Server-Supported Signatures *

N. Asokan†
IBM Research Division, Zurich Research Laboratory, Switzerland and
University of Waterloo, Canada

G. Tsudik‡
University of Southern California,
Information Sciences Institute, USA

M. Waidner§
IBM Research Division, Zurich Research Laboratory, Switzerland

Journal of Computer Security, to appear in Fall 1997

Abstract

Non-repudiation is one of the most important security services. In this paper we present a novel non-repudiation technique, called **Server-Supported Signatures, S^3.** It is based on *one-way hash functions* and *traditional digital signatures.* One of its highlights is that for ordinary users the use of asymmetric cryptography is limited to signature *verification.* S^3 is efficient in terms of computational, communication and storage costs. It also offers a degree of security comparable to that of existing techniques based on asymmetric cryptography.

Keywords: digital signatures, non-repudiation, electronic commerce, network security, distributed systems, mobility.

*A previous version of this work was presented at ESORICS '96 [1].

†e-mail: aso@zurich.ibm.com
‡e-mail: gts@isi.edu
§e-mail: wmi@zurich.ibm.com

1 Introduction

Computers and communication networks have become an integral part of many people's daily lives. Systems to facilitate commercial and other transactions have been built on top of large open computer networks. These transactions must often have some legal significance if they are to be useful in real life. Non-repudiation is one of the essential services necessary for attaching legal significance to transactions and information transfer in general.

Existing techniques for non-repudiation are based primarily on either symmetric or asymmetric cryptography. Practically secure symmetric techniques are computationally more efficient but require unconditional trust in third parties. "Unconditional" means that if such a third party cheats, the victim cannot prove this to an arbiter (e.g., a court). Practically secure asymmetric techniques (which we refer to as "traditional digital signatures") are computationally less efficient but can be constructed in a way that allows one to prove cheating by any third parties involved. We call a third party whose cheating can be proven to an arbiter a **verifiable third party.**

We present a novel non-repudiation technique called **Server-Supported Signatures, S^3.** It is based on *one-way hash functions* and *traditional digital signatures.* Like well-constructed asymmetric techniques, S^3 uses only verifiable third parties. However, for ordinary users, S^3 limits the use of asymmetric cryptographic techniques to signature *verification.* All signature *generations* are done by third parties, called *signature servers.* For some signature schemes, e.g., RSA with a public exponent of 3, verifying signatures is significantly more efficient than generating them.

2 Background and Motivation

The International Standardization Organization (ISO) is in the process of standardizing techniques to provide non-repudiation services in open networks. Current versions of the draft ISO standards [5] identify various classes of non-repudiation services. Two of these are of particular interest:

- **Non-repudiation of Origin (NRO)** guarantees that the originator of a message cannot later deny having originated that message.
- **Non-repudiation of Receipt (NRR)**[1] guarantees that the recipient of a message cannot deny having received that message.

Non-repudiation for a particular message is obtained by constructing a **non-repudiation token.** The non-repudiation token must be such that it can be verified by:

- the intended recipients of the token (e.g., in the case of NRO, the recipient of the message; in the case of NRR, the originator of the message), and
- in case of a dispute, by a mutually acceptable *arbiter.*

The draft ISO standards divide non-repudiation techniques into two classes:

- *Asymmetric non-repudiation techniques* are based on digital signature schemes using public-key cryptography. The main (and probably the only) difficulty in using digital signature schemes is the computational cost involved. This is a particularly serious issue when "anemic" portable devices (like mobile phones) are involved.

  Non-repudiation is based on certification of the signer's public key by a *certification authority.* Trust in this certification authority can be minimized by an appropriate *registration procedure.* For example, the signer and the authority may be required to sign a paper contract listing the signer's and certification authority's public keys, responsibilities, and liabilities, possibly in front of a notary public. In the worst case, the certification authority could cheat the user by issuing a certificate with a public key

[1] ISO documents call this "non-repudiation of delivery (NRD)." We use the term "receipt" because we feel that the term "delivery" is more appropriate to describe the function performed by the message transport system.

chosen by a cheater. But the supposed signer could deny all signatures based on this forged certificate by citing the contract signed during registration. Thus, trust is reduced to trust in the verifiability of the registration procedure.

- *Symmetric non-repudiation techniques* are based on symmetric message authentication codes (MACs) and trusted third parties that act as witnesses. Generating and verifying message authentication codes are typically low-cost operations compared to digital signature operations.

  The signer has to trust the third party unconditionally, which means that the third party could cheat the user without giving the user any chance to deny forged messages. One could reduce this trust by using several third parties in parallel or by putting the third party into tamper resistant hardware. These two approaches increase both cost and complexity but neither of them solves the problem completely.

In the following we present a new, low-cost technique for non-repudiation services, called *server-supported signatures*. It uses both traditional digital signatures (based on asymmetric cryptographic techniques) and one-way hash functions in order to minimize the computational costs for ordinary users. Our main motivation arises from the typical mobile computing environments where the mobile entities have considerably less computing power than do static entities.

