistic scenario would involve, for example, a 60- to 100-MhZ PDA as the user platform and a 1.7- to 2-GhZ PIV as a SEM.

As is evident from Table 8.2, all four user platforms experience noticeable speed-up as a result of using SAS, as compared with plain RSA. It is not surprising that the two low-end clients (233-MHz and 500-MHz) obtain a factor 4 to 6 speed-up depending on the key size. It is interesting, however, that the seemingly most powerful client platform (1.2-GHz PIV) also experiences a small speed-up. However, looking at Table 8.2, it becomes clear that the 1.2-GHz PIV is not the fastest platform after all. The explanation for this oddity rests with the chip maker.

Processor	**Key length (bits)**			
	1024	*2048*	*4096*	*8192*
PI-233 MHz	13.3	52.4	322.5	2,143.4
PIII-500 MHz	9.1	46.3	302.0	2,070.2
PIII-700 MHz	8.5	45.1	299.0	2,059.6
PIV-1.2 GHz	8.5	45.4	299.0	2,061.0

Table 2: SAS signature timings (ms)

To summarize, as Tables 8.2 and 8.2 illustrate, despite large variances in the four clients' CPU speeds, the difference in SAS sign time is very small. Moreover, the SAS sign time is only slightly higher than the corresponding value for the SEM (PIII-933 MHz) in Table 8.2, meaning that – communication delay aside – a SAS client can sign almost as fast as the SEM. The reason is that, to obtain a SAS signature, a user's cryptographic computation (which dominates the overall time) amounts to message hashing and signature verification. Hashing is almost negligible as compared to public key operations. RSA signature verification is also quite cheap in comparison to signing since we use small public exponents.

Processor	**Key length (bits)**			
	1024	*2048*	*4096*	*8192*
PI-233 MHz	0.6	0.7	1.1	1.7
PIII-500 MHz	0.4	0.5	0.8	1.2
PIII-700 MHz	0.1	0.2	0.2	0.3
PIV-1.2 GHz	0.4	0.5	0.8	1.2

Table 3: Network round-trip delay (ms)

9 Benefits and Drawbacks

In summary, the SAS signature scheme offers several important benefits as described below: **Efficient Signatures.** As follows from the protocol description and our experimental results, the SAS signature scheme significantly speeds up signature computation for slow, resource-limited devices. Even where speed-up is not as clearly evident (e.g., with small key sizes), SAS signatures conserve CPU resources and, consequently, power, for battery-operated devices.

Fast revocation. To revoke a SAS certificate, it is sufficient for the CA to communicate to the correct SEM. This can be achieved, for example, with CA simply issuing a new CRL and sending it to the SEM. Thereafter, the SEM will no longer accept SAS signature requests for the revoked certificate.

We remark that, with traditional signature schemes, the user who suspects that his key has been compromised can ask the CA to revoke the certificate binding this key to the user. However, the adversary can continue *ad infinitum* to use the compromised key and the verification burden is placed on all potential verifiers who must have access to the latest CRL. With SAS, once the SEM is notified of a certificate's revocation, the adversary is no longer able to interact with the SEM to obtain signatures. Hence, potential compromise damage is severely reduced.

More secure signatures. Since only a SEM performs real RSA public key operations (key generation, signature computation), it can do so with stronger RSA keys than would otherwise be used by the users. Indeed, a small PDA-like device is much less likely to generate high-quality (or sufficiently long) RSA factors (p, q) and key-pairs than a much more powerful and sophisticated SEM.

Signature Causality. Total order can be imposed over all SAS signatures produced by a given user. This is a direct consequence of the hash chain construction and the **SAS Invariant**. In other words, total ordering can be performed using the monotonically increasing signature counter included in each SAS signature.

Dispute Resolution. Signature Causality can be used to provide unambiguous dispute resolution in case of private key compromise. Recall that the compromise of a private key in a traditional signature scheme results in chaos. In particular, all prior signatures become worthless unless the use of a secure timestamping service is explicitly mandated for all signer and signatures. In SAS, once the time of compromise is established, signatures can be easily sorted into pre- and post-revocation piles.

Attack Detection. As discussed in Section 7, an adversary can succeed in obtaining a single fraudulent half-signature (not a full SAS signature) by substituting a message of its own choosing in the user's signature request. This essentially closes the door for the adversary since it is unable to obtain further service (short of inverting the hash function). The real user will detect that an attacks has taken place the next time when it tries to run the SAS signature protocol with its SEM.

