Equipping “Smart” Devices With Public Key Signatures

Xuhua Ding
xhding@smu.edu.sg

Daniele Mazzocchi
mazzocchi@ismb.it

Gene Tsudik
gts@ics.uci.edu

Abstract

One of the major recent trends in computing has been towards so-called “smart” devices, such as PDAs, cell phones and sensors. Such devices tend to have a feature in common: limited computational capabilities and equally limited power, as most operate on batteries. This makes them ill-suited for public key signatures. This paper explores practical and conceptual implications of using Server-Aided Signatures (SAS) for these devices. SAS is a signature method that relies on partially-trusted servers for generating (normally expensive) public key signatures for regular users. Although the primary goal is to aid small, resource-limited devices in signature generation, SAS also offers fast certificate revocation, signature causality and reliable timestamping. It also has some interesting features such as built-in attack detection for users and DoS resistance for servers. Our experimental results also validate the feasibility of deploying SAS on smart devices.

1 Introduction

One of the major recent trends in computing has been towards so-called “smart” devices, such as PDAs, cell phones and sensors. Although these devices come in many shapes and sizes and are used for a variety of purposes, they tend to have two features 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. However, their weakness in computation is offset by their communication strength and operation fashion. For instance, cell phones and sensors always interact with their base stations; PDAs are regularly synchronized with PCs. Even though the advances in hardware technology increasingly strengthens the computation power of these devices, they are still in a demanding situation of resisting attacks from “stronger” adversaries. In fact, the computation power disparity between the smart devices and the adversary continues to become even larger.

Digital signatures are a basic building block for many secure applications. Two crucial features of digital signatures are non-repudiation and strong authentication of both origin and data. These features, especially non-repudiation,) are needed for the signatures to serve as a fair digital witness for commercial transactions and critical information transfer in general. However, traditional digital signatures are based on asymmetric (public key) cryptographic techniques which usually incur expensive computations.

Digital signatures are rapidly becoming ubiquitous. At the same time, increased use of digital signatures accentuates the need for effective revocation of cryptographic credentials and certificates. While this has been an issue for a long time, the problem is now becoming more evident. For example, in the recent Verisign fiasco, 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.

Effective revocation is 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 where a trusted user, Alice, does something that warrants immediate revocation of her security privileges. Alice might be fired, transferred or her private key may have been compromised. Ideally, immediately following revocation, no one should be able to perform any cryptographic operations involving Alice’s private key.

In addition, when a cryptographic certificate is revoked or simply expires, to establish the validity of digital signatures generated prior to revocation or expiration becomes a difficult issue due to the challenge of determining the exact generation time. Though a secure timestamping service may provide a means of distinguishing between pre- and post-revocation signatures, it hasn't been widely adopted due to its well-known prohibitive cost. Finally, compromise of a private key can lead to an unlimited number of fraudulent signatures being generated and distributed by the adversary. Therefore, it is important to find a way to limit potential damage.

Our Contribution The basic idea of Server-Aided Signatures was introduced in [1] as a non-repudiation technique. In this paper, we advance the previous results and present a full-fledged signature scheme to address 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

Moreover, we provide detailed security and performance analysis on both theoretical and application levels. We implement a prototype of the SAS signature scheme for experimental purposes. The computation and communication performance from our experiments validates our analysis and proves its feasibility for low-end devices.

Synopsis

The signature method discussed here, SAS, is based largely on a weak non-repudiation technique developed by Asokan et al. [1]. The most notable feature of the SAS method is the introduction of an online partially trusted entity. Specifically, each SAS signature is generated with the aid of a partially-trusted server called SEM (short for **SE**curity **M**ediator). This feature can be viewed as a mixed blessing. Although the requirement for on-line help for each signature is clearly a burden, it offers a number of benefits. We discuss the pros and cons, both real and perceived, in Section 9. The system model is elaborated in Section 4.

Informally, the basic SAS signature protocol is as follows:

- First, a prospective signer, Alice, contacts her SEM and provides the data to be signed as well as a one-time *token*.
- The SEM checks Alice's certificate validity and, if not revoked, computes a half-signature over the data as well as other parameters, including the one-time token. SEM then returns the results to Alice.
- Alice verifies the 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 the 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 fact that the SEM's half-signature is a traditional public key 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 [2] over the other half. Note that computing one-time signatures requires little computation resource.

Verifying a SAS signature is easy: after obtaining the signature, a verifier first verifies the correctness of the SEM's public key signature, then checks the link between two halves, i.e, verifies the user's (Alice's) one-time signature.

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. Moreover, because of its "superior" status, a SEM is much less likely to be revoked or compromised. (An organization's certificate usually has much longer life length than a personal certificate). Since a signer (Alice) is assumed to have much less computing power than 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 compute any further signatures on Alice's behalf. 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 already been 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 challenge is to offload the computation to an *untrusted* server without undermining users' security. We emphasize that utilizing a *fully-trusted* server to assist the low-end devices has several drawbacks. First, full trust usually implies poor scalability. Therefore the fully-trusted server, being a single point of failure in terms of security and availability, becomes an attractive target for various attacks. Second, in many applications, it is impractical to establish a centralized fully-trusted entity. Third, a fully-trusted server actually puts the users' security at risk, as a server compromise exposes all users' secret information.

2 Related Work

In this section, we review some related work and make comparisons against our scheme.

Offloading expensive computation from weak devices is a main functionality of SAS. A well-known related approach is online/offline signatures proposed by Even et al. [3]. The notion of an online/offline signature means that the signing process is broken into two phases. The first phase, performed offline, is independent of the particular message to sign while the second phase, performed online, is per message. An ordinary signature scheme $\mathcal{S}$ can be transformed into its online/offline variant by combining: 1) The said signature scheme $\mathcal{S}$; 2) A fast one-time signature scheme σ; and 3) A fast strong collision-free hash function $\mathcal{H}$, for which it is infeasible to find two inputs having the same hash value. The underlying idea is to use the $\mathcal{S}$ to sign offline a random construction of information which is later signed online using σ. This signature scheme is well suited for chip-card applications, where the expensive public key signature part is computed offline by a server, independent from the messages, while the cheap one-time signature is computed online by the chipcard. Its main disadvantage is that the resulted signature length is much longer than a normal one, even though Shamir made an improvement in [4]. Modadugu et al. [5], instead of targeting signature computation, proposed a method which helps handhelds generate RSA keys by using an untrusted server.

