Efficient Implementation of Keyless Signatures with Hash Sequence Authentication

Ahto Buldas, Risto Laanoja, and Ahto Truu

GuardTime AS, Tammsaare tee 60, 11316 Tallinn, Estonia.

Abstract. We present new ideas for decreasing the size of secure memory needed for hardware implementations of hash-sequence based signatures proposed recently by Buldas, Laanoja and Truu (in the following referred to as BLT). In their scheme, a message m is signed by time-stamping a concatenation $m\|z_t$ of the message and the one-time pseudo-random password z_t intended to sign messages at a particular time t. The signature is valid only if the time-stamp points to the same time t. Hence, the one time passwords cannot be abused after their use.

To efficiently and securely implement such a scheme at the client side, dedicated hardware is needed and thereby, the solutions that save the (secure) memory and computational time are important. For such schemes, the memory consumption directly depends on the efficiency of the *hash sequence reversal algorithms*. The best known reversal algorithm for the BLT scheme uses $O(\log^2 \ell)$ memory. This means that for a signing key that is valid for one year (i.e. $\ell \approx 2^{25}$ with one-second time resolution), the device needs to store about $25^2 = 625$ hash values which for SHA-256 hashing algorithm means about 20 K bytes of secure memory. Another problem with hash sequence reversal algorithms is that they mostly assume that the signature device is always connected to the computer or has an independent power supply. This is a serious limitation for smart-card implementations of the scheme.

We show first that a mini Public Key Infrastructure in the signature device can be used to lower the memory consumption about twice. There is a master key (i.e. a hash sequence) that is used to certify short term (about five minutes) signing keys so that a signature consists of a "short term certificate" which is a hash chain in the master hash tree (used to authenticate the master hash sequence), and a hash chain that is used to authenticate a particular hash value z_t in the sequence.

We also discuss how to implement hash sequence signatures in devices that have no power supply and are not regularly connected to computers, such as smart-cards which are often used as personal digital signature devices. General-purpose cryptographic smart-cards also have many restrictions that limit the use of hash sequence signatures. For example, their hashing speed is relatively low: up to 500 hashing steps per second; their secure memory is of limited size, etc. This all combined with irregular usage patterns makes the use of hash sequence signatures questionable. We show why the hash sequence signature (in its original form) cannot be used as the CA signature in the mini PKI solution.

Finally, we propose a new type of hash sequence signature that is more suitable for smart-card implementations.

1 Introduction

Keys are the most vulnerable components of any digital signature solution. Key exposure is not only dangerous to the key owner but also to the parties that rely on digital signatures as potential evidence to protect their rights. The validity (integrity) of digital signatures that use traditional public-key based mechanisms depends on assumptions that some private keys are secure. On one hand, instant revocation of keys is necessary to protect the signer. On the other hand, the possibility of instant revocation makes the signature verification procedure very complex, because one has to prove that the key was not revoked at the time of creating the signature. This means that many additional confirmations (such as OCSP responses, time-stamps, etc.) must be added to the signature. If instant revocation is possible, the signature key can be revoked immediately after signing and hence, verification mechanisms must be very precise in determining the chronological

order of the signing and the revocation events. Due to the evidentiary function of signatures, this order should also be provable to third parties decades after the document has been signed. The public-key time-stamping solutions such as that proposed in RFC3161 do not eliminate this problem because the validity of time-stamps also depends on the validity of (some) keys.

Traditional electronic signature schemes are also vulnerable to quantum-computational attacks [10, 11, 20] against public-key systems like RSA. Quantum attacks may soon be practical and it is reasonable to take them into account when designing long-term solutions for digital signatures. This is why it becomes more and more important to study alternative schemes that are not vulnerable to quantum attacks.

Considering these two goals (secret-independent integrity and quantum immunity), the so-called hash function based digital signatures are becoming an increasingly popular subject to study. The history of such signature schemes goes back to early 80-ties. Hash function based signatures were first studied by Lamport [14, 15] and Merkle [16]. Lamport (one-time) signatures were designed for signing a message bitwise. For each bit of the message, two secret random numbers r_0 and r_1 are chosen and their one-way images $f(r_0)$ and $f(r_1)$ published as the public key. For signing the corresponding bit $b \in \{0, 1\}$ of the message, the random number r_b is revealed. Obviously, the key pair $f(r_0), f(r_1)$ can only be used once and must be immediately revoked after the use. The signature can be made more efficient if the message is hashed before signing and the bits of the hash are signed. Merkle proposed the use of hash trees for more efficient publishing and verifying the public keys of the Lamport signature. Haber et al [12] showed how to use hash functions for time-stamping so that the validity of time-stamps does not depend on any secrecy assumptions, hence providing the Lamport scheme with a key-independent revocation check mechanism. It was proven in [6, 5] that the hash function based time-stamps cannot be back-dated if the hash function used is collision resistant.

