More Efficient Server Assisted One Time Signatures

Vipul Goyal
Department of Computer Science & Engineering
Institute of Technology
Banaras Hindu University
Varanasi, India
vipulg@cpan.org

Abstract

Server assisted one time signature scheme was presented in the last edition of CT-RSA as a non-repudiation service for mobile and constrained devices. However, the scheme suffered with high storage requirements for the virtual server and high memory requirements for the mobile client. We significantly improve the scheme by dramatically reducing virtual server storage requirements as well as mobile client memory requirements. More precisely, the virtual server storage requirements in our scheme are more than 80 times less than that in the original scheme. Further, memory requirements for the mobile client are reduced by a factor of more than 130.

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 [7].

Non-repudiation is usually provided through a digital signature. However, digital signature generation and even verification are known to be highly computationally intensive processes. It is not feasible to implement public key cryptography and hence digital signatures on a constrained mobile device having limited computational resources and memory. Hence the question arises i.e. how to provide a means of non-repudiation to such devices on which public key cryptography is not possible.

The answer is to employ a third party. If unconditionally trusted, very efficient non-repudiation service may be provided by the third party. In a straightforward setting, the mobile user, also called the originator, and the trusted third party may share a secret key. The originator would supply the message to be signed encrypted with that key and the third party would sign that message on behalf of the originator. Clearly, it is easy for the third party to cheat in this setting since it could sign any message on behalf of the user.

Such schemes are of limited usage since they require the originator to have unconditional trust in the third party. More practical, though less efficient, techniques are possible in which if the third party cheats, the originator is able to prove this cheating to an arbiter. The third party in this setting is called a verifiable server (VS). The first practical scheme under this category was devised by Asokan et al [7]. However, it suffered from several serious limitations as discussed in section 3. This scheme was recently improved by Bicakci and Baykal [1] to propose server assisted one time signatures (SAOTS). SAOTS, although very practical, suffered from storage problems. The storage requirements for the VS in this scheme were quite high and ever increasing. More precisely, for every signature generated, the VS was required to add approximately 5 KB to its storage. Considering minimal parameters, assume that a user signs just a single message per day and the VS supports just a thousand users, the storage requirements for the VS starts becoming prohibitive in just 1 year. Clearly, it is not feasible to use this scheme on a large scale in a commercial setting. Additionally, the mobile device was required to store about 2.7 KB of secret data in its memory in order to be able to use this scheme. For a device which cannot generate or verify signatures, storing this data in its memory could be also prohibitive.

OUR CONTRIBUTION. We improve the SAOTS construction trying to decrease the storage requirements for the VS as well as the memory requirements for the mobile client. By using a new technique to generate one time signature keys and introducing a new server storage scheme, we are able to dramatically reduce the VS storage requirements as well as the mobile client memory requirements. Only 60 bytes per signature generation are added to the VS storage instead of 5 KB in the previous scheme. Further, the memory requirements for the mobile client are reduced from 2.7 KB to just 20 bytes. The computational requirements in our scheme remain pretty much the same as in the previous scheme [1].

Rest of the paper is organized as follows- Section 2 provides some background material on hash chains and one time signatures. Section 3 discusses the related work. Section 4 discusses the proposed construction of the SAOTS scheme and various issues involved in it. Section 5 concludes the paper.

2 Background on Hash Chains and One Time Signatures

2.1 Hash Chains

Hash chains are based a public function h that is easy to compute but computationally infeasible to invert, for suitable definitions of “easy” and “infeasible”. Such functions are called one-way functions (OWF) and were first employed for use in login procedures by Needham [16]. If the output of a one-way function is of fixed length, it is called a one-way hash function (OWHF). More precisely, the definition of OWHF is given as [15]-

Definition A function h that maps bit strings, either of an arbitrary length or a predetermined length, to strings of a fixed length is a OWHF if it satisfies three additional properties-
→ Given x, it is easy to compute h(x)
→ Given h(x), it is hard to compute x
→ It is hard to find two values x and y such that $h(x) = h(y)$, but $x \neq y$.

The idea of “hash chain” was first proposed by Lamport [1] in 1981 and suggested to be used for safeguarding against password eavesdropping. However being an elegant and versatile low-cost technique, the hash chain construction finds a lot of other applications.