In terms of architecture, the SAS method is an example of a mediated cryptographic protocol. Recent work by Boneh et al. [6] on mediated RSA (mRSA) and Ding et al. [7] on mediated group signatures are other examples. Mediated cryptographic protocols share the feature of utilizing a partially trusted online server which, coupled with individual endusers, operates under a two-party computation model. With a similar model, Reiter and McKenzie proposed in [8] a technique to improve the security for portable devices where the private-key operations are password-protected. They also proposed another scheme for the more challenging problem of mediated (2-party) DSA signatures [9]. Earlier, in 1996, Ganesan [10] also exploited the same idea for improving Kerberos security as part of the Yaksha system.

Certificate revocation, another main feature of SAS, is a well-recognized problem in all current PKI-s. The main existing approaches are based on *Certificate Revocation Lists* (CRLs) or *Online Certificate Status Protocol* (OCSP).

CRLs and their variants are the most common way to handle certificate revocation. A Validation Authority (VA) *periodically* posts a signed list of all revoked certificates. These lists are placed on designated servers . Since these lists can get quite long, the 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 the signature was issued, the signer's certificate was not on any CRLs.

Compared with CRLs-based solutions, the approach using *OCSP* avoids the transmission of long CRLs to every user and provides more timely revocation information. 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 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. To improve the protocol scalability and remove the single point of failure, Kocher proposed *Certificate Revocation Trees* (CRT)[11] for OCSP. His idea is to have a single highly secure VA periodically post a signed CRL-like hash tree to many insecure VA servers. Users then query these insecure VA servers, which correspondingly produce a convincing proof that the certificate is (or is not) on the CRT. Further improvement on CRT include Naor and Nissim's 2-3 tree [12] and Goodrich's skip-lists [13]. Both of them enable the secure VA to issue efficient updates, instead of re-computing and distributing the entire CRT.

Both CRL-based and OCSP-based revocation approaches have essential drawbacks. Both approaches ask the verifier to take the risk and burden of checking the signer's certificate status. Besides the tremendous overhead, the verifier has difficulty in determining the exact signature generation time, which, in certain scenarios, is a vital issue . Furthermore, even though a signer's certificate has been revoked, she is still free to compute signatures and leaves the verifier to validate them. In practice, many users, including security professionals, simply ignore CRLs or are not willing to query an OCSP server.

The key cryptographic component in SAS systems is hash chains, which have been exploited in many protocols, e.g., Lamport's method [14] for password authentication and Perrig et al.'s protocol [15, 16] for packet stream authentication in multicast/broadcast settings. In [17], Bickaci et al. claim several improvements on our previous work [18]. Instead of using a hash chain as outlined in our scheme, they employ a one-time signature chain, which brings three major benefits. First, the user does not make any large number operations. Second, the resulted signature is exactly a traditional public key signature. Third, only one round of communication is required. However, due to the inefficiency of one-time signature size, its expensive communication cost offsets the benefits above.[1] The bit length of messages sent by the user is larger than the square of the hash length. Recent attacks on MD5 and SHA-1 [20] demand that hash functions have larger bit length. For instance, SHA-224 produces a 224-bit message digest and SHA-256 generates 256 bits. Using SHA-256 in [17] will result in user requests of more than 64K bits. Unfortunately, most low-end mobile devices have upstream channels with limited bandwidth. (Even in the forthcoming 3G cellular network, the upper bound of upstream bandwidth is only 64kbps.) It takes several hundreds milliseconds to one second to send a request. The incurred power consumption is even higher than regular public key operations, which takes less than one hundred milliseconds. Thus, our approach is more practical for a wide range of low-end devices whereas [17] is only fit for low-end devices with sufficient communication capability.

3 Preliminaries

Before proceeding to the core protocol description, we recall two cryptographic primitives which are utilized in constructing the SAS protocol.

Cryptographic Hash Function

A hash function $h()$ operates on arbitrary-length input 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 cryptographic hash function is usually required to be *one-way* and *collision-resistant*. Informally, a function $f() : X \rightarrow Y$ is one-way if, given an input element $x \in_{\mathcal{R}} X$, it is easy to compute $y = f(x)$, meanwhile, given a randomly chosen $y \in_{\mathcal{R}} Y$, it is computationally infeasible to find an x such that $f(x) = y$. Thus, a one-way hash function ensures that it is hard to compute a pre-image of a given value.

Since the domain of a hash function is much larger than its range, collisions do exist. However, for *collision-resistant* hash functions, it is difficult to find out a collision. More formally, $h()$ is collision-resistant if it is computationally intractable to find any two distinct input strings x, x' such that $h(x) = h(x')$. A collision-resistant one-way hash function can be recursively applied to an input string. We refer to $h^i(x)$ as

[1]Goyal proposed in [19] to use a seed to generate a one-time key pair, so that the memory cost at both the server and user ends is significantly reduced.

the result of applying $h()$ i times starting with the input x, that is:

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

Recursive application results in a *hash-chain* beginning with the original input:

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

Several secure and efficient collision-resistant one-way hash functions have been proposed. Due to the recent attack on hash functions [20], MD5 and SHA-1 have become insecure. SHA-224 and SHA-256 from the SHA family are promising candidates for SHA-1 and MD5. SHA-224 digests an arbitrary-sized message into a 224 bits string while SHA-256 produces a 256 bits string for any input. Both are much faster than large number computation.

Public Key Signatures

Digital signatures serve as a means for providing data integrity and non-repudiation. A public key based digital signature scheme $\mathcal{SIG}$ consists of three algorithms, namely $\mathcal{SIG} = (\texttt{Gen}, \texttt{Sign}, \texttt{Ver})$. On inputting an array of security parameters, a user $\mathcal{U}$ runs the probabilistic Gen to obtain a pair of public and private keys (pk, sk). On inputting a private key sk and a message m, $\mathcal{U}$ runs the probabilistic Sign to produce a signature $\sigma = \texttt{Sign}_{\mathcal{U}}(m)$. On inputting a public key pk, a message m and a tag σ, anyone can run the deterministic Ver to check whether σ is a valid signature. We require a signature scheme to be existentially unforgeable under the adaptive chosen-message attack model.[2] Many signature schemes, such as probabilistic RSA signatures [22, 23] and Schnorr[24] signatures, satisfy the security requirement of the SAS protocol.

Usually Sign and Ver have different computation complexity. For example, the RSA signature verification is much cheaper than RSA signature generation if its public exponent is particularly chosen, e.g., 65537. This fact will be exploited in the SAS protocol where the users will perform signature verification as shown in following sections.