In [4], a new type of hash function based signature scheme was proposed in which the signature process involves a signature server. For every time unit t, the client has a one-time pseudo-random password z_t, and the passwords are computed using the recurrent scheme $z_t = f(z_{t+1})$, where f is a one-way (hash) function. The last element z_0 of the one-way chain $z_0 \stackrel{f}{\leftarrow} z_1 \stackrel{f}{\leftarrow} z_2 \stackrel{f}{\leftarrow} \ldots \stackrel{f}{\leftarrow} z_\ell$ is the public key of the signer. To sign a message M at time t, the signer: (1) combines M with a one-time password z_t that is dedicated for signing messages at time t; and (2) sends a hash $x = h(M, z_t)$ to the server to obtain a hash-tree based time-stamp $T_{t'}(x)$ for x. The signature $\langle x, z_t, T_{t'}(x)\rangle$ is considered valid only if $t = t'$ and z_t is verified to be the t-th element of the chain. This can be done by iterating f (exactly t times) on z_t and comparing the result with z_0 (the public key).

For signing messages during the next time unit $(t + 1)$, the pre-image z_{t+1} of z_t must be found and released. As the hash function cannot be inverted, the only way is to compute z_{t+1} from the perviously saved hash values z_j with $j > t + 1$ (the so-called "pebbles"). In order to design an efficient (hardware) device for such signatures, one has to find efficient ways of gradually releasing the pre-images of the one-time passwords. Secure memory is a relatively costly resource in hardware devices and therefore even a slight decrease in memory consumption is valuable. Trivial solutions are (1) to store the whole chain (i.e. use $O(\ell)$ memory) and hence make the traversal possible in $O(1)$-time; or (2) to store only z_ℓ (i.e. use $O(1)$ memory) and compute the next password in $O(\ell)$ time. The best traversal algorithms by Jakobsson et al [13, 8] proposed a technique with $O(\log \ell)$ memory and $O(\log \ell)$ traversal step.

In [4], the hash sequence has an additional hash tree structure in order to make the verification procedure more efficient (from $O(\ell)$ to $O(\log \ell)$ hashing operations), which means that not only the

hash values in the sequence must be released but also the verification (authentication) hash chains from particular hashes to a root hash value (that is included into the signer's certificate). Based on Jakobsson's hash sequence traversal algorithm [13, 8] they construct a hash-calendar traversal algorithm that requires $O(\log^2 \ell)$ units of memory and $O(\log^2 \ell)$ time per one pre-image.

We propose a new method for signatures of this kind that enables to reduce the memory consumption about twice. The main idea is to use a mini Public Key Infrastructure in the signature device, so that there is a master key (i.e. a hash sequence) that is used to certify short term (about five minutes) signing keys. So, a signature consists of a "short term certificate" which is a hash chain in the master hash tree (used to authenticate the master hash sequence), and a hash chain that is used to authenticate a particular hash value z_t in the sequence. We point out, though, that the "pebbling" solutions [13, 8, 4] as well as the new hierarchical solution are efficient only if the signature device is constantly up and running, which is not the case for smart-cards often used as personal digital signature devices.

Therefore, we also discuss how to implement hash sequence signatures in devices that have no power supply and are not regularly connected to computers. General-purpose cryptographic smart-cards also have many restrictions that limit the use of hash sequence signatures. For example, their hashing speed is relatively low: up to 500 hashing steps per second; their secure memory is of limited size, etc. This all combined with irregular usage patterns makes the use of hash sequence signatures questionable. We show why the hash sequence signature (in its original form) cannot be used as the CA signature in the mini-PKI solution. Therefore, we propose a new type of hash sequence signature that is more suitable for that purpose.

The paper is organized as follows. In Section 2, we describe the state of the art. In Section 3, we present the hierarchical approach to decrease the memory consumption of hardware/software implementations of the BLT-type schemes. We also point out that for smart-card solutions, the hash chain reversal algorithms do not fit very well for the BLT-type signatures if the index i of the password has a direct connection with physical time. In Section 4, we present a new type of hash sequence signature scheme that is much more suitable for smart-card applications and for which the traversal algorithms suit well. In Section 5, we discuss several practical use cases of the new scheme.

2 Preliminaries and State of the Art

2.1 One-Way Hash Functions

A function f is said to be *one-way* if, given an output $f(x)$ of a randomly chosen input x, it is hard to find x' such that $f(x') = f(x)$. In the following, we also need a somewhat non-standard one-wayness concept: if f and h are functions with equal domains, then we say h to be *f-one-way* if, given an output $h(x)$ of a randomly chosen input x, it is hard to find $f(x)$.

2.2 Server-Based Signatures and Non-Repudiation

The BLT-type signatures [4] must certainly be *server-based*, i.e. electronic signature solutions in which a publicly available server participates in the signature creation process. This is because a reliable publication process is an essential part of the solution. The conventional solutions based on public-key cryptography assume that users may sign their documents off-line without any communication with servers. For many reasons, server-based signatures are preferable. For example,

they can reduce the computational cost of creating digital signatures, and make it easier to detect possible abuses of signature keys.

Many different forms of server-based signatures exist. For example, Lamport signatures [14] are server based, and also the so-called *on-line/off-line signatures* first proposed in 1989 by Even, Goldreich and Micali [9] in order to speed up the signature creation procedure, which is usually much more time consuming than verification. The so-called *Server-Supported Signatures (SSS)* proposed in 1996 by Asokan, Tsudik and Waidner [1] delegate the use of time-consuming operations of asymmetric cryptography from clients (ordinary users) to a server. Clients use hash chain authentication [15] to send their messages to a signature server in an authenticated way and the server then creates a digital signature by using an ordinary public-key digital signature scheme. In SSS, signature servers are not considered Trusted Third Parties because the transcript of the hash chain authentication phase can be used as evidence. In SSS, servers cannot create signatures in the name of their clients. The so-called *Delegate Servers (DS)* proposed in 2002 by Perrin, Burns, Moreh and Olkin [17] reduce the problems and costs related to individual private keys. In their solution, clients (ordinary users) delegate their private cryptographic operations to Delegation Servers that, after authenticating the clients, sign messages on their behalf.