A hash chain of length *N* is constructed by applying a one-way hash function *h*() recursively to an initial seed value s.

$$h^N(s) = h(h(h(...h(s)...))) \quad \text{(N times)}$$

The last element $h^N(s)$ also called the tip T of the hash chain resembles the public key in public key cryptography i.e., by knowing $h^N(s)$, $h^{N-1}(s)$ can not be generated by those who do not know the value *s*, however given $h^{N-1}(s)$, its correctness can be verified using $h^{N}(s)$. This property of hash chains has directly evolved from the property of one-way hash functions.

In most of the hash-chain applications, first $h^{N}(s)$ is securely distributed and then the elements of the hash chain are spent (or used) one by one by starting from $h^{N-1}(s)$ and continuing until the value of *s* is reached. At this point the hash chain is said to be exhausted and the whole process should be repeated again with a different s to reinitialize the systems.

2.2 One Time Signatures

The Concept of One time signatures (OTS) has been known for over two decades. It was initially proposed by Lamport and was the first digital signatures scheme ever designed. Interestingly, OTS schemes employ nothing more than OWHFs. The concept of OTS was subsequently enhanced by Merkle [17-18], Winternitz [17] and Bicakci et al [22]. Bleichenbacher et al [19-21] formalized the concept of OTS using directed acyclic graphs (DAGs).

SIGNING A ONE BIT MESSAGE. The signer chooses as the secret key two values x1 and x2 (representing ‘0’ and ‘1’) and publishes their images under a one-way function y1 = h(x1) and y2 = h(x2) as the public key. These x’s and y’s are called the *secret key components* and the *public key components*, respectively. To sign a single bit message, reveal the pre-image corresponding to the actual ‘0’ or ‘1’ i.e. reveal x1 or x2 based upon whether the message to be signed is 0 or 1.

For signing longer messages, several instances of this basic scheme may be used. Thus we note that to sign an n bit message, 2n x’s and thus 2n y’s are required and the size of signatures generated is equal to n x’s i.e. n times the size of random number.

There are several improvements to this basic scheme. Merkle [17, 18] proposed an improvement which reduces the number of public as well as secret key components in the Lamport method by almost two-fold. Instead of generating two x’s and two y’s for each bit of the message, the signer generates only one x and one y for each bit of the message to be signed. When one of the bits in the message to be signed is a ‘1’, the signer releases the corresponding value of x; but when the bit to be signed is a ‘0’, the signer releases nothing. Because this allows the receiver to pretend that he did not receive some of the x’s, and therefore to pretend that some of the ‘1’ bits in the signed message were ‘0’, the signer must also sign count of the ‘0’ bits in the message. Now, when the receiver pretends that a ‘1’ bit was actually a ‘0’ bit, he must also increase the value of the count field, which can’t be done. Because the count field has only $\log_2(n)$ bits in it, the number of public and secret key components is decreased by almost a factor of two i.e. from 2n to $n + \log_2(n)$ (or to $n + [\log_2(n)] + 1$ if n is not a power of 2). This also results in the decrease of signature size by almost a factor of two.

As an example, if we wished to sign the 8-bit message ‘0100 1110’ we would first count the number of ‘0’ bits (there are 4) and then append a 3-bit count field (with the value 4) to the original 8-bit message producing the 11-bit message ‘0100 1110 100’which we would sign by releasing x[2], x[5], x[6], x[7] and x[9]. The receiver cannot pretend that he did not receive x[2], because the resulting erroneous message ‘0000 1110 100’ would have 5 ‘0’s in it, not 4. Similarly, pretending he did not receive x[9] would produce the erroneous message ‘0100 1110 000’ in which the count field indicates that there should be no ‘0’s at all. There is no combination of x’s that the receiver could pretend not to have received that would let him concoct a legitimate message.

Winternitz [17] proposed an improvement which reduces the signature size by several folds at the expense of increased computational effort. In Winternitz's method, the OWF is applied to two secret key components iteratively for a fixed number of times, resulting in a two-component public key.

3 Related Works

Server assisted signatures can be explained in three subgroups depending on the trust relationship between the user and the server. More specifically, the server employed may be either (1) fully trusted, (2) un-trusted, or (3) verifiable.