4 Model and Notation

System Model

We now show the details of the whole architecture. It involves three entities:

- *Regular Users* – entities who sign messages using the SAS signature protocol. Some regular users may have limited computational resources and are incapable of executing expensive large-number operations.
- *Security Mediators (SEMs)* – partially-trusted entities assisting a set of regular users in generating SAS signatures.
- *Certification Authorities (CAs)* – trusted off-line entities that issue certificates and associate each regular user with one or multiple SEMs.

SEMs and CAs are high-end workstations or servers equipped with abundant computation and communication resources so that they are capable of handling heavy computation and communication load. Furthermore, they are better protected and less likely to be compromised than regular users.

As in common PKIs, a CA in the SAS protocol issues public key certificates for SEMs and regular users[3]. In addition, a CA binds each regular user with one (or several) SEM(s). A user can request signature generation assistance from the assigned SEM(s).

[2]Informally [21], the adaptive chosen-message attack model allows an adversary to access a *signature oracle*, which returns signatures on a polynomial number of messages chosen by the adversary, except the one he is challenged to sign.

[3]Regular users have a new form of public key certificates which is explained in Section 5.

Communication Channel

We do not assume that the communication channels between users and SEMs are private or authentic. Instead, they are assumed to be **reliable**. This implies that the underlying communication system provides a sufficient error handling mechanism to detect, with overwhelming probability, all benignly corrupted packets. Furthermore, timeouts and retransmissions are likewise handled by the communication system with the assumption that a packet eventually gets through. We assume that all communication channels connected with CAs are secure. The assumption is reasonable considering that a CA is an offline entity and any communication can be physically protected.

Trust Model

A remarkable feature of a SEM is that she is *partially* trusted. A SEM is trusted to honestly execute instructions from the administrator, e.g., helping in computing signatures for certain users or rejecting requests from others. However, a SEM may maliciously attempt to attack regular users. For instance, she may try to forge users' signatures or deny her contribution of a given signature. A SEM may even collude with other adversaries (including other dishonest SEMs, users, or external attackers) to mount attacks.

The trust model of CAs follows those in standard public key infrastructures. A CA is trusted not to introduce any phantom users and not to attack users or SEMs. We also assume the security of a CA's private key which is used to sign certificates.

Notation

Throughout the rest of the paper, we use the notations listed in Table 1.

U_i	Regular user U_i
SEM	Security mediator
$Cert_i$	User U_i's public key certificate
PK_i	User U_i's public key
SK_i^k	User U_i's k-th private key
$Cert_{sem}$	SEM's public key certificate
e_{sem}	SEM's RSA public key
d_{sem}	SEM's RSA private key
$\mathcal{H}_{u_i}()$	A cryptographically secure hash function used by U_i.
$SIG_k()$	A signing function using key k.
$VFY_k()$	A signature verification function using key k .

Table 1: SAS Protocol Notations

5 SAS Description

The SAS system consists of five component algorithms: *Setup, Sign, Verify, Handoff, Renew. Setup* initializes the settings for SEMs and regular users; *Sign* computes SAS signatures on given messages, which can later be validated by running *Verify*. *Handoff* algorithm allows a regular user to switch from one SEM to another. *Renew* algorithm allows a user to use new one-time private keys (a hash chain as shown below) without applying for a new certificate. We now proceed to describe each algorithm.

5.1 Setup

The system administrator sets up a CA and initializes a system-wide cryptographic setting. Specifically, the administrator selects a collision-resistant one-way hash function $\mathcal{H}()$ for the users. The choices of $\mathcal{H}()$

include SHA-224 and SHA-256, which are assumed to be secure. A public key signature scheme, which is secure against adaptive chosen message attacks, is selected for SEMs. In order to minimize computation overhead for regular users, the chosen public key signature scheme should 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.) Therefore, we choose RSA signature scheme [22, 23] with a small public exponent, such as 3 and 65,537 for SEMs.

To become a SAS signer, U_i customizes $\mathcal{H}()$ into $\mathcal{H}_{u_i}()$. In essence, $\mathcal{H}_{u_i}()$ is a keyed hash (e.g., [25]) with a known key set to the identity of the signer. Then, U_i generates a secret random element SK_i^0 and chooses n as the number of messages to sign. Starting with this value, U_i computes:

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

$$SK_i^j = \mathcal{H}_{u_i}(SK_i^{j-1}) = \mathcal{H}_{u_i}^j(SK_i^0) \text{ for } 1 \leq\ j \leq\ n$$

The hash chain of $(SK_i^0, SK_i^1, \ldots SK_i^n)$ is called U_i's *key chain.* Each SK_i^j, for $0 < j < n$ is U_i's j-th (one-time) private key. It subsequently enables U_i to produce $(n-1)$ SAS signatures, since as shown below, each of them will be used only once. The first value, SK_i^0, is referred to as U_i's *seed private key.* The last value, SK_i^n, together with n, are referred to as U_i's *root public key* PK_i.

Each SEM initializes its own secret/public RSA key-pair (d_{sem}, e_{sem}) of sufficient length. (We use the notation $[x]^{d_{sem}}$ 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 assigns associations between users and SEMs by listing SEMs in users' certificates. Each user has a unique *Registration SEM* in her home domain. Roaming users are allowed to have associations with *Alternative SEMs* in other domains, as shown in Section 5.4. One SEM serves a multitude of users. We expect the number and placement of SEMs in an organizational network to closely resemble that of OCSP Validation Agents (VAs) [26].

In order to obtain a SAS certificate $Cert_i$, U_i composes a certificate request and submits it to the CA via some (usually off-line) channel. U_i'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 and so forth. Additionally, a SAS certificate contains two other fields:

1. U_i's root public key PK_i, i.e $< SK_i^n, n >$
2. a pair of distinguished name and certificate serial number for each SEM associated with U_i.

Once issued, U_i's SAS certificate $Cert_i$ can be made publicly available via a directory service such as LDAP [27].

5.2 SAS Signature Protocol

To get the first signature from the SEM, U_i needs to register herself with her assigned SEM either off- or on-line. In the off-line case, the SEM obtains U_i's SAS certificate via manual (local or remote) installation by an administrator or by fetching it from the directory service. To register online, U_i 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 it in the certificate database and creates a new user entry.