Limited Damage. Even if the entire SAS hash chain is compromised (i.e., an adversary obtains the seed of the hash chain), the damage is contained since the adversary can generate at most n signatures. Furthermore, a user whose hash chain is compromised will detect the compromise the very next time she attempts to contact the SEM. (This is because the SEM will reply with its last half-signature ostensibly computed for the requesting user.)

Alas, the SAS scheme has some notable drawbacks as well:

• Each SEM is a single point of failure and a performance bottleneck for the users it serves.

• As discussed in Section 7, a SEM signs (with RSA, to produce its half-signature) a response to every well-formed signature request. This feature can be exploited by an adversary in order to mount a DoS attack. However, even the best attack can succeed in making a SEM sign at most once for each user it serves. Of course, an adversary can still flood any SEM with malformed requests which can certainly render a SEM unavailable to legitimate users.

• Unlike other mediated or multi-party signature methods (such as mRSA or 2-party DSA), SAS signatures are not compatible with any other basic signature type. In other words, SAS signatures are not transparent to verifiers. Therefore, all potential verifiers must avail themselves of at least the SAS verification method.

• It is possible, but neither easy nor elegant, for

a user to switch among different SEMs in SAS. One way is to have multiple SAS certificates; one for a distinct SEM. Another way is to use on-line hand-over of a SAS certificate among two SEMs. Neither solution is particularly attractive due to the difficulty of replication of a statfule server. (In mRSA [4], for example, a user can swich among SEMs transparently, where SEM is stateless.)

• SAS involves on-going state retention for regular users and SEMs. This burden is particularly heavy for SEMs (users can off-load their state periodically) since they must keep complete signature histories for all users served.

Acknowledgements

We thank Dan Boneh for some useful discussions, Ignacio Solis for early prototyping of the SAS library and Yongdae Kim for comments on the draft of this paper. We also gratefully acknowledge the anonymous referees whose comments served to greatly improve the final version of this paper.

References

[1] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, "RFC2560: Internet public key infrastructure online certicate status protocol - OCSP," June 1999.

[2] N. Asokan, G. Tsudik, and M. Waidner, "Server-supported signatures," *Journal of Computer Security*, vol. 5, no. 1, 1997.

[3] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology – CRYPTO '87* (C. Pomerance, ed.), no. 293 in Lecture Notes in Computer Science, (Santa Barbara, CA, USA), pp. 369–378, Springer-Verlag, Berlin Germany, Aug. 1988.

[4] D. Boneh, X. Ding, G. Tsudik, and B. Wong, "Instanteneous revocation of security capabilities," in *Proceeding of USENIX Security Symposium 2001*, Aug. 2001.

[5] P. MacKenzie and M. K. Reiter, "Networked cryptographic devices resilient to capture," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 12–25, May 2001.

[6] P. MacKenzie and M. K. Reiter, "Two-party generation of dsa signatures," in *Advances in Cryptology – CRYPTO '01* (J. Kilian, ed.), no. 2139 in Lecture Notes in Computer Science, pp. 137–154, Springer-Verlag, Berlin Germany, Aug. 2001.

[7] R. Ganesan, "Argumenting kerberose with pubic-key crytography," in *Symposium on Network and Distributed Systems Security* (T. Mayfield, ed.), (San Diego, California), Internet Society, Feb. 1995.

[8] C. Boyd, "Digital mulitsignatures," *Cryptography and Coding*, pp. 241–246, 1989.

[9] Y. Desmedt and Y. Frankel, "Threshold cryptosystems," in *Advances in Cryptology – CRYPTO '89* (G. Brassard, ed.), no. 435 in Lecture Notes in Computer Science, (Santa Barbara, CA, USA), pp. 307–315, Springer-Verlag, Berlin Germany, Aug. 1990.

[10] O. Goldreich, "Secure multi-party computation (working draft)," 1998.

[11] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc, 1996.

[12] L. Lamport, "Password authentication with insecure communication," *Communications of the ACM*, vol. 24, pp. 770–772, Nov. 1981.

[13] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Journal of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

[14] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology – CRYPTO '96* (N.Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 1–15, Springer-Verlag, Berlin Germany, 1996.

[15] S. Boeyen, T. Howes, and P. Richard, "RFC 2559: Internet x.509 public key infrastructure operational protocols - LDAPv2," 1999.

[16] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato, "Internet x.509 public key infrastructure time stamp protocol (tsp), draft-ietf-pkix-time-stamp-15.txt," May 2001.

[17] S. Micali, "Enhanced certificate revocation system," Tech. Rep. TM-542b, MIT/LCS, May 1996.