3 Server-Supported Signatures for Non-repudiation of Origin

3.1 Preliminaries: One-way Hash Functions

Intuitively, a one-way function $f()$ is a function such that given an input string x it is easy to compute $f(x)$, but given a randomly chosen y it is computationally infeasible to find an x' such that $f(x') = y$. A one-way hash function is a one-way function $h()$ that operates on arbitrary-length inputs to produce a fixed length value. The term x is called a *pre-image* of $h(x)$. A one-way hash function $h()$ is said to be *collision-resistant* if it is computationally infeasible to find any two strings x and x' such that $h(x) = h(x')$. Collision-resistance implies one-wayness [13, Section 7.2]. A number of efficient and allegedly one-way hash functions, such as SHA[8], have been invented. One-way hash functions can be recursively applied to an input string. The notation $h^i(x)$ denotes the result of applying $h()$ i times recursively to an input x. That is,

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

Such recursive application results in a *hash-chain* that is generated from the original input string:

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

3.2 Model and Notation

We distinguish three types of entities in the system:

- *Users* – participants in the system who wish to avail themselves of the non-repudiation service while sending and receiving messages among themselves.
- *Signature Servers* – special entities responsible for actually generating the non-repudiation tokens on behalf of the users.
- *Certification Authorities* – special entities responsible for linking public keys with identities of users and servers.

Signature servers and certification authorities will be verifiable third parties from the users' point of view.

All entities agree on a one-way, collision-resistant hash-function $h()$ and a digital signature scheme. Entities should "personalise" the hash function. For example, this can be done by always including their unique name as an argument: using $h(O, x)$, where O is the entity computing the one-way hash. We use $h_O()$ to refer to the personalised hash function used by O.

The result of digitally signing a message x with signature key SK is denoted by $(x)SK$. We also assume that, given $(x)SK$, anyone can extract x from it provided they have the public key corresponding to SK. The users' security depends on the one-way property of $h_O()$, which must hold even against the servers. In practice, this is not a problem because hash functions such as SHA are one-way for *all* parties. We note, however, that so-called cryptographically strong hash-functions are usually invertible for the party that generated the hash-function.

In order to minimize the computational overhead for users, $h_O()$ must be efficiently computable, and digital signatures must be efficiently *verifiable*. Only signature servers and certification authorities need to have the ability to *generate* signatures. SHA as hash function and RSA with public exponent 3 as signature scheme would be reasonable choices.

Each user, O (O as in Originator) generates a secret key, K_O, randomly chosen from the range of $h_O()$. Based on K_O, user O computes the hash chain $K_O^0, K_O^1, \ldots K_O^n$, where

$$K_O^0 = K_O,\ K_O^i = h_O^i(K_O) = h_O(K_O^{i-1})$$

$PK_O = K_O^n$ constitutes O's *root public key*. Root public key K_O^n will enable O to authenticate n messages. This is not a limitation: before the old root public key is consumed completely a new root public key can be generated and authenticated using the old root public key.

Each signature server S generates a pair of secret and public keys (SK_S, PK_S) of the digital signature scheme. Each certification authority, CA, does the same. The CA is responsible for verifiably binding a user O (server S) to her root public key PK_O (its public key PK_S). We assume that the registration procedure is constructed such that CA becomes a verifiable third party.

Notation Summary
$h_O()$ - one-way collision-resistant hash function for O
SK_X - secret key known only to entity X
K_O^i - user O's $(n-i)$-th public key
$(x)SK$ - digital signature on message x with secret key SK
$[m]$ - Message m sent via a confidential channel

3.3 Initialization

To participate in the system, a user O chooses a signature server S that shall be responsible for generating signatures on O's behalf, generates a random secret key K_O, and constructs the hash chain. As will be described below, O can cause S to transfer the signature generation responsibility to another signature server S', if required (e.g., because O is a mobile user who wishes to always use the closest server available).

O submits the root public key $PK_O = K_O^n$ to a CA for certification. A certificate for O's root public key is of the form

$$Cert_O = (O, n, PK_O, S)SK_{CA}$$

We ignore all information typically contained in a certificate but not relevant to the discussion at hand, e.g. organizational data such as serial numbers and expiration dates. The registration performed by O and CA must be verifiable, as discussed above. CA may make the certificate available to anyone via a directory service. O then deposits the certificate received from CA with S.

Each signature server S acquires a certificate containing PK_S from its CA. As these are ordinary public key certificates, we do not describe them here.

For the sake of simplicity, we do not include the certificates in the following protocols. They might be attached to other messages or retrieved using a directory service. We assume that the necessary certificates are always available to anyone who needs to verify a signature.

3.4 Generating NRO Tokens

The basic idea is to exploit the digital signature generation capability of a signature server to provide non-repudiation services to ordinary users. The basic protocol, providing non-repudiation of origin, is illustrated in Figure 1. We assume that a user O wants to send a message m along with an NRO token to some recipient R. The first protocol run uses $i = n$; i is decreased during each run.