In the first category, after receiving an authentic message from a user (A MAC algorithm which can be implemented very efficiently may be used for authentication), a more powerful proxy server on behalf of the user generates a public key digital signature for the message [2]. Notice that the user himself does not need to perform any public key operation, he just computes a MAC using secret key cryptography. The drawback here is that this simple design is only applicable when the user fully trusts the proxy server i.e. the server can generate forged signatures and the cheating cannot be proven by the user. On the other extreme, a totally un-trusted server might be utilized i.e. the server executes only non-security sensitive computations for the user. Unfortunately, there are not many secure and practical schemes under this category. Most of the schemes proposed so far in this category have been found flawed. For instance the protocol proposed by Bequin and Quisquater [4] was later broken by Nguyen and Stern [5]. To our knowledge, for RSA signatures, designing a secure server-assisted protocol that utilizes an untrusted server is still an open problem. But the situation for DSA is not the same. A secure and unbroken example for DSA is the interesting approach of Jakobson and Wetzel [6]. However in their approach, to generate the signature, public key operations although in reduced amount are still needed to be performed on the constrained device.

The last server-assisted signature alternative is to employ a verifiable server (VS). A VS is the one whose cheating can be proven to an arbiter. This approach can be considered somewhere in between the other two since the server in this case can cheat but subsequently, the user would have the ability to prove this situation to other parties (e.g. an arbiter).

The first work that aims to reduce the computational costs to generate digital signatures for low-end devices by employing a powerful VS is SAS protocol [7]. In [8], the authors extend this work by providing implementation results and some other details of the scheme. Now we provide a brief summary of SAS protocol. For a more comprehensive treatment, the reader is referred to the original papers [7, 8].

There is an initialization phase in which each user (originator) gets a certificate from an offline certification authority specifying $h^N(s)$, the tip of the hash chain, where s is kept secret by the originator O. In addition, O should register to a VS (which has the traditional public-key based signing capability) before operation. Then the SAS protocol works in three rounds-

1. The originator (O) sends m and $h^{N-i}(s)$ to VS where
   – m is the message
   – $h^{N-i}(s)$ is the ith element of the hash chain. The counter i is initially set to 1 and incremented after each run.

2. Having received O's request, VS checks the followings:

– Whether O's certificate is revoked or not.
– Whether h^i(supplied ith link) = $h^N(s)$ or in a more efficient way h(supplied ith link) = $h^{N-i+1}(s)$ since $h^{N-i+1}(s)$ has already been received as the (i-1)th link
If these checks are OK, VS signs m concatenated with $h^{N-i}(s)$ and sends it back to *O*.

3. After receiving the signed message from VS, O verifies the VS's signature, attaches $h^{N-i-1}(s)$ to this message and sends it to the receiver R.
Upon receipt of the signed message, the receiver verifies VS's signature and checks whether $h(h^{N-i-1}(s)) = h^{N-i}(s)$

Note that VS may try to sign m' instead of m once O releases the ith link. But then, since O verifies the signature, it would not release the (i-1)th link thus rendering the signature on m' incomplete. The best VS can do is to sign two messages m and m' with the same link embedded in both signatures. For this, O should store every message signed by the VS corresponding to each link in the hash chain. This leaves O with a cryptographic proof of server fraud if she encounters any other message singed by the server since O can then produce two message signed by VS with the same hash chain link embedded in them.

[1] observes the following limitations of SAS protocol-

1. Verifying VS's signature: In step 3 of the SAS protocol, before sending the signed message to R, O should verify the VS's signature to be safe against the possibility of changing the message either by an adversary in transit or by the VS itself. Remember that for some constrained devices, public key cryptography is simply untenable no matter it is used for signing or verifying. Even when public key cryptography is acceptable for the user's device, the efficiency provided by this protocol is based on an assumption which is not always valid. More precisely, if the VS uses RSA [14] signature scheme, where verification is much more efficient as compared to signature generation, SAS brings efficiency. On the other hand, if instead of RSA, other digital signature schemes like DSS [12] where verification is at least as costly as signing are used to sign the message, SAS protocol apparently becomes less-efficient than even the traditional signing methods.

2. Incompatible Verification: As observed in [8], unlike the proxy signatures, SAS signatures are not compatible with other primary signature types i.e. the signature generated by the SAS protocol is not a standard signature. Therefore, the receiver must utilize the custom-built verification method of SAS protocol.

3. Storing VS's signatures: In SAS protocol, the user must store all the signatures generated by VS on its behalf in order to be able to prove VS's cheating to an arbiter [1]. For many devices having a limited storage capacity, this might also be an unrealistic assumption.

