Server-Supported Signatures

N. Asokan[1,2], G. Tsudik[1,3], M. Waidner[1]

[1] IBM Zürich Research Laboratory, CH-8803 Rüschlikon, Switzerland
email: {aso,wmi}@zurich.ibm.com
[2] Department of Computer Science, University of Waterloo, Waterloo, Canada.
[3] Information Sciences Institute, University of Southern California, USA.
email: gts@isi.edu

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 existing techniques based on asymmetric cryptography.

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

1 Introduction

Computers and communication networks have become an integral part in the daily lives of many people. Systems to facilitate commercial and other transactions have been built on top of large open computer networks. Oftentimes these transactions must have some legal significance if they are to be useful in real life. Non-repudiation is one of the essential services that must be provided in order to attach 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 arbitrator (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 third parties. We call a third party whose cheating can be proven to an arbitrator 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 [Sch96].

This paper is organized as follows. We begin in the next section by briefly reviewing the standardization activities in the area of non-repudiation. Section 2 provides some more motivation for alternative non-repudiation techniques. The actual design of S^3 is described in Sections 3 and 4. Some variations are addressed in Section 5 and Section 6 discusses performance and storage costs. The paper concludes in Section 7 with a brief review of related work.

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 [ISO95a, ISO95b, ISO95c] 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)**[4] 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 *arbitrator.*

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

- *Asymmetric non-repudiation techniques* are based on digital signatures schemes, i.e., on public-key cryptography. The main (and likely 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,* e.g., if the signer and the authority have to sign a paper contract listing the signer's and certification authority's public keys, responsibilities, and liabilities, maybe in front of a notary public. In the worst case the certification authority could cheat the user by

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

issuing a certificate with a public key 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 lightweight 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 in tamper resistant hardware. Both approaches increase bith cost and complexity while none of them solves the problem completely.

In the following we present a new, light-weight technique for non-repudiation services, *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 the 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. x is called the *pre-image* of $h(x)$.

A large number of practically secure and efficient one-way hash functions have been invented, e.g., SHA or MD5 [Sch96].

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,

$$h^i(x) = h(h(h(\ldots(i\ times)\ldots h(x)\ldots)))$$

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 that are 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 hash-function $h()$ and a digital signature scheme. Entities should "personalize" the hash function by always including their unique name as an argument. **Therefore, every occurrence of $h(x)$ below really means $h(O, x)$, where O is the entity computing the one-way hash.**

The result of digitally signing a message x with signature key SK is denoted by $(x)SK$. The users' security depends on the one-way property of $h()$, i.e., the one-way property must hold even against the servers.[5]

In order to minimize the computational overhead for users, $h()$ must be efficiently computable, and digital signatures must be efficiently *verifiable*. Only signature servers and certification authorities must be able to *generate* signatures. MD5 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 $h()$. Based on K_O, user O computes the hash-chain $K_O^0, K_O^1, \ldots K_O^n$, where $K_O^0 = K_O$, $K_O^i = h^i(K_O) = h(K_O^{i-1})$. $PK_O = K_O^n$ constitutes O's *root public key*. Root public key K_O^n will enable O to authenticate n messages[6].

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. 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 in a way such that CA becomes a verifiable third party.

[5] Hash functions such as SHA or MD5 are one-way for *all* parties, i.e., practically this is no problem. But note that usually so-called cryptographically strong hash-functions are invertible for the party that generated the hash-function.

[6] This is no real 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.

Notation Summary
$h()$ - one-way hash function
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

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[7] $(O, n, PK_O, S)SK_{CA}$. The registration performed by O and CA must be verifiable, as discussed earlier. 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 on PK_S from a certification authority. Since these are ordinary certificates for digital signatures we do not need to 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.[8]

[7] 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, expiration dates, etc.

[8] In case O does not want to reveal the message to S for privacy reasons, m can be replaced by a hash of m.

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 $h^{n-i}(K_O^i) = PK_O$. 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 back to O in the second flow.
3. O verifies the received signature and records K_O^i as consumed, i.e., replaces i by $i-1$. The NRO token for R consists now of

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

   O produces this token, i.e., actually authenticates m, by revealing K_O^{i-1}. In Figure 1 we assumed that the NRO token is sent to R via S is the third flow. Alternatively, O can send the token to R directly.