1. O begins by sending (O, m, i) to its signature server S along with O's current public key K_O^i in the first protocol flow (in case O does not want to reveal the message to S for privacy reasons, m can be replaced by a randomised hash of m, computed using a collision-resistant hash function).

2. S verifies the received public key based on O's root public key (and O's certificate obtained from CA), i.e., checks that $h_O^{n-i}(K_O^i) = PK_O$. The signature server S has to ensure that only one NRO can be created for a given (O, i, K_O^i). If a message on behalf of O containing K_O^i has not yet been signed, S signs (O, m, i, K_O^i), records K_O^i as consumed, and sends the signature (which we call the *candidate non-repudiation token*) back to O in the second flow.

3. O verifies the received signature and records K_O^i as consumed by replacing i by $i-1$. The NRO token for R now consists of

$$(O, m, i, K_O^i)SK_S, K_O^{i-1}$$

   O produces this token which actually authenticates m, by revealing K_O^{i-1}.

   In Figure 1 we assumed that the NRO token is sent to R via S in the third flow. Alternatively, O can send the token directly to R.

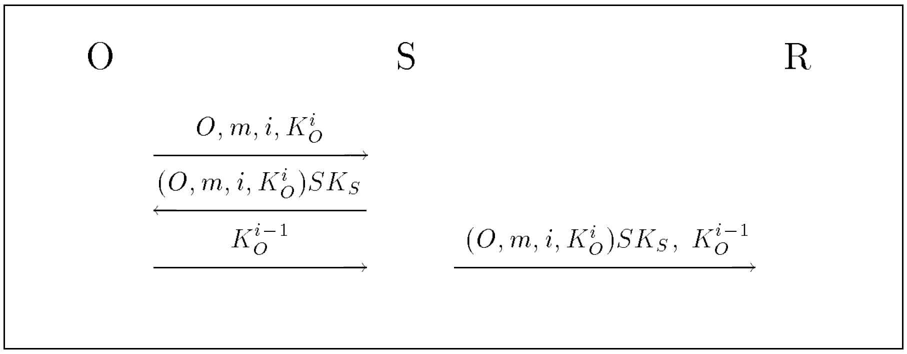

Figure 1: Protocol providing non-repudiation of origin.

K_O^i is referred as the **token public key** of the $(n-i+1)$th non-repudiation token, $(O, m, i, K_O^i)SK_S, K_O^{i-1}$. Note that O must consume the token public keys in sequence and must not skip any of them. In particular, O must not ask for a signature using K_O^{i-1} as token public key unless she has received S's signature under K_O^i. Otherwise, S could use that to create a fake non-repudiation token, which O cannot repudiate during a dispute.

3.5 Dispute Resolution

In case of a dispute, R can submit the NRO to an arbiter. The arbiter will verify the following:

- the public keys are certified by CA,
- the signature in the token by the signature server is valid,
- the token public key is in fact a hash of the alleged pre-image in the token, and
- the root public key can be derived from the token public key by repeated hashing.

If these checks are successful, then the originator is allowed the opportunity to *repudiate* the token by

- proving that CA cheated:
  - If O has registered with CA, O can show a certificate on a different root public key.
  - Otherwise CA will be asked to prove that the root public key was registered by O (i.e., by showing the signed contract with O).
- proving that S cheated by showing a different non-repudiation token corresponding to the same token public key.

Note that in case CA is honest, to claim falsely that O has sent a message m', a cheating R has to produce an NRO token of the form:

$$(O, m', i, K_O^i)SK_S, K_O^{i-1}$$

If O has not revealed K_O^{i-1} yet, then one-wayness of $h_O()$ implies that anyone else will find it computationally infeasible to generate this NRO token, even if K_O^i is known. If O has already revealed K_O^{i-1} she must have sent K_O^i to S before. According to the protocol, O reveals K_O^{i-1} only if she has received a signature from S under K_O^i which satisfied her. Therefore, O can show a different token corresponding to the same token public key.

Suppose an adversary of O successfully breaks the one-wayness of $h_O()$ and obtains an NRO token of the form:

$$(O, m', i, K_O^i)SK_S, x$$

where $h_O(x) = h_O(K_O^{i-1})$. If x is different from K_O^{i-1}, then on being challenged with this NRO token, O can reveal K_O^{i-1}, proving that the system has been broken. This is known as the "fail-stop" property. Assuming that h_O has a uniform distribution, the domain used must be larger than the range of h_O in order to achieve a reasonable level of fail-stop property. We can do this by slightly modifying the building procedure to include a random padding to the input of h_O during the computation of every link in the chain. The sequence of random pads used are generated using a pseudo-random number generator whose seed is committed to in the certificate.