4. Network Overhead: One of the factors that affect the overall performance of the SAS protocol is the round-trip delay between O and VS which is related to the number of steps in the protocol execution. SAS has an extra step in which the VS returns back the generated signature to the user in order to enable him to verify this signature. Other server assisted signature schemes like SAOTS (explained next) have only 2 steps as compared to 3 in SAS.

SAS was recently improved by Bicakci and Baykal [1] to propose server assisted one time signatures (SAOTS) which is the first VS based approach where the user does not need to perform any public key operation at all. SAOTS is completely transparent to verifiers since the signatures are indistinguishable from standard signatures. Further, in SAOTS unlike other alternatives, the VS not the user is required to save the signatures for dispute resolution. Operating in two rounds as opposed to three, SAOTS eliminates all the four aforementioned drawbacks of SAS protocol. The basic idea is that the user signs the message with a one time signature key pair and sends it to VS which in turn stores the user's one time signature and signs the message with the traditional public key.

For system setup, every user registers to the VS and generates a one-time key pair by randomly generating secret key components. In a secure fashion, the user distributes the public key to the server.

For getting a message signed by the VS, the user precomputes a second one-time key pair. When the message to be signed is ready, he concatenates the message with the new one-time public key and signs this by his previous one-time private key. He then sends the message and the new one-time public key as well as the one-time signature to VS. VS verifies the received one-time signature using the one time public key received in the previous step. He stores the new one-time public key the user has signed for the verification of next message. VS also stores the received signature. He is now ready to sign the message with the user's private key. Finally, the signed message is transmitted to the intended receiver(s).

The user can sign any further messages easily by repeating the above steps. Dispute resolution is straightforward since VS stores all the public keys and signatures received form the user.

The main problems in this scheme are the high storage requirements for the VS and memory requirements for the user. Recall that for SHS and merkle's construction of one time signatures, the number of secret (or public) key components = 160 + log(160) = 168. Hence for every signature, since the VS is required to store the message to be signed = 20 bytes, the public key = 168*20 = 3360 bytes and the signature = (168/2)*20 = 1680 bytes, the total storage addition for every signature is 20 + 3360 + 1680 = 5060 bytes or approximate 5 KB. As explained in section 1, even for the minimal parameters, the storage requirements for the VS start becoming prohibitive soon in such a setting. Further, the user is required to remember the last secret key components in order to be able to sign the next message. For 128 bit random numbers, the user memory requirements comes out to be about 168*16 = 2.7 KB approx. This can also be prohibitive for a device which cannot even generate or verify signatures.

4 The Proposed Construction

Before going further, we introduce some basic notations used in this section-

U	The mobile user or the originator
VS	Verifiable Server
R	Receiver of the signature
L	Length of the output of OWHF employed e.g. 160 bit for SHS
m	Number of public/secret key components used in the OTS scheme. Equal to $L+\log_2(L)$ for Merkle's construction
p	Average number of components in a one time signature. Usually equal to m/2.
P_U^i	i^{th} one time public key of user U. Equal to the collection (or Concatenation) of m public key components
S_U^i	i^{th} one time secret key of user U. Equal to the collection (or Concatenation) of m secret key components

$S_U^i(M)$	The message M signed with the one time secret key S_U^i. Equal to the collection (or Concatenation) of the relevant secret key components required to sign M
P_U	Traditional public key of the user U
$S_U(M)$	Message M signed with the traditional secret key of the user U

4.1 Setup

In order to initialize the system, the mobile user U initiates a counter i = 1 and generates the following-

1) A secret key K
2) A one time key pair as follows-
   $S_U^i = \{h(K, i, 1), h(K, i, 2), \dots , h(K, i, m)\}$
   $P_U^i = \{h^2(K, i, 1), h^2(K, i, 2), \dots , h^2(K, i, m)\}$

Now, the user securely stores K and i and transfers P_U^1 to the VS in a non-repudiable manner. This can be accomplished by a public key signature if U has a capability of traditional signing or paper means or by getting a certificate from a CA specifying the one-time public key P_U^1.

$$\mathbf{U \rightarrow VS: \quad P_U^1}$$

In addition, to produce traditional public key signatures on behalf of the user, the VS generates a public/secret key pair i.e. P_U/S_U on behalf of the user and obtains a certificate from the CA specifying the public key P_U. Note that the secret key S_U is not revealed even to U itself.