One of the basic functions of digital signatures is non-repudiation, i.e. potential ability to use the signature as evidence against the signer. Solutions in which trusted third parties are (technically) able to sign on behalf of their client are not desirable for non-repudiation because clients may use that argument to (fraudulently) call their signatures into question. Therefore, solutions that presume clients having their personal signature devices are preferable to those relying entirely on trusted parties. As we also want the signatures to be quantum-immune, we cannot use ordinary digital signature mechanisms (like RSA, DSA, etc.) in the signing devices. One way of avoiding quantum threats is using one-time hash-chain type password schemes.

One-Time Hash-Password Schemes The main idea behind iterated hash chain authentication [15] is that the client first generates a chain of hash values (with reverse order of indices). Let ℓ be the number of possible authentication sessions (i.e. the number of one-time passwords). Then z_ℓ is a uniformly random seed, and $z_i \leftarrow f(z_{i+1})$ for all $i < \ell$. The last element $z_0 = f(z_1)$ in the chain is the so-called *public key*, which is published and also given to the server.

Now, the client will use z_1 in the first authentication session. Server does not know z_1 before the client uses it, but as it knows z_0 (the public key) it is possible to verify the password by checking the relation $z_0 = f(z_1)$. After the first session, the server already knows z_1 and hence it is possible for the server to check z_2 used by the client (by the relationship $z_1 = f(z_2)$), etc.

Time-Dedicated Passwords The indices i may also be related to time, i.e. z_i is assumed to be published by the client not before time $t_0 + i$, where t_0 is a certain initial time that is also published together with the public key z_0. If now for example a message m and a z_i are time-stamped together, this may be considered as a signature of m. The signature is correct only if the date of the time-stamp is not later than $t_0 + i$. The signature cannot be forged because all published passwords $z_1, \ldots, z_i$ are useless for creating signatures after $t_0 + i$, as no suitable time-stamps can be obtained any more (at least if the time-stamps were not intentionally back-dated by the time-stamping authority).

This idea in the context of authentication was first used in the so-called TESLA protocol [18]. However, as it was described by the authors of TESLA, the scheme is not quite suitable for digital

signatures, because of inefficiency of off-line verification. TESLA was designed to authenticate parties who are constantly communicating with each other. This is not the case for digital signatures and if one would convert TESLA to autonomous digital signatures, their size would grow to $O(\ell)$.

Efficient Hash Sequence Reversal Considering possible security problems in clients' general-purpose computers, the password sequence should be maintained by a dedicated cryptographic hardware device that generates the random seed, computes the chain and then consecutively reveals the pre-images. Naive solutions would either require $O(\ell)$ amount of memory (if the whole hash chain is pre-computed and stored in the device) or $O(\ell)$ time to compute the next pre-image (if only the seed is stored). It was shown by Jakobsson et al [13, 8] that $O(\log \ell)$ memory and $O(\log \ell)$ time (per one pre-image) is sufficient. The traversal algorithm of Jakobsson [13] uses $O(\log \ell)$ precomputed hash values (called *pebbles*) as shortcuts for consecutive pre-image computation. By suitably placing the pebbles, the pre-image computation time can be reduced from $O(\ell)$ to $O(\log \ell)$. Even more efficient algorithm was presented by Schoenmakers [19].

Efficiently Verifiable Hash Sequences For some applications, especially for signatures, it should be possible to verify whether the one-time password z_i that was just used is indeed the i-th element of the chain. An obvious way to do it is to iterate z_i exactly i times and compare the result with the public key z_0. For long hash sequences, it may take a lot of time, though. Therefore, an additional hash-tree structure may be used to speed up the process [4]. Let $r = \mathcal{T}^h(z_1, \ldots, z_\ell)$ be the root hash of a hash tree, created with a hash function h. The public key is then a pair (z_0, r). If the one-time password z_i is used, a hash chain from z_i to r can be presented to show that z_i is in the right place of the chain. The proof is of size $O(\log \ell)$, and one only needs $O(\log \ell)$ steps (instead of $O(\ell)$) to verify it.

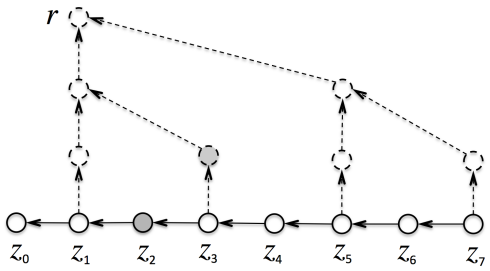

Fig. 1. Hash sequence with an additional hash tree structure (dashed).

Note that h must be f-one way to avoid premature disclosure of one-way passwords. If the tree and the chain were computed with the same one-way hash function, then the hash chain for z_1 would contain z_2 (the two gray nodes in Fig. 1 would have equal values) and hence z_2 would be prematurely disclosed.