We present below the signature protocol run by U_i and her registration SEM. (To run the same protocol with an alternative SEM, U_i must run a handoff protocol presented in Section 5.4.) In the initial run of the protocol, the signature counter k is set to $n-1$. Both the SEM and U_i consistently maintain the counter by decrementing it after each run. The protocol is illustrated in Figure 1.

Step 1. U_i starts by sending a request containing: $\{U_i, m, k, SK_i^k\}$ to its assigned SEM. If for privacy reasons U_i does not wish to reveal the message to the SEM, m can be replaced with $h(m)$. U_i may optionally enclose her SAS certificate.

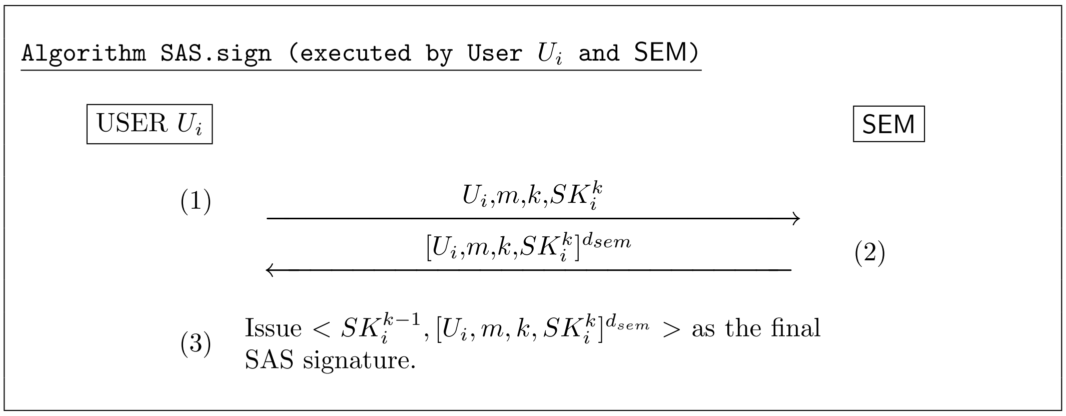
Algorithm SAS.sign (executed by User U_i and SEM)
USER U_i
SEM
(1)
U_i,m,k,SK_i^k
$[U_i,m,k,SK_i^k]^{dsem}$
(2)
(3)
Issue $< SK_i^{k-1}, [U_i, m, k, SK_i^k]^{d_{sem}} >$ as the final SAS signature.

Figure 1: SAS Signature Algorithm

Step 2. On receiving U_i's request, the SEM obtains $Cert_i$ (either from the request or from local storage) and checks its status. If revoked, the SEM replies with an error message and halts the protocol. Otherwise, the SEM compares the signature counter in the request to its own signature counter. In case of a mismatch, the SEM replies to U_i with the half-signature produced in the last protocol run and aborts. (Note that the SEM keeps a record of all previously generated half-signatures.)

Then, the SEM proceeds to verify the received k-th "private" key (SK_i^k) with U_i's root public key in $Cert_i$. Specifically, the SEM checks that $\mathcal{H}_{u_i}^{n-k}(SK_i^k) = PK_i$. In case of a mismatch, the SEM replies to U_i with the last recorded half-signature and aborts the protocol.

Otherwise, the SEM signs the requested message with its RSA private key d_{sem} using the RSASSA-PSS scheme specified in [23]. For simplicity, the result is denoted as $SIG_i = [Cert_i, m, k, SK_i^k]^{d_{sem}}$. Other attributes may also be included in the SEM's half-signature, e.g., a timestamp. the SEM decrements U_i's signature counter, records the half-signature and returns the latter to U_i.

In the above, the SEM assures that for a given SAS certificate, exactly *one* signature is created for each SK_i^k. We refer to this property as the **SAS Invariant**. This concept enables non-repudiation for SAS signatures and protects users from being framed by SEMs.

Step 3. U_i (who is assumed to be in possession of the SEM's certificate) verifies the SEM's half-signature, records it and decrements her signature counter. If the 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 $j \neq k$), U_i aborts the protocol and concludes that a hostile attack has occurred.[4]

In the end, U_i's SAS signature on message m has the following format:

$$[Cert_i, m, k, SK_i^k]^{d_{sem}}, SK_i^{k-1}$$

The second part, namely SK_i^{k-1}, is U_i's half-signature. As mentioned earlier, it is actually a one-time signature since $\mathcal{H}_{u_i}(SK_i^{k-1}) = SK_i^k$.
□

Note that U_i must use her one-time keys strictly in the reverse order of key generation, i.e. starting from SK_i^{n-1}, SK_i^{n-2}, SK_i^{n-3} and so on. In particular, U_i must not request a SEM half-signature using SK_i^{k-1} unless, in the last protocol run, she obtained SEM's half-signature containing SK_i^k.

[4]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. If a verifier trusts a SEM to honestly check user requests and verify user certificate status, he can choose light verification. Otherwise, he chooses full verification.

Light verification involves the following steps:

1. Obtain and verify[5] $Cert_{sem}$;
2. Verify the SEM's RSA half-signature: $[Cert_i, m, k, SK_i^k]^{d_{sem}}$;
3. Verify U_i's half-signature: $\mathcal{H}_{u_i}(SK_i^{k-1}) \overset{?}{=} SK_i^k$.

Full verification requires, in addition:

4. Verify $Cert_i$ and obtain n from $Cert_i$;
5. Check that $k < n$, otherwise abort;
6. Verify U_i root public key: $\mathcal{H}_{u_i}^{n-k}(SK_i^k) \overset{?}{=} SK_i^n$

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

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 SAS Handoff

The SAS protocol users could be low-end devices, which trade off their computational resource for better mobility. Binding the user to a single SEM impedes their mobility. The SAS handoff protocol addresses this problem by allowing users to switch their SAS service providers during roaming. A roaming user's SAS certificate contains a list of SEMs, among which one is designated as the *Registration SEM* to handle the user's first SAS request. Other SEMs on the list, referred to as the *Alternative SEM*, will not provide service unless an authorization token is presented.

The handoff protocol is similar to those in secure mobile IP networks. Their main idea is that the home server issues authorization tokens that helps the foreign server to authenticate the guest users' requests. In SAS, the user obtains a SAS signature on a pre-defined macro message from the current SEM before moving to a foreign domain. Suppose the next token for U_i to use is SK_i^k and U_i wants to switch service from the current server, SEM_a, to SEM_b. The following protocol is executed by U_i, SEM_a and SEM_b:

Step 1. U_i starts by sending a request containing: $\{M(a,b,i,k), SK_i^k\}$ to SEM_a. The enclosed $M(a,b,i,k)$ is a pre-defined macro message of U_i's request for handoff from SEM_a to SEM_b starting with token SK_i^k. $M(a,b,i,k)$ also contains other necessary information required by SEM_a to make an approval.