4 Server-Supported Signatures for Non-repudiation of Origin and Receipt

Non-repudiation of receipt (NRR) can be easily added to the basic protocol. Before sending K_O^{i-1} to R, S can ask R for an NRO token for "NRR"$|m$, which is then passed on to O. This is illustrated in Figure 2. The NRR token consists of:

$$(R, \text{“NRR”}|m, j, K_R^j)SK_S, K_R^{j-1}, r$$

Square parentheses ([ ]) indicate that the message contained within them is sent via a confidential channel. As this protocol is just two interleaved instances of the basic NRO protocol, it still guarantees that O and R can repudiate all forged NRO and NRR tokens, respectively. Note that this protocol actually implements *fair-exchange* of the NRO token for m and its NRR token, based on S as a trusted third party. If S behaves dishonestly, no fairness can be guaranteed: O might not receive the NRR token or R might not receive the NRO token.

The protocol as depicted in Figure 2 allows the possibility that R may refuse to send the NRR token after having received the candidate NRR token from S (from which R can extract m). An alternative approach is to include only a commitment to the message m in the candidate NRO token instead of the actual message itself. However, R has to trust that S will in fact send m *after* R has already acknowledged having received it. Note that if O and R happen to use different signature servers, additional inter-server message flows will be necessary.

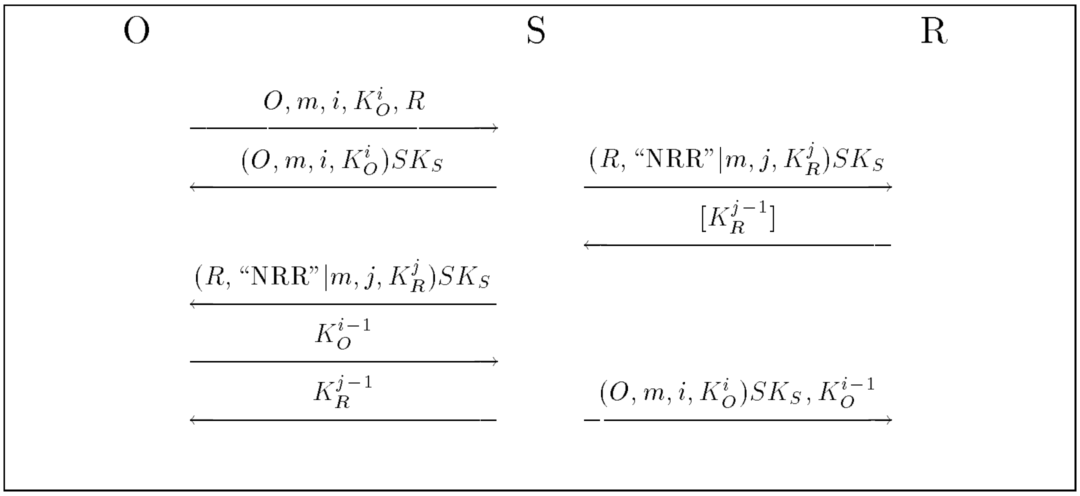

Figure 2: Protocol providing non-repudiation of origin and receipt.

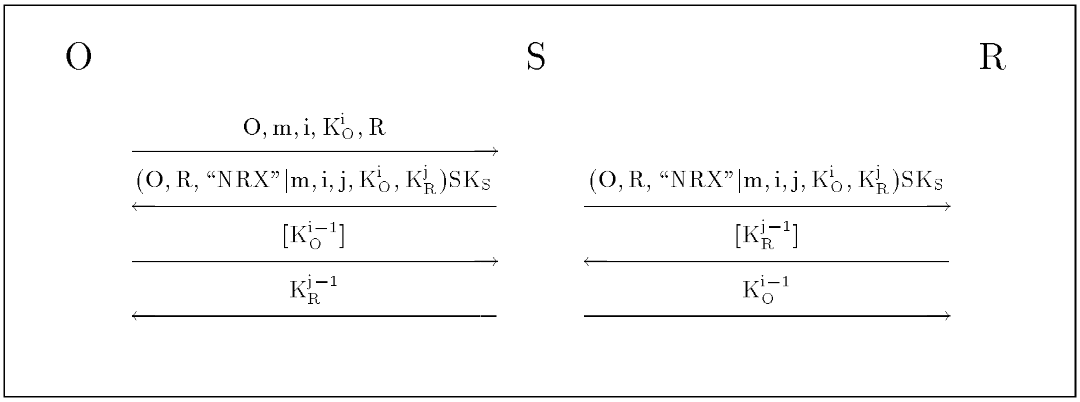

Figure 3: Protocol providing integrated non-repudiation of origin and receipt.

This protocol allows the possibility that either the NRO token or the NRR token may be optional, at the cost of an extra signature by S. The entire protocol has eight message flows. Further, the NRO and NRR tokens are linked only by the hash of the message. In environments where both NRO and NRR are mandatory, a modified protocol as shown in Figure 3 can be used. It results in a combined NRO and NRR token:

$$(O, \text{“NRX”}|m, i, j, K_O^i, K_R^j)SK_S, K_O^{i-1}, K_R^{j-1}$$

The modified protocol has only seven message flows and requires only a single signature by S.