[18] "SAS plug-in web page," available at: *http://sconce.ics.uci.edu/sucses/*.

[19] "Qualcomm eudora mailer," available at: *http://www.eudora.com*.

[20] R. Laboratories, "Cryptographic message syntax standard," Public Key Cryptography Standards 7, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: ftp://ftp.rsa.com/pub/pkcs/. 1993.

[21] R. Housley, W. Ford, W. Polk, and D. Solo, "RFC 2459: Internet x.509 public key infrastructure certificate and crl profile," Jan. 1999.

[22] "The openssl project web page," *http://www.openssl.org*.

[23] P. Kocher, "On certificate revocation and validation," in *Financial Cryptography – FC '98, Lecture Notes in Computer Science, Springer-Verlag, Vol. 1465*, pp. 172–177, 1998.

[24] M. Naor and K. Nissim, "Certificate revocation and certificate update," in *Proceedings 7th USENIX Security Symposium (San Antonio, Texas)*, Jan 1998.

[25] M. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an authenticated dictionary with skip lists and commutative hashing," in *Proceedings of DARPA DISCEX II*, 2001.

[26] W. Aiello, S. Lodha, and R. Ostrovsky, "Fast digital identity revocation," in *Advances in Cryptology – CRYPTO '98* (H. Krawczyk, ed.), no. 1462 in Lecture Notes in Computer Science, Springer-Verlag, Berlin Germany, Aug. 1998.

Appendix A: SAS Implementation Details

A.1 SAS Signature Format

The well-known PKCS#7 [20] standard defines a general cryptographic message syntax for digital signatures. In it, `SignerInfor` includes an optional set of signed attributes as well as an optional set of unsigned attributes. This flexibility allows us to easily extend the PKCS#7 signature syntax to accommodate SAS signatures. This is because a SAS signature can be viewed as a regular public key signature with an appended extra value, i.e., the hash pre-image.

The format changes are only a few new requirements for `authenticatedAttributes` and `unauthenticatedAttributes` of the `SignerInfor` field. In a SAS signature, `SignerInfor` is the same as in plain PKCS#7, except:

- `authenticatedAttributes`:
  this field is not OPTIONAL, but MANDATORY. It must contain, at a minimum, two more attributes aside from those set in PKCS#7:
  - SAS_issuer_sn: IssuerAndSerialNumber – specifies the SAS client's certificate by issuer name and issuer-specific serial number
  - SAS_signed_token_index: INTEGER – specifies the SAS client signed one-time signature index (counter)
  - SAS_signed_token_value: OCTET STRING – specifies the SAS client signed one-time public key

  Note that PKCS#7 requires `issuerAndSerialNumber` in `SignerInfo` to identify signer's key. In SAS, this corresponds to SEM's key. Therefore, we require another field `SAS_issuer_sn` to identify the user's SAS certificate containing the SAS root key. The signed token is not placed into `ContentInfo` so that the message digest handling is the same as with any other public key signature type. Moreover, the token can be extracted from PKCS#7 independently, if necessary.

- `unauthenticatedAttributes`:
  this field is not OPTIONAL, but MANDATORY. It must contain:
  - `SAS_preimage_token_value`: OCTET STRING – specifies the SAS user's one-time hash pre-image of the signed token specified in `SAS_signedtoken_value`. This attribute is unsigned. It is inserted by the user when the SEM's half-signature is received and verified.

Because of format compatibility, a SAS signature can be shipped as a normal PKCS#7 signature. However, the verification method is obviously different. The normal PKCS#7 verification routines can only verify the SEM half-signature (i.e., RSA public key signature).

The extra step in (light) verification of a SAS signature is the comparison of the hash of `SAS_preimage_token_value` and the `SAS_signed_token_value` assuming light verification is used. Otherwise, as described above, the verifier checks the validity of `SAS_signed_token_value` and

`SAS_signed_token_index` by computing the iterative hash and comparing the result with the SAS root key in the signer's SAS certificate.

The fact that two parties participate in signing result in a semantic issue when SAS signatures are used in conjunction with S/MIME. Most S/MIME applications enforce a policy requiring the sender of the message (as shown in the RFC822 `From:` field) to match the e-mail address in the signer certificate. Unfortunately, in SAS, the sender is the holder of the SAS certificate, e.g., alice@wonderland.com. Whereas, the "signer" is the SEM, e.g., sem@wonderland.com. Therefore, a SAS verifier should be aware of the presence of the unsigned attribute and use the proper email address in comparison.

