TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
11/2020

Hash-Based Server-Assisted Digital Signature Solutions

AHTO TRUU

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

The dissertation was accepted for the defence of the degree of Doctor of Philosophy (Computer Science) on 28/04/2020

Supervisor: Prof. Ahto Buldas
Department of Software Science
School of Information Technologies
Tallinn University of Technology
Tallinn, Estonia

Opponents: Ass. Prof. Andreas Hülsing
Eindhoven University of Technology
Eindhoven, Netherlands

Ass. Prof. Elena Andreeva
Technical University of Denmark
Lyngby, Denmark

Defence of the thesis: 28/05/2020, Tallinn

Declaration:
Hereby I declare that this thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology, has not been submitted for any academic degree elsewhere.

Ahto Truu

signature

European Union
European Regional
Development Fund

Investing
in your future

ISSN 2585-6898 (publication)
ISBN 978-9949-83-546-1 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9949-83-547-8 (PDF)

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
11/2020

Räsifunktsioonidel põhinevad serveri toega digitaalse signeerimise lahendused

AHTO TRUU

To Enok,
who introduced me to computing.

To Piret,
who has suffered the consequences.

Contents

List of Publications

The thesis is based on the following publications, referred to by Roman numbers.

Publication I: A. Buldas, R. Laanoja, and A. Truu. A server-assisted hash-based signature scheme. In *NordSec 2017*, *Proceedings*, volume 10674 of *LNCS*, pages 3–17. Springer, 2017

Publication II: A. Buldas, R. Laanoja, and A. Truu. A blockchain-assisted hash-based signature scheme. In *NordSec 2018*, *Proceedings*, volume 11252 of *LNCS*, pages 138–153. Springer, 2018

Publication III: A. Buldas, D. Firsov, R. Laanoja, H. Lakk, and A. Truu. A new approach to constructing digital signature schemes (short paper). In *IWSEC 2019*, *Proceedings*, volume 11689 of *LNCS*, pages 363–373. Springer, 2019

Publication IV: A. Buldas, A. Truu, R. Laanoja, and R. Gerhards. Efficient record-level keyless signatures for audit logs. In *NordSec 2014*, *Proceedings*, volume 8788 of *LNCS*, pages 149–164. Springer, 2014

Author’s Contributions to the Publications

Publication I: I was the lead author, contributing the primary idea and most of the writing.

Publication II: I contributed the authenticated data structure and the protocol for auditing the server’s behavior, and some of the writing.

Publication III: My contribution was technical work on the proofs of forward-resistance of the proposed tag schemes, and some of the writing.

Publication IV: I contributed most of the writing, the specifications of the aggregation and extraction algorithms, and also the technical specification not included in the academic paper, but underlying the reference implementation described in the paper.

1 Introduction

1.1 Background and Motivation

In any modern society, one would be hard pressed to find a field of activity where important decisions would not be made based on digital data, documents, or messages. It is therefore vital to verify that the information has not been tampered with, before relying on it in such a manner. Digital signatures are the core technology for ensuring integrity and authenticity of electronic information.

All the digital signature schemes in wide use today (RSA [71], DSA [40], ECDSA [52]) are known to be vulnerable to quantum attacks by Shor's algorithm [77]. While the best current experimental results are still toy-sized [59], it takes a long time for new cryptographic schemes to be accepted and deployed, so it is of considerable interest to look for post-quantum secure alternatives already now. These considerations triggered the Post-Quantum Cryptography project, announced by NIST in December 2016 [64].

Error-correcting codes, lattices, and multi-variate polynomials have been used as foundations for proposed replacement schemes [7]. However, these are relatively complex structures and new constructions in cryptography, so require significant additional scrutiny before gaining trust.

Hash functions, on the other hand, have been studied for decades and are widely believed to be quite resilient to quantum attacks. The best currently known quantum results against hash functions are using Grover's algorithm [46] to find a pre-image of a given k-bit value in $2^{k/2}$ queries instead of the 2^k queries needed by a classical attacker and its adaptation by Brassard *et al.* [10] to find a collision in $2^{k/3}$ instead of $2^{k/2}$ queries. To counter these attacks, it would be sufficient to deploy hash functions with correspondingly longer outputs when moving from pre-quantum to post-quantum setting. The crucial point is that the attack costs remain exponential even for quantum adversaries.

Another advantage of hash function based schemes is the minimization of assumptions. It has been shown that secure digital signatures can exist if and only if one-way and second pre-image resistant hash functions exist [74]. As the signature schemes we study in the following depend on the hash function having additional properties, they are formally not based on minimal assumptions. However, all real-life deployments that we are aware of implement the hash-then-sign model that relies on a collision resistant hash function in addition to a signature scheme. Therefore, it can be argued that in practice, collision resistance should also be counted among the minimal assumptions.

1.2 Approach and Summary of Results

In this work, we propose the BLT family of signature schemes (named after the initials of the inventors) built from hash functions as the sole underlying cryptographic primitive. We analyze most of the proposed schemes formally, providing security reductions to properties of hash functions.

The proposed schemes are really templates that can be instantiated with different hash functions. A benefit of this approach is that when the security of one hash function becomes insufficient (whether by advances in cryptoanalytic techniques or in computing power, or a combination of the two), we can replace the hash function with a stronger one and the security of the new instantiation of the signature scheme follows from the security of the replacement hash function.

We start by reviewing related work in Chapter 2 and then outlining the security definitions and properties of hash functions and hash trees in Chapter 3.

In Chapter 4 we describe the scheme we call BLT-TB (for "time-bound") that essentially consists of authenticating messages with one-time keys pre-bound to fixed time slots and proving correct usage of these keys by time-stamping the authenticators. We prove the scheme to be $2^{k/2-1}$-secure under the assumption that the hash function with k-bit outputs behaves like a random oracle. While this puts quite a high requirement on the hash function, the security level of the scheme is very close to the principal upper bound of $2^{k/2}$ common to all signature schemes able to sign messages of arbitrary length. The main drawback of the scheme is the need to pre-generate keys for all possible signing times, which can be prohibitive for constrained devices, such as smart cards.

In Chapter 5 we propose a way to reduce the key generation costs for personal devices that are used only occasionally. The BLT-BC scheme (for “blockchain”) makes the keys one-time and uses a blockchain-backed validator (perhaps implemented as a consensus-based cluster) to enforce the one-time property of the signing keys. Thus, the scheme trades savings on the client side against higher requirements on the supporting service, both in the trust placed on the service and also the computing resources required to operate it.

In Chapter 6 we propose another way to relax the requirement to pre-bind the keys to time by developing the concept of forward-resistant tags. We prove that the combination of a forward-resistant tag system and a backdating-resistant time-stamping service yields an unforgeable signature scheme. We then propose several forward-resistant tag systems, in particular some that allow dynamic binding of keys to time, and derive their security properties from those of the underlying hash function. Based on that, we then define the BLT-OT (for “one-time”) signature scheme where each key can be used just once, but at the time of keyholder’s choosing. The resulting scheme has competitive performance both as a one-time scheme and also as a component of a many-time scheme.

All these signature schemes depend on access to a time-stamping service and are thus inherently server-supported. On one hand this comes with the need to critically trust an external service for the security of the signatures and the restriction that signing is only possible on-line. On the other hand, most practical deployments of digital signatures in open systems need to track key revocations and are on-line even when the underlying signature scheme theoretically supports off-line signing and the server-supported nature of our schemes also provides a natural choke point for implementing instant key revocation and possibly enforcing other kinds of usage policies as well.

In Chapter 7 we apply hash function based techniques to efficient signing of audit logs. We propose a scheme where log records are signed in batches to reduce computational and storage overhead, but can be proven (and the proofs verified) individually, without leaking any information about the contents of other records in the log.

A limitation of the current work, in particular considering the motivation of quantum threats, is that the security proofs are done in classical setting. Formal analysis in quantum setting is hindered, among other difficulties, by the fact that our signature schemes depend on time-stamping and there is currently no well-defined notion of back-dating resistance of time-stamping in quantum setting. There is hope, however, that the collapsing property [82, 81, 35] of hash functions could be useful in moving to quantum setting.

2 Related Work

2.1 Hash-Based Signatures

The earliest signature scheme constructed from hash functions is due to Lamport [37]. His scheme, as well as the refinements proposed in [61, 42, 39, 12, 49], are *one-time*: they require generation of a new key pair and distribution of a new public key for each message to be signed.

Merkle [61] introduced the concept of *hash tree* which aggregates a (potentially large) number of inputs into a single hash value so that any of the inputs can be linked to it with a compact proof. This allowed combining N instances of a one-time signature scheme into an N-*time* scheme. This approach has been further studied in [34, 73, 36]. A common drawback of these constructs is that the whole tree has to be built at once.

In [62], Merkle proposed a method to grow the tree gradually as needed. However, to authenticate the lower nodes of the tree, a chain of one-time signatures (rather than of much smaller hash values) is needed, unless the scheme is used in an interactive environment and the recipient keeps the public keys already delivered as part of earlier signatures. This multi-level approach has subsequently been refined in [58, 11, 14, 13, 50, 51].

A complication with the N-time schemes is that they are *stateful*: as each of the one-time keys may be used only once, the signer has to keep track of them. If this information is lost (for example, when a previous state is restored from a backup or when multiple concurrent processes use the same key), key re-use may result in a catastrophic security loss. Perrig [66] proposed a *few-time* scheme where a private key can be used to sign several messages, and the security level decreases gradually with each additional use.

Bernstein *et al.* [8] combined the optimized few-time scheme of [70] with the multi-level tree of [13] to obtain a *stateless* scheme that uses keys based on a pseudo-random schedule, making the risk of re-use negligible even without tracking the state.

2.2 Server-Assisted and Interactive Signatures

In server-assisted schemes, signers can't produce signatures on their own, but have to co-operate with servers. The two main motivations for such schemes are: (a) performance: costly computations can be offloaded from an underpowered signing device (such as a smart card) to a more capable server; and (b) security: risks of key misuse can be reduced by either keeping the keys in a server environment (which can presumably be managed better than an end-user's personal computer) or by having the server perform additional checks as part of the signature generation protocol.

An obvious solution would be to just let the server handle the asymmetric-key operations based on requests from the signers [68]. In this case the server has to be completely trusted, but it's not immediately clear whether that is in fact less secure than letting the end-users manage their own keys [30].

To reduce the need to trust the server, Asokan *et al.* [5] proposed a method where asymmetric-key operations are performed by a server, but a user can prove the server's misbehavior when presented with a signature that the server created without the user's request. However, such signatures appear to be valid to a verifier until challenged by the signer. Thus, this protocol is usable in contexts where a dispute resolution process exists, but unsuitable for applications with immediate and irrevocable effects, such as authentication for access control purposes or committing transactions to append-only ledgers. This also applies to later improvements of the approach in [9, 45].

Several methods have been proposed for outsourcing the more expensive computation steps of specific signature algorithms, notably RSA, but most early schemes have subsequently been shown to be insecure. In recent years, due to increasing computational power of handheld devices and wider availability of hardware-accelerated implementations, attention has shifted to splitting keys between end-user devices and back-end servers to improve the security of the private keys [33, 18].

Interactive signature protocols were first considered by Anderson *et al.* [2]. They proposed a protocol, where, once bootstrapped, a message is preceded by publishing the hash of the message and each message is authenticated by accompanying it with a secret whose hash was published together with an earlier message. Although the verification is limited to a single party, the protocol is shown to be a signature scheme according to several definitions. Similar concept was used in [67] to authenticate parties who are constantly communicating with each other. Due to this, it has the same inflexibility of not supporting multiple independent verifiers.

2.3 Non-Repudiation and Cryptographic Time-Stamping

An important property of digital signatures as an alternative to hand-written ones [41] is non-repudiation, i.e. the possibility to use the signature as evidence against the signer. Solutions where trusted third parties are (technically) able to sign on behalf of their clients are not desirable for non-repudiation, because clients may use that argument to fraudulently call their signatures into question. Therefore, solutions where clients have personal signature devices are preferable to those relying entirely on trusted parties.

Another real-world complexity is key revocation. Without such capability clients may fraudulently claim that their private keys were stolen and someone else may create signatures in their name. With revocation tracking, signatures created before a key revocation event can be treated as valid, whereas signatures created afterwards can be considered invalid. Usually this is implemented using cryptographic time-stamping and certificate status distribution services. No matter the implementation details, this can't be done without online services, which means that most practical deployments of digital signatures are actually server-supported.

Cryptographic time-stamps prove that some data existed before a particular time. The proof can be a statement that the data hash existed at a given time, cryptographically signed by a trusted third party.

Haber and Stornetta [47] made the first steps towards trustless time-stamping with a scheme where each time-stamp would include some information from the immediately preceding one and a reference to the immediately succeeding one. Benaloh and de Mare [6] proposed to increase the efficiency of hash-linked time-stamping by operating in rounds, where messages to be time-stamped within one round would be combined into a hierarchical structure from which a compact proof of participation could be extracted for each message. The aggregation structures would then be linked into a linear chain. Buldas *et al.* [27, 26, 28] proposed a series of time-stamping schemes based on binary linking that allowed any two tokens to be ordered in time, even if they were issued within the same aggregation round.

