Efficient Quantum-Immune Keyless Signatures with Identity

Ahto Buldas[1,2], Risto Laanoja[1,2], and Ahto Truu[1]

[1] Guardtime AS, Tammsaare tee 60, 11316 Tallinn, Estonia. ahto.buldas@guardtime.com
[2] Tallinn University of Technology, Ehitajate tee 5, 12618 Tallinn, Estonia.

Abstract. We show how to extend hash-tree based data signatures to server-assisted personal digital signature schemes. The new signature scheme does not use trapdoor functions and is based solely on cryptographic hash functions and is thereby, considering the current state of knowledge, resistant to quantum computational attacks. In the new scheme, we combine hash-tree data signature (time-stamping) solutions with hash sequence authentication mechanisms. We show how to implement such a scheme in practice.

1 Introduction

Electronic signatures use secret keys to protect the integrity of data. Keys can be compromised because of technical failures, human factor, or both threats in combination. In the creation process of electronic signatures, some types of secrets (keys, passwords, etc.) seem unavoidable, because secrets are necessary for authenticating the signer. Even if one uses biometric authentication, there must be a secure channel between the biometric reader and the signature creation device, and even if the signature creation device is physically (and securely) connected to biometric reader, the signature itself must be protected against modification (the integrity of signatures) and traditional solutions use public-key cryptography for that. This means that the validity of signatures depend on assumptions that some private keys are secure. For example, the keys of Public Key Infrastructure (PKI) service providers like the OCSP responders are used to sign the validity statements of public-key certificates. Thereby, signatures created with secret keys look more like testimonies than independent (irrefutable) proofs of facts and it is hard to achieve certainty beyond reasonable doubt. Large service providers are often even interested in denying the fact that their keys might have been abused.

Instant revocation has been a serious problem for traditional PKI signatures. On one hand, revocation is necessary to protect the signer if the signature key becomes insecure. On the other hand, the possibility of revocation makes the signature verification procedure much more complex, because it must be proven that the key was not revoked at the time of creating the signature. This means that many additional confirmations (such as OCSP responses, timestamps, etc.) must be added to the signature together with public-key certificates. Note that if instant revocation is possible, a signature key might be revoked right after signing, which means that electronic signature solutions must be very precise in determining the chronological order of the signing and the revocation events. For some types of documents, such a determination must be possible decades after these documents have been signed.

Traditional electronic signature schemes are also vulnerable to quantum-computational attacks against traditional public-key cryptosystems (like RSA) that use the so-called *trapdoor one-way functions*. Quantum attacks have become more and more practical and it is reasonable to take them into account when designing electronic signature solutions for long-term use, i.e. for maintaining evidence collection—the so-called *chain of custody* management. The main goal of this work is to construct a signature system in which:

- Signatures are compact in size and independently verifiable, their verifiability does not require any communication with third parties.
- Instant revocation of signature creation keys is possible and does not involve certificate revocation list management.
- Integrity of signatures does not depend on the secrecy of keys, i.e. signatures stay verifiable after their creation so that no secrecy assumptions (i.e. that some keys are secret during the verification) are needed.
- Trusted third parties are unable to forge clients' signatures, i.e. the signatures can be used for *non-repudiation.*
- Signatures are immune to quantum computational attacks, which means that we must avoid of using traditional digital signature schemes like RSA.

To achieve these goals, we use *server-based signatures* where all signatures are created with assistance of *signature servers*—third parties that on clients' requests create signatures in the name of clients. Instead of using traditional digital signature schemes that might be vulnerable to quantum attacks, we use the so-called hashing and publishing mechanism, where the server creates a Merkle hash tree [21] from the requests it receives and publishes the root hash of the tree as a trust anchor. The server authenticates the clients and adds their identities to the hash tree. Everyone who has an authentic copy of the published root hash and certain hash values in the tree can verify the signature. As hash functions are keyless, the integrity of such signatures does not depend on any secrets.

As in such a scheme, servers are able to create signatures in the name of their clients, the non-repudiation feature is not achieved. For still having the non-repudiation property, we have to include a *proof of authentication* to a signature, which can only be created in the presence of the client. One example of a proof of authentication is an RSA signature of the client but we have to avoid using RSA because of the goals of this paper—we have to avoid modular arithmetic based one-way functions because they are vulnerable to quantum attacks. Note also that one cannot use *message authentication codes* (MACs) for non-repudiation because the server and the client have copies of the same key and the server is able to create signatures in the name of its clients.