5 Variations on the Theme

5.1 Reducing Storage Requirements for Users

In order to deny forged non-repudiation tokens, O has to store *all* signatures received from S, which might be a bit unrealistic for a device that is not even able to compute signatures. One can easily avoid this storage problem by including an additional field H in S's signature that serves as a commitment on all the previous signatures made by S for that hash chain; i.e., an NRO token looks like

$$NRO^i := ((O, m_i, i, K_O^i, H^i)SK_S, K_O^{i-1})$$

The value H^i is recursively computed by $H^n := Cert_O$ and $H^{i-1} := f(H^i, NRO^i)$. The function $f()$ is a collision-resistant one-way hash function[2]. NRO^i is an NRO token on message m_i using the token public key K_O^i.

O has to store only the last value H^i and the last signature received from S. S has to store all signatures, and has to provide them to O in case of a dispute. If S cannot provide a sequence of signatures that fits the hash value contained in the last signature received by O, the arbiter allows O to repudiate all signatures and assumes that S cheated.

This idea of chaining previous signatures was used by Haber and Stornetta [3] for the construction of a time stamping service, based on the observation that the sequence of messages in H^i cannot be changed afterwards. One can combine their protocols with ours, using S as a time stamping server, as explained in Section 6.1.

5.2 Increasing Robustness

As mentioned above, a signature server must sign exactly one message for a given user per public key (K_O^i) in the hash chain. However, anyone can send a signature request in the form of the first flow, i.e., (O, m, i, K_O^i).

If the signature server does not subsequently receive the corresponding pre-image of the current public key (K_O^{i-1}), the current public key is rendered invalid in any case. This implies that an attacker can succeed in invalidating an entire chain of a user by generating fake signature requests in her name.

An obvious solution would be to require O and S to share a secret key to be used for computing (and verifying) a message authentication code over the first protocol flow.

An alternative solution is to give users the ability to invalidate token public keys without having to create a new chain. The construction is only slightly more complicated than the basic protocol: instead of one chain, each user generates *two* chains (computed with two **different** hash functions): $K_O^n, \ldots, K_O^0$ and $\hat{K}_O^n, \ldots, \hat{K}_O^0$.

Each token public key is now a pair of hash values, say, $(K_O^i, \hat{K}_O^j)$. If O receives the candidate token $(O, m, i, K_O^i, \hat{K}_O^j)SK_S$, she can either *accept* or *reject* it;

- O accepts by revealing K_O^{i-1}. The next token public key is $(K_O^{i-1}, \hat{K}_O^j)$.
- O rejects by revealing $\hat{K}_O^{j-1}$. The next token public key is $(K_O^i, \hat{K}_O^{j-1})$.

On receiving K_O^{i-1} or $\hat{K}_O^{j-1}$, server S creates

- the **non-repudiation token** $(O, m, i, j, \hat{K}_O^j, K_O^{i-1})SK_S$ or
- the **invalidation token** $(O, \text{“INV”}|m, i, j, K_O^i, \hat{K}_O^{j-1})SK_S$

respectively.

The additional signature by S is necessary because for one signature

$$(O, m, i, j, K_O^i, \hat{K}_O^j)SK_S$$

it can easily happen that both K_O^{i-1} and $\hat{K}_O^{j-1}$ become public, i.e., the combination of the first signature with one pre-image would not be unambiguous and recipient R could not depend on what he receives. Instead of making two signatures, S can instead include two commitments $h(a_{NRO})$ and $h(a_{INV})$ of two random numbers a_{NRO} and a_{INV} in the first signatures. Then, S can release one of the two random numbers to O. The random number together with the first signature serves as either the NRO token or the invalidation token. Note that a cheating S could generate both tokens for the same token public key, but O could easily prove that S cheated by showing the token received.

[2] Note that $f()$ may be the same as $h_O()$. However, collision-resistance is a mandatory property for $f()$ (if S succeeds in breaking the collision-resistance of $f()$, it can forge signatures which O may not be able to refute since O no longer retains all past signatures). As we mentioned in Section 3.5, collisions in $h_O()$ are not equally catastrophic from the point-of-view of O.

5.3 Support for Roaming Users

In the basic protocol, the trust placed on the signature server is quite limited — it is trusted only to protect its secret key from intruders and to generate signatures in a secure manner. This limited trust enables a mobile user to make use of a signature server in foreign domains while travelling. Normally the signature server in the user's home domain will be in charge of the user's hash chain. Whenever the user requests to be transferred to a signature server in a different domain, an agreement could be signed by the user and the old signature server authorising the transfer of charge of the user's hash chain. As usual, the pre-image of the current token public key, used to sign this agreement will become the next token public key.

In other words, instead of having a single root public key certificate (which includes the identity of the "home" signature server), a chain of public key certificates could be used. The chain consists of the root public key certificate signed by the home CA and one *hand-off certificate* every time the charge for the user's public key has changed hands:

$$(O, n, K_O^n, S_0)SK_{CA}$$

$$(O, n_l, K_O^{n_l}, S_l)SK_{S_{l-1}},\ for\ 0 < n_l < n, l > 0$$

where, $S_0 = S$ and n_l is the index of the token public key used to sign the request for the l^{th} hand-off (from S_{l-1} to S_l).