The security of hash-then-publish time-stamping schemes has been proven in a very strong model where the service provider does not have to be trusted [31, 29, 20, 21], making them particularly suitable for our use-case. Note that it is possible to provide such a service efficiently and in global scale [19].

3 Preliminaries

3.1 Notation

In the following, $\{0,1\}^k$ denotes the set of k-bit strings and $\{0,1\}^*$ the set of bit-strings of arbitrary (finite) length.

We will write $h : \{0,1\}^* \to \{0,1\}^k$ to mean a general hash function mapping arbitrary-length inputs to fixed-length outputs and $h : \{0,1\}^m \to \{0,1\}^k$ to mean a hash function restricted to fixed-length inputs.

We will also write $h(x_1,x_2)$ or $h(x_1,x_2,\ldots,x_n)$ to mean the result of applying h to a bit-string encoding the pair (x_1,x_2) or the tuple $(x_1,x_2,\ldots,x_n)$, respectively.

We will use $x \overset{\$}{\leftarrow} D$ to mean that x is obtained by sampling from the distribution D (sampled uniformly randomly in case D is a set) and $y \leftarrow C(x)$ to mean that y is obtained as the output of (possibly probabilistic) computation C on input x.

We will use $\mathcal{A}^{C(\cdot)}$ to denote that $\mathcal{A}$ has oracle access to C. This means that $\mathcal{A}$ can make queries to the public interface of C, but has no access to implementation of C.

We will write $\Pr\big[E\big]$ for the probability of the event E happening and $\Pr\big[P : E\big]$ for the probability of the event E happening assuming the pre-condition P.

3.2 Security Framework

In the following sections, we analyze the security levels of the proposed signature schemes by relating them to the security properties of the underlying hash function.

As most security attacks are probabilistic, we will use a slight generalization of Luby's *time-success ratio* [56] to express the resilience of cryptographic schemes as the relationship of the probability that an attack will succeed to the amount of resources the adversary can spend on the attack:

Definition 1. *A cryptographic scheme is S-secure if no adversary can break the security properties of the scheme with probability greater than or equal to $\frac{\rho}{S}$, where ρ represents the computational resources available to the adversary.*

The resources represented by ρ are computation time and memory. The total resource budget of the adversary is $\rho = \alpha \cdot \mathrm{time} + \beta \cdot \mathrm{memory}$, where α and β are the costs of a unit of computation time and a unit of memory, respectively.

The notion of S-security is equivalent to the condition that every adversary with success probability δ and computational resources ρ has $\rho/\delta > S$. This means that the cost of an attack is measured by the ratio ρ/δ, which is motivated by the following thought experiment: if the benefit to be gained from a successful attack is γ, the attack is economically viable if the expected outcome $\delta \cdot \gamma$ exceeds the cost ρ, i.e. if $\gamma > \rho/\delta$. Hence, the ratio ρ/δ can be considered as a "market-driven" limit on the cost of the attack.

3.3 Hash Functions

In general, a hash function h maps arbitrary-sized input data to fixed-size output values:

$$h : \{0,1\}^* \to \{0,1\}^k .$$

Even though some actual hash functions technically are defined only for inputs up to a specific length, these limits are so high that for all practical purposes the functions can be considered unlimited.

Hash values are often used as representatives of data that are either too large or too confidential to be used directly. For example, in the hash-then-publish time-stamping model, a hash value of a document is published to establish evidence of the existence of the document without disclosing its contents. Likewise, in the hash-then-sign model, a document's hash value is signed instead of the document itself.

To facilitate such uses, cryptographic hash functions are expected to have several additional properties, such as one-wayness (it's infeasible to reconstruct the input from the output), second pre-image resistance (it's infeasible to change the input so that it still maps to the same output), and collision resistance (it's infeasible to find two distinct inputs mapping to the same output). These have received extensive formal treatment in [72, 3, 4], from which we summarize the minimum needed for our purposes.

Definition 2 (One-way function)**.** *A function $f : D \to R$ is S-secure one-way if every f-inverting adversary $\mathcal{A}$ using computational resources ρ has success probability*

$$\Pr\left[x \xleftarrow{\$} D,\ x' \leftarrow \mathcal{A}(f(x)) : f(x') = f(x)\right] < \frac{\rho}{S}\ .$$

The one-wayness property models a hash function's ability to keep the confidentiality of the inputs. There is a natural upper bound of 2^k on the one-wayness of a hash function with k-bit outputs: assuming uniform distribution of outputs, a randomly generated x' will map to the desired value with probability 2^{-k} and an adversary that can afford to generate 2^k random inputs and evaluate the function on all of them is expected to find a match. (If the function has non-uniform output distribution, the probability of the attacker finding a match grows faster, so 2^k is still an upper bound.)

Definition 3 (Second pre-image resistant function)**.** *A function $f : D \to R$ is S-secure second pre-image resistant if every adversary $\mathcal{A}$ using computational resources ρ to find a second pre-image of a given input has success probability*

$$\Pr\left[x \xleftarrow{\$} D,\ x' \leftarrow \mathcal{A}(x) : x' \neq x,\ f(x') = f(x)\right] < \frac{\rho}{S}\ .$$

The second pre-image resistance models the strength of the hash values as commitments secure against attackers trying to replace or modify the inputs after the commitment. In other words, one might say that the second pre-image resistance models a hash function's security against post-commitment attacks by second or third parties. The second pre-image resistance of a hash function with k-bit outputs is also bounded by 2^k.

Definition 4 (Collision resistant function)**.** *A function $f : D \to R$ sampled from a family $\mathcal{F}$ is S-secure collision resistant if every collision-finding adversary $\mathcal{A}$ using computational resources ρ has*

$$\Pr\left[f \xleftarrow{\$} \mathcal{F},\ (x_1,x_2) \leftarrow \mathcal{A}(f) : x_1 \neq x_2,\ f(x_1) = f(x_2)\right] < \frac{\rho}{S}\ .$$

The collision resistance models the strength of the hash values as commitments secure against malicious users trying to prepare multiple inputs that would all match the same committed output, or in other words, security against pre-commitment attacks by first parties. The upper bound for the collision resistance of a hash function with k-bit outputs is $2^{k/2}$: in a set of randomly generated inputs, the probability of any one pair forming a collision is 2^{-k}; as the number of pairs grows quadratically with the size of the set, an adversary that can generate $2^{k/2}$ inputs and check for duplicates in the set of corresponding outputs is expected to find a matching pair.

Definition 5 (Random oracle)**.** *A random oracle $\Omega : D \to R$ is a function picked uniformly randomly from all the functions $D \to R$.*

Hash functions are often viewed as random oracles to model the unpredictability of their outputs. A random oracle achieves the upper bounds for one-wayness, second pre-image resistance and collision resistance.

Definition 6 (Undetectable function)**.** *A function $f : D \to D$ is S-secure undetectable if every detecting adversary $\mathcal{A}$ using computational resources ρ has:*

$$\left|\Pr\left[x \overset{\$}{\leftarrow} D : \mathcal{A}(x) = 1\right] - \Pr\left[x \overset{\$}{\leftarrow} D : \mathcal{A}(f(x)) = 1\right]\right| < \frac{\rho}{S} .$$

This property is essentially the one-shot version of the unpredictability of random oracles. We will also need to rely on this property for iterated hash functions:

Lemma 1. *If $f : D \to D$ is S-secure undetectable, then $f^n = \overbrace{f \circ f \circ \ldots \circ f}^{n \text{ times}}$ is at least $\frac{S}{n}$-secure undetectable.*

The proof is given in [15] which is an extended version of Publication III.

3.4 Hash Trees and Hash Chains

Introduced by Merkle [61], a hash tree is a tree-shaped data structure built using a 2-to-1 hash function $h : \{0,1\}^{2k} \to \{0,1\}^k$. The nodes of the tree contain k-bit values. Each node is either a leaf with no children or an internal node with two children. The value x of an internal node is computed as $x \leftarrow h(x_l, x_r)$, where x_l and x_r are the values of the left and right child, respectively. There is one root node that is not a child of any node. We will use $r \leftarrow T^h(x_1, \ldots, x_N)$ to denote a hash tree whose N leaves contain the values $x_1, \ldots, x_N$ and whose root node contains r.

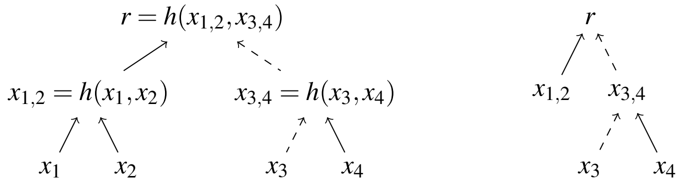

Figure 1: The hash tree $T^h(x_1, \ldots, x_4)$ and the corresponding hash chain $x_3 \rightsquigarrow r$.

In order to prove that a value x_i participated in the computation of the root hash r, it is sufficient to present values of all the sibling nodes on the unique path from x_i to the root in the tree. For example, to claim that x_3 belongs to the tree shown on the left in Figure 1, one has to present the values x_4 and $x_{1,2}$ to enable the verifier to compute $x_{3,4} \leftarrow h(x_3, x_4)$, $r \leftarrow h(x_{1,2}, x_{3,4})$, essentially re-building a slice of the tree, as shown on the right in Figure 1. We will use $x \overset{c}{\rightsquigarrow} r$ to denote that the hash chain c links x to r in such a manner.

Intuitively, it seems obvious that if the function h is collision-resistant, the existence of such a chain whose output equals the original r is a strong indication that x was indeed the original input. However, this result was not formally proven until 25 years after the hash tree construct was proposed [31, 34].

A binary tree with N leaves has $N-1$ internal nodes, so it takes $N-1$ evaluations of the hash function to compute the root hash value of the tree. In a balanced binary tree, the distance of any leaf from the root is $\log_2 N$, thus $\log_2 N$ hash function evaluations are needed to verify a hash chain extracted from such a tree.

The cost of extracting hash chains depends on how much of the tree is cached. If the whole tree is kept (causing a storage overhead of $N-1$ hash values in addition to the leaves), the hash chain for any leaf can be extracted with no additional hash function evaluations. If nothing is kept, the whole tree has to be re-populated, at the cost of $N-1$ hash function evaluations.

Various trade-offs between these extremes are possible. For example, if the internal nodes halfway between the root and the leaves are kept (an overhead of approximately $\sqrt{N}$ fixed hash values), the hash chain for any leaf can be extracted by re-populating two sub-trees at the total cost of $2\sqrt{N}$ hash function evaluations.

Assuming the chains are extracted consecutively for all leaves in the left-to-right order, there are several more efficient dynamic algorithms, such as the one by Szydlo [79] that keeps only $3\log_2 N$ nodes of the tree and spends $2\log_2 N$ hash function evaluations per chain extraction.

3.5 Hash-Then-Publish Time-Stamping

Following Buldas and Saarepera [31], we model the hash-and-publish time-stamping service as consisting of a repository $\mathcal{R}$ and an aggregation layer S (Figure 2). We consider the repository $\mathcal{R}$ to be an ideal object that works as follows:

- The time t is initialized to 1, and all the cells $\mathcal{R}_i$ to $\perp$.
- The query $\mathcal{R}.\texttt{time}$ is answered with the current value of t.
- The query $\mathcal{R}.\texttt{get}(t)$ is answered with R_t.
- On the request $\mathcal{R}.\texttt{put}(x)$, first $\mathcal{R}_t \leftarrow x$ is assigned and then $t \leftarrow t+1$.

The aggregation layer S operates in fixed-duration rounds. During each round, S collects client requests. At the end of the round, S aggregates the received requests $x_1,\ldots,x_N$ into a hash tree $r \leftarrow T^h(x_1,\ldots,x_N)$, queries t via a call to $\mathcal{R}.\texttt{time}$, commits the root r of the hash tree via $\mathcal{R}.\texttt{put}(r)$, and finally returns to each client the hash chain a linking that client's input x to the committed root r.

$$\begin{tikzcd}
\boxed{S} \arrow[r, shift right, "r_t"'] \arrow[d, shift left, "(t, x \overset{a_t}{\rightsquigarrow} r_t)"] & \boxed{\mathcal{R}} \arrow[l, shift right, "t"'] \arrow[d, dashed, "r_t"'] \\
\boxed{C} \arrow[u, shift left, "x"] \arrow[r, "(x,t,a_t)"] & \boxed{V}
\end{tikzcd}$$

Figure 2: Interactions in the hash-then-publish time-stamping between the client C, the aggregation service S, the ideal repository $\mathcal{R}$, and the verifier V.

A verifier V, receiving an input x and a time-stamp (t,a_t), first obtains r_t by querying $\mathcal{R}.\texttt{get}(t)$ and then checks that the hash chain a_t links the input x to r_t.

To simplicfy presentation, we count time in aggregation rounds of the time-stamping service and use the expression "at time t" to mean "during aggregation round t".