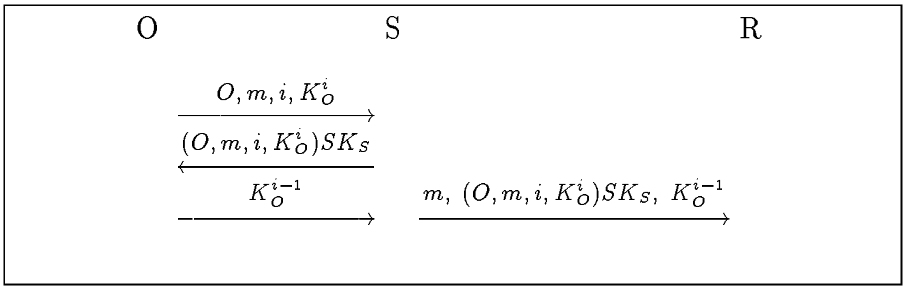

Fig. 1. Protocol providing non-repudiation of origin

K_O^i is referred as the **token public key** of the $(n-i+1)$st 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, i.e., 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.

3.5 Dispute Resolution

In case of a dispute, R can submit the NRO to an arbitrator. The arbitrator 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 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 root public key was registered by O (i.e., showing the signed contract with O).
- proving that S cheated by showing a different token corresponding to the same token public key.

Note that in case CA is honest, to falsely claim 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, it is presumed 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.

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 m to R, S can ask R for an NRO token for ("NRR", $h(m)$) which is then passed on to O. This is illustrated in Figure 2. The NRR token consists of:

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

Since 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.

Notice that the present protocol actually implements *fair-exchange* of m (NRO token) and its receipt (NRR token), based on S as trusted third party. If S behaves dishonestly, no fairness can be guaranteed, i.e., O might not receive the NRR token or R might not receive m or the NRO token.

Depending on the application R might request to receive m already in the second flow. This allows R to refuse generating the NRR token after he has actually received m, but avoids the problem that R has to trust that he will receive m *after* he has already acknowledged having received it.

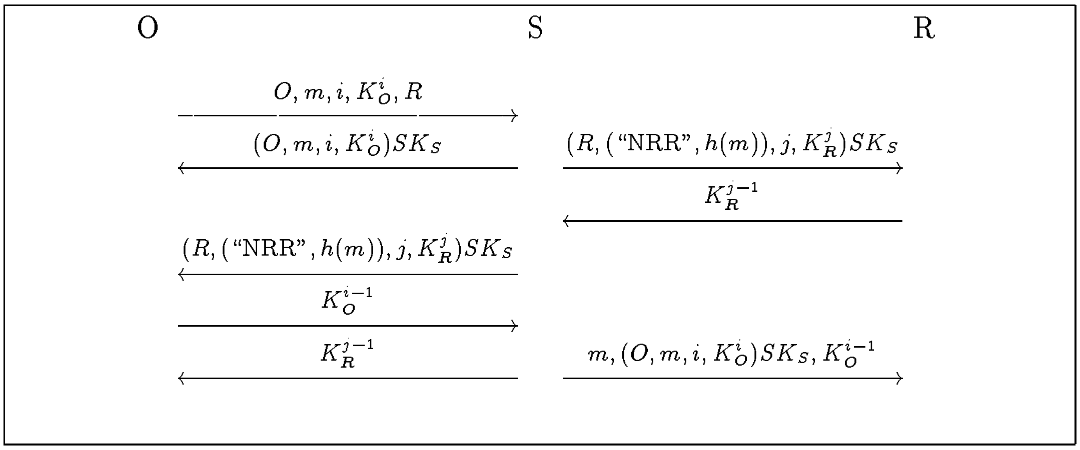

Fig. 2. Protocol providing non-repudiation of origin and receipt

5 Variations on the Theme

5.1 Reducing Storage Requirements

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 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

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

Value H^i is recursively computed by $H^n := K_O^n$ and $H^{i-1} := f(H^i, NRO^i)$. $f()$ is a one-way hash function (it may be the same as $h()$).

O has to store the last value H^i and the last signature received from S only. 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 to the hash value contained in the last signature received by O, the arbitrator allows O to repudiate all signatures and assumes that S cheated.