To effect a change in charge during a handoff, the following procedure is carried out:

1. The user O sends a hand-off request to both the current signature server S_{l-1} and the intended signature server S_l. As this request must be non-repudiable, this step is essentially a run of the basic protocol to generate a NRO token with a message that means "hand-off from S_{l-1} to S_l requested." S_{l-1} will issue a candidate NRO token for the request using the current token public key $K_O^{n_l}$ and O will validate the token by revealing $K_O^{n_l-1}$).

2. When the NRO token is received and verified by S_{l-1}, it generates a corresponding hand-off certificate described in the previous paragraph and sends it to both O and S_l. It will no longer generate signatures on behalf of O for that hash chain unless charge is explicitly transferred back to it at some point. In addition, it will store both the hand-off certificate and the corresponding NRO token.

3. When S_l has received both the NRO token and the hand-off certificate, it will be ready to generate signatures on behalf of O, starting with $K_O^{n_l-1}$ as the first token public key.

5.4 Key Revocation

As with any certificate-based system, there must be a way for any user O to revoke her hash chain.[3] If the currently secret portion of O's hash chain (say K_O^i, for $i = p-1, p-2, \ldots 1$) has been compromised, O will detect this when she attempts to construct an NRO the next time for the token public key K_O^p: S will return an error indicating the current token public key $K_O^q (q < p)$ from S's point of view. O can attempt to limit the damage by doing one of the following:

1. invalidate all remaining token public keys $K_O^i (i = q, q-1, \ldots 1)$ by requesting NRO tokens for them, or

2. notifying S to invalidate the remaining hash chain by sending it a non-repudiable request to that effect and receiving a non-repudiable statement from S stating that the hash chain has been invalidated. This can be implemented similar to the invalidation tokens described in Section 5.2 — except in this case the token would invalidate the entire chain and not just a single key.

[3] Revocation by authorities is not an issue in this system because the user has to interact with the signature server for the generation of every new NR token anyway.

5.5 General Signature Translation

In a more general light, the signature server in S^3 can be viewed as a "translator" of signatures: *it translates one-time signatures based on hash-functions into traditional digital signatures.* The same approach can be used to combine other techniques such that the result provides some features that are not available from the constituent techniques by themselves.

For example, one could select a traditional digital signature scheme (say D_1) where signing is easier than verification (e.g. DSS) and one (say D_2) where verification is easier than signing (e.g. RSA with a low public exponent) and construct a similar composite signature scheme. The signature key of an entity X in digital signature scheme D is denoted by SK_X^D. To sign a message m, an originator O would compute $(m)SK_O^{D_1}$ and pass it along with m to the signature server S. If the server can verify the signature, it will translate it to $(m, (m)SK_O^{D_1})SK_S^{D_2}$. In other words, the composite scheme allows digital signatures where both signing and verification are computationally inexpensive.

6 Applications

6.1 Building a Secure time stamping Service

In Section 3.5, we argued that S^3 meets the standard requirements of a signature scheme. The structure of the non-repudiation tokens result in an additional property: non-repudiation tokens issued by a given user have a strict temporal ordering among them.

Recall the structure of the non-repudiation tokens described in Section 5.1:

$$NRO^i := ((O, m_i, i, K_O^i, H^i)SK_S, K_O^{i-1})$$

where,

$$H^n := K_O^n, \quad H^{i-1} = f(H^i, NRO^i)$$

If $f()$ is collision-resistant, the chaining factor H^i imposes an order among the messages signed. We call this a *token chain.* Suppose that

$$\{NRO^i\}, i = n \ldots p, p-1 \ldots q$$

indicates the token chain of (NRO tokens issued by) a certain user O at a given time. It is easy to see that all $NRO^q, q < p$, must have been created after NRO^p. As NRO^p is a commitment on m_p, it follows that O knew m_p and showed it to S before NRO^q was created. O and S, either by themselves or in collusion, cannot create NRO^p after NRO^q was created. This enables us to build a time stamping service based on S^3.

Ideally, a time stamping system must be able to impose a total order on all the messages time stamped. We can adapt the approach used in [3], where S generates a chaining factor from a *single*, global chain. Every signature generated by the server has a chaining factor from this global chain. To verify a given time stamp, one needs to know the owners of the previous (and if necessary the subsequent) time stamps generated by the time stamping server.