The security of such schemes can be proven in a model where only the repository $\mathcal{R}$ is assumed to operate correctly and the service S does not have to be trusted. We refer to [31, 29, 20, 21] for detailed analyses, including the requirements on the hash function h used by S in the general setting, and list our assumptions case by case in the proofs in the following sections.

3.6 Digital Signatures

A signature scheme consists of a triple $(\mathrm{Gen}, \mathrm{Sig}, \mathrm{Ver})$ of algorithms, where:

- Gen is a probabilistic key-generation algorithm that, given as input a security parameter k, produces a secret key sk and a public key pk.
- Sig is a signature-generation algorithm that, given as input the secret key sk and a message m, produces a signature $\sigma \leftarrow \mathrm{Sig}(\mathsf{sk}, m)$.
- Ver is a verification algorithm that, given as input a signature σ, a message m, and the public key pk, returns either 0 or 1, such that

$$\mathrm{Ver}(\mathrm{Sig}(\mathsf{sk}, m), m, \mathsf{pk}) = 1$$

  whenever $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathrm{Gen}(1^k)$.

Goldwasser *et al.* [44] proposed a framework for studying security of signature schemes where the attackers have various levels of access to signing oracles and various requirements on what they need to achieve for the attack to be considered successful (and the scheme broken). As the highest security level, they defined the concept of *existential unforgeability* where an attacker should be unable to forge signatures on any messages, even nonsensical ones.

They also defined the *chosen-message attack* where the attacker can submit a number of messages to be signed by the signing oracle before having to come up with a forged signature on a new message, and in particular, as the one giving the attacker the most power, the *adaptive chosen-message attack* where the attacker will receive each signature immediately after submitting the message and can use any information gained from previous signatures to form subsequent messages.

Definition 7 (Existential unforgeability)**.** *A signature scheme is S-secure existentially unforgeable against adaptive chosen-message attacks if any adversary using computational resources ρ, having access to a signer's public key p and to a signing oracle $\mathcal{S}$ to obtain signatures $\sigma_1 \leftarrow \mathcal{S}(m_1), \ldots, \sigma_n \leftarrow \mathcal{S}(m_n)$ on adaptively chosen messages $m_1, \ldots, m_n$, can produce a new message-signature pair (m, σ) such that $m \notin \{m_1, \ldots, m_n\}$, but σ is a valid signature on m, with probability less than ρ/S.*

4 Time-Stamped Scheme with Time-Bound Keys

We now have the foundations to start describing and analyzing our proposed new signature schemes.

The principal idea of our first scheme is to have the signer commit to a sequence of keys so that each key is assigned a time slot when it can be used to sign messages and will transition from a signing key to a verification key once the time slot has passed.

Signing itself then consists of time-stamping the message-key pair in order to prove that the signing operation was performed at the correct time.

4.1 Description of the Scheme

More formally, the procedures for key generation, signing, and verification are as follows.

Key Generation. To prepare to sign messages at times $1, \ldots, T$, the signer:

1. Generates T unpredictable k-bit values as signing keys: $(z_1, \ldots, z_T) \leftarrow \mathcal{G}(T, 1^k)$.
2. Binds each key to its time slot: $x_i \leftarrow h(i, z_i)$ for $i \in \{1, \ldots, T\}$.
3. Aggregates the key bindings into a hash tree: $p \leftarrow T^h(x_1, \ldots, x_T)$.
4. Publishes the root hash p as the public key.

The resulting data structure is shown in Figure 3 and its purpose is to be able to extract hash chains c_i such that $h(i, z_i) \overset{c_i}{\rightsquigarrow} p$ for $i \in \{1, \ldots, T\}$.

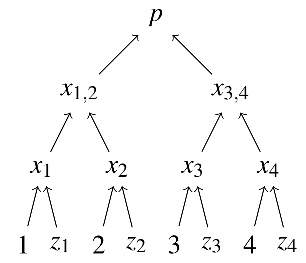

Figure 3: Computation of the public key for $T = 4$: $z_1, \ldots, z_4$ are the private keys.

Signing. To sign message m at time t, the signer:

1. Uses the appropriate key to authenticate the message: $y \leftarrow h(m, z_t)$.
2. Time-stamps the authenticator: sends y to the time-stamping service and receives in response a_t such that $y \overset{a_t}{\rightsquigarrow} r_t$, where r_t is the root hash of the aggregation tree built by the time-stamping service for the aggregation round t. We assume the root hash is committed to in some reliable way, such as broadcasting it to all interested parties, but place no other trust in the service.
3. Outputs the tuple (t, z_t, c_t, a_t), where t is the signing time, z_t is the signing key for time slot t, c_t is the hash chain linking the binding (t, z_t) to the signer's public key p, and a_t is the hash chain from the time-stamping service linking the message-key pair (m, z_t) to the round commitment r_t.

Note that the signature is composed and emitted after the time-stamping step, which makes it safe for the signer to release the key z_t as part of the signature: the time-stamping aggregation round t has ended and any future uses of the key z_t can no longer be stamped with time t.

Verification. To verify that the message m and the signature $s = (t, z, c, a)$ match the public key p, the verifier:

1. Checks that z was committed as signing key for time t: $h(t,z) \overset{c}{\rightsquigarrow} p$.
2. Checks that m was authenticated with key z at time t: $h(m,z) \overset{a}{\rightsquigarrow} r_t$.

4.2 Security of the Scheme

To formalize our security assumptions, we introduce three oracles:

- We model the publishing of the root hashes of the time-stamping aggregation trees as the oracle $\mathcal{R}$ that allows each r_t to be published just once.
- The signing oracle $\mathcal{S}$ will compute the message authenticators at any time, but will release only the keys that have already expired for signing (transitioned to verification keys).
- We model the hash function h as a random oracle using the *lazy sampling* technique: every time h is queried with a previously unseen input, a new return value is generated by uniform random sampling from $\{0,1\}^k$; when h is queried with a previously seen input, the same value is returned as the first time.

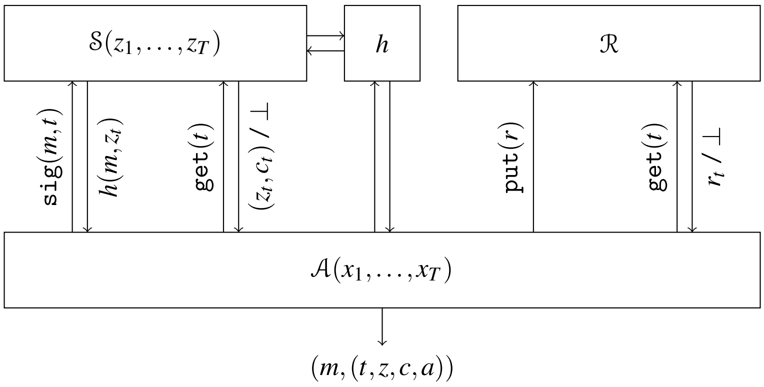

Figure 4: The adversary's interactions with the oracles: $\mathcal{S}$ is the signing oracle producing message authenticators and releasing private keys no longer valid for signing; h is the hash function modeled as a random oracle; $\mathcal{R}$ is the repository of round commitments of the time-stamping service.

The adversary $\mathcal{A}$ will be interacting with the oracles as shown in Figure 4 with the goal of producing a forgery.

To model the fact that the signer needs to keep secret only the keys $z_1,\ldots,z_T$, we explicitly initialize the adversary with $x_1,\ldots,x_T$. Note that the verification rule still assumes that the verifier has access only to the signer's public key p, which means the adversary is not limited to presenting hash chains that were actually extracted from $T^h(x_1,\ldots,x_T)$.

Also note that we leave the aggregation process of the time-stamping service fully under the adversary's control; only the repository $\mathcal{R}$ needs to be trusted to operate correctly.

As normally signing message m in our scheme involves first calling $\mathcal{S}.\texttt{sig}(m,t)$, then committing to $\mathcal{R}$ the root of a hash tree that includes the return value, and then calling $\mathcal{S}.\texttt{get}(t)$, we formalize the forgery condition by demanding that the adversary can't make the two $\mathcal{S}$ calls in that order:

Definition 8 (Forgery of a server-assisted signature). *The pair (m, σ) produced by an adversary is a successful forgery if σ is a valid signature on m, but the adversary did not make the calls $\mathcal{S}.\texttt{sig}(m,t)$, $\mathcal{S}.\texttt{get}(t)$, in that order, for any $t \in \{1, \dots, T\}$.*

Theorem 1. *Our signature scheme, when instantiated with a hash function $h : \{0,1\}^{2k} \to \{0,1\}^k$ modeled as a random oracle, is at least $2^{k/2-1}$-secure existentially unforgeable against adaptive chosen-message attacks.*

The proof, which is given in Publication I, is a hybrid argument by case analysis: we first assume that the adversary does not call $\mathcal{S}.\texttt{get}(t)$ and establish an upper bound on the probability that it could succeed in creating a forgery; we then show that an adversary that does call $\mathcal{S}.\texttt{get}(t)$ would not gain any advantage from calling $\mathcal{S}.\texttt{sig}(m,t)$ after that; we then assume as an alternative case that the adversary indeed did not call $\mathcal{S}.\texttt{sig}(m,t)$ and establish an upper bound on the probability of a successful forgery in this case; finally, we obtain our result as an upper bound on the sum of these two success probabilities.

4.3 Implementation Considerations

Key Generation. In the description of the scheme we assumed that the signing keys $z_1, \dots, z_T$ are unpredictable values drawn from $\{0,1\}^k$, but left unspecified how they might be generated in practice.

Obviously they could be generated as independent truly random values, but this would be rather expensive and also would necessitate keeping a large number of secret values over a long time. It would be more practical to generate them pseudo-randomly from a single random seed s. There are several known ways of doing that:

- Iterated hashing: $z_T \leftarrow s$, $z_{i-1} \leftarrow h(z_i)$ for $i \in \{2, \dots, T\}$.

  This idea of generating a sequence of one-time keys from a single seed is due to Lamport [55] and has also been used in the TESLA protocol [67]. Implemented this way, our scheme would also bear some resemblance to the Guy Fawkes protocol [2].

  Note that the keys have to be generated in reverse order, otherwise the earlier keys released as signature components could be used to derive the later ones that are still valid for signing. To be able to use the keys in the direct order, the signer would have to either remember them all, re-compute half of the sequence on average, or implement a traversal algorithm such as the one proposed by Schoenmakers [76].

- Counter hashing: $z_i \leftarrow h(s,i)$ for $i \in \{1, \dots, T\}$.

  With a hash function behaving as a random oracle, this scheme would generate keys indistinguishable from truly random values, but we are not aware of any strong results on the security of practical hash functions when used in this mode.

- Counter encryption: $z_i \leftarrow E_s(i)$ for $i \in \{1, \dots, T\}$.

  The signing keys are generated by encrypting their indices with a symmetric block cipher using the seed as the encryption key. This is equivalent to using the cipher in the counter mode as first proposed by Diffie and Hellman [38]. The security of this mode is extensively studied for all practical block ciphers. Another benefit of this approach is that it can be implemented using standard hardware security modules where the seed is kept in protected storage and the encryption operations are performed in a security-hardened environment.

Time-Stamping. As already mentioned, we side-step the key state management problems [60] common for most N-time signing schemes by making the signing keys not one-time, but *time-bound* instead. This in turn raises the issue of clock synchronization.

We first note that even when the signer's local clock is running fast, premature key release is easy to prevent by having the signer verify the time-stamp on $h(m, z_t)$ before releasing z_t.

The next issue is that the signer needs to select the key z_t before computing $h(m, z_t)$ and submitting it to time-stamping. If, due to clock drift or network latency, the time in the time-stamp received does not match t, the signature can't be composed. To counter clock drift and stable latency, the signer can first time-stamp a dummy value and use the result to compare its local clock to that of the time-stamping service.

To counter network jitter, the signer can compute the message authenticators $h(m, z_{t'})$ for several consecutive values of t', submit all of them in parallel, and compose the signature using the components whose t' matches the time t in the time-stamps received. Buldas *et al.* [19] have shown that with careful scheduling the latency can be made stable enough for this strategy even in an aggregation network with world-wide scale.

Finally, we note that time-stamping services operating in discrete aggregation rounds are particularly well suited for use in our scheme, as they only return time-stamps once the round is closed, thus eliminating the risk that a fast adversary could still manage to acquire a suitable time-stamp after the signer has released a key.

4.4 Discussion

Performance. In the following estimates, we assume the use of SHA-256 [63], a common 256-bit hash function. On small inputs, a moderate CPU core can perform about a million SHA-256 evaluations per second.[1] We also assume a signing key sequence containing one key per second for a year, for a total of a bit less than 32 million, or roughly 2^{25} keys.

Using the techniques described above, generation of T signing keys takes T applications of either a hash function or a symmetric block cipher. Aggregating them into a public key takes $2N - 1$ hashing operations. Thus, the key generation in our example takes about 100 seconds on a single core (and is well parallelizable if either of the counter-based generator mechanisms is used).