A.2 SAS Certificate

To support SAS attributes, we extended X509v3 handling [21] in the popular Openssl library [22]. In addition to the usual X509v3 fields, a SAS certificate also certifies the following:

- `SASHashType`: `DigestAlgorithmIdentifier` – identifies the hash algorithm used in generating the hash chain;
- `SASPublicKeyIdentifier`: `OCTET STRING` – SAS root key in the hash-chain.
- `SASPublicKeyPara`: `INTEGER` – length of the hash-chain.
- `SASServerName`: `STRING` – SEM's host name. This field indicates the location of SEM and has no security meaning.
- `SASSerialNumber`: `INTEGER` – SEM's certificate serial number. (Here it is assumed that the SEM and the user share the same CA). Uniquely identifies SEM's certificate and the corresponding public key.

A.3 Eudora Plug-in Details

We implemented the SAS plug-in as two email translators defined in Eudora's plug-in API [19]. Specifically, SAS signing is a *Q4-Transmission* translator and SAS verification is an *On-Display* translator.

SAS signing translator is invoked when Eudora is ready to send email and is fed with the entire email message, including its MIME header. When SAS signature protocol terminates, the whole SAS signature in PKCS#7 format is appended to the email body as an attachment with the MIME subtype `''x.SAS-signature''`.

SAS verification translator is called when Eudora is about to display a SAS-signed email. As in traditional signature verification, a certificate chain must be at hand. Our plug-in allows users to specify the root CA certificate, assuming, of course, that the SEM and the SAS client share the same certificate issuer. It is easy to build a chain by extracting SEM and client's certificate from the PKCS#7 signature. In this implementation, we chose not to adopt opaque signing. If the signature is invalid, an error message window is popped up while the original email body is still displayed.

Since SAS signature verification is different from normal S/MIME, non-Eudora applications, like Netscape or Outlook, cannot verify it without a special verification program. We provide such a standalone

Appendix B: Related Work on Certificate Revocation

• **CRLs and Δ-CRLs:** Certificate Revocation Lists are the most common way to handle certificate revocation. The Validation Authority (VA) periodically posts a signed list of all revoked certificates. These lists are placed on designated servers called CRL distribution points. Since these lists can get quite long, a VA may alternatively post a signed Δ-CRL which only contains the list of revoked certificates since the last CRL was issued. When verifying a signature on a message, the verifier checks that, at the time that the signature was issued, the signer's certificate was not on the CRL.

• **OCSP:** The Online Certificate Status Protocol (OCSP) [1] improves on CRLs by avoiding the transmission of long CRLs to every user and by providing more timely revocation information. The VA sends back a signed response indicating whether the specified certificate is currently revoked. When verifying a signature, the verifier sends an OCSP (certificate status request) query to the VA to check if the enclosed certificate is currently valid. The VA answers with a signed response indicating the certificate's revocation status. Note that OCSP prevents one from implementing stronger semantics: it is impossible to ask an OCSP VA whether a certificate was valid at some time in the past.

• **Certificate Revocation Trees:** Kocher [23] suggested an improvement over OCSP. Since the VA is a global service, it must be sufficiently replicated to handle the load of all validation queries. This means the VA's signing key must be replicated across

many servers which is either insecure or expensive (VA servers typically use tamper-resistance to protect the VA's signing key). Kocher's idea is to have a single highly secure VA periodically post a signed CRL-like data structure to many insecure VA servers. Users then query these insecure VA servers. The data structure (CRT) proposed by Kocher is a hash tree where the leaves are the currently revoked certificates sorted by serial number The root of the hash tree is signed by the VA.

A user wishing to validate a certificate issues a query to the closest VA server. Any insecure VA can produce a convincing proof that the certificate is (or is not) on the CRT. If n certificates are currently revoked, the length of the proof is $O(\log n)$. In contrast, the length of the validity proof in OCSP is $O(1)$.

• **Skip-lists and 2-3 trees:** One problem with CRTs is that, every time a certificate is revoked, the entire CRT must be recomputed and distributed in its entirety to the various VA servers. A data structure allowing for dynamic updates would solve this problem since the secure VA would only need to send small updates to the data structure along with a signature on the new root of the structure. Both 2-3 trees proposed by Naor and Nissim [24] and skip-lists proposed by Goodrich [25] are natural data structures for this purpose. Additional data structures were proposed in [26]. When a total of n certificates are already revoked and k new certificates must be revoked during the current time period, the size of the update message to the VA servers is $O(k \log n)$ (as opposed to $O(n)$ with CRT's). The proof of certificate's validity is $O(\log n)$, same as with CRTs.