For that reason, we use a solution where clients are authenticated via one-time passwords that are generated by applying a one-way hash function iteratively to a secret random seed. Such an authentication technique was first proposed by Lamport [19]. The idea of using iterated hashing for non-repudiation was first proposed in [22] but their scheme is not suitble for creating independently verifiable compact signatures.

In this paper, we show how to use the technique proposed in [22] in combination with the idea of server-based signatures to create compact signatures without using any trapdoor functions. The key idea is to replace hash chains with the so-called *inverse hash calendars* that instead of the $O(\ell)$-time verification and $O(\ell)$-size signatures we have $O(\log \ell)$-size signatures with $O(\log \ell)$-time verification, where ℓ is the number of one-time passwords in the scheme. Based on Jakobsson's hash sequence traversal algorithm [17, 11], we construct a hash-calendar traversal algorithm that requires $O(\log^2 \ell)$ units of memory and $O(\log^2 \ell)$ time per one preimage. We also point out several advantages that the new signature solution has over the traditional solutions.

2 Preliminaries

2.1 Traditional Signature Solutions

In a traditional signature system, each user has a private key d, and a public key e which is made available to all potential verifiers. To sign a message m, the signer applies a signature function $\sigma = S(d, m)$. To verify a digital signature, one has to use a verification function $V(e, m, \sigma)$ which returns 1 if and only if the signature is correct.

For reliable distribution of public keys, a trusted *Certification Authority* (CA) is used. The CA also have a private key which is used to sign *public key certificates*—statements that bind public keys to identities of users. The certificate is mostly added to the signature for reliable verification of the key-identity relationship.

It is also foreseen that users may *revoke* their keys (certificates) in case the keys are suspected of being compromised. Hence, one also needs *time stamps* in order to prove that the signature was created before the key was revoked. Therefore, the PKI signature schemes also need a trusted *Time Stamping Authority* (TSA). Time-stamping is a mechanism to prove that certain data was created at (or before) certain time. In the context of signatures it helps to determine whether the key was valid (not revoked) at the time of creating the signature. In PKI, time-stamping service is provided by trusted third parties who just sign the hash of the data (sent by a client) together with current time-reading.

Actually, one needs even more trusted parties in the PKI-based signature solutions. Usually there are also the so-called *On-line Certificate Status Protocol* (OCSP) responders, whose duty is to confirm the validity of public key certificates. The OCSP-responders also use private keys to sign the validity statements. Also the OCSP responses must be reliably dated.

Due to the complicated structure of trusted third parties that are necessary in the traditional signature solutions, the structure of a signature is very complicated. It consists of the signature, the certificate, the time stamp of a signature, the OCSP-response, etc. Still, in spite of its complexity, the pki signatures do not provide a complete solution to the fundamental question of what happens if the keys of the trusted third parties become compromised. We need another layer of timestamps, OCSP-responses, etc. that are signed by higher-level trusted third parties, and so on. So, we have a chicken and egg problem. Moreover, there are some more fundamental threats for traditional digital signatures like the fast developing field of quantum computations.

2.2 Quantum Computing

Quantum computing was visioned in early 80-ties by Manin [20] and Feynman [13]. In 1997, Shor [24] presented the first quantum algorithm for factoring integers which means that if a quantum computer can be built in practice, many traditional cryptographic signing algorithms like RSA become totally insecure. Hash functions however seem to stay secure even in the presence of quantum computers. In 1996, Grover [14, 15] presented a quantum algorithm that is capable of searching a record from an N-element unsorted database in time $O(N^{1/2})$. In 1998, Brassard, Høyer and Tapp [3] presented an efficient collision-search algorithm which they claimed is capable of finding collisions for n-bit hash functions in time $2^{n/3}$, whereas in ordinary computers it would take $2^{n/2}$ steps instead. In practice, this would just mean that one has to use about 1.5 times larger hash functions. In 2009, Bernstein [2] called their practical implication into question and for now, the collision search on ordinary computers is at least as fast as it would be in quantum computers, which means that hash functions seem to be quantum-immune.

Traditional electronic signatures use trapdoor one-way functions such as RSA and they will be insecure for the *quantum age*.

2.3 Hash-Tree Digital Signature Schemes

Hash functions seem to stay secure and there exist signatures schemes that are entirely based on hash functions. The first such scheme was proposed in 1979 by Lamport [18]. The Lamport signature scheme is very inefficient because of signature size (a few hundred hash values) and the public key size (two hash values for every signed bit). In 1980, Merkle [21] proposes of using hash-trees in order to efficiently publish a large public key. Though the Lamport signature scheme and its more efficient modification by Merkle are secure against known quantum attacks, they still have a fundamental weakness related to secret keys—the secrecy of keys is a necessary assumption not only for the security of signing but also for the security of verification. If the key is compromised, all signatures become unreliable. So, the revocation problem still remains.