The resulting public key consists of just one hash value. In the private key, only the seed s has to be kept secret. The signing keys $z_1, \ldots, z_T$ can be erased once the public key has been computed, and then re-generated as needed for signing.

The hash tree $T^h(x_1, \ldots, x_T)$ presents a space-time trade-off. It may be kept (in regular unprotected storage, as it contains no sensitive information), taking up $2N - 1$ nodes, or about 1 GB, and then the key authentication hash chains can be just read from the tree with no additional computations needed. Alternatively, one can use a hash tree traversal algorithm, such as the one proposed by Szydlo [79], to keep only $3 \log_2 N$ nodes of the tree and spend $2 \log_2 N$ hash function evaluations per chain extraction, assuming all chains are extracted consecutively.

The size of the signature (t, z_t, c_t, a_t) is dominated by the two hash chains. The key authentication chain consists of $\log_2 N$ hash values, for a total of about 800 B for our 1-year key sequence. The time-stamping chain consists of $\log_2 M$ hash values, where M is the number of requests received by the time-stamping service in the round t. Assuming the use of the KSI service described in [19] operating at its full capacity of 2^{50} requests per round, this adds about $1\,600$ B. Thus we can expect signatures of less than 3 kB.

[1]As reported by OpenSSL 1.0.2 speed test on a laptop with 2.3 GHz Intel Core i5 CPU.

As the verification means re-computation of the hash chains, it amounts to less than a hundred hash function evaluations.

We will compare the performance of this scheme to the state of the art in Section 6.3.

Security Model. The security model of our scheme is markedly different from that of the "standalone" schemes. The correct operation of the repository $\mathcal{R}$ backing the time-stamping service is critical for the unforgeability of the signatures. An adversary that can tamper with the contents of $\mathcal{R}$ could back-date time-stamps and thus reuse keys already released as components of legitimate signatures to create forgeries.

There are practical ways to mitigate this risk. For example, the repository could be based on a data structure where each record includes a hash value of the preceding one and new records could be added by consensus among multiple independently operated nodes, essentially implementing a distributed robust public transaction ledger [43].

This, however, poses additional engineering challenges when deploying the scheme in practice.

Also the verifiers of signatures need to be able to authenticate the responses received from the repository when they query the round commitments.

5 Blockchain-Backed Scheme with One-Time Keys

The signing keys in the scheme proposed in the previous chapter are really not one-time, but rather time-bound: every key can be used for signing only within a specific time interval. For that reason, we will refer to the scheme as BLT-TB (for “time-bound“) in the following.

The architecture of the BLT-TB scheme can be modeled as interactions between the following parties (Figure 5, left):

- The signer who uses trusted functionality in secure device D to manage private keys.
- Server S that aggregates key usage events from multiple signers in fixed-length rounds and posts the summaries to append-only repository $\mathcal{R}$.
- Verifier V who can verify signatures against the signer's public key p and the round summaries r_t obtained from the repository.

Note that S and $\mathcal{R}$ together implement a hash-then-publish time-stamping service where neither the signer nor the verifier needs to trust S; only $\mathcal{R}$ has to operate correctly for the scheme to be secure.

5.1 Design of the Scheme

One-Time Keys. The design of BLT-TB incurs quite a large overhead as keys must be pre-generated even for time periods when no signatures are created. To avoid this inherent inefficiency, we now propose to spend the keys sequentially, one-by-one, as needed.

As the first idea, we could have the signer time-stamp each signature, just as in BLT-TB. In case of a dispute, the signature with the earlier time-stamp would win and the later one would be considered a forgery. This would obviously make verification very difficult, but more importantly would give the signer a way to deny any signature: before signing a document d with a key z, the signer could use the same key to privately sign some dummy value x; when later demanded to honor the signature on d, the signer could show the signature on x and declare the signature on d a forgery.

To prevent this, we assign every signer to a designated server which allows each key to be used only once. A trivial solution would be to just trust the server to behave correctly. This would still not achieve non-repudiation, as the server could collect spent keys and create valid-looking signatures on behalf of the signer.

Validating the Server's Behavior. If either the signer or the server published all signing events, including the key index for each one, then the server could not reuse keys and would not have to be treated as a trusted component. This would be quite inefficient, though, because of the amount of data that would have to be distributed and processed during verification, and would also leak information about the signer's behavior.

To avoid publishing all transactions, we use spent key counters both at the signer and the server side and an authenticated data structure for validating that the server ever only increases the counter values.

If proofs of correct operation were included in signatures, verifiers could reject signatures without valid proofs. This approach would have quite large overhead, however, as the verifiers would have to be able to validate the counters throughout their entire lifetime. Other parties who could perform such validation are the repository, the signers, or independent auditors. Both signers and auditors could only discover a forgery after the fact, not early enough to avoid creation of forged signatures.

A promising idea is to validate the server's correct operation by the repository itself. We require the server to provide a proof of correctness with each update to the repository.

The repository accepts the update only after validating the proof and then broadcasts the accepted root hashes. Because signatures are verified based on published root hashes in the repository, forgery by temporarily decrementing key usage counters is prevented.

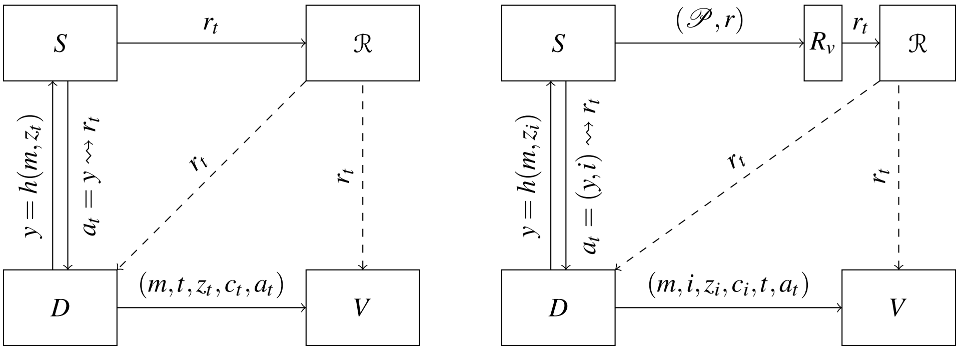

Figure 5: Interactions in BLT-TB (left) and BLT-BC (right): the signer uses the trusted device D to manage private keys; the server S aggregates client requests and posts round commitments to the repository $\mathcal{R}$, which both the signing device D and the verifier V query for the commitments; in BLT-BC, the repository has an additional validation layer R_v checking the proofs of correct operation accompanying the commitments before accepting them for inclusion in the repository.

5.2 Description of the Scheme

Our proposed new scheme, which we will refer to as BLT-BC (for "blockchain"), consists of the following parties (Figure 5, right):

- The signer uses trusted device D to generate keys and then sign data, as in BLT-TB.
- The server S assists signers in generating signatures. S keeps a counter of spent keys for each signer and sends updates to the repository.
- The repository performs two tasks. The validation layer R_v verifies the correctness of each operation of S before accepting it and periodically commits the summary of current state to a public append-only storage layer $\mathcal{R}$.
- The verifier V is a relying party who verifies signatures, as in BLT-TB.

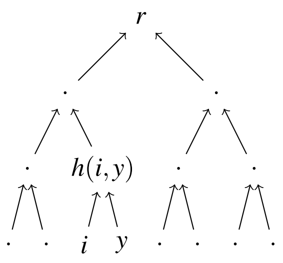

Figure 6: Server tree, showing the last key index i and the corresponding message authenticator y of the second client only.

The server maintains a hash tree with a dedicated leaf for each client (Figure 6). The value of the leaf is computed by hashing the pair (i, y) where i is the index of the last spent key and y is the last message received from the client (as detailed in **Signing** below).

Each public key must verifiably have just one leaf assigned to it. Otherwise, the server could set up multiple parallel counters for a client, increment only one of them in response to client requests, and use the others for forging signatures with keys the signer has already used and released.

One way to achieve that would be to have the server return the *shape* (that is, the directions to move to either the left or the right child on each step) of the path from the root of the tree to the assigned leaf when the client registers for service, and the client to include that shape when distributing its public key to verifiers. Another option would be to use the bits of the public key itself as the shape. Because most possible bit sequences are not actually used as keys, the hash tree would be a *sparse* one in this case.

Setup: Signer. To prepare to sign up to N messages, the signer:

1. Generates N unpredictable k-bit signing keys: $(z_1, \ldots, z_N) \leftarrow \mathcal{G}(N, 1^k)$.
2. Binds each key to its sequence number: $x_i \leftarrow h(i, z_i)$ for $i \in \{1, \ldots, N\}$.
3. Aggregates the key bindings into a hash tree: $p \leftarrow T^h(x_1, \ldots, x_N)$.
4. Registers the public key p with the server S.

The data structure giving the public key is similar to the one in the BLT-TB scheme (Figure 3), and also has the same purpose: to be able to extract the hash chains c_i linking the private key bindings to the public key: $h(i, z_i) \overset{c_i}{\rightsquigarrow} p$ for $i \in \{1, \ldots, N\}$.

Setup: Server. Upon receiving registration request from a signer, the server dedicates a leaf in its tree and sets i to 0 and y to an arbitrary value in that leaf.

Signing: Signer. Each signer keeps the index i of the next unused key z_i in its state. To sign message m, the signer:

1. Uses the current key to authenticate the message: $y \leftarrow h(m, z_i)$.
2. Sends the authenticator y to the server.
3. Waits for the server to return the hash chain a_t linking the pair (i, y) to the new published summary r_t: $h(i, y) \overset{a_t}{\rightsquigarrow} r_t$.
4. Checks that the shape of the received hash chain is correct and its output value matches the authentic r_t acquired directly from the repository.
5. If validation succeeds then outputs the tuple (i, z_i, c_i, t, a_t), where i is the key index, z_i is the i-th signing key, c_i is the hash chain linking the binding (i, z_i) to the public key p, and a_t is the hash chain linking (i, y) to the published r_t.
6. Increments its key counter: $i \leftarrow i + 1$.

Signing: Server. Upon receiving request y' from a signer, the server:

1. Extracts the hash chain a linking the current state of the client record (i, y) to the current root r of the server tree: $h(i, y) \overset{a}{\rightsquigarrow} r$.
2. Updates the client's record from (i, y) to $(i' \leftarrow i + 1, y')$ and computes the corresponding new root hash r' of the server tree.
3. Submits the tuple (i, y, a, r, y', r') to the repository for validation and publishing.
4. Waits for the repository to end the round and publish r_t.
5. Uses the state of its hash tree corresponding to the published r_t to extract and return to all clients with pending requests the hash chains a_t linking their updated (i', y') records to the published r_t: $h(i', y') \overset{a_t}{\rightsquigarrow} r_t$.

Signing: Repository. The validation layer R_v of the repository $\mathcal{R}$ keeps as state the current value r^* of the root hash of the server tree. Upon receiving the update (i, y, a, r, y', r') from S, the validator verifies its correctness:

1. The claimed starting state of the server tree matches the current state of R_v: $r = r^*$.
2. The claimed starting state of the signer record agrees with the starting state of the server tree: $h(i, y) \overset{a}{\rightsquigarrow} r$.
3. The update of the client record increments the counter: $i' \leftarrow i + 1$.
4. The new state of the server tree corresponds to just this one change: $h(i', y') \overset{a}{\rightsquigarrow} r'$.
5. If all the above checks pass, R_v updates its own state accordingly: $r^* \leftarrow r'$.

Note that the hash chain a is the same in the verification of the starting state of the signer record against the starting state of the server tree and in the verification of the new state of the signer record against the new state of the server tree. This ensures no other leaves of the server tree can change with this update.

R_v operates in rounds. During a round, it receives updates from the server, validates them, and updates its own state accordingly. At the end of the round, it publishes the current value of its state as the new round commitment r_t in the append-only storage $\mathcal{R}$.

Verification. To verify that the message m and the signature $s = (i, z, c, t, a)$ match the public key p, the verifier:

1. Checks that z was committed as the i-th signing key: $h(i, z) \overset{c}{\rightsquigarrow} p$.
2. Retrieves the commitment r_t for the round t from repository $\mathcal{R}$.
3. Checks that the use of the key z to compute the message authenticator $y \leftarrow h(m, z)$ matches the key index i: $h(i, y) \overset{a}{\rightsquigarrow} r_t$.

Note that the signature is composed and sent to verifier only after the verification of r_t, which makes it safe for the signer to release the key z_i as part of the signature: the server has already incremented its counter i so that only z_{i+1} could be used to produce the next valid signature.

5.3 Implementation Considerations

Server-Supported Signing. The model of server-supported signing is a higher-level protocol not directly comparable to traditional signature algorithms like RSA. The model has some useful properties:

- It is possible to create a server-side log of all signing operations, so that in case of either actual or suspected key compromise there is a complete record, making damage control and forensics manageable.
- Key revocation can be implemented by setting the client's counter to some sentinel "infinite" value, and the server can also return a proof of this update after it has been committed to the repository.
- The server can add custom attributes, and even *trusted attributes* which can't be forged by the server itself: cryptographic time-stamp, address, policy ID, etc.