This idea of chaining previous signatures was used by *Haber and Stornetta* [HS] 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 could combine their protocols with ours, using S as time-stamping sever.

5.2 Increasing Robustness

As mentioned above, a signature server will 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 regardless. 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) 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 $(O, m, i, K_O^i, \hat{K}_O^j)SK_S$, she can either *accept* or *reject* that:

- 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** $(O, m, i, K_O^i, \hat{K}_O^j, K_O^{i-1})SK_S$ or
- the **invalidation token** $(O, m, i, K_O^i, \hat{K}_O^j, \hat{K}_O^{j-1})SK_S$

respectively. The new token public key is $(K_O^{i-1}, \hat{K}_O^j)$ or $(K_O^i, \hat{K}_O^{j-1})$, respectively.

The additional signature by S is necessary since for one signature

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

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. 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.

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 roaming (or 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 two signature servers involving the transfer of the user's current 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:

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

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

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

1. The user O sends a request for change of charge to both the current signature server S_{l-1} and the intended signature server S_l. Since this request must be non-repudiatable, this step is essentially a run of the basic protocol to generate a NRO token with a message that means "change of charge from S_{l-1} to S_l requested" for token public key $K_O^{n_l-1}$.
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.

5.4 Key Revocation

As with any certificate-based system, there must be a way for any user O to revoke her hash-chain[9]. If the currently secret portion O's hash-chain (say K_O^i, for $i = p-1, p-2, \ldots 1$) has been compromised, O will detect the fact 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-repudiatable request to that affect and receiving a non-repudiatable 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.

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

5.5 Similar Constructions

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 in such a way 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 the message 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 Analysis

Computation: Users need to be able to compute one-way hashes, and to verify digital signatures. Only the signature servers and CAs are required to generate signatures.

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 in case they request them. 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, since 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, while the token generation is a two-party protocol. 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 just has to invalidate the current chain.

Security: In the preceding sections, we demonstrated that as long as the registration procedure, digital signature scheme and 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 one-way hash function and not on the security of the digital signature scheme.**

7 Related Work

Although non-repudiation of origin and receipt are among the most important security requirements, only a few basic protocols exist. See [For94] for a summary of the standard constructions. We are not aware of any previous work that aims to minimize the computational costs (on 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. [Sch96] 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.

There had been other proposals to use one-way hash functions to construct signatures. Merkle's paper [Mer87] includes an overview of these efforts. The original proposals in this category were impractical: A proposal by Lamport/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, with a single public key and (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 using a tree structure [Mer87]; in one version of his proposals, with just a single public key, it is possible to sign an arbitrary number of messages. But, it still 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.* [PPW91, Pfi] 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, 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.[10] Because of the specific security requirements, all parties have to perform complex cryptographic operations, and signatures are not easily transferable.

[10] 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.

8 Acknowledgments

We are indebted to Michael Steiner for his perceptive feedback on previous versions of this paper. We would also like to thank Didier Samfat for making us interested in the original problem.

References

[For94] Warwick Ford. *Computer Communications Security – Principles, Standard Protocols and Techniques*. Prentice Hall, New Jersey, 1994.

[HS] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. Journal of Cryptology 3/2 (1991) 99-111.

[ISO95a] ISO/IEC JTC1, Information Technology, SC 27. 2nd ISO/IEC CD 13888-1 Information Technology - Security Techniques - Non-repudiation - Part1: General Model. ISO/IEC JTC 1/SC 27 N 1105, May 1995.

[ISO95b] ISO/IEC JTC1, Information Technology, SC 27. 2nd ISO/IEC CD 13888-2 Information Technology - Security Techniques - Non-repudiation - Part2: Using symmetric encipherment algorithms. ISO/IEC JTC 1/SC 27 N 1106, July 1995.

[ISO95c] ISO/IEC JTC1, Information Technology, SC 27. ISO/IEC CD 13888-3 Information Technology - Security Techniques - Non-repudiation - Part3: Using asymmetric techniques. ISO/IEC JTC 1/SC 27 N 1107, September 1995.

[Mer87] 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.

[Pfi] Birgit Pfitzmann. Fail-stop signatures; principles and applications. Proc. Compsec '91, 8th world conference on computer security, audit and control, Elsevier, Oxford 1991, 125-134.

[PPW91] 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.

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons Inc., New York, second edition, 1996.