Step 2. SEM_a checks U_i's request integrity as in Step 2 in Section 5.2. In addition, SEM_a checks whether SEM_b is listed in U_i's certificate. If so, SEM_a returns a normal half SAS signature and stops the normal SAS signature service for U_i; otherwise a signed error message is returned to U_i and SEM_a halts the protocol.

Step 3. U_i receives the half signature from SEM_a and constructs a full SAS signature

$$\mathrm{TK}(U_i, \mathsf{SEM}_a, \mathsf{SEM}_b, k) = \{[M(a,b,i,k), SK_i^k]^{d_{\mathsf{SEM}_a}}, SK_i^{k-1}\}$$

as in Step 3 in Section 5.2. U_i presents $\mathrm{TK}(U_i, \mathsf{SEM}_a, \mathsf{SEM}_b, k)$ to SEM_b as an authorization token. SEM_b checks that: (1) both SEM_a and herself are listed on U_i's certificate; and (2) $\mathrm{TK}(U_i, \mathsf{SEM}_a, \mathsf{SEM}_b, k)$ is a

[5]This may be done infrequently.

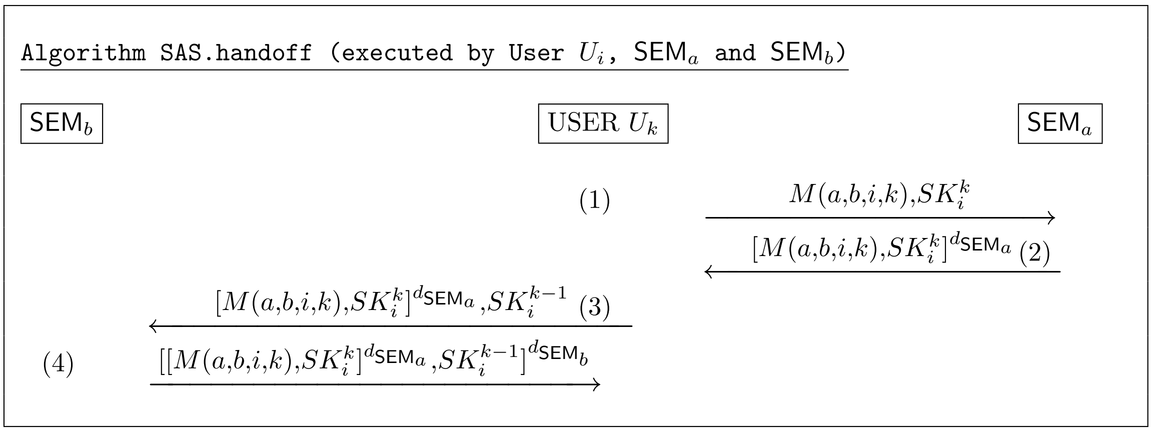
Algorithm SAS.handoff (executed by User U_i, SEM_a and SEM_b)
SEM_b
USER U_k
SEM_a
(1) $M(a,b,i,k),SK_i^k$
$[M(a,b,i,k),SK_i^k]^{d_{\mathsf{SEM}_a}}$ (2)
$[M(a,b,i,k),SK_i^k]^{d_{\mathsf{SEM}_a}},SK_i^{k-1}$ (3)
(4) $[[M(a,b,i,k),SK_i^k]^{d_{\mathsf{SEM}_a}},SK_i^{k-1}]^{d_{\mathsf{SEM}_b}}$

Figure 2: SAS Handoff Algorithm

valid SAS signature with respect to SEM_b and SEM_a. If both hold, SEM_b deposits $\mathrm{TK}(U_i, \mathsf{SEM}_a, \mathsf{SEM}_b, k)$ to a secure database and returns to an RSA signature on it to U_i. In case of dispute, $\mathrm{TK}(U_i, \mathsf{SEM}_a, \mathsf{SEM}_b, k)$ should be presented to prove the handoff authorization from SEM_a. U_i constructs a SAS signature:

$$< [\mathrm{TK}(U_i, \mathsf{SEM}_a, \mathsf{SEM}_b, k)]^{d_{SEM_b}}, SK_i^{k-2} >$$

which enables U_i to prove SEM_b's approval on handoff. To sign future messages, U_i starts from SK_i^{k-2}.

5.5 SAS Renewal

A renewal is needed when the messages to sign outnumber the length of the key chain or the states between the SEM and the user are inconsistent due to attacks or system failures. The renewal protocol allows a user to use a new chain of private keys without applying for a new certificate, on the condition that her seed private key is not compromised.

Suppose user U_i is currently using the hash chain seeded with SK_i^0 and the SEM is expecting SK_i^k. To shift to a new chain seeded with $SK_i'^0$, U_i and SEM run the following protocol shown in Figure 3:

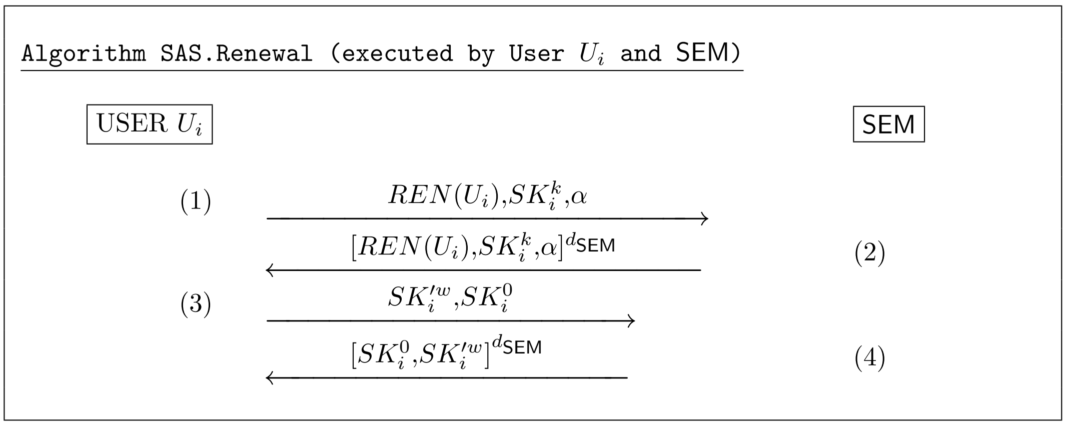
Algorithm SAS.Renewal (executed by User U_i and SEM)
USER U_i
SEM
(1) $REN(U_i),SK_i^k,\alpha$
$[REN(U_i),SK_i^k,\alpha]^{d_{\mathsf{SEM}}}$ (2)
(3) $SK_i'^w,SK_i^0$
$[SK_i^0,SK_i'^w]^{d_{\mathsf{SEM}}}$ (4)