2.4 Data Signatures using Merkle Trees

Data signature is a mechanism to protect the integrity of data, i.e. to prevent unauthorized modification of data. The so-called hashing and publishing mechanism is used for that purpose. Data is hashed by using public and standard cryptographic one-way hash functions. The hash is published in widely witnessed ways (in newspapers etc.) Many hashes are published together by using a Merkle hash tree [21] (Fig. 1). The leaves ($x_1, \ldots, x_4$ in Fig. 1) of the tree are paired and hashed recursively to compute the root hash r of the tree which is published. A data signature (of a data record) is a proof that the data record took part of creating a global hash tree (for certain time unit). The proof is a *slice* of the global tree (a *hash chain*) that consists of the data that is absolutely necessary to re-compute the root of the global hash tree. For example, the time stamp for x_1 in Fig. 1 consists of x_2 (for re-computing r_{12}) and r_{34} (for recomputing r from r_{12}). Time stamps are compact, because in case of N leaves, their size is just $O(\log N)$.

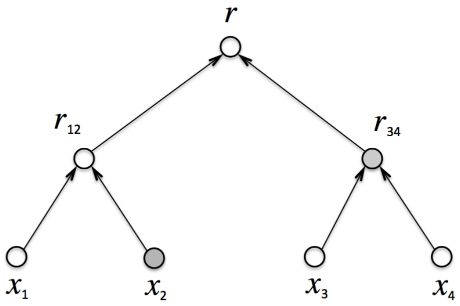

Fig. 1. Merkle hash tree.

Such a technique was first described in [16] and it has been proven [9, 5–7] that hash-tree schemes are secure against back-dating attacks, assuming collision-freeness and other security properties of hash functions.

2.5 Hash Calendar

Hash calendar [4] is a special kind of hash tree. At any given moment, the tree contains a leaf node for each second since 1970-01-01 00:00:00 UTC. The leaves are numbered left to right starting from zero and new leaves are always (every second) added to the right (Fig. 2). The internal nodes of the tree can only be computed if the corresponding predecessor nodes have been computed. For example at time 6 by UTC, we have three subtrees computed (Fig. 2, left): one for the leaves 0-3, one for 4-5 and a singleton tree for 6. To compute the published hash value at time 6 (in case this was the publishing time), a temporary hash value $x = h(x_{45}, x_6)$, where x_{45} is the root of the tree with leaves 4,5, and x_6 is the 6th leaf, and finally the published hash R is computed as $R = h(x_{03}, x)$, where x_{03} is the root hash of the tree with leaves 0,1,2,3. In this example, the calendar hash chain for the 5th leaf is depicted in Fig. 2 right. It hash a right link as the first link with sibling hash 4, a right link structure with sibling hash 6 as the second link, and right link structure with sibling x_{03} as the third and final link.

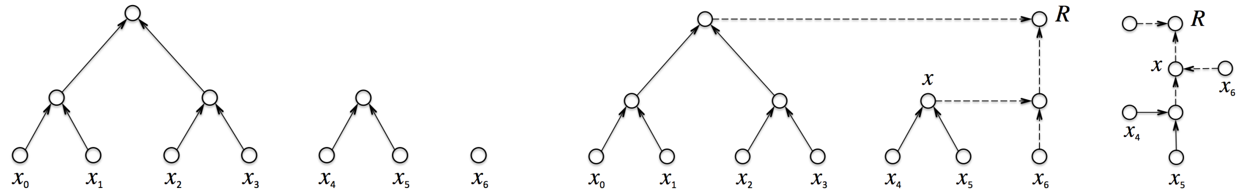

Fig. 2. Hash calendar at time 6 by UTC (left), publishing computations (the middle tree), and the calendar hash chain for the 5th UTC second.

Since the calendar tree is built in a deterministic manner, the shape of the tree for any moment can be reconstructed knowing just the number of leaf nodes in the tree at that moment, which is one more than the number of seconds from 1970-01-01 00:00:00 UTC to that moment. Therefore, given the time when the calendar tree was created and a hash chain extracted from it, the UTC time value that corresponds to the leaf node is uniquely determined.