Finally, in scenarios where non-repudiation must be provided, all traditional schemes and algorithms must be supplemented with some server-provided functionalities like cryptographic time-stamping.

Blockchain-Backed Repository. The proposed scheme dictates that the repository must have the following properties:

- Updates are only accepted if their proof of correctness is valid.
- All commitments are final and immutable.
- Commitments are public, and their immutability is publicly verifiable.

To minimize trust requirements on the repository, we propose to re-use the patterns used for creating *blockchains*. We do not consider proof-of-work, focusing on byzantine fault tolerant state machine replication model.

Instead of full transactions, we record in the blockchain only aggregate hashes representing batches of transactions. This provides two benefits: (1) the size of the blockchain grows linearly in time, in contrast with the usual dependency on the number of transactions and storage size of transaction records; and (2) recording and publishing only aggregate hashes ensures privacy.

When implemented as a distributed robust public transaction ledger [43], no single component of the repository needs to be trusted.

Scalable Architecture. Although presented above as a list of components, envisioned real-life deployment of the scheme is hierarchical, as shown in Figure 7.

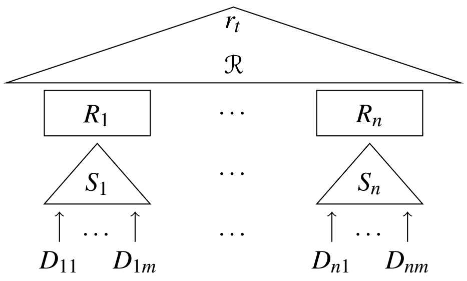

Figure 7: A scalable deployment architecture for BLT-BC: each client device D_{ij} is served by a designated server S_i audited by validation cluster R_i; round commitments from all validation clusters are aggregated into a meta-commitment by the repository $\mathcal{R}$.

The topmost layer is a distributed cluster of blockchain consensus nodes, each possibly operated by an independent "permissioned" party. The blockchain can accept inputs from multiple signing servers, each of which may in turn serve many clients. Because of this hierarchical nature the scheme scales well performance-wise. In terms of the amount of data, as stated earlier, the size of blocks and the number of blocks does not depend on the number of clients and number of signatures issued.

5.4 Discussion

Performance. The efficiency of the new scheme for both signers and verifiers is at least on par with the state of the art.

The performance considerations for key generation and management on the client side are similar to the BLT-TB scheme (Section 4.3), except the number of private keys required is much smaller (assuming 10 signing operations per day, just $3\,650$ keys are needed for a year, compared to the 32 million keys in BLT-TB) and the effort required to generate and manage them, which was the main weakness of BLT-TB, is also correspondingly reduced.

Like in BLT-TB, the size of the signature in the new scheme is also dominated by the two hash chains. The key sequence membership proof contains $\log_2 N$ hash values, which is about 12 for the 3650-element yearly sequence. The blockchain membership proof has $\log_2 K$ hash values, where K is the number of clients the service has. Even when the whole world (8 billion people) signs up, it's still only about 33 hash values. Conservatively assuming the use of 512-bit hash functions, the two hash chains add up to less than 3 kB in total.

Verification of the signature means re-computing the two hash chains and amounts to about 45 hash function evaluations.

Admittedly, the above estimates exclude the costs of querying the blockchain to acquire the committed r_t that both the signer and the verifier need. However, that is comparable to the need to access a time-stamping service when signing and an OCSP (Online Certificate Status Protocol) responder or a CRL (Certificate Revocation List) when verifying signatures in the traditional PKI (Public-Key Infrastructure) setup.

Security Model. The trusted repository $\mathcal{R}$ and its implications for the security model are similar to the BLT-TB scheme (Section 4.4).

In BLT-BC, the validation layer R_v is also critical for security, which obviously increases the trust base. As discussed in previous chapter for the repository $\mathcal{R}$, the security risks of introducing another trusted component into the system can be somewhat mitigated by implementing R_v as a distributed consensus cluster, but this adds additional engineering challenges and upkeep costs.

While the scheme also introduces state on the client side, this is mostly a usability concern and not a significant security risk. If a client loses track of its state and tries to re-use a key, the server will block the signing attempt.

6 Time-Stamped Scheme with One-Time Keys

Both the BLT-TB and BLT-BC signature schemes prevent other parties from misusing keys by making each key expire immediately after a legitimate use.

In BLT-BC, this is achieved by having a server track the first use of each key and prove the correctness of its operation through an auditing and publishing mechanism. Thus the security of BLT-BC rests on the reliability of the auditors and the publishing channel.

In BLT-TB, each key is explicitly bound to a time slot at the key-generation time and expires automatically when that time slot passes. The legitimate use of a key is proven by time-stamping the message-key pair at the correct time and the security of the scheme rests on the resilience of the time-stamping service against back-dating attacks, which is arguably a much smaller trust base.

6.1 Forward-Resistant Tags

Tag Systems. To combine the efficiency of the one-time keys of BLT-BC with the smaller trust base of BLT-TB, we can note that back-dating resistance of the time-stamp already prevents any attackers from moving the key usage events back in time. Thus, it would be sufficient for the key binding to only prevent it from being moved forward. Based on this observation, we introduce the concept of *forward-resistant tags*.

Definition 9 (Tag system). *By a tag system we mean a triple* $(\mathrm{Gen}, \mathrm{Tag}, \mathrm{Ver})$ *of algorithms, where:*

- Gen *is a probabilistic key-generation algorithm that, given as input the tag range* T*, produces a secret key* sk *and a public key* pk*.*
- Tag *is a tag-generation algorithm that, given as input the secret key* sk *and an integer* $t \in \{1, \ldots, T\}$*, produces a tag* $\tau \leftarrow \mathrm{Tag}(\mathsf{sk}, t)$*.*
- Ver *is a verification algorithm that, given as input a tag* τ*, an integer* t*, and the public key* pk*, returns either* 0 *or* 1*, such that*

$$\mathrm{Ver}(\mathrm{Tag}(\mathsf{sk}, t), t, \mathsf{pk}) = 1$$

whenever $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathrm{Gen}(T)$ *and* $1 \le t \le T$*.*

The above definition of a tag system is quite similar to that of a signature scheme consisting of procedures for key generation, signing, and verification. The fundamental difference is that a signature binds the use of the secret key to a message, while a tag binds the use of the secret key to a time.

Definition 10 (Forward-resistant tag system). *A tag system* $(\mathrm{Gen}, \mathrm{Tag}, \mathrm{Ver})$ *is* S*-secure forward-resistant if every tag-forging adversary* $\mathcal{A}$ *using computational resources* ρ *has success probability*

$$\Pr\left[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen}(T),\ (\tau, t) \leftarrow \mathcal{A}^{\mathrm{Tag}(\mathsf{sk}, \cdot)}(\mathsf{pk}) : \mathrm{Ver}(\tau, t, \mathsf{pk}) = 1\right] < \frac{\rho}{S},$$

where $\mathcal{A}$ *makes one oracle call* $\mathrm{Tag}(\mathsf{sk}, t')$ *with* $1 \le t' < t$*.*

The restriction for $\mathcal{A}$ to make just one oracle call stems from the fact that the very purpose of a tag system is to bind the use of the secret key to a specific time.

Informally, in order to implement a forward resistant tag system, we have to bind each tag to a time t so that the tag can't be re-bound to a later time. As already mentioned, this notion could be seen as dual to time-stamping that prevents back-dating.

Induced Signature Scheme. We can now formalize the signature scheme induced by a tag system and a time-stamping repository.

Definition 11. *A tag system* $(\mathrm{Gen}, \mathrm{Tag}, \mathrm{Ver})$ *and a time-stamping repository* $\mathcal{R}$ *induce a one-time signature scheme as follows:*

- *The signer* $\mathcal{S}^{\mathcal{R}}(m)$ *queries* $t \leftarrow \mathcal{R}.\mathtt{time}$, *creates* $\tau \leftarrow \mathrm{Tag}(\mathsf{sk}, t)$, *stores* $\mathcal{R}.\mathtt{put}((m, \tau))$, *and then returns* (τ, t) *as the signature.*
- *The verifier* $\mathcal{V}^{\mathcal{R}}(m, (\tau, t), \mathsf{pk})$ *queries* $x \leftarrow \mathcal{R}.\mathtt{get}(t)$, *and checks that* $x = (m, \tau)$ *and* $\mathrm{Ver}(\mathsf{pk}, t, \tau) = 1$.

We use the simplistic model of the time-stamping service (omitting the aggregation layer) for convenience of formal analysis. A more refined model would make the security reductions really complex. For example, even for a seemingly trivial change, where $\mathcal{R}$ publishes a hash $h(m, \tau)$ instead of just (m, τ), one needs non-standard security assumptions about h such as non-malleability. In this section, we try to avoid these technicalities and focus on the basic logic of the tag-based signature scheme.

Definition 12 (Existential unforgeability of one-time signatures)**.** *A one-time signature scheme is* S*-secure existentially unforgeable if every forging adversary* $\mathcal{A}$ *using computational resources* ρ *has success probability*

$$\Pr\Big[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen}(T),\ (m, \sigma) \leftarrow \mathcal{A}^{\mathcal{S}^{\mathcal{R}}, \mathcal{R}}(\mathsf{pk}) : \mathcal{V}^{\mathcal{R}}(m, \sigma, \mathsf{pk}) = 1\Big] < \frac{\rho}{S},$$

where $\mathcal{A}$ *makes only one* S*-query and not with* m*.*

Theorem 2. *If the tag system is* S*-secure forward-resistant then the induced one-time signature scheme is (almost)* S*-secure existentially unforgeable.*

The proof, which is given in Publication III, is by reduction: we assume an adversary A able to forge signatures and construct an adversary B able to forge tags, by having B simulate the repository $\mathcal{R}$ for A and use the data from intercepted $\mathcal{R}$-calls to produce a forged tag.

The BLT-TB Tag System. To simplify the analysis, we omit the aggregation of individual time-bound keys into a hash tree, and model the essence of the BLT-TB signature scheme as a tag system as follows:

- The secret key sk is a list $(z_1, \ldots, z_T)$ of T unpredictable values.
- The public key pk is the list $(f(z_1), \ldots, f(z_T))$, where f is a one-way function.
- The tagging algorithm $\mathrm{Tag}(z_1, \ldots, z_T; t)$ outputs z_t.
- The verification algorithm Ver, given as input a tag τ, an integer t, and the public key $(x_1, \ldots, x_T)$, checks that $1 \leq t \leq T$ and $f(\tau) = x_t$.

Theorem 3. *If* f *is* S*-secure one-way, then the BLT-TB tag system is* $\frac{S}{T}$*-forward-resistant.*

The proof, which is given in Publication III, is again by reduction: we assume a tag-forging adversary A and construct an f-inverting adversary B by making a call to A with a BLT-TB public key where a randomly selected component has been replaced with the target hash value; if the adversary A succeeds, then with probability $\frac{1}{T}$ it found a pre-image of the target value and thus B also succeeds.

The BLT-OT Tag System. We now define the BLT-OT tag system (inspired by Lamport's one-time signatures [37]) as follows:

- The secret key sk is a list $(z_0, \ldots, z_{\ell-1})$ of $\ell = \lceil \log_2(T+1) \rceil$ unpredictable values.
- The public key pk is the list $(f(z_0), \ldots, f(z_{\ell-1}))$, where f is a one-way function.
- The tagging algorithm $\mathrm{Tag}(z_0, \ldots, z_{\ell-1}; t)$ outputs an ordered subset $(z_{j_1}, \ldots, z_{j_m})$ of components of the secret key sk such that $0 \le j_1 < \ldots < j_m \le \ell - 1$ and $2^{j_1} + \ldots + 2^{j_m} = t$.
- The verification algorithm Ver, given as input a sequence $(z_{j_1}, \ldots, z_{j_m})$, an integer t, and the public key $(x_0, \ldots, x_{\ell-1})$, checks that:
  1. $f(z_{j_1}) = x_{j_1}, \ldots, f(z_{j_m}) = x_{j_m}$; and
  2. $0 \le j_1 < \ldots < j_m \le \ell - 1$; and
  3. $2^{j_1} + \ldots + 2^{j_m} = t$; and
  4. $1 \le t \le T$.

Theorem 4. *If f is S-secure one-way, then the BLT-OT tag system is $\frac{S}{\ell}$-forward-resistant.*

The proof is very similar to Theorem 3 and given in [15] which is an extended version of Publication III.

The BLT-W Tag System. We now define the BLT-W tag system (inspired by Winternitz's idea [62] for optimizing the size of Lamport's one-time signatures) as follows:

- The secret key sk is an unpredictable value z.
- The public key pk is $f^T(z)$, where f is a one-way function.
- The tagging algorithm $\mathrm{Tag}(z; t)$ outputs the value $f^{T-t}(z)$.
- The verification algorithm Ver, given as input a tag τ, an integer t, and the public key x, checks that $1 \le t \le T$ and $f^t(\tau) = x$.