Figure 3: SAS Hash-Chain Renewal Protocol

Step 1: In order to sign w messages in the future, U_i generates a new hash chain of length $w + 1$: $SK_i'^0 \ldots SK_i'^w$, and computes $\alpha = \mathcal{H}_{u_i}(SK_i^0, SK_i'^w)$. In the protocol message, $REN(U_i)$ is a pre-defined macro message indicating U_i's hash-chain renewal request; w is the index of new root public key $SK_i'^w$; SK_i^k

is a current hash token to use in the current hash chain; α serves as a commitment to the seed private key of the old chain and the root public key of the new chain.
Step 2: SEM checks the authenticity of SK_i^k and U_i's certificate status as in the SAS signature protocol. If the renewal is approved, SEM returns a signature on the request as a normal SAS signature. Meanwhile, the state is updated so that any future SAS signature requests using this chain will be rejected and an attack alarm should be signalled.
Step 3: If SEM's signature in the second round is verified as valid, U_i reveals to SEM the $SK_i'^w$ and SK_i^0.
Step 4: SEM checks if $\mathcal{H}_{u_i}(SK_i^0, SK_i'^w)$ equals α received in the first round. If true, SEM replies with an RSA signature on the SK_i^0 and $SK_i'^w$. The signature acts as a special "certificate" which, together with the certificate from CA, are attached with U_i's future SAS signatures.

6 Analysis

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

6.1 Security Analysis

We argue that the SAS signature protocol described in Section 5 is as secure as RSASSA-PSS, which is secure against adaptive chosen message attacks in the random oracle model. Namely, it is infeasible for an adversary, including a user, to forge a SAS signature on any message without the aid of a SEM mounting adaptive chosen message attacks. Moreover, it is infeasible for a SEM to frame a user without being held accounted. The security against existential forgery is informally discussed below, with a formal version presented in Appendix A. Section 6.3 shows how to identify a malicious SEM.

Intuitively, to forge one U_i's SAS signature, an adversary may attempt to:

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

TYPE II: find a quantity SK^* such that $\mathcal{H}_{u_i}(SK^*) = SK_i^k$. Recall that SK_i^k is signed by the SEM with the message.

Clearly, a TYPE I attack is on the underlying public key signature scheme, i.e., RSA, and, as such, is not specific to the SAS method. Therefore, we only consider TYPE II attacks. 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. Thus, the security of the signatures is dependent not only on the strength of the cryptographic algorithm, but also the hash algorithm. Breaking the hash function results in signature forgeries. In TYPE II attacks, finding SK^* essentially implies breaking either the collision-resistance or the one-wayness property of the underlying hash function $\mathcal{H}_{u_i}()$. Such an attack is, at the same time, an attack on the digital signature scheme, which leads to the conclusion that SAS signatures are virtually as secure as the public key signatures used by SEM.

Caveat We observe that malicious users may employ SEM as a signature oracle to answer their queries. Hence, the SAS protocol requires SEM's signature scheme be secure against adaptive chosen message attacks, so that SAS queries will not be abused to forge SEM's signatures.

6.2 State Maintenance

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

$$Cert_i, Cert_{sem}, SK_i^0, k, \{SIG_n, ..., SIG_{n-k-1}\}$$

The first three values are self-explanatory. The fourth is U_i's signature counter k, and the rest is the list of previously received signatures from SEM for the same $Cert_i$. Among the values, it is straightforward that SK_i^0 should be kept secret. Though other information can be publicly readable, the list of signatures should be protected from unauthorized deletion which will nullify the non-repudiation of the SAS signature.

The state for U_i kept by the SEM is similar:

$$Cert_i, k, \{SIG_n, ..., SIG_{n-k-1}\}$$

Note that the U_i's key index k (i.e., the number of signatures) is critical in authenticating U_i's requests. Hence, the integrity of relevant values should be protected by SEM against illegal tampering.

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 U_i 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.

6.3 Disputes

In case of a dispute between a signer (U_i) and a verifier (V) on the origin of a SAS signature, V submits the disputed SAS signature $\{[Cert_i, m, k, SK_i^k]^{d_{sem}}, SK_i^{k-1}\}$ to an unbiased arbitrator $\mathcal{A}$. $\mathcal{A}$ first executes a full SAS signature verification as described earlier. If the signature is false, $\mathcal{A}$ rules that U_i is not the originator and dismisses the case. Otherwise, U_i is asked to produce a different SAS signature with the same one-time key (i.e., same one-time signature). If U_i can come up with such a signature (meaning that the signed message is different from the one in the disputed signature), the arbitrator concludes that U_i's SEM cheated or was compromised. This conclusion is based on the apparent violation of the **SAS Invariant**. If U_i fails to provide a different version, the arbitrator concludes that U_i is the originator of this signature. For roaming users using multiple SEMs, the arbitrator can construct a chain of authorization rooted from U_i's Registration SEM. The arbitrator can detect a collusion between U_i and an Alternative SEM if the latter produced SAS signatures for U_i without an authorization token.

7 Denial of Service

The SAS signature protocol, unlike traditional signature schemes, involves multiple parties and communications. Although it resists forgery attacks, it is subject to Denial of Service (DoS) attacks. Since we assume that the communication channel is reliable, only hostile DoS attacks are of interest. Also, our channel assumption states that all messages eventually get through; thus, attacks on the communication infrastructure, e.g., bandwidth and routing, are ruled out. In general, we consider two types of DoS attacks on SAS service: user attacks and SEM attacks. The purpose of a user attack is to thwart a particular user from getting services whereas a SEM attack attempts to block the SEM from providing services to regular users.

7.1 User Attacks

User attacks can be further divided into request and reply attacks. Request attacks involves tampering with (or injecting) a user's signature request and a reply attack, basically modifying a SEM's reply. Suppose that an adversary, Eve, intercepts[6] the signature request from U_i and substitutes the original message with her own choice. In this case, the SEM receives a request that is perfectly legitimate (well-formed) from its point of view. It proceeds to sign and return it to U_i, via Eve again. Clearly, U_i will detect Eve's attack and discard the reply because it contains a signature for a different message. If Eve prevents the reply from reaching U_i, she gains no advantage since, as explained above, forging a signature requires Eve to come up with a one-time key which contradicts our assumption on the hash function.