Hash Calendar in action. A publicly available hash calendar can be used as an audit mechanism in order to prevent malicious behavior of the server. Real world's publishing mechanisms are usually not efficient enough to publish every hash in the calendar immediately after creation. So during the time period between creating a calendar hash and publishing it the server may try to reorder the calendar hash values. To prevent that happening, the server should reply promptly after creating a new calendar has so that the reply contains a unique representation on the whole previous calendar. For example, when creating x_6 (Fig. 2), the server has to reply with the two root hash values: the root of the subtree with leaves x_0, x_1, x_2, x_3 and the subtree with the leaves x_4, x_5. If the server still tries to reorder the calendar hash values, then it has to reply inconsistently to the clients, which will be detected sooner or later.

2.6 Signatures with Identity and Keyless Signatures' Infrastructure (KSI)

Data signature is not the only possible use of keyless hash-tree signatures. Keyless signatures may also be used as personal signatures or device signatures that bind the identity of the signer to the signed data. In this case of course signers need to be authenticated by the service and (at least to current knowledge) some sort of secrets for authentication would be necessary. Still, even if keys or passwords are used for authentication, the integrity and reliability of the signatures does not depend on the secrecy of any keys. During the creation of a personal keyless signature, the signer is *authenticated* by the service and *signers identity* is added to the global hash tree (together with the signed hash value).

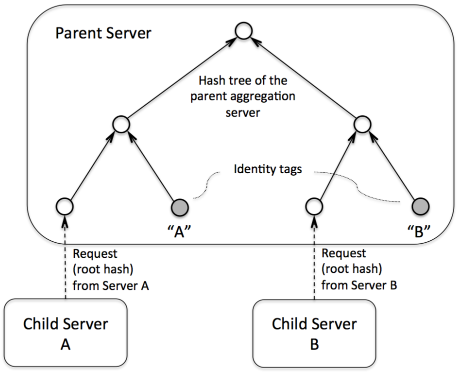
Parent Server
Hash tree of the parent aggregation server
Identity tags
"A"
"B"
Request (root hash) from Server A
Request (root hash) from Server B
Child Server A
Child Server B

Fig. 3. Identities in a keyless signature.

Note that for scalability reasons a hierarchy of *aggregation servers*, the so-called *Keyless Signatures' Infrastructure, KSI* [4], is needed to create the global hash tree. Such an infrastructure consists of a hierarchy of aggregation servers that, in co-operation, create the per-round global hash trees. First layer aggregation servers, so-called gateways, are responsible of collecting requests directly from clients, and every aggregation server receives requests from a set of lower level servers, hashes them together into a hash tree, and sends the root hash value of the tree as a request to higher-level servers. The server then waits for the response from a higher-level server and by combining the received response with suitable hash chains from its own hash tree responds to lower-level servers.

Parent servers also check the identity of child servers and add their identities into the hash trees together with the corresponding requests (Fig. 3). Having a keyless signature, it can be checked which servers in the global KSI actually created the signature.

2.7 Server-Based Signatures and Trust Issues

Keyless signatures must certainly be *server-based*, i.e. electronic signature solutions in which a publicly available server participates in the signature creation process. The conventional solutions based on public-key cryptography and public-key infrastructure assume that signers use their personal trusted computing bases for signing documents and signatures can be created off-line without any communication with servers. There are several reasons why server-based signatures are preferable to off-line solutions, such as: (1) to reduce the computational cost of creating digital signatures of ordinary users; (2) to reduce possible misuses of cryptographic keys by ordinary users; (3) to have better control over the number of forged signatures in case of malicious abuses of signature keys.

Many different forms of server-based signatures exist. For example, Lamport signatures [18] are server based, and also the so-called *on-line/off-line signatures* first proposed in 1989 by Even, Goldreich and Micali [12] 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 [19] 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 assumed to be Trusted Third Parties (TTPs) 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 [22] reduce the problems and costs related to individual private keys. In their solution, clients (ordinary users) delegate their private cryptographic operations to a Delegation Server (DS). Users authenticate to DS and request to sign messages on their behalf by using the server's own private key. The main motivation behind DS was that private keys are difficult for ordinary users to use and easy for attackers to abuse. Private keys are not memorable like passwords or derivable from persons like biometrics, and cannot be entered from keyboards like passwords. Private keys are mostly stored as files in computers or on smart-cards, that may be stolen.