Verifiable Reversal is a hash sequence traversal, except that in addition to consecutively revealing pre-images, the hash chains to the root hash r must be revealed. Therefore, the original algorithm of Jakobsson (that uses pebbles with one hash value) was modified in [4] by adding to each pebble

a hash chain (consisting of $O(\log \ell)$ hash values) to the global root hash. As there are $O(\log \ell)$ pebbles, the memory requirement in this algorithm is $O(\log^2 \ell)$.

2.3 The BLT Signature Scheme

Based on the idea of efficiently verifiable hash sequences, Buldas, Laanoja and Truu [4] proposed a new type of signature scheme (referred to as the BLT-scheme in the following), which is quantum immune (at least against the known quantum attacks) and in which the integrity of the signatures does not depend on the secrecy of keys. In the following, we summarize the basic procedures of their signature scheme as well as some additional procedures related to secure client-side implementations of the scheme.

Key Generation: The client generates a random seed z_ℓ and a key-hash chain $z_1, z_2, \ldots, z_\ell$ by using the recursive relation $z_{i-1} = f(z_i)$ (for all $i = \ell \ldots 1$). Every z_i is a one-time password for a particular time. The client also computes the key hash-tree (Fig. 1) and its root hash r. The client's public key consists of $z_0 = f(z_1)$ and r.

Public Key Certificates: A public key certificate for client is a 5-tuple $\langle \mathrm{ID}_c, z_0, r, t_0, \mathrm{ID}_s \rangle$, where ID_c is the identity of the client, (z_0, r) is the public key, t_0 is the time when the certificate becomes valid (i.e. z_1 is intended to sign documents at time $t_0 + 1$, z_2 is for signing at $t_0 + 2$, etc.), and ID_s is the identity of the signature server that is authorized to serve the client.

The certificate is sent to the signature server and is also published in a way that is not controlled by the signature server, i.e. the server cannot create or change clients' certificates. To *revoke* the certificate, it is sufficient to send the server a revocation note, after which the server will not create the signatures any more.

Signing a Document: To sign a message m (or a hash of a message) at time $t > t_0$ (where $t = t_0 + i$), the client computes $x = h(m, z_i)$ and sends x together with its identity ID_c as a request to the signature server. The server checks that the certificate of the client has not been revoked and (if not) creates a hash-tree time-stamp S_t for the pair (x, ID_c), and sends S_t back to the client.

The signature for m is $\langle \mathrm{ID}_c, i, z_i, c_i, S_t \rangle$, where c_i is the hash chain which proves that z_i is the i-th element of the key-hash chain.

Verifying a Signature: To verify the signature $\langle \mathrm{ID}_c, i, z_i, c_i, S_t \rangle$ on the message m with the certificate $\langle \mathrm{ID}_c, z_0, r, t_0, \mathrm{ID}_s \rangle$, the following is checked:

- Client identities in the certificate and in the signature coincide.
- The key z_i and the hash chain c_i lead to the root hash value r, or that f iterated on z_i exactly i times leads to z_0.
- S_t is a valid time-stamp on $(h(m, z_i), \mathrm{ID}_c)$.
- Time t extracted from S_t satisfies $t = t_0 + i$, i.e. the correct key was used.
- Server identities in S_t and the certificate coincide, i.e. the server was authorized by the client to create the signature.

Security: The security of the signature scheme relies on the fact that if z_i is used right before $t_0 + i$ (when z_i expires), then it is impossible to abuse z_i. If z_i is used too early (sufficiently long before $t_0 + i$), then z_i can be abused by anyone who has the signature with z_i. So, for the security of the scheme, it is viable that *the signer verifies the signature before disclosing it* to other parties. This guarantees, due to the condition $t = t_0 + i$ that z_i is safe to disclose.

Signatures are considered valid only if $t = t_0 + i$, where t is the time indicated by the data signature (time-stamp) S_t obtained from the service. In practical implementations, the value of t depends on the service delay. Hence, t may vary but the values t_0 and i are fixed before sending the signature request to the server. Therefore, the equality $t = t_0 + i$ does not necessarily hold.

The service can be organized so that the delay is predictable and is no more than a few seconds. Then the client may send several requests in parallel using i, $i+1$, $i+2$, $\ldots$, $i+\Delta$, where Δ is the maximum accepted service delay. Hence, there is always $i' \in [0, \ldots, \Delta]$ for which $t = t_0 + i'$. The client keeps the signature with such i' and deletes the rest.