A slight variation on the above occurs when Eve has in her possession the last SAS signature generated by U_i. In this case, Eve can contact U_i's SEM with a well-formed request and without U_i's knowledge, i.e.,

[6]This does not contradicts our assumption on the communication channel. Eve could be a router sitting between users and a SEM. She is able to manipulate all packets passing through her without changing their routes.

U_i is off-line. However, this attack results in the same outcome as the above. This is because of the **SAS Invariant**. Eventually, U_i requests a new signature and SEM replies with the last (signed) reply since a duplicated hash is used. U_i, once again, detects an attack. We note that this type of attack can be prevented: one way to do so is for U_i not to reveal her i-th signature until $(i + 1)$-st signature is computed.

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 her own certificate.

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

7.2 SEM Attacks

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 [26] (or even worse, a TSP [28]) 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 hash token, i.e., the pre-image of the previously used one-time token. All other requests will not incur any signature computation.

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 non-trivial 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 U_i loses 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 U_i to store them in more permanent storage. Because of her "amnesia," U_i 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 U_i's SAS certificate. Once she verifies the reply, U_i will realize her loss of state and resort to off-line means.

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

Any loss of SEM's state is much more serious. Most importantly, if the SEM loses all state pertaining to U_i, the **SAS Invariant** property can no longer be guaranteed. (Consider, for example, malicious U_i re-establishing the state of her SAS certificate on the SEM and then obtaining n signatures with the same hash chain.) Note that a regular user's state can be recovered from the SEM's state whereas the inverse is not secure, as a malicious user can cheat the SEM by hiding the last part of the signature list while a dishonest SEM can be detected by the user.

7.4 SEM Compromise

SEM compromise is clearly the greatest risk in SAS. The adversary who gains control of a SEM can unrevoke 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 U_i for each (SK_i^k, SK_i^{k-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_i^k, SK_i^{k-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 U_i when composing her certificate request. To self-revoke the certificate, U_i 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 U_i's certificate while the latter places a reference to the said certificate on the next CRL.

A similar technique, with the value revealed by the CA instead, has been suggested by Micali [29] as part of a proposal for an efficient revocation scheme.

8 Performance 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. We first analyze the protocol performance and then provide numerical experimental data.

8.1 Efficiency

The overall time cost for U_i to generate a SAS signature can be broken up as follows:

1. Network overhead: round-trip delay between U_i and the SEM. U_i needs to send less than one hundred bytes and receive a few hundred bytes (up to a few kilobytes if the SEM's public key certificate is attached). The traffic load is trivial in wired networks. Even in a wireless setting, modern wireless technology can easily handle such amount of data.

2. SEM computation: public key signature computation plus other overhead (including hash verification of user's one-time public key, database processing, etc.) Note that SEMs are high-end servers. If needed, they might even be equipped with specialized cryptographic hardware/firmware. An RSA signature generation only takes a few milliseconds.

3. User computation: verification of the SEM half-signature and other overhead, e.g., commitment to storage. Note that SEMs use an RSA signature scheme with a small public exponent so that the verification computation is considerably light. Hash computations are negligible.

Heuristically SEM's computation dominates the overall signature generation cost. Compared with public key signatures, regular users pay much fewer CPU cycles while they are well protected with the same security strength. The speed-up will be higher when SEM's computation power is stronger. To support our analysis, we provide below numerical results from our experiments.

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 100Mbps 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 233-MHz PI (Pentium I) to a respectable 1.2-GHz PIV (Pentium IV). Note that we selected the lowest-end platform conservatively: only high-end PDAs and palmtops approach 200-MHz processor speed. Our choice of the SEM platform is similarly conservative: a 933-MHz PIII. (At the time of this experiment, 1.7-GHz platforms were 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 2: Plain RSA signature timings (ms)

First, we present in Table 2 plain RSA timings conducted with OpenSSL on the five hardware platforms. It is interesting that the 1.2-GHz PIV is not the fastest platform for RSA operations after all. The explanation for this oddity rests with the chip maker. A possible explanation could be that the skimpy L1 cache in PIV results in more cache misses for cryptographic operations than PIII.

Table 3 illustrates the SAS timing measurements on the four user platforms with the SEM daemon running on a 933-MHz PIII. All SAS timings in Table 3 include the SEM and user processing time as well as network transmission cost. The size of the signature request is determined by the digest size of the hash function, whereas the SEM's replies vary from roughly 164 bytes for 1024-bit RSA key to around $1,060$ bytes for an 8K-bit RSA key. Although the network delay varies with the packet size, its effect on performance is negligible when compared to the corresponding computation cost.

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 3: SAS signature timings (ms)

Despite large variances in the four clients' CPU speeds shown in Table 2, the difference in SAS sign time is very small, as shown in Table 3. Moreover, the SAS sign time is only slightly higher than the corresponding value for the SEM (PIII-933 MHz) in Table 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. A decomposition of the SAS signature cost for PI-233 is shown in Figure 4. It shows the percentage of SEM computation, communication cost and user computation in computing a SAS signature. The dominance of SEM's computation cost becomes more significant when stronger security is demanded, which matches our analysis in Section 8.1. This performance characteristic motivates an organization to invest in a high-speed SEM to enhance its security.

The performance speed-up for all clients is shown in Figure 5, where data items in Table 2 are divided by their counterparts in Table 3. It is not surprising that the client with 233-MHz CPU obtains a factor 3

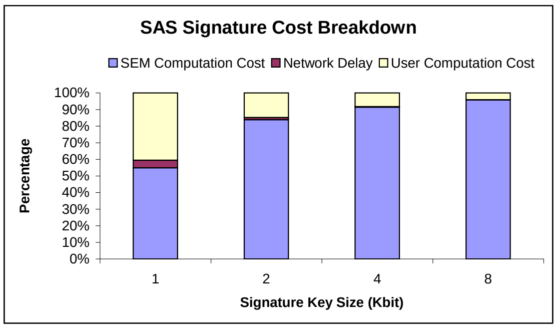

Figure 4: Decomposition of SAS signature cost

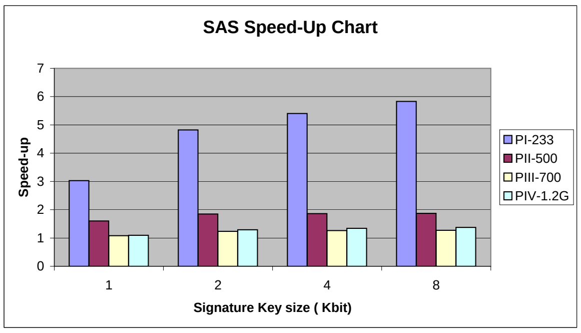

Figure 5: SAS Performance Speed-up

to 6 speed-up depending on the key size. When compared against clients with faster CPU, computational resource-constrained devices benefit more from the SEM's presence. The SAS signature scheme is not helpful for the purpose of improving performance for high-end devices.