Trust Issue. The main drawback in server-based signature solutions is that the server must be completely trusted. Still, it is somewhat unclear whether such a server-based signature system is less trustworthy than the traditional PKI based signature solutions, where the signature keys are held by end-users. In [8], "average persons" creating electronic signatures are compared to illiterate persons signing paper documents in the presence of a notary public. Indeed, the electronic devices for creating electronic signatures are far more complicated than pens used to sign paper documents. Having full control over the electronic signature technology is out of question for most users. Therefore, blind trust is inevitable in the electronic signature systems and it does not make much difference whether to trust your personal computer or a signature server. Considering the limited knowledge average persons have about elementary security procedures, server-based solutions are even preferable to the traditional ones.

2.8 Verifiable Client Authentication and Non-Repudiation

It is desirable to have a "trace of authentication" inside the signature that is verifiable and cannot be forged by the server (or anyone else). We also want the signatures dated to t be unforgeable at any later time $t' > t$ no matter if the secret authentication key is secure at t' or not.

Otherwise, the non-repudiation function of the signature is not covered. Clients may not accept the situation that signature servers are capable of signing in their names and must therefore be blindly trusted.

As we also want the signatures to be quantum-immune, we cannot use ordinary digital signature mechanisms (like RSA, DSA, etc.) as the traces of authentication. 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 [19] 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)

$$\begin{aligned} z_\ell &\leftarrow \{0,1\}^n \\ z_i &\leftarrow h(z_{i+1}) \quad \text{for } i \in \{\ell-1, \ell-2, \ldots, 0\} \end{aligned}$$

The last element $z_0 = h(z_1)$ 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 $z_0 = h(z_1)$. After the first session, the server already knows z_1 and hence it is possible for the server to check z_2 after it is used by the client (by the relationship $z_1 = h(z_2)$), etc.

Time-Dedicated Passwords The indices i may also be related with 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 is timestamped together, this may be considered as a signature of m. The signature is correct only if the date of the timestamp 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 $t_0 + i$, because the time stamps obtained later will not be suitable for later-created signatures (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 [23]. 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 in the use of digital signatures and if one wants to convert TESLA to autonomous digital signatures, their size is $O(\ell)$.

Efficient Hash Sequence Traversal Considering possible security problems in clients' computers, the password sequence could be maintained by a dedicated cryptographic hardware device that generates the random seed, computes the chain and then consequtively reveals the preimages. 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 (if only the seed is stored) to compute the next preimage. In was shown by Jakobsson et al [17, 11] that $O(\log \ell)$ memory and $O(\log \ell)$ time (per one pre-image) is sufficient.

3 New Keyless Signature Scheme

We show that if we use Keyless Signatures combined with Merkle hash tree type structures instead of hash chains in an authentication procedure analogous to hash chain authentication, then we can produce compact and independently verifiable signatures, which are quantum immune (at least against the known quantum attacks) and the integrity of which does not depend on the secrecy of keys.

3.1 Compactness of Signatures

In our solution, together with a one-time password z_i, we also add a hash chain to the signature, which proves that z_i participated in the computation of the public key z_0 and that z_i is in the right place in the hash tree. To verify the proof, one only needs $O(\log N)$ steps instead of $O(N)$ that would be needed if we use the standard TESLA authentication scheme.

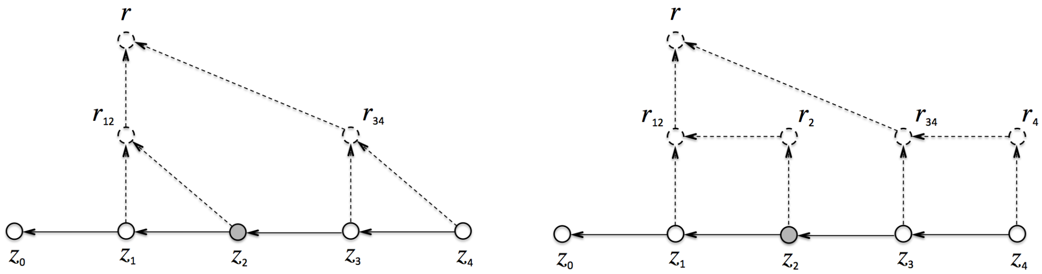

Fig. 4. Key hash-chain with Merkle tree (left) and a secure version of it (right).

Note however that such a hash tree must have a special form and shape to prevent premature disclosure of one-way passwords. if we just add a Merkle tree structure to the key hash chain like shown in Fig. 4 (left) then the hash chain for z_1 contains z_2 and hence z_2 would be prematurely disclosed. Thereby, we use a new type of tree structure (Fig. 4), where the keys $z_2, z_4, \ldots$ with even indices are replaced respectively with hash values $r_2 = h(z_2) = z_1$, $r_4 = h(z_4) = z_3$, etc.