Hardware Implementation: For better protecting the keys z_i against client-side malware, it would be preferable to hold these keys in dedicated hardware devices. To avoid premature disclosure of keys, there should be a mechanism that prevents malware from abusing the hardware device, i.e. making the device disclose future keys. There are two ways how this can be done:

- *Independent hardware clock* in the signature device that cannot be externally adjusted and prevents the use of z_i earlier than $t_0 + i(1-\delta) - \Delta$, where δ is the maximum expected clock drift per time unit and Δ is the maximum service delay. The shortcoming of this approach is that ordinary quartz clocks may drift a few seconds per day, so the yearly drift can be about 10 minutes. More precise clocks might be too expensive for personal signature devices.
- *Secure channel between the server and the signature device* that enables the server to securely announce precise time to the signature device, so that there is no need to have a clock in the signature device itself. The device and the server may have a shared secret key K that is not known to any programs (and hence, also to malware) running in the client's computer. To sign a hash m of a message, the client sends m to the device together with an index i (of the key z_i). The device does not reveal z_i directly, but only the request $x = h(m, z_i)$ that will be sent to the signature server. Together with the signature $S_t(x, \mathrm{ID}_c)$, the server also sends to the client a Message Authentication Code $\mathsf{MAC}_K(t)$. The signature device does not reveal z_i and c_i before having seen a correct $\mathsf{MAC}_K(t)$ such that $t_0 + i < t$, i.e. after z_i has already been expired. This guarantees, assuming that there is no co-operating malware both in the server and in the client's computer, that the keys z_i are never prematurely exposed.

Memory Consumption Problem: For signing messages during the next time unit $(t+1)$, the pre-image z_{t+1} of z_t must be found and released. As the hash function cannot be inverted, the only way is to compute z_{t+1} from the perviously saved hash values z_j with $j > t+1$ (the so-called "pebbles"). Secure memory is a relatively costly resource in hardware devices and therefore even a slight decrease in memory consumption is valuable. The best backwards traversal algorithms for hash sequences require $O(\log \ell)$ memory, where ℓ is the length of the sequence. In our case, the hash sequence has an additional hash tree structure in order to make the verification procedure more efficient (from $O(\ell)$ to $O(\log \ell)$ hashing operations), which means that not only the hash values in the sequence must be released but also the verification (authentication) hash chains from particular hashes to a root hash value (that is included in the signer's certificate). The best known algorithm

for such an authenticated hash sequence traversal uses $O(\log^2 \ell)$ memory. This means that for the signing key that is valid for one year (i.e. $\ell \approx 2^{25}$ key-hash chain elements, assuming one-second time resolution), the device needs to store about $25^2 = 625$ hash values which for SHA-256 hashing algorithm means about 20 KB of secure memory.

In the next section, we propose a new method that can reduce the memory consumption about twice. The main idea is to use a mini Public Key Infrastructure in the signature device, so that there is a master key (i.e. a hash sequence) that is used to certify short term (about five minutes) signing keys. So, a signature consists of a "short term certificate" which is a hash chain in the master hash tree (used to authenticate the master hash sequence), and a hash chain that is used to authenticate a particular hash value z_t in the sequence.

3 Hierarchical Approach

All the solutions use the following approach. We divide the validity period of ℓ seconds into A-second sub-periods. There are $\frac{\ell}{A}$ of such periods. All periods have different password sequences with independently generated random seed values. Only one period (buffer) is active at any moment of time. At the same time, the next period may be in the phase of key generation. When a period ends, the next period is activated by:

- Generating a new A-second password sequence $z_0 \xleftarrow{f} z_1 \xleftarrow{f} z_2 \xleftarrow{f} \ldots \xleftarrow{f} z_{A-1} \xleftarrow{f} z_A$, where z_A is a uniformly chosen random element in the domain of f.
- Signing the last hash value of the new sequence by using the so-called master hash sequence of length $\frac{\ell}{A}$.

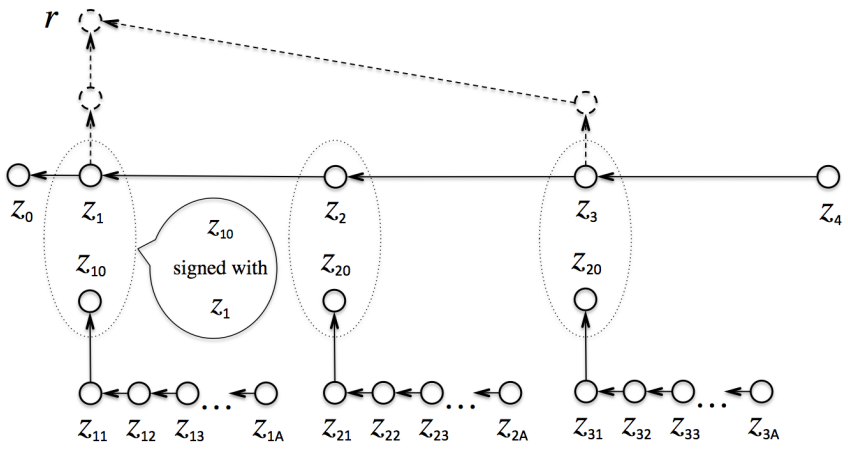

Fig. 2. Hierarchical signature scheme. The upper chain $z_1, z_2, \ldots$ is used to sign short-term public keys.

3.1 Buffer without Pipelining

We assume the use of the $\log^2$ pebbling algorithm of [4] for both chains. Therefore, the memory requirement (the number of stored passwords) is about:

$$\log^2 A + \log^2 \frac{\ell}{A} = 2\log^2 A - 2\log A \cdot \log \ell + \log^2 \ell = 2x^2 - 2\lambda x + \lambda^2 \ ,$$

where $x = \log A$ and $\lambda = \log \ell$. It is easy to see that this quadratic polynomial has global minimum $\frac{\lambda^2}{2}$ at the point $x = \frac{\lambda}{2}$. This means that the memory size is decreased by half compared to [4]. For example, for one-year certificate validity period and one-second time resolution (i.e. $\ell \approx 2^{25}$ and $\lambda \approx 25$), and with 256-bit f (i.e. 32 output bytes), the memory size would be about $\frac{25^2}{2} \cdot 32 \approx 10$ KB.