8.3 SAS Application Example: Eudora Plug-in

To demonstrate the ease and utility of the SAS signatures, we developed a plug-in [30] (on top of the SAS user library) for the both the Outlook and Eudora [31] mailers.

Using Eudora as an example, the sender simply clicks on the plug-in button when composing emails. 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’’`).

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 SAS implementation can be found in Appendix A.

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, by the 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 relevant public key certificate. 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 SEMs perform expensive RSA operations (key generation, signature computation), they can do so with stronger RSA keys and better randomness 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.

Clear Dispute Resolution. Signature Causality and timestamps from SEM can be combined to provide unambiguous dispute resolution in the case of a 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 signers 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 attack has taken place the next time it tries to run the SAS signature protocol with its SEM.

Limited Damage (if Renewal is not allowed). 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. The hand-off protocol mitigates the problem to a certain extent.

• As discussed in Section 7, a SEM signs 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, like other online servers, with malformed requests which can 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. Therefore, all potential verifiers must avail themselves of at least the SAS verification method.

• SAS involves ongoing state retention for regular users and SEMs. This burden is particularly heavy for SEMs since they must keep complete signature histories for all users served. However, it is not as heavy for users because they can off-load their state periodically.

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 thank anonymous reviewers for constructive comments.

References

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

[2] 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.

[3] S. Even, O. Goldreich, and S. Micali, "On-line/off-line digital signatures," *Journal of Cryptology*, vol. 9, no. 1, pp. 35 – 67, 1996.

[4] A. Shamir and Y. Tauman, "Improved online/offline signature schemes," in *Advances in Cryptology – CRYPTO '2001*, pp. 355–367.

[5] N. Modadugu, D. Boneh, and M. Kim, "Generating rsa keys on a handheld using an untrusted server," in *RSA Conference, Cryptography Track, 2000.*

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

[7] X. Ding, G. Tsudik, and S. Xu, "Leak-free group signatures with immediate revocation," in *Proccedings of IEEE ICDCS 2004.*

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] A. Perrig, R. Canetti, D. Song, and D. Tygar, “Efficient and secure source authentication for multicast,” in *Proceedings of NDSS 2001.*

[16] A. Perrig, “The biba one-time signature and broadcast authentication protocol,” in *Proceedings of ACM CCS 2001.*

[17] K. Bicakci and N. Baykal, “Server assisted signatures revisited,” in *Proceedings of RSA Conference' Cryptography Track 2004.*

[18] X. Ding, D. Mazzocchi, and G. Tsudik, “Experimenting with server-aided signatures,” in *Proceedings of NDSS 2002.*

[19] V. Goyal, “More efficient server assisted one time signatures,” *available at http://eprint.iacr.org/2004/135.*

[20] X. Wang and H. Yu, “How to break md5 and other hash functions,” in *Advances in Cryptology – EUROCRYPT '2005.*

[21] S. Goldwasser, S. Micali, and R. Rivest, “A digital signature scheme secure against adaptive chosen-message attacks,” *SIAM J. Computing*, vol. 17, no. 2, 1998.

[22] 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.

[23] RSA Laboratory, “PKCS #1v2.1: RSA cryptography standard,” June 2002.

[24] C. P. Schnorr, “Efficient identification and signatures for smart cards,” in *Advances in Cryptology – CRYPTO '89*, pp. 239–252.

[25] 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.

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

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

[28] 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.

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

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

[31] “Qualcomm eudora mailer,” available at: *http://www.eudora.com.*

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

[33] “The openssl project web page,” *http://www.openssl.org.*

Appendix A: The Security of SAS Signatures

Theorem 1 *The SAS protocol in Section 5 is secure against existential forgery under adaptive chosen message attacks.*

Proof: Suppose there exists an algorithm $\mathcal{F}$ which succeeds in forging a SAS signature with probability P. Then, we are able to construct a simulator $\mathcal{S}$ which is allowed to access an RSA signature oracle $\mathcal{O}$ and succeeds in forgery with the same probability.

To forge an RSA signature with respect to key d, $\mathcal{S}$ runs $\mathcal{F}$ and simulates a SAS signature scheme where the SEM's key is set as d. $\mathcal{S}$ simulates users by computing hash chains for every simulated user. To simulate SEM and handle $\mathcal{F}$'s signature queries, $\mathcal{S}$ utilizes $\mathcal{O}$ as follows:

- For any SAS signature request needed by $\mathcal{F}$, $\mathcal{S}$ checks its legitimacy as in the SAS protocol. If valid, it is forwarded to $\mathcal{O}$ as an RSA signature query. The reply from $\mathcal{O}$ is returned to $\mathcal{F}$. Otherwise, an error message is returned as in the SAS protocol.
- For any RSA signature query from $\mathcal{F}$, it is forwarded to $\mathcal{O}$, whose reply is forwarded to $\mathcal{F}$ accordingly.

When $\mathcal{F}$ halts, it outputs a forged SAS signature $\{r, [m, k, SK^k]^d\}$ corresponding to message m and a user's i-th private key. If $[m, k, SK^k]$ has never been sent to $\mathcal{O}$, $\mathcal{S}$ outputs $[m, k, SK_i]^d$ as an RSA signature. It contradicts the fact that RSASSA-PSS has been proven secure under an adaptive chosen message attack, assuming reverting RSA function is hard under the random oracle model. Otherwise, $\mathcal{F}$ either reverses the one-wayness of the hash function, if SK^{k-1} is not revealed yet, or finds a second pre-image of SK^k, if $SK^{k-1} \neq r$. However, both contradict our assumption on hash function.
□

Appendix B: SAS Certificate

To support SAS attributes, we extended X509v3 handling [32] in the popular Openssl library [33]. 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` – root public 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.