3.2 Public Key Certificates

A client generates the random seed z_N and a key-hash chain $z_1, z_2, \ldots, z_N$ by using the recursive relation $z_{i-1} = h(z_i)$ (for all $i = 1 \ldots N$). Every hash value is a one-time password for a particular second of time. Client also computes the key hash-tree (Fig. 4) and its root hash r. Client's public key consists of $z_0 = h(z_1)$ and r.

A public key certificate for client consists of the following data:

- Identity ID of Client.
- The public key (z_0, r).
- Validity time t_0 after which 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.
- Identity and connection parameters of a keyless signature servers that are 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 that contains z_N with the corresponding hash chain sufficient to recompute r from z_N.

3.3 Signing a Document

To sign a document M at time $t > t_0$ (where $t = t_0 + i$), the Client:

- Computes a hash $m = h(M)$; and
- $x = h(m, z_i)$.

- Sends x together with client's identity ID as a request to the signature server.
- The server checks if 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), and sends S_t back to the client.

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

3.4 Verifying a Signature

To verify a signature $\langle \mathrm{ID}, i, z_i, c_i, S_t \rangle$ with a certificate $\langle \mathrm{ID}, z_0, r, t_0, \mathrm{ID}_s \rangle$ the following checks are made:

- 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.
- Extract time t from the timestamp S_t.
- $t = t_0 + i$, i.e. if a proper hash key was used.
- Extract server's identity from S_t and check whether it coincides with ID_s, i.e. if the server was authorized by the client to create the keyless signature.

3.5 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.

3.6 How to Achieve $t = t_0 + i$?

Signatures are considered valid only if $t = t_0 + i$, where t is the time indicated by the data signature S_t (time stamp) 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 befor sending the signature request to the server. Therefor, the equality $t = t_0 + i$ not necessarily holds.

The service can be organized so that the delay is predictable an is no more that a few seconds. Hence, 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.

4 Personal Signature Device

For better protecting the keys z_i, it might be reasonable to hold them in a hardware device that may resemble a memory stick that is connected to the computer with a USB connector. The hardware device has the following properties.

- It has an independent clock that cannot be externally adjusted, because that would be a security hole.
- It enables to use z_i on client's request but not 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.

As the ordinary quartz clocks may drift a few seconds per day, the yearly drift can be about 10 minutes. More precise clocks are too expensive or impossible to include into the personal signature device and therefore, the clock drift must be taken into account.

It is reasonable to consider two different types of personal signature devices: the light version and the strong version.

4.1 Signature Device: Light Version

The light version of the signature device, given as input an index i and a password p, checks if $t' > t_0 + i(1-\delta) - \varDelta$ and releases z_i and c_i, if the inequality holds. Here, t' means the internal clock-reading of the signature device.

The service we need for the light version of the signature device is the ordinary keyless signature service used only for time-stamping the requests, i.e. the time t in the time stamp S_t must be the standard UTC time. There is no need for the service to authenticate the client so we only need a pure data signature.

The downside of this solution is that malware in the client's computer may steal z_i and abuse it to forge client signatures. As there are no restrictions on the number of signatures that may be created, the number of potential forgeries is unlimited. Note also that as z_i is revealed to client's computer, malware is able to abuse it even after the personal signature device is disconnected from the computer.

4.2 Signature Device: Strong Version

In order to make the system more secure, we assume that the device and the server have a shared secret key K which is unknown to client and to anybody else. To sign a message M, the client hashes first the message $m = h(M)$ and sends the hash m and an index i to the device as a request. The device computes the request $x = h(m, z_i)$ and outputs x. After that, (x, ID) is sent to the server who proceeds as described above. Together with S_t, the server also sends to the client the Message Authentication Code (MAC) $\mu = \mathsf{MAC}_K(t)$ where K is the secret key. The client computer then sends MAC μ as a request into the personal signature device.

The device reveals z_i and c_i only if it has received $\mathsf{MAC}_K(t)$ with t that satisfies $t_0 + i < t$, i.e. after z_i has already expired. This guarantees, assuming that the service knows the correct time and there are no co-operating malware both in the server and in the client's computer, that the keys z_i are never exposed before they expire, even if the client does not verify hir/her own signatures before disclosing them to third parties. So, the signatures can only be created if the personal signature device is connected to the computer.