4.2 Operation

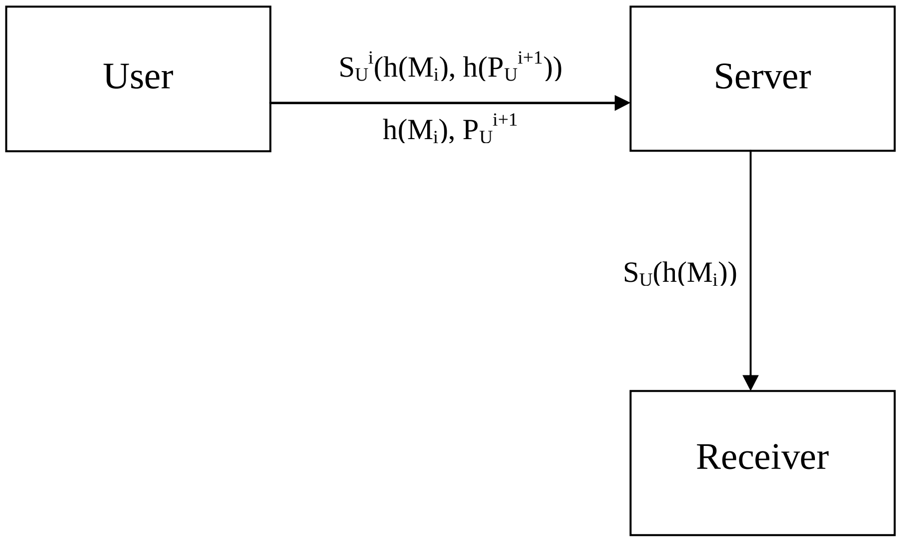

Figure 1: Operation of the Proposed Scheme

For generating the ith signature, the protocol works as follows-

1) The user precomputes the next i.e. (i+1)th one time public/secret key pair (see section 4.1). When the message to be signed is ready, he concatenates the hash of the message

with the hash of the computed one-time public key and signs the resulting quantity with his current i.e. ith one time secret key. He then sends this signature along with the hash of the message and the computed public key to VS.

$$\mathbf{U \rightarrow VS:\quad S_U^i(h(M_i), h(P_U^{i+1})), h(M_i), P_U^{i+1}}$$

2) VS checks the validity of the received signature using the stored P_U^i. It then stores the following-
   a) $h(M_i)$
   b) $h(S_U^i(h(M_i), h(P_U^{i+1})))$
   c) $h(P_U^i)$

   VS then replaces the stored P_U^i with the received P_U^{i+1}. Now, VS produces a traditional public key digital signature to sign the message $h(M_i)$ using user's secret key S_U. This signature is then directly sent to the receiver R if specified in the user's request. Alternatively, the signature may also be returned to the user.

$$\mathbf{VS \rightarrow R:\quad S_U(h(M_i))}$$

This completes our description of the proposed scheme. Observer that as with SAOTS, the scheme is fully transparent to the receiver since the signature is a standard signature. The certificate of the user specifying P_U may be supplied along with the signature if required[1]. The receiver does not need to have a custom built software to check the validity of the signature. This is in contrast to SAS. Further, the user is not required to perform any public key operations whereas in SAS, digital signature verification was required by the user.

4.3 Analysis

First observe that for every signature generated, the VS has to add 3 hash values to its storage i.e. for the ith signature, the VS has to store the following- 1) $h(M_i)$, 2) $h(S_U^i(h(M_i), h(P_U^{i+1})))$, and 3) $h(P_U^i)$. Thus, a signature costs only 60 bytes to the VS. Clearly, this is a significant improvement over [1] in which the VS had to add about 5 KB to its storage as explained in section 3. After generating i signatures for the user U, the VS stores the following-

$$\mathbf{\{h(M_1), h(S_U^1(h(M_1), h(P_U^2))), h(P_U^1)\}, \{h(M_2), h(S_U^2(h(M_2), h(P_U^3))), h(P_U^2)\}, \ldots, \{h(M_i), h(S_U^i(h(M_i), h(P_U^{i+1}))), h(P_U^i)\}, P_U^{i+1}}$$

Now, consider memory requirements for the mobile user. In our scheme, the user is only required to store the 128 bit secret key K along with the counter i. Assuming a 32 bit counter, the memory requirements for the user comes out to be 20 bytes. This is about 135 times less than that in the original scheme [1] for which the requirements are about 2.7 KB as explained previously.