3.2 Buffer with Pipelining

We assume again the use of the $\log^2$ pebbling algorithm of [4] for both chains. In addition, to avoid delays during the new sequence generation phase, we generate the new sequence in parallel with using the current one, which means additional memory buffer of size $\log^2 A$. Therefore, the memory requirement (the number of stored passwords) is about:

$$2\log^2 A + \log^2 \frac{\ell}{A} = 3\log^2 A - 2\log A \cdot \log \ell + \log^2 \ell = 3x^2 - 2\lambda x + \lambda^2 \ ,$$

where again $x = \log A$ and $\lambda = \log \ell$. This time the minimum is $\frac{2\lambda^2}{3}$ at the point $x = \frac{\lambda}{3}$, which means memory size decrease by a third compared to [4]. Using again the example from the previous section (i.e. $\ell \approx 2^{25}$ and $\lambda \approx 25$, 256-bit f), the memory size would be about $2\frac{25^2}{3} \cdot 32 \approx 13$ KB.

3.3 Simple Buffer

We may also assume that the lower-level buffer is relatively short and can be used without the supporting hash tree structure. So, we use the $\log^2$ pebbling algorithm of [4] only for the upper chain. Therefore, the memory requirement (the number of stored passwords) is about:

$$\log^2 \frac{\ell}{A} = \log^2 A - 2\log A \cdot \log \ell + \log^2 \ell = x^2 - 2\lambda x + \lambda^2 \ ,$$

where once more $x = \log A$ and $\lambda = \log \ell$. In this case, the memory size is the smaller the larger A we choose. Hence, we have to specify what the reasonable A is. For example, if we accept one second delays and the device we have is capable of computing 500 (about 2^9) hashes per second, then the reasonable choice of A is 2^9. Then for $\ell \approx 2^{25}$ and $\lambda \approx 25$, and 256-bit f, the memory size would be about $(25-9)^2 \cdot 32 \approx 8.2$ KB.

3.4 Irregular Usage Problem

The solutions above are good if the signature device is constantly powered and running. This is a reasonable assumption in case of device signatures, but not in the case of personal signatures. Personal signature devices are not used very often, and so most of the one-time passwords in the chain will never be used.

Even if we accept that, two problems still remain. Firstly, the hash sequence traversal algorithms are not intended for irregular usage patterns and become very inefficient if there is a need to catch up. The time needed for catching up is $O(\ell)$. Secondly, the CA signatures must be created at the seconds intended for that. These seconds are $t_0 + i \cdot A$, where A is the length of the sub-period in seconds. If the validity of the certificates is 10 minutes (i.e. $A = 600$), and the signature device is connected to client's computer right after $t_0 + i \cdot A$, then for about 10 minutes, there are no valid certificates for signing, and hence the client will not be able to sign for almost 10 minutes.

4 A New Hash Sequence Signature Scheme

To overcome the irregular usage problem, we propose a new hash sequence signature scheme that has no predetermined schedule of using the one-time passwords z_i, i.e. z_i is used to sign the i-th message and there is no direct relationships between i and the current time. The challenge is ensuring that z_i will not be abused by third parties (to sign more messages) after having been made public. The main idea is to use the signatures themselves as (server-signed) revocation notes of z_i: the signature request also contains the index i of z_i, which is included into the signature. The server is not allowed to sign two messages with the same index. In case it uses z_i (possibly co-operating with a malicious adversary) twice, the client is able to prove the forgery by showing the original (older) signature with z_i.

The main advantage of the new scheme is that clients can use the one-time passwords whenever they want to. The signature counter mechanism on the server side prevents malicious verifiers (third parties that verify signatures) from abusing the already-used passwords. Even if a malicious verifier co-operates with a malicious server, clients are still capable of defending themselves with the list of already created signatures or the re-confirmed time-lists (the latter will be introduced in Sec. 4.2).

4.1 Description of the Scheme

Key Generation: The client generates a random seed z_ℓ and a key-hash chain $z_1, z_2, \ldots, z_\ell$ by using the recursive relation $z_{i-1} = f(z_i)$ (for all $i = \ell \ldots 1$). Every hash value is a one-time password. The client also computes the key hash-tree and its root hash r. The client's public key consists of $z_0 = f(z_1)$ and r.

Public Key Certificates: A public key certificate for client contains the identity ID_c of the client, the public key (z_0, r), the validity period $t_0 \ldots t_1$, and the identity ID_s and connection parameters of the signature server that is authorized to serve the client.

The certificate is sent to the signature server and is also published in a way that is not controlled by the signature server, i.e. the server cannot create or change clients' certificates. To *revoke* the certificate, it is sufficient to send the server a revocation note.

Signing a Document: We assume that there is a shared secret authentication key between the server and the client. This key is used to authenticate the messages sent between them. For every client, the server stores the number of signatures already created by this client. To sign the i-th message m (or a hash of a message):

- The client (with identity ID_c) computes $x = h(m, z_i)$ and sends (x, ID_c, i) as a request to the signature server.