Note that the check $t' > t_0 + i(1-\delta) - \varDelta$ can be omitted, because x does not reveal useful information about z_i. This means that the signature device does not need an clock and a separate power supply. This means that the signature device is inexpensive and costs a few dollars.

5 Hash Calendar Traversal

Hash calendar traversal is similar to hash sequence traversal, except that in addition to consecutively revealing pre-images, the hash chains that prove the to the global root must be revealed. We use a similar traversal algorithm to that of Jakobsson [17] which uses $O(\log \ell)$ precomputed hash values (called *pebbles*) used as shortcuts for consecutive pre-image computation. By suitably placing the pebbles, the preimage computation time can be reduced from $O(\ell)$ to $O(\log \ell)$.

While in the original algorithm of Jakobsson a pebble consists of one hash value, we modify this algorithm 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 the new algorithm is $O(\log^2 \ell)$.

While moving a pebble to left, we also have to create a new hash chain for the moved pebble. We show that that this can be done with $O(\log \ell)$ hash computations. Hence, each traversing step takes $O(\log^2 \ell)$ hash steps. Note that our algorithm works for hash trees of any shape. It is very likely that for hash trees of special shape, there exist somewhat better algorithms.

5.1 Graph-Theoretical Background

Definition 1 (Proper Subtree). *By a proper subtree P of a binary tree T we mean a subset of the vertices of T such that there exists $r \in P$ (the root of P) such that P is the smallest set of vertices for which (1) $r \in P$; (2) if $v \in P$ and v_L, v_R are the left- and the right children of v, then $v_L, v_R \in P$.*

Definition 2 (Orderd Tree). *By an ordered tree T we mean a tree the leaves of which are ordered so that for any vertex w, if w_L and w_R are the left- and the right siblings of v respectively, and T_L and T_R are the proper subtrees with roots w_L and w_R, respectively, then $v_L < v_R$ for any leaves $v_L \in T_L$ and $v_R \in T_R$.*

Definition 3 (Left- and Right Subtrees). *For any leaf v of an ordered tree T, by the right subtree R_v of v we mean the maximum proper subtree with v as the first leaf. By the left subtree L_v of v we mean the maximum proper subtree of T that has v as the last leaf.*

Definition 4 (Chain). *By a chain from a vertex v to vertex r in a tree T we mean the set of all sibling vertices in the unique path from v to r (if such a path exists).*

Lemma 1. *If v is a leaf of an ordered tree and v' is its left neighbor, then the roots r' and r of L_v and R_v respectively then there is a vertex w for which r' and r are the left- and the right children, respectively.*

Proof. There is a vertex w that is the common "grandparent" of r' and r respectively. Let w_L and w_R be the left and the right children of w, respectively. If $w_L \neq r'$ or $w_R \neq r$, then there would exist leaves between v' and v, which is impossible. □

Definition 5 (Cuts). *By the* right cut *[*left cut*] of a leaf v of an (ordered) tree T we mean respectively the set of all right[left] siblings in the chain from v to the root of T.*

Note that the right cut of v always contains a chain from v to R_v and the left cut contains a chain from v to L_v.

Lemma 2 (Left Move of a Pebble). *Let v be a leaf of an ordered tree T and v' be its left neighbor. Let C be the right cut of v and r be the root of R_v. Then the set $C \cup \{r\}$ contains the right cut of v'.*

Proof. Let r' be the root of $L_{v'}$ and r be the root of R_v. By Lemma 1, there is a vertex w with left child r' and right child r (i.e. r' and r are siblings). As v' is the last leaf of $L_{v'}$, the hash chain from v' to r' consists only of left siblings. In w, the hash chains from v' and v (to the root of T) meet and will further be the same. As r is the first right sibling in the hash chain from v' to the root of T, and all other right siblings are already in C, we conclude, that $C \cup \{r\}$ contains the right cut of v'. □

Lemma 3 (Right Move of a Pebble). *Let v be a leaf of an ordered tree T and v' be its left neighbor. Let C' be the left cut of v' and r' be the root of $L_{v'}$. Then the set $C' \cup \{r'\}$ contains the left cut of v.*

Proof. Dual of Lemma 2. □

Corollary 1 (Moving Lead Pebble to Right). *Let v be a leaf of an ordered tree T and v' be its left neighbor. Let C' be the left cut of v' and C be the right cut of v. Let r' be the root of $L_{v'}$. Then the set $S = C' \cup C \cup \{r'\}$ contains a chain from v to the root of T.*