Theorem 5. *If f is S_1-secure one-way, S_2-secure collision resistant, and S_3-secure undetectable, then the BLT-W tag system is $\frac{\min(S_1,S_2,S_3)}{2 \cdot T}$-secure forward-resistant.*

The proof is similar to Theorem 3 and given in [15], an extended version of Publication III.

6.2 Description of the Scheme

The signature scheme induced by the BLT-OT tag system according to Definition 11 would come with the requirement that the signer must know in advance the time at which its request reaches the time-stamping service. This is hard to achieve in practice, in particular for personal signing devices such as smart cards that lack built-in clocks. To overcome this limitation, we construct the BLT-OT one-time signature scheme as follows.

Key Generation. Let ℓ be the number of bits that can represent any time value t when the signature may be created (e.g. $\ell = 32$ for POSIX time up to year 2106). The private key is generated as sk $= (z_0, \ldots, z_{\ell-1})$, where z_i are unpredictable values, and the public key as pk $= h(X), X = (x_0, \ldots, x_{\ell-1}), x_i = f(z_i)$, where h is a second pre-image resistant hash function and f a one-way function.

Public Key Distribution. To aid instant key revocation, also the identity ID_c of the client and the identity ID_s of the designated time-stamping service should be distributed along with the public key (within the public key certificate in a typical PKI-like setup). Upon receiving a revocation notice, the service stops serving the affected client, and thus it is not possible to generate signatures using revoked keys.

Signing. To sign a message m, the client:

1. Gets a time-stamp S_t on the record (m, X, ID_c) from the time-stamping service designated by ID_s.
2. Extracts the ℓ-bit time value t from S_t and creates the list $W = (w_0, \ldots, w_{\ell-1})$, where
   - $w_i = z_i$ if the i-th bit of t is 1, or
   - $w_i = x_i = f(z_i)$ otherwise.
3. Disposes of the private key $(z_0, \ldots, z_{\ell-1})$ to prevent its re-use.
4. Emits (W, S_t) as the signature.

Verification. To verify the signature (W, S_t) on the message m against $(\mathsf{pk}, ID_c, ID_s)$, the verifier:

1. Extracts time t from the time-stamp S_t.
2. Recovers the list $X = (x_0, \ldots, x_{\ell-1})$ by computing
   - $x_i = f(w_i)$ if the i-th bit of t is 1, or
   - $x_i = w_i$ otherwise.
3. Checks that the computed X matches the public key: $h(X) = \mathsf{pk}$.
4. Checks that S_t is a valid time-stamp issued at time t by service ID_s on the record (m, X, ID_c).

Using the reduction techniques from previous sections to formally prove the security of this optimized signature scheme is complicated by both the iterated use of f and the more abstract view of the time-stamping service, and is left as future work.

6.3 Discussion

The BLT-TB scheme works well for powerful devices that are constantly running and have reliable clocks. These are not reasonable assumptions for personal signing devices such as smart cards, which have very limited capabilities and are not used very often. Generating keys could take hours or even days of non-stop computing on such devices. This is clearly impractical, and also wasteful as most of the keys would go unused.

The BLT-OT scheme proposed in Section 6.2 solves these problems at the cost of introducing state on the client side. As the scheme is targeted towards personal signing devices, the statefulness is not a big risk, because these devices are not backed up and also do not support parallel processing. The benefit in addition to improved efficiency is that the device no longer needs to know the current time while preparing a signing request. Instead, it can just use the time from the time-stamp when composing the signature.

Performance as One-Time Scheme. When implemented as described in Section 6.2, the cost of generating a BLT-OT key pair is ℓ random key generations and $\ell + 1$ hashing operations, the cost of signing $\ell + 1$ hashing operations and one time-stamping service call, and the cost of signature verification at most $\ell + 1$ hashing operations and one time-stamp verification. In this case the private key would consist of ℓ one-time keys and the public key of one hash value, and the signature would contain ℓ hash values and one time-stamp token. The private storage size can be optimized by generating the one-time keys from one true random seed using a pseudo-random generator. Then the cost of signing increases by ℓ operations, as the one-time keys would have to be re-generated from the seed before signing. This version is listed as BLT-OT in Table 1.

Winternitz's idea [62] for optimizing the size of Lamport's one-time signatures [37] can also be applied to BLT-OT. Instead of using one-step hash chains $z_i \rightarrow h(z_i) = x_i$ to encode

Table 1: Efficiency of hash-based one-time signature schemes. We assume 256-bit hash functions, 32-bit time values, and time-stamping hash-tree with 33 levels. Times are in hashing operations and signature sizes in hash values. TS in BLT schemes stands for the time-stamping service call.

Scheme	Key gen. time	Sig. time	Ver. time	Sig. size
Lamport	1 025	1 024	513	256
Winternitz ($w = 4$)	1 089	1 088	1 021	68
BLT-OT	65	64 + TS	33 + 33	32 + 33
BLT-W ($w = 2$)	65	64 + TS	49 + 33	16 + 33

single bits of t, we can use longer chains $z_i \to h(z_i) \to \ldots \to h^n(z_i) = x_i$ and publish the value $h^{n-j}(z_i)$ in the signature to encode the value j of a group of bits of t. When encoding groups of w bits of t in this manner, the chains have to be $n = 2^w$ steps long. This reduces the size of the signature by w times, but increases the costs of key generation and signing by a bit less than $\frac{2^{w-1}}{w}$ times and the cost of verification by a bit less than $\frac{2^w-1}{w}$ times. Note that for $w = 2$, only the verification cost increases by about 50%! Also note that in contrast to applying this idea to Lamport's signatures, in BLT-W no additional countermeasures are needed to prevent an adversary from stepping the hash chains forward: the time in the time-stamp takes that role. This version is listed as BLT-W in Table 1.

To compare BLT-OT signature sizes and verification times to other schemes, we also need to estimate the size of hash-trees built by the time-stamping service. Even assuming the whole world (8 billion people) will use the time-stamping service in every aggregation round, an aggregation tree of 33 layers will suffice. We also assume that in all schemes one-time private keys will be generated on-demand from a single random seed and public keys will be aggregated into a single hash value. Therefore, the key sizes will be the same for all schemes and are not listed in Table 1.

Performance as Many-Time Scheme. A one-time signature scheme is not practical by itself. Merkle [61] proposed aggregating multiple public keys of a one-time scheme using a hash tree to produce so-called N-time schemes. Assuming 10 signing operations per day, a set of 3 650 BLT-OT keys would be sufficient for a year. The key generation costs would obviously grow correspondingly. The change in signing time would depend on how the hash tree would be handled. If sufficient memory is available to keep the tree (which does not contain private key material and thus may be stored in regular memory), the

Table 2: Efficiency of hash-based many-time signature schemes. We assume key supply for at least 3 650 signatures, 256-bit hash functions, 32-bit time values, and time-stamping hash-tree with 33 levels. Times are in hashing operations and signature sizes in hash values. TS in BLT schemes stands for the time-stamping service call.

Scheme	Key gen. time	Sig. time	Ver. time	Sig. size
XMSS	897 024	8 574	1 151	79
SPHINCS	ca 16 000	ca 250 000	ca 7 000	ca 1 200
BLT-TB	ca 96 000 000	50 + TS	25 + 33	25 + 33
BLT-OT-N	240 900	64 + TS	45 + 33	44 + 33
BLT-W-N ($w = 2$)	240 900	64 + TS	61 + 33	28 + 33

authenticating hash chains for individual one-time public keys could be extracted with no extra hash computations. Signature size and verification time would increase by the 12 additional hashing steps linking the one-time public keys to the root of the aggregation tree. This scheme is listed as BLT-OT-N in Table 2, where we compare it with the following schemes:

- XMSS is a stateful scheme like BLT-OT-N; the values in Table 2 are computed by taking $N = 2^{12} = 4\,096$ and leaving other parameters as in [13];
- SPHINCS is a stateless scheme and can produce an indefinite number of signatures; the values in Table 2 are inferred from [8] counting invocations of the ChaCha12 cipher on 64-byte inputs as equivalent to hash function evaluations;
- The values for BLT-TB in Table 2 are inferred from Section 4.4.

Security Model. Like those of BLT-TB and BLT-BC, the security model of BLT-OT also relies on the repository $\mathcal{R}$ for the unforgeability of the signatures. This is in contrast with XMSS and SPHINCS, both traditional "standalone" signature schemes.

Unlike BLT-BC, management of client-side state to track spent keys is a real security concern for BLT-OT. Indeed, even using a private key just twice may in the worst case leak all key components and give an adversary an easy opportunity to forge signatures on any chosen messages at any chosen time in the future. Although this is not necessary with correct private key management, an external service similar to the validation layer of BLT-BC may in fact provide a useful safety net to reduce this risk. Such state management concerns also apply to XMSS, while SPHINCS, being a stateless scheme, is not affected.

A weakness of the BLT family compared to XMSS and SPHINCS is the higher requirements that the BLT schemes place on the underlying hash function. The unforgeability proof for BLT-TB signature scheme in Chapter 4 assumes the hash function models a random oracle, which is a very high bar.

The forward-resistance proofs of BLT-TB and BLT-OT tag systems given in Section 6.1 only assume one-wayness from the underlying hash function, but these proofs cover only a small part of the whole signature scheme. Extending the security proofs to complete signature schemes while keeping the assumptions minimal is likely to require changes in the definitions of the schemes.

For example, the plain hash trees aggregating the one-time or time-bound key pairs into a many-time key set will likely have to be replaced by more complicated constructs, like has already been done in XMSS and SPHINCS. How such hardening will affect the performance of the signature schemes remains to be determined by future research.

7 Application: Efficient Log Signing

In this chapter, we will slightly change the subject and take a look at applying hash function cryptography to protecting the integrity and authenticity of logs.

Increasingly, logs from various information systems are used as evidence and with that, the requirements on maintenance and presentation of the log data are growing. Integrity and authenticity—confidence that the log has not been tampered with or replaced with another—are obvious requirements in this context.

As information systems usually log all their activities sequentially, often the details of the relevant transactions are interspersed with other information in a log. To protect the confidentiality of unrelated events, it is desirable to be able to extract records from the signed log and still prove their integrity.

An example of such a case is a dispute between a bank and a customer. On one hand, the bank can't just present the whole log, as the log contains also information about transactions of other customers. On the other hand, the customer involved in the dispute should have a chance to verify the integrity of the relevant records.

Hence, an ideal log signing scheme should have the following properties:

- The integrity of the whole log can be verified by the owner of the log: no records can be added, removed or altered undetectably.
- The integrity of any record can be proven to a third party without leaking any information about the contents of any other records in the log.
- The signing process is efficient in both time and space.
- The extraction process is efficient in both time and space.
- The verification process is efficient in both time and space.

7.1 Related Work

Schneier and Kelsey [75] proposed a log protection scheme that encrypts the records using one-time keys and links them using cryptographic hash functions. The scheme allows both for verification of the integrity of the whole log and for selective disclosure of the one-time encryption keys. However, it needs a third party trusted by both the logger and the verifier and active participation of this trusted party in both phases of the protocol.

Holt [48] replaced the symmetric cryptographic primitives used in [75] with asymmetric ones and enabled verification without the trusted party. However, his scheme requires public-key signatures on individual records, which adds high computational and storage overhead to the logging process. Also, the size of the information required to prove the integrity of one record is at least proportional to the square root of the distance of the record from the beginning of the log. Other proposed amendments [78, 1] to the protocol from [75] have similar weaknesses.

Kelsey *et al.* [53] proposed a log signing scheme where records are signed in blocks, by first computing a hash value of each record and then signing the sequence of hash values. This enables efficient verification of the integrity of the whole block, significantly reduces the overhead compared to having a signature per record, and also removes the need to ship the whole log block when a single record is needed as evidence. But still the size of the proof of a record is linear in the size of the block. Also, other records in the same block are not protected from the informed brute-force attack discussed in Section 7.2.

Ma and Tsudik [57] constructed a logging protocol which provides so-called *forward-secure stream integrity*, retaining the provable security of the underlying primitives. They also proposed a possibility to store individual signatures to gain better granularity, at the cost of extra storage overhead.

7.2 Description of the Scheme

A computational process producing a log may, in principle, run indefinitely and thus the log as an abstract entity may not have a well-defined beginning and end. In the following, we model the log as an ordered sequence of blocks, where each block in turn is an ordered sequence of records. Many practical logging systems work this way, for example in the case of `syslog` output being sent to a log file that is periodically rotated.