– The server checks that i is consistent with the stored signature count and that the certificate of the client has not been revoked.
– If the checks were successful, the server obtains a time-stamp $S_t(x, \mathrm{ID}_c, i)$ for (x, ID_c, i), and sends $S_t(x, \mathrm{ID}_c, i)$ back to the client.
– The client stores all the signatures $S_{t_1}(x_1, \mathrm{ID}_c, 1)$, $S_{t_2}(x_2, \mathrm{ID}_c, 2), \ldots$ he/she ever creates. This is necessary for later disputes where client has to deny forgeries created by a malicious server.

The signature for m is $\langle \mathrm{ID}_c, i, z_i, c_i, S_t \rangle$, where c_i is the tree hash chain which proves that z_i is the i-th element of the key hash chain.

Verifying a Signature: To verify a signature $\langle \mathrm{ID}_c, i, z_i, c_i, S_t \rangle$ with a certificate $\langle \mathrm{ID}_c, z_0, r, t_0, t_1, \mathrm{ID}_s \rangle$ the following is checked:

– Client identities in the certificate and in the signature coincide.
– The key z_i and the hash chain c_i lead to the root hash value r, i.e. the correct key was used.
– S_t is a valid time-stamp on $(h(m, z_i), \mathrm{ID}_c, i)$.
– Server identities in S_t and the certificate coincide, i.e. if the server was authorized by the client to create the signature.

Denying a Forgery: Dishonest third parties in co-operation with the signature server may try to abuse the one-time keys z_i that are already used and public. Say, $S_{t'}(x', \mathrm{ID}_c, i)$ (where $x' = h(m', z_i)$) is a new signature created with z_i at time $t' > t$.

Then the client can show the stored signature $S_t(x, \mathrm{ID}_c, i)$ and prove thereby that $S_{t'}(x', \mathrm{ID}_c, i)$ is a server-assisted forgery. The service provider is then fully responsible for the consequences.

In this scheme, the client has to store all the signatures that have been created. For later availability of the signatures, it is desirable to store the signatures inside the signature device or to any other medium that is not controlled by the signature server. Note that the memory in which the signatures are held should not be secret and hence the price of storing the signatures may be acceptable.

4.2 Storing Time-Lists Instead of Signatures

To save storage, the scheme can be modified by using hashed time-lists. For example, instead of storing two signatures $S_{t_1}(x_1, \mathrm{ID}_c, 1)$ and $S_{t_2}(x_2, \mathrm{ID}_c, 2)$, the client may store only the second one, assuming the server signs $(t_1, x_2, \mathrm{ID}_c, 2)$ at t_2 to confirm that the previous signature was created (and the previous password z_1 revoked) at t_2. To develop this idea further, assume that (when signing the i-th message m_i), the server signs $(y_{i-1}, x_i, \mathrm{ID}_c, i)$, where y_{i-1} is the incremental hash of the time-list $t_1, \ldots t_{i-1}$ computed by the recursive scheme $y_j = h(y_{j-1}, t_j)$, $y_0 = \lfloor\rfloor$. After creating the signature $S_{t_i}(y_{i-1}, x_i, \mathrm{ID}_c, i)$, the server computes $y_i = h(y_{i-1}, t_i)$ and stores y_i. The previous value y_{i-1} is no more needed.

The client has to store the time-list $t_1, t_2, \ldots, t_{i-1}$ and the last signature $S_{t_i}(y_{i-1}, x_i, \mathrm{ID}_c, i)$, which indirectly confirms that $z_1, \ldots, z_{i-1}$ were revoked at $t_1, \ldots, t_{i-1}$, respectively. If the size of the signature is 3 KB, the time values are 32-bit integers, and the client generates 500 signatures, then the storage required on the client side for the time-list and the signature is about 5 KB.

A modification of this scheme is that the server is made responsible for storing the time-list $t_1, t_2, \ldots, t_{i-1}$ while the client only stores and updates the hash y_{i-1} of the list and keeps only the

last signature. It is presumed that once the server signed $(y_{i-1}, x_i, \mathrm{ID}_c, i)$, it must be able to present a time list with $i-1$ elements that hashes to y_{i-1}. This further reduces the memory requirements on the client side.

5 Discussion: Typical Use Cases of Digital Signatures

To lower the memory requirements, one may use a hierarchical scheme (Sec. 3) in which the new signature scheme is used at the upper level and the ordinary BLT-scheme [4] at the lower level. Such a two-level scheme solves the irregular usage problem. Every time, the client wants to sign a message, the signature device can immediately generate a short-term (say, five-minute) certificate that can then be used to sign messages with the ordinary BLT-scheme. Note that the time-lists only contain the times at which the short-term certificates were signed, as there is no need for time lists in the ordinary BLT signature scheme.