Proof. As S contains (as subsets) both the right cut C and the left cut of v (by Lemma 3), then S also contains the chain from v to the root of T. □

5.2 Traversal Algorithm

The new traversal algorithm extends the hash sequence traversal algorithms [17, 11] in the following way:

- We add the tree structure to hash (password) sequence.
- We add a new pebble, the so-called *lead pebble* that points to the hash value v' in the sequence that was previously released. The lead pebble contains the left cut $C^{-}_{v'}$ of v'.
- We extend every ordinary pebble at v with the right cut C^{+}_{v} of v. Every time the pebble moves to the left, its right cut is also updated (by Lemma 2).
- For releasing the next hash value (and the corresponding hash chain), the first pebble v (with C^{+}_{v}) has to be moved to the position next to the lead pebble v' (with $C^{-}_{v'}$), after which (by Corollary 1) the hash chain from v to the root hash is derived and released together with v. The lead pebble is moved to v.

Otherwise, the algorithm works like an ordinary hash sequence traversal algorithm. If we use the Jakobsson's algorithm for placing the pebbles, then we have $O(\log \ell)$ pebbles. In each step, we have to move some of these pebbles to the left by a constant number of steps. Hence, as in the new algorithm we also have to disclose the hash chains, each pebble contains $O(\log \ell)$ hash steps and hence, the computational cost of every step is measured by $O(\log^2 \ell)$. For example, if we issue certificates per one year with SHA-256 and we have a hash value per each second, the size of memory we need is about 20 K bytes.

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., Kroonmaa, A., Laanoja, R.: Keyless signatures infrastructure: How to build global distributed hash-trees. In: H. Riis Nielson and D. Gollmann (Eds.): NordSec 2013, LNCS 8208, pp. 313–320 (2013)
5. Buldas, A., Laur, S.: Knowledge-binding commitments with applications in time-stamping. In: PKC 2007, LNCS 4450, pp. 150–165 (2007)
6. Buldas, A., Laanoja, R.: Security proofs for hash tree time-stamping using hash functions with small output size. In: Boyd, C., Simpson, L. (eds.): ACISP 2013. LNCS 7959, pp. 235–250 (2013)
7. Buldas, A., Niitsoo, M.: Optimally tight security proofs for hash-then-publish time-stamping. In: ACISP 2010. LNCS 6168, pp. 318–335 (2010)
8. Buldas, A., Saarepera, M.: Electronic signature system with small number of private keys. In: 2nd Annual PKI Research Workshop, pp.96–108. NIST Gaithersburg MD, USA. April 28–29 (2003)
9. Buldas, A., Saarepera, M.: On provably secure time-stamping schemes. In: ASIACRYPT 2004, LNCS 3329, pp. 500–514 (2004)
10. 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)
11. Coppersmith, D., Jakobsson, M.: Almost optimal hash sequence traversal. In: Blaze, M. (ed.): FC 2002, LNCS 2357, pp. 102–119 (2003)
12. Even, S., Goldreich, O., Micali, S.: On-line/off-line digital signatures. J. Cryptology (1996) 9: 35–67.
13. Feynman, R.P.: Simulating physics with computers. International Journal of Theoretical Physics 21 (6): 467–488 (1982)
14. Grover L.K.: A fast quantum mechanical algorithm for database search, Proceedings, 28th Annual ACM Symposium on the Theory of Computing, p. 212 (1996)
15. Grover L.K.: From Schrödinger's equation to quantum search algorithm. American Journal of Physics 69(7): 769–777 (2001)
16. Haber, S., Stornetta, W.-S.: How to time-stamp a digital document. Journal of Cryptology 3(2), 99–111 (1991)
17. 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)
18. Lamport, L.: Constructing digital signatures from a one way function. Comp. Sci. Laboratory. SRI International (1979)
19. Lamport, L.: Password authentication with insecure communication. Comm. ACM (1981) 24(11): 770–772.
20. Manin, Yu. I.: Vychislimoe i nevychislimoe (Computable and Noncomputable) (in Russian). Sov.Radio. pp. 13–15 (1980)
21. Merkle, R.C.: Protocols for public-key cryptosystems. In: Proceedings of the 1980 IEEE Symposium on Security and Privacy, pp. 122–134 (1980)
22. 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)
23. Perring, A., Canetti, R., Tygar, J.D., Song, S.: The TESLA broadcast authentication protocol. In CryptoBytes, 5:2, pp. 2–13 (2002)
24. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, SIAM J. Comput. 26 (5): 1484–1509 (1997)