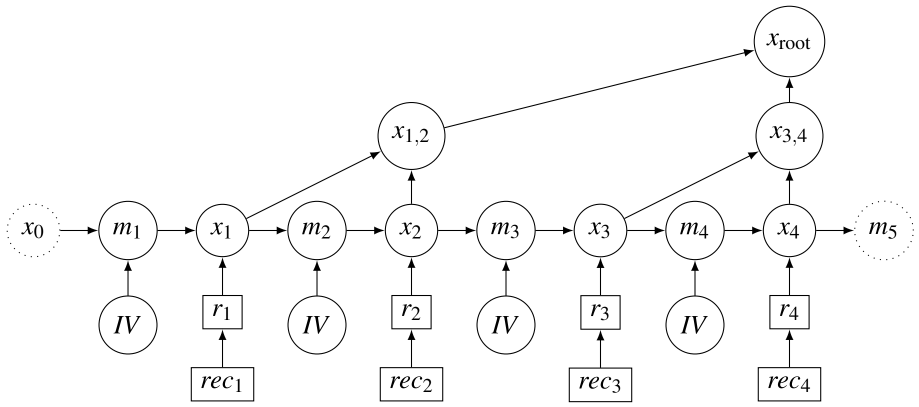

Figure 8: Log aggregation using a hash tree with interlinks and blinding masks: rec_i are the log records; r_i are the hash values of the records; IV is the random seed; m_i are the blinding masks; x_i are leaves and $x_{a,b}$ are internal nodes of the hash tree; x_{root} is the value to be signed.

Hash Tree Aggregation. To reduce the size of the evidence for a single record compared to [48] and [53], the records can instead be aggregated using a hash tree.

Blinding Masks. The hash chain extracted from the hash tree for one leaf contains the values of other nodes. A cryptographic hash function can't be directly reversed to learn the input value from which a hash value in the chain was created. However, a typical log record may contain insufficient entropy to make that argument—an attacker who knows the pattern of the input could exhaustively test all possible variants to find the one that yields the hash value actually in the chain and thus learn the contents of the record. To prevent this kind of informed brute-force attack, a blinding mask with sufficient entropy can be added to each record before aggregating the hash values.

Generating cryptographically secure random values is expensive. Also, when an independent random mask would be used for each record, all these values would have to be stored for later verification. It is therefore much more efficient to derive all the blinding masks from a single random seed, as in the data structure shown on Figure 8.

The Scheme. The steps to aggregate records $rec_1, \ldots, rec_N$ for signing are as follows:

1. The seed IV is generated as a k-bit random value.
2. Each record is hashed: $r_i \leftarrow h(rec_i)$.
3. For each record, a blinding mask is computed and applied: $m_i \leftarrow h(x_{i-1}, IV)$, $x_i \leftarrow h(m_i, r_i)$.
4. The blinded record hashes are aggregated into a hash tree: $r \leftarrow T^h(x_1, \ldots, x_N)$.
5. The root hash r is signed.

7.3 Security of the Scheme

The security properties of hash trees in protecting the integrity of inputs is already well studied in the context of hash-then-publish time-stamping (Section 3.5). In particular, the use of a collision resistant hash function is sufficient to protect the integrity of aggregated records if the length of the hash chains accepted is restricted [31, 29], and the requirements on the hash function are higher if unbounded aggregation is desired [31, 21].

In this section we focus on confidentiality: that the integrity proof for any record does not leak any information about the contents of any other records. To formalize the privacy-preserving property, we adapt the concept of *indistinguishability under chosen-plaintext attack* (IND-CPA), originally proposed by Luby [56] for symmetric encryption schemes.

Definition 13 (Content concealing log signing). *A log signing scheme is S-secure IND-CPA content concealing if every detecting adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ using computational resources ρ has advantage $\delta < \frac{\rho}{S}$ in the following attack scenario:*

1. *The first stage $\mathcal{A}_1$ of the adversary chooses a position i and a list of records $rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell$, as well as two test records rec_i^0, rec_i^1 and an advice string a.*
2. *The environment sets $x_0 \leftarrow 0^k$, picks randomly $IV \xleftarrow{\$} \{0,1\}^k$ and $b \xleftarrow{\$} \{0,1\}$, assigns $rec_i \leftarrow rec_i^b$ and for every $j \in \{1, \ldots, \ell\}$ computes: $m_j \leftarrow f(IV, x_{j-1})$, $r_j \leftarrow h(rec_j)$, and $x_j \leftarrow f(m_j, r_j)$.*
3. *The second stage $\mathcal{A}_2$ of the adversary, given as input the advice string a and the lists of hash values $x_1, \ldots, x_\ell$, and $m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell$, tries to guess the value of b by outputting the guessed value $\hat{b}$.*

The advantage of $\mathcal{A}$ is defined as $\delta = 2\left|\Pr\left[\hat{b} = b\right] - \frac{1}{2}\right|$.

We give the proof of the content concealing property of our signing scheme under the assumption that the hash function used models a pseudo-random function family. Informally, the assumption means that a 2-to-1 hash function $f : \{0,1\}^{2k} \to \{0,1\}^k$ can be assumed to behave like a random function $\Omega : \{0,1\}^k \to \{0,1\}^k$ when the first half of the input is a randomly chosen secret value.

Definition 14 (Pseudo-random function family). *A two-argument function $f : D_1 \times D_2 \to R$ is an S-secure pseudo-random function family if every adversary $\mathcal{A}$ using computational resources ρ to distinguish between a restriction of f fixing its first argument to a randomly chosen value r and a true random oracle $\Omega : D_2 \to R$ has success probability*

$$\left|\Pr\left[r \xleftarrow{\$} D_1 : \mathcal{A}^{f(r,\cdot)} = 1\right] - \Pr\left[\mathcal{A}^{\Omega(\cdot)} = 1\right]\right| < \frac{\rho}{S} .$$

Theorem 6. *If f is an S-secure pseudorandom function family, then our log signing scheme is $\frac{S}{4}$-secure IND-CPA content concealing.*

The proof, which is given in Publication IV, is game-based: we define three games where the first one is the attack game against our aggregation scheme, the second one is attacking a version of the scheme where the blinding masks are independently generated random values, and in the third one the input to the second stage of the adversary does not depend on b; we then show that the difference of success probabilities between the first and second game is bound by the indistinguishability of f from a pseudo-random function family, and the same for the second and third game; we then obtain our result as the sum of these two success probabilities.

7.4 Implementation Considerations

Canonical Binary Trees. As our log signing scheme only signs the root value of the aggregation hash tree, it is crucial to build the tree in a deterministic manner so that a verifier would be able to construct the exact same tree as the signer did. To achieve the logarithmic size of the integrity proofs of the individual records, the tree should not be overly unbalanced. Thus, we define the *canonical binary tree* with n leaves (shown for $n = 11$ on Figure 9) to be built as follows:

1. The leaves are laid out *from left to right* (square nodes on the figure).
2. The leaves are collected into perfect binary trees *from left to right*, making each tree as big as possible using the leaves still available (adding the white circles on the figure).
3. The perfect trees are merged into a single tree *from right to left* which means joining the two smallest trees on each step (adding the black circles on the figure).

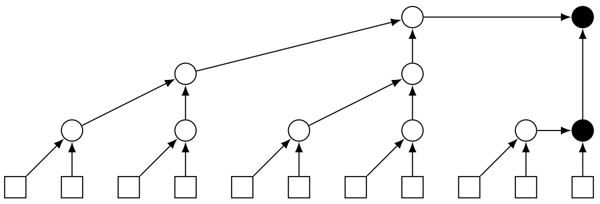
Figure 9: A canonical binary tree: 11 leaves (squares) grouped into three perfect trees (white circles) and merged into a single tree with minimal height (black circles).

A useful property of canonical trees is that they can be built on-line, as the leaves arrive, without knowing in advance the eventual size of the tree, and keeping in memory only logarithmic number of nodes (the root nodes of the perfect binary trees constructed so far). Such on-line algorithms both for aggregating a sequence of records and also for extracting a hash chain linking a specific input record to the root of the tree are given in Publication III.

Performance. Aggregating a log record for signing with our scheme incurs four hash function evaluations on average: one to hash the record itself, one to generate the blinding mask, one to apply the blinding mask to the record hash, and amortized one evaluation to link the record into the hash tree.

Storage overhead depends on whether the record and tree hashes are stored or re-computed on demand, hash algorithm output size, and log block size. In case of 256-byte log records and 32-byte hash values, the storage overhead is about 12% for keeping the record hashes and about 25% for keeping the tree hashes. The storage overhead caused by signatures themselves is negligible in practical scenarios.

There are two potential benefits to be gained from storing the hashes. The first one is resilience to small localized changes in the signed log data (whether the changes are malicious or accidental). If only the block-level signatures are kept, the hash tree has to be re-built for signature verification and a mismatch between the newly computed root hash and the signature only indicates that something has changed in the data. If, on the other hand, the record hashes or the tree hashes (or both) are kept and they happen to be intact (which is likely in case of accidental data decay), the integrity of those records that have indeed not changed can still be proven by first verifying the integrity of the

Table 3: Storage, runtime and verification feature trade-offs (N is log block size).

Characteristic	**No hashes kept**	**Record hashes**	**Tree hashes**
	Signing		
Per-record storage	none	1 hash value	2 hash values
Per-record computation	4 hashings	4 hashings	4 hashings
Per-block storage	1 signature value	1 signature value	1 signature value
Per-block computation	1 signing	1 signing	1 signing
Memory	$O(\log N)$	$O(\log N)$	$O(\log N)$
	Whole log verification		
Report granularity	block	record	record
Time	$O(N)$	$O(N)$	$O(N)$
Memory	$O(\log N)$	$O(\log N)$	$O(\log N)$
	Record proof extraction		
Per-record storage	$O(\log N)^*$	$O(\log N)^*$	$O(\log N)^*$
Time	$O(N)$	$O(N)$	$O(\log N)$
Memory	$O(\log N)$	$O(\log N)$	$O(1)$
	Record proof verification		
Report granularity	record	record	record
Time	$O(\log N)$	$O(\log N)$	$O(\log N)$
Memory	$O(1)$	$O(1)$	$O(1)$

* It's $O(\log N)$ asymptotically, but in practice often the $O(1)$ signature size dominates over the $O(\log N)$ hash chain size. For example, for the 3600-byte signatures and 32-byte hash values used in the case study in Publication III, the signature size exceeds the hash chain size for all $N < 2^{100}$.

stored hashes against the signature and then comparing the records one by one against the respective hashes.

If the tree hashes are kept in a data structure that allows efficient access to individual hash values, the integrity proofs for individual records can be extracted without the need to re-build the tree, and also without the need to access the underlying records. In addition to increased efficiency, the latter property also improves confidentiality.

Table 3 summarizes the performance of our log signing scheme and the trade-offs between the hash storage policies.

7.5 Discussion

While using hash trees to aggregate data before signing is not new, we are not aware of previous applications in logging context. Aside from application case study, we contributed a method for generating multiple blinding masks from a single random value to achieve blinding that is provably as good as using independently generated random masks.

Compared to previous log signing schemes with selective disclosure, our method offers improvements as follows: unlike [75], or scheme does not require a trusted third party; the proof of integrity of a record is $O(\log N)$ in our scheme, compared to $O(\sqrt{N})$ in [48] and $O(N)$ in [53]; while the asymptotic complexities are not directly comparable, based on the performance comparison table provided in [57], where the best-case signer computation cost is 5.55 ms per log record (albeit on slightly weaker hardware than in our experiment), we can estimate that our scheme is faster by two to three orders of magnitude.

8 Conclusions and Outlook

We have proposed several hash-based server-assisted digital signature schemes. A novel design element of the schemes is their reliance on time-stamping service as an inherent component. The performance of the new schemes is very competitive, as indicated in Tables 1 and 2, but the reliance on time-stamping service adds dependence on a security-critical external component.

The BLT-TB scheme described in Chapter 4 is suitable for use in server applications that need to produce a lot of signatures. The scheme features efficient signing and verification and small signatures. Only the key generation is quite expensive, but still tolerable on full-sized computers. The scheme also requires the signer to have a reliable clock and direct network access, which are reasonable assumptions for servers. A benefit is that the scheme is stateless in the sense that the key to be used is determined by the signing time and thus the signer does not need to explicitly track spent keys and also access to the private key does not need to be synchronized in a parallel execution environment.

In contrast, the BLT-OT scheme described in Chapter 6 is suitable for personal signing devices that are used only occasionally. In addition to small signatures and efficient signing and verification, also key generation costs are relatively low. The price for this efficiency is the introduction of state on the client side. However, as typically personal key management devices, such as smart cards and USB tokens, are not backed up and do not support multi-processing, the risks related to managing key state are significantly reduced.

A weakness of the formal analysis of the tag-based schemes in Chapter 6, in particular compared to the analysis in Chapter 4, is the simplistic modeling of the time-stamping service. Some progress towards addressing this issue has been made in [17].

Another limitation of the present work is that the formal security reductions are done in classical setting and post-quantum security of the schemes is supported only indirectly, by referring to the quantum resilience of hash functions in general.

Formal analysis in quantum setting is hindered, among other difficulties, by the fact that our signature schemes depend on time-stamping and there is currently no well-defined notion of back-dating resistance of time-stamping in quantum setting. There is hope, however, that the collapsing property [82, 81, 35] of hash functions could be useful in moving to quantum setting.

We also proposed a privacy-protecting hash-function based aggregation mechanism for efficient signing of audit logs. An interesting avenue for future work in that application is to augment the aggregation with an evolving key mechanism so that an attacker who has gained control of a system would not be able to re-sign any logs after they have been tampered with.