In Section 4, we outlined a protocol that results in a combined NRO/NRR token. Chaining factors can be included in this token as well. The resulting NR token will be

$$NR_{AB}^{ij} = (A, \text{"NRX"}|m_{ij}), i, j, K_A^i, K_B^j, H_A^i, H_B^j)SK_S, K_A^{i-1}, K_B^{j-1}$$

where

$$H_A^n := K_A^n, \quad H_A^{i-1} = f(H_A^i, NRO_A^i)$$

$$H_B^m := K_B^m, \quad H_B^{j-1} = f(H_B^j, NRO_B^j),$$

and NR_{AB}^{ij} is the same as NRO_A^i and NRO_B^j. Each time A sends a message to B using the modified protocol resulting in a combined NRX token, the token chains of A and B are "synchronised:" any $NRO_A^p, p > i$ must have been created before any $NRO_B^q, q < j$. Although this does not necessarily result in a total order, the more chains are synchronised after a message has been signed, the greater the number of witnesses to the time of signing.

Thus, S^3 can be used to temporally link the tokens generated by S on behalf of multiple users. A practical implementation of a time stamping service can be constructed by requiring that S include a time stamp in each signature generated. The aim of this construction is not to provide an absolute guarantee that the time stamp in a document is precisely correct. Instead, the temporal ordering property of S^3 signatures is used to verify if the time stamp is plausible. In case of a dispute about the time stamp on a signature by A, the token chains of all parties synchronised to A's chain after the signature was made are examined (suppose there are n such parties). If these token chains satisfy all the temporal ordering relationships discussed above *and* if there is a sufficient number of honest parties among those linked to A's token chain, then the time stamp is probably correct. In this scenario, at least all but one of the $n+2$ parties involved must collude in order to produce a fake time stamp which cannot be proved to be a fake.

When digital signatures are used as a means to provide accountability, it is crucial to have unforgeable time stamps embedded in the signatures. The usual technique to achieve this is to use a seperate, external time stamping service is used in conjunction with a traditional digitial signature mechanism. The structure of S^3 makes it a signature scheme with an integrated unforgeable time stamping facility.

6.2 Applications with a Fixed Recipient

As the role of the signature server is verifiable, the recipient can also play that role. This is useful in applications where several non-repudiable messages need to be sent to the same fixed recipient. An example of this is a home-banking" (or electronic funds transfer) application, where customers send signed payment orders to their bank.

Payments messages are of the form

$$m = (payee,\ amount,\ date)$$

When a payer wants to make a payment, he constructs a message of the above form and executes the normal S^3 protocol with the bank, resulting in an NRO token for the message. The bank then transfers *amount* to the account of *payee* and issues a special NRR token, which can be its signature on the entire NRO token. Optionally, it may also get a S^3 NRR token from the payee and forward it to the payee. The non-repudiation tokens serve as evidence of the transaction.

The idea of using a hash chain for repeated, *fixed-value* payments was suggested recently [9, 4]. We have been able to use S^3 for payments of arbitrary values because S^3 provides non-repudiation of origin for arbitrary messages.

7 Analysis

Computation: Ordinary users of S^3 need to be able to compute one-way hashes and to verify traditional digital signatures. Only the signature servers and CAs are required to generate traditional digital signatures. Key generation for ordinary users is also relatively simple: the user needs to be able to generate a random number. In contrast, key generation in traditional digital signature systems is typically more complex, involving, for example, the generation of large prime numbers. In summary, the computational requirements for ordinary users of S^3 are less than those using a traditional digital signature scheme offering comparable security.

Storage: Using the improvement described in Section 5.1, users need to store only the last signature received from S, the pre-image of the current token public key and the sequence number, and the public keys needed to verify certificates.

Signature servers need to store all generated signatures in order to provide them to the users on demand. The stored signatures are necessary only in case of a dispute. Therefore, they can be periodically down-loaded to a secure archive.

Communication: The communication overhead of S^3 is comparable to that of standard symmetric non-repudiation techniques because a third party, S, is involved in each generation of a non-repudiation token.

Using traditional digital signatures, the involvement of third parties can be restricted to exception handling, whereas token generation is usually non-interactive. The price to be paid for this gain in efficiency is that revocation of signature keys becomes more complicated. Note that in S^3, revoking a key is trivial. O simply has to invalidate the current chain.

Security: In the preceding sections, we demonstrated that as long as the registration procedure, the digital signature scheme, and the one-way hash function are secure, both users and signature servers are secure with respect to their respective objectives. Furthermore, the security of originators depends on the strength of the hash function and not on the security of the digital signature scheme.

Additionally, we observe that in practice, traditional digital signature algorithms are not applied directly to arbitrarily long messages. Instead, a collision-resistant, one-way hash function is first applied to the message to produce a fixed-length digest or fingerprint which is then signed using the signature algorithm. The overall security therefore depends on both the traditional digital signature algorithm and the hash function. Signature servers typically have significantly more computational resources available to them than do ordinary users. Hence they can choose a higher grade security (e.g. much longer signature keys) from a given digital signature algorithm. Thus, S^3 gives ordinary users the ability to produce stronger signatures than they could have been able to by using traditional signatures by themselves in the standard way.

8 Related Work

Although non-repudiation of origin and receipt are among the most important security services, only a few basic protocols exist. See [2] for a summary of the standard constructions. We are not aware of any previous work that aims to minimize the computational costs (at the protocol level) for ordinary users while providing the same security as standard non-repudiation techniques based on asymmetric cryptography.