The choice of a suitable configuration of a BLT-signature solution highly depends on the particular usage scenario. We discuss three different use cases of digital signatures and show that there are suitable solutions for all of them. The use cases we analyse are:

- *Personal Digital Signatures*: In this case, the signature is typically used not every day and no more than few times a day which means that the signature is rarely used more than 500 times a year. Typically, the usage pattern is irregular. The signature device is mostly not connected to client's computer. This is the case, where the original BLT-scheme is not quite suitable and the new scheme can help. For example, if we use the simple buffer scheme (Sec. 3.3) with $A = 2^9$ (i.e. with 8.5 minute certificates) and $\ell = 2^{25}$ (one-year validity period), the size of a signature is 2 KB, and the device is used 500 times, then the required memory size for the device is about 12 KB (8 KB for pebbles + 2 KB for time-lists + 2 KB for the stored signature). If the device is capable of computing 500 hashes per second, then the signing time is about two seconds: one second for the generation of a one-time certificate and one second for obtaining a signature from the server.
- *Employee's Digital Signatures*: In this case, many signatures can be created in one day. The signature device is regularly connected to employee's computer (during every working day). Still there are long time periods during which there is no connectivity and that is why the original BLT-scheme is still not quite suitable. Due to the regular connectivity, we can use a hierarchical combination of the new scheme (for the upper chain) and the original BLT-scheme. The certificates can be relatively long term (for one day) and that is why the size of time-lists is not large (no more than 365 time values per year) which is about 1.5 KB. If we use a dual version of the simple buffer scheme (Sec. 3.3) with the new scheme (without trees) in the upper chain and the original BLT-scheme in the lower chain (with trees) and choose $A = 2^{16}$, then the memory consumption is about 11.5 KB (8 KB for pebbles + 1.5 KB for time-lists + 2 KB for the stored signature). For the devices capable of computing 500 hashes per second, short term keys are generated in 1-2 minutes, which is not a problem as this is done just once in the beginning of the working day.
- *Device Signatures*: In this case, signatures are created automatically by a computer and the signature device is constantly connected to the computer. In this case, the original BLT-scheme or the buffer with pipelining solution (Sec. 3.2) with memory consumption about 13 KB can be used.

References

1. Asokan, N., Tsudik, G., Waidner, M.: Server-supported signatures. J. Computer Security (1996) 5: 131–143.
2. Bernstein, D.J.: Cost analysis of hash collisions : will quantum computers make SHARCS obsolete?. In: Proceedings 4th Workshop on Special-purpose Hardware for Attacking Cryptographic Systems–SHARCS'09, Lausanne, Switserland, September 9-10, 2009), pp. 105–116 (2009)
3. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. In: Lucchesi, C.L., Moura, A.V., (Eds.), LATIN'98: LNCS 1380, pp. 163–169 (1998)
4. Buldas, A., Laanoja, R., Truu, A.: Efficient Quantum-Immune Keyless Signatures with Identity. In: Cryptology ePrint Archive 2014/321 (2014)
5. Buldas, A., Niitsoo, M.: Optimally tight security proofs for hash-then-publish time-stamping. In: ACISP 2010. LNCS 6168, pp. 318–335 (2010)
6. Buldas, A., Saarepera, M.: On provably secure time-stamping schemes. In: ASIACRYPT 2004, LNCS 3329, pp. 500–514 (2004)
7. Catalano, D., Di Raimondo, M., Fiore, D., Gennaro, R.: Off-line/on-line signatures; theoretical aspects and experimental results. In Cramer, R. (Ed.) PKC 2008, LNCS 4939, pp. 101–120. Springer Heidelberg (2008)
8. Coppersmith, D., Jakobsson, M.: Almost optimal hash sequence traversal. In: Blaze, M. (ed.): FC 2002, LNCS 2357, pp. 102–119 (2003)
9. Even, S., Goldreich, O., Micali, S.: On-line/off-line digital signatures. J. Cryptology (1996) 9: 35–67.
   21 (6): 467–488 (1982)
10. Grover L.K.: A fast quantum mechanical algorithm for database search, Proceedings, 28th Annual ACM Symposium on the Theory of Computing, p. 212 (1996)
11. Grover L.K.: From Schrödinger's equation to quantum search algorithm. American Journal of Physics 69(7): 769–777 (2001)
12. Haber, S., Stornetta, W.-S.: How to time-stamp a digital document. Journal of Cryptology 3(2), 99–111 (1991)
13. Jakobsson, M.: Fractal hash sequence representation and traversal. In Proceedings of the 2002 IEEE International Symposium on Information Theory (ISIT 02), pp. 437–444 (2002)
14. Lamport, L.: Constructing digital signatures from a one way function. Comp. Sci. Laboratory. SRI International (1979)
15. Lamport, L.: Password authentication with insecure communication. Comm. ACM (1981) 24(11): 770–772.
16. Merkle, R.C.: Protocols for public-key cryptosystems. In: Proceedings of the 1980 IEEE Symposium on Security and Privacy, pp. 122–134 (1980)
17. Perrin, T., Burns, L., Moreh, J., Olkin, T.: Delegated cryptography, online trusted parties, and PKI. In 1st Annual PKI Research Workshop—Proceedings, pp. 97–116 (2002)
18. Perring, A., Canetti, R., Tygar, J.D., Song, S.: The TESLA broadcast authentication protocol. In CryptoBytes, 5:2, pp. 2–13 (2002)
19. Schoenmakers, B.: Explicit optimal binary pebbling for one-way hash chain reversal. In: Cryptology ePrint Archive 2014/329 (2014)
20. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, SIAM J. Comput. 26 (5): 1484–1509 (1997)