4.4 Dispute Resolution

A dispute arises in case a user U claims that the VS has signed a message M on his behalf which he did not request the VS to sign. The prerequisite for this claim is that U should produce the signature on M generated by the VS on his behalf. Now, the arbiter asks VS to prove that the message M was indeed requested by U to be signed. Additionally, since in our scheme the VS

[1] Here one possible option is to have the VS sign the message with its own public key after appending a statement to the message indicating the user on whose behalf it has signed the message. The advantage of this approach is that the VS is not needed to obtain a new certificate for each registered user.

does not store full one time signatures and public keys, the arbiter also asks U to cooperate in the process (although U may not do so honestly).

At this point, assume that VS has generated n signatures on behalf of the user U. Thus, VS stores the following-

$\{h(M_1), h(S_U^1(h(M_1), h(P_U^2))), h(P_U^1)\}$, $\{h(M_2), h(S_U^2(h(M_2), h(P_U^3))), h(P_U^2)\}$, … , $\{h(M_n), h(S_U^n(h(M_n), h(P_U^{n+1}))), h(P_U^n)\}$, P_U^{n+1}

The dispute resolution protocol proceeds in steps. The i^th step of the process takes place as follows for $1 \leq i \leq n$

(1) The VS submits the stored $h(S_U^i(h(M_i), h(P_U^{i+1})))$ to the arbiter.

(2) VS demands P_U^i and $S_U^i(h(M_i), h(P_U^{i+1}))$ from U after supplying him $h(M_i)$ and $h(P_U^{i+1})$. U generates S_U^i and P_U^i using the secret key K (see footnote [2]) and signs the concatenation of supplied $h(M_i)$ and $h(P_U^{i+1})$ using S_U^i. U cannot supply a wrong P_U^i since he signed $h(P_U^i)$ in the previous i.e. (i-1)th step[3]. He cannot supply a wrong signature $S_U^i(h(M_i), h(P_U^{i+1}))$ since the signature can be verified using the supplied key P_U^i.

(3) The arbiter computes the hash of the signature $h(S_U^i(h(M_i), h(P_U^{i+1})))$ supplied by U and matches it with hash of the signature submitted by VS in (1). If they do not match, the VS is concluded to be the cheater. Otherwise, the arbiter concludes the following-
    a) VS was requested by U to sign $h(M_i)$
    b) Hash of the next one time public key to be supplied by U should be $h(P_U^{i+1})$.

If the hash of the disputed message i.e. h(M) equals $h(M_i)$, the dispute is resolved in the favour of VS. Else if i equal n, i.e. all message stored by VS have been checked and none equals M, the dispute is resolved in the favour of the user U. Otherwise, i is incremented and the process continues with (1) again.

Now, we summarize the computational and storage requirements for SAOTS and our scheme for all the 3 parties involved-

Notations:-
H: hash computation
S: generation of traditional public key signature
V: verification of traditional public key signature
E: OTS encoding computation (costs less than one hash)
p: number of hash computations to verify OTS
m: number of public key components in the OTS scheme
K: size of secret key of the user
C: size of the signature counter

[2] Note that it is immaterial how U generates one time key pairs S_U^i's and P_U^i's. VS cannot and do not need to ensure that U is following the correct formula for generating one time key pairs during any phase in the scheme. The sole purpose of specifying a formula for S_U^i and P_U^i is to put the responsibility of supplying signatures and public keys on U instead of the VS.
[3] for i = 1, U cannot supply a wrong key since P_U^1 was the initial key registered using non-repudiable means.

	Party Name	SAOTS original	Proposed Construction
Computational Requirements	User	1H + 1E	1H + 1E
	Server	1E + (p+2)H + 1S	1E + (p+3)H + 1S
	Receiver	1H + 1V	1H + 1V
Storage Requirements	User	mH	1K + 1C
	Server	(m+p+1)H	3H
	Receiver	_	_

Table 1: Objective comparison of the proposed scheme with the original SAOTS scheme

5 Conclusion

Server assisted signature schemes try to provide a means of non-repudiation for constrained and mobile devices having limited computational and memory resources. SAOTS seems to be the most practical server assisted signature scheme. However, the main problem with SAOTS was high storage requirements for the VS and memory requirements for the mobile client.

We improved the SAOTS construction by addressing both of the above problems. By using a new technique to generate one time signature keys and introducing a new server storage scheme, we were able to dramatically reduce the VS storage requirements as well as the mobile client memory requirements. Only 60 bytes per signature generation are added to the VS storage instead of 5 KB in the previous scheme. Further, the memory requirements for the mobile client are reduced from 2.7 KB to just 20 bytes. The resulting construction seems to be practical from both computational as well as storage point of view.