The efficiency problem as addressed by specific designs of signature schemes was mainly motivated by the limited computing power of smart cards and smart tokens. [12] lists most known proposals. Typically they are based on pre-processing or on some asymmetry in the complexity of signature generation and verification (i.e., either sender or recipient must be able to perform complex operations, but not both). Note that although server-supported signatures use a signature scheme that is asymmetric with respect to signature generation and verification, ordinary users are *never* required to generate signatures; thus, both sender and recipient are assumed to be computationally weak.

In his well-known paper [6], Lamport proposed using hash chains for password authentication over insecure networks. There had been other, earlier proposals to use one-way hash functions to construct signatures. Merkle has presented an overview of these efforts [7]. The original proposals in this category were impractical: a proposal by Lamport and Diffie requires a "public key" (i.e. an object that must be bound to the signer beforehand) and two hash operations to sign *every* bit. Using an improvement attributed to Winternitz involving a single public key (which is the n^{th} hash image of the private key) and n hash operations, one can sign a single message of size $log_2 n$ bits. Merkle introduced the notion of a tree structure [7]; in one version of his proposals, with just a single public key, it is possible to sign an arbitrary number of messages. Nevertheless, it took either a large number of hash operations or a large amount of storage in order to sign more than a handful of messages corresponding to the same public key.

Motivated by completely different factors, Pfitzmann et al. [10][11, Section 6.3.3] proposed a fail-stop signature protocol which uses the same ideas as S^3. There, the signature server is also the recipient of the signature (which is a sub-case in the scope of S^3), and the goal is to achieve unconditional security for the signer against the server (in the sense of fail-stop signatures). The protocol has a similar structure as the one in Section 1.[4] Because of the specific security requirements, all parties have to perform complex cryptographic operations, and signatures are not easily transferable.

[4] It uses a so-called bundling function $h()$ instead of the conceptionally simpler hash function used in S^3. A value $h(x)$ is used as O's current public key. To give a NRO token for message m to S, the signer O sends m to S, which answers by $(m, h(x))SK_S$. Finally O sends x to S, which terminates the protocol. The NRO token for S is $(m, h(x))SK_S, x$. The consumed public key $h(x)$ can be renewed by including a new public key $h(x')$ in m.

9 Acknowledgments

We thank Jay Black and the anonymous referees for their valuable comments on previous drafts of this paper, Michael Steiner and Ceki Gülcü for many insights and thought-provoking discussions, Lilli-Marie Pavka for helping us polish the presentation, and Didier Samfat for suggesting the original problem and piquing our interest.

References

[1] N. Asokan, G. Tsudik, and M. Waidner. Server-supported signatures. In Elisa Bertino et al., editors, *Proceedings of the Fourth European Symposium on Research in Computer Security (ESORICS)*, number 1146 in Lecture Notes in Computer Science, pages 131–143. Springer-Verlag, Berlin Germany, September 1996.

[2] Warwick Ford. *Computer Communications Security – Principles, Standard Protocols and Techniques.* Prentice Hall, New Jersey, 1994.

[3] S. Haber and W. S. Stornetta. How to time-stamp a digital document. In *Advances in Cryptology – CRYPTO '90*, pages 437–455. Springer-Verlag, Berlin Germany, 1991.

[4] Ralf Hauser, Michael Steiner, and Michael Waidner. Micro-payments based on ikp. Research Report 2791 (# 89269), IBM Research, Feb 1996.

[5] ISO/IEC JTC1, Information Technology SC 27. Information technology - security techniuqes - non-repudiation. ISO/IEC JTC 1/SC 27, 1996. Contains 3 parts; Current version dated June 1996; Next version expected in early 1997.

[6] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, November 1981.

[7] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO '87*, number 293 in Lecture Notes in Computer Science, pages 369–378, Santa Barbara, CA, USA, August 1987. Springer-Verlag, Berlin Germany.

[8] NIST National Institute of Standards and Technology (Computer Systems Laboratory). Secure hash standard. Federal Information Processing Standards Publication FIPS PUB 180-1, April 1995.

[9] Torben P. Pedersen. Electronic payments of samll amounts. In *Cambridge Workshop on Security Protocols*, volume 1189 of *Lecture Notes in Computer Science*, pages 59–68, April 1996.

[10] Andreas Pfitzmann, Birgit Pfitzmann, and Michael Waidner. Practical signatures where individuals are unconditionally secure. Unpublished manuscript, available from the authors (pfitzb@informatik.uni-hildesheim.de), 1991.

[11] Birgit Pfitzmann. *Digital Signature Schemes — General Framework and Fail-Stop Signatures.* Number 1100 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1996.

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

[13] Douglas R. Stinson. *Cryptography: theory and practice.* CRC Press Series on Discrete Mathematics and Its Applications, edited by Kenneth Rosen. CRC Press, Boca Raton, Florida, 1995.