References

[1] Kemal Bicakci and Nazife Baykal, "Server Assisted Signatures Revisited", T. Okamoto (Ed.): CT-RSA 2004, LNCS 2964, pp. 143–156, 2004.

[2] M. Burnside, D. Clarke, T. Mills, A. Maywah, S. Devadas, and R. Rivest. Proxy- Based Security Protocols in Networked Mobile Devices. Proceedings of the 17th ACM Symposium on Applied Computing (Security Track), March 2002.

[3] A. Boldyreva, A. Palacio, and B. Warinschi. Secure Proxy Signature Schemes for Delegation of Signing Rights. Cryptology ePrint Archive, Report 2003/096, 2003, http://eprint.iacr.org.

[4] P. Beguin and J. J. Quisquater. Fast server-aided RSA signatures secure against active attacks. CRYPTO 95, LNCS No. 963, Springer-Verlag, 1995.

[5] P. Nguyen and J. Stern. The Beguin-Quisquater server-aided RSA protocol from Crypto '95 is not secure. ASIACRYPT 98, LNCS No. 1514, Springer-Verlag, 1998.

[6] M. Jakobsson and S. Wetzel. Secure Server-Aided Signature Generation. In Proc. of the International Workshop on Practice and Theory in Public Key Cryptography (PKC 2001), LNCS No. 1992, Springer, 2001.

[7] N. Asokan, G. Tsudik and M. Waidners. Server-supported signatures. Journal of Computer Security, November 1997.

[8] X. Ding, D. Mazzocchi and G. Tsudik. Experimenting with Server-Aided Signatures. Network and Distributed Systems Security Symposium (NDSS'02), February 2002.

[9] K. Bicakci and N. Baykal. SAOTS: A New Efficient Server Assisted Signature Scheme for Pervasive Computing. In Proc. of 1st International Conference on Security in Pervasive Computing, SPC 2003, LNCS No. 2802, March 2003, Germany.

[10] K. Bicakci and N. Baykal. Design and Performance Evaluation of a Flexible and Efficient Server Assisted Signature Protocol. In Proc. of IEEE 8th Symposium on Computers and Communications, ISCC 2003, Antalya, Turkey.

[11] R. Gennaro and P. Rohatgi. How to Sign Digital Streams. CRYPTO 1997, LNCS No. 1294, Springer-Verlag, 1997.

[12] National Institute for Standards and Technology. Digital Signature Standard (DSS). Federal Register, 56(169), August 30, 1991.

[13] National Institute of Standards and Technology (NIST), "Announcing the Secure Hash Standard", FIPS 180-1, U.S. Department of Commerce, April 1995.

[14] R. L. Rivest, A. Shamir, and L. M. Adleman, “A method for obtaining digital signatures and public-key cryptosystems”, Communications of the ACM, 21(2):120–126, 1978.

[15] T.A. Berson, L. Gong and T.M.A. Lomas, Secure, “Keyed, and Collisionful Hash Functions, Technical Report” SRI-CSL-94-08, SRI International, May 1994.

[16] M. V. Wilkes, “Time-Sharing Computer Systems”, New York: Elsevier, 1972.

[17] R.C. Merkle, A Digital Signature Based on a Conventional Encryption Function, Proc. CRYPTO’87, LNCS 293, Springer Verlag, 1987, pp 369-378.

[18] R.C. Merkle, A Certified Digital Signature, Proc. CRYPTO’89, LNCS 435, Springer Verlag, 1990, pp 218-238.

[19] D. Bleichenbacher and U.M. Maurer, Directed Acyclic Graphs, One-way Functions and Digital Signatures, Proc. CRYPTO’94, LNCS 839, Springer Verlag, 1994, pp 75-82.

[20] D. Bleichenbacher, U.M. Maurer, Optimal Tree-Based One-time Digital Signature Schemes, Proc. STACS’96, LNCS 1046, Springer-Verlag, pages: 363-374, 1996.

[21] D. Bleichenbacher, U.M. Maurer, On the efficiency of one-time digital signatures, Proc. ASIACRYPT’96, LNCS 1163. Springer-Verlag, pages: 145-158, 1996.

[22] K. Bicakci, G. Tsudik, B. Tung, How to construct optimal one-time signatures, Computer Networks (Elsevier), Vol.43(3), pp. 339-349, October 2003.