References

[1] R. Accorsi. BBox: a distributed secure log architecture. In *EuroPKI 2010*, *Proceedings*, volume 6711 of *LNCS*, pages 109–124. Springer, 2011.

[2] R. J. Anderson, F. Bergadano, B. Crispo, J.-H. Lee, C. Manifavas, and R. M. Needham. A new family of authentication protocols. *Operating Systems Review*, 32(4):9–20, 1998.

[3] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton. Seven-property-preserving iterated hashing: ROX. In *ASIACRYPT 2007*, *Proceedings*, volume 4833 of *LNCS*, pages 130–146. Springer, 2007.

[4] E. Andreeva and M. Stam. The symbiosis between collision and preimage resistance. In *IMACC 2011*, *Proceedings*, volume 7089 of *LNCS*, pages 152–171. Springer, 2011.

[5] Asokan, G. Tsudik, and M. Waidner. Server-supported signatures. *Journal of Computer Security*, 5(1):91–108, 1997.

[6] J. Benaloh and M. de Mare. Efficient broadcast time-stamping. Technical report, Clarkson University, 1991.

[7] D. J. Bernstein, J. Buchmann, and E. Dahmen, editors. *Post-Quantum Cryptography*. Springer, 2009.

[8] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn. SPHINCS: Practical stateless hash-based signatures. In *EUROCRYPT 2015*, *Proceedings*, *Part I*, volume 9056 of *LNCS*, pages 368–397. Springer, 2015.

[9] K. Bicakci and N. Baykal. Server assisted signatures revisited. In *CT-RSA 2004*, *Proceedings*, volume 2964 of *LNCS*, pages 143–156. Springer, 2004.

[10] G. Brassard, P. Høyer, and A. Tapp. Quantum cryptanalysis of hash and claw-free functions. In *LATIN'98*, *Proceedings*, volume 1380 of *LNCS*, pages 163–169. Springer, 1998.

[11] J. A. Buchmann, L. C. Coronado García, E. Dahmen, M. Döring, and E. Klintsevich. CMSS—An improved Merkle signature scheme. In *INDOCRYPT 2006*, *Proceedings*, volume 4329 of *LNCS*, pages 349–363. Springer, 2006.

[12] J. A. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert. On the security of the Winternitz one-time signature scheme. *IJACT*, 3(1):84–96, 2013.

[13] J. A. Buchmann, E. Dahmen, and A. Hülsing. XMSS—A practical forward secure signature scheme based on minimal security assumptions. In *PQCrypto 2011*, *Proceedings*, volume 7071 of *LNCS*, pages 117–129. Springer, 2011.

[14] J. A. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, and C. Vuillaume. Merkle signatures with virtually unlimited signature capacity. In *ACNS 2007*, *Proceedings*, volume 4521 of *LNCS*, pages 31–45. Springer, 2007.

[15] A. Buldas, D. Firsov, R. Laanoja, H. Lakk, and A. Truu. A new approach to constructing digital signature schemes (extended paper). Cryptology ePrint Archive, Report 2019/673, 2019. `https://eprint.iacr.org/2019/673`.

[16] A. Buldas, D. Firsov, R. Laanoja, H. Lakk, and A. Truu. A new approach to constructing digital signature schemes (short paper). In *IWSEC 2019*, *Proceedings*, volume 11689 of *LNCS*, pages 363–373. Springer, 2019.

[17] A. Buldas, D. Firsov, R. Laanoja, and A. Truu. Verified security of BLT signature scheme. In *ACM SIGPLAN CPP 2020*, *Proceedings*, pages 244–257. ACM, 2020.

[18] A. Buldas, A. Kalu, P. Laud, and M. Oruaas. Server-supported RSA signatures for mobile devices. In *ESORICS 2017*, *Proceedings*, *Part I*, volume 10492 of *LNCS*, pages 315–333. Springer, 2017.

[19] A. Buldas, A. Kroonmaa, and R. Laanoja. Keyless signatures' infrastructure: How to build global distributed hash-trees. In *NordSec 2013*, *Proceedings*, volume 8208 of *LNCS*, pages 313–320. Springer, 2013.

[20] A. Buldas and R. Laanoja. Security proofs for hash tree time-stamping using hash functions with small output size. In *ACISP 2013*, *Proceedings*, volume 7959 of *LNCS*, pages 235–250. Springer, 2013.

[21] A. Buldas, R. Laanoja, P. Laud, and A. Truu. Bounded pre-image awareness and the security of hash-tree keyless signatures. In *ProvSec 2014*, *Proceedings*, volume 8782 of *LNCS*, pages 130–145. Springer, 2014.

[22] A. Buldas, R. Laanoja, P. Laud, and A. Truu. Bounded pre-image awareness and the security of hash-tree keyless signatures. In *ProvSec 2014*, *Proceedings*, volume 8782 of *LNCS*, pages 130–145. Springer, 2014.

[23] A. Buldas, R. Laanoja, and A. Truu. Keyless signature infrastructure and PKI: Hash-tree signatures in pre- and post-quantum world. *IJSTM*, 23(1/2):117–130, 2017.

[24] A. Buldas, R. Laanoja, and A. Truu. A server-assisted hash-based signature scheme. In *NordSec 2017*, *Proceedings*, volume 10674 of *LNCS*, pages 3–17. Springer, 2017.

[25] A. Buldas, R. Laanoja, and A. Truu. A blockchain-assisted hash-based signature scheme. In *NordSec 2018*, *Proceedings*, volume 11252 of *LNCS*, pages 138–153. Springer, 2018.

[26] A. Buldas and P. Laud. New linking schemes for digital time-stamping. In *ICISC'98*, *Proceedings*, pages 3–14. KIISC, 1998.

[27] A. Buldas, P. Laud, H. Lipmaa, and J. Villemson. Time-stamping with binary linking schemes. In *CRYPTO'98*, *Proceedings*, volume 1462 of *LNCS*, pages 486–501. Springer, 1998.

[28] A. Buldas, H. Lipmaa, and B. Schoenmakers. Optimally efficient accountable time-stamping. In *PKC 2000*, *Proceedings*, volume 1751 of *LNCS*, pages 293–305. Springer, 2000.

[29] A. Buldas and M. Niitsoo. Optimally tight security proofs for hash-then-publish time-stamping. In *ACISP 2010*, *Proceedings*, volume 6168 of *LNCS*, pages 318–335. Springer, 2010.

[30] A. Buldas and M. Saarepera. Electronic signature system with small number of private keys. In *2nd Annual PKI Research Workshop*, *Proceedings*, pages 96–108. NIST, 2003.

[31] A. Buldas and M. Saarepera. On provably secure time-stamping schemes. In *ASIACRYPT 2004*, *Proceedings*, volume 3329 of *LNCS*, pages 500–514. Springer, 2004.

[32] A. Buldas, A. Truu, R. Laanoja, and R. Gerhards. Efficient record-level keyless signatures for audit logs. In *NordSec 2014*, *Proceedings*, volume 8788 of *LNCS*, pages 149–164. Springer, 2014.

[33] J. Camenisch, A. Lehmann, G. Neven, and K. Samelin. Virtual smart cards: How to sign with a password and a server. In *SCN 2016*, *Proceedings*, volume 9841 of *LNCS*, pages 353–371. Springer, 2016.

[34] L. C. Coronado García. *Provably Secure and Practical Signature Schemes*. PhD thesis, Darmstadt University of Technology, Germany, 2005.

[35] J. Czajkowski, L. G. Bruinderink, A. Hülsing, C. Schaffner, and D. Unruh. Post-quantum security of the sponge construction. In *PQCrypto 2018*, *Proceedings*, volume 10786 of *LNCS*, pages 185–204. Springer, 2018.

[36] E. Dahmen, K. Okeya, T. Takagi, and C. Vuillaume. Digital signatures out of second-preimage resistant hash functions. In *PQCrypto 2008*, *Proceedings*, volume 5299 of *LNCS*, pages 109–123. Springer, 2008.

[37] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[38] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proc. IEEE*, 67(3):397–427, 1979.

[39] C. Dods, N. P. Smart, and M. Stam. Hash based digital signature schemes. In *Cryptography and Coding*, *Proceedings*, volume 3796 of *LNCS*, pages 96–115. Springer, 2005.

[40] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985.

[41] European Commission. Regulation no 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/EC (eIDAS regulation). *Official Journal of the European Union*, L 257:73–114, 2014.

[42] S. Even, O. Goldreich, and S. Micali. On-line/off-line digital signatures. *J. Cryptology*, 9(1):35–67, 1996.

[43] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT 2015*, *Proceedings*, *Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, 2015.

[44] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[45] V. Goyal. More efficient server assisted one time signatures. Cryptology ePrint Archive, Report 2004/135, 2004.

[46] L. K. Grover. A fast quantum mechanical algorithm for database search. In *28th ACM STOC*, *Proceedings*, pages 212–219. ACM, 1996.

[47] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *J. Cryptology*, 3(2):99–111, 1991.

[48] J. E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Proceedings of the 2006 Australasian Workshops on Grid Computing and e-Research*, pages 203–211. Australian Computer Society, 2006.

[49] A. Hülsing. W-OTS+—Shorter signatures for hash-based signature schemes. In *AFRICACRYPT 2013, Proceedings*, volume 7918 of *LNCS*, pages 173–188. Springer, 2013.

[50] A. Hülsing, L. Rausch, and J. A. Buchmann. Optimal parameters for XMSS MT. In *CD-ARES 2013, Proceedings*, volume 8128 of *LNCS*, pages 194–208. Springer, 2013.

[51] A. Hülsing, J. Rijneveld, and F. Song. Mitigating multi-target attacks in hash-based signatures. In *PKC 2016, Proceedings, Part I*, volume 9614 of *LNCS*, pages 387–416. Springer, 2016.

[52] D. Johnson, A. Menezes, and S. A. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *Int. J. Inf. Sec.*, 1(1):36–63, 2001.

[53] J. Kelsey, J. Callas, and A. Clemm. Signed syslog messages. IETF RFC 5848, 2010.

[54] M. Keren, A. Kirshin, J. Rubin, and A. Truu. MDA approach for maintenance of business applications. In *ECMDA-FA 2006, Proceedings*, volume 4066 of *LNCS*, pages 40–51. Springer, 2006.

[55] L. Lamport. Password authentification with insecure communication. *Commun. ACM*, 24(11):770–772, 1981.

[56] M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, 1996.

[57] D. Ma and G. Tsudik. A new approach to secure logging. *ACM Transactions on Storage*, 5(1):2:1–2:21, 2009.

[58] T. Malkin, D. Micciancio, and S. K. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *EUROCRYPT 2002, Proceedings*, volume 2332 of *LNCS*, pages 400–417. Springer, 2002.

[59] E. Martín-López, A. Laing, T. Lawson, R. Alvarez, X.-Q. Zhou, and J. L. O'Brien. Experimental realization of Shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, 6(11):773–776, 2012.

[60] D. A. McGrew, P. Kampanakis, S. R. Fluhrer, S.-L. Gazdag, D. Butin, and J. A. Buchmann. State management for hash-based signatures. In *SSR 2016, Proceedings*, volume 10074 of *LNCS*, pages 244–260. Springer, 2016.

[61] R. C. Merkle. *Secrecy, Authentication and Public Key Systems*. PhD thesis, Stanford University, 1979.

[62] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO'87, Proceedings*, volume 293 of *LNCS*, pages 369–378. Springer, 1987.

[63] NIST. Secure hash standard (SHS). FIPS 180-4, 2001.

[64] NIST. Post-quantum cryptography. `https://www.nist.gov/pqcrypto`, 2016.

[65] M. Opmanis, V. Dagienė, and A. Truu. Task types at Beaver contests. In *ISSEP 2006, Proceedings*, pages 509–519. Institute of Mathematics and Informatics, Vilnius, Lithuania, 2006.

[66] A. Perrig. The BiBa one-time signature and broadcast authentication protocol. In *ACM CCS 2001, Proceedings*, pages 28–37. ACM, 2001.

[67] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. The TESLA broadcast authentication protocol. *CryptoBytes*, 5(2):2–13, 2002.

[68] T. Perrin, L. Bruns, J. Moreh, and T. Olkin. Delegated cryptography, online trusted third parties, and PKI. In *1st Annual PKI Research Workshop, Proceedings*, pages 97–116. NIST, 2002.

[69] T. Poranen, V. Dagienė, Åsmund Eldhuset, H. Hyyrö, M. Kubica, A. Laaksonen, M. Opmanis, W. Pohl, J. Skūpienė, P. Söderhjelm, and A. Truu. Baltic olympiads in informatics: Challenges for training together. In *Olympiads in Informatics: The International Conference joint with the XXI International Olympiad in Informatics, Proceedings*, pages 112–131. Institute of Mathematics and Informatics, Vilnius, Lithuania, 2009.

[70] L. Reyzin and N. Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In *ACISP 2002, Proceedings*, volume 2384 of *LNCS*, pages 144–153. Springer, 2002.

[71] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[72] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE 2004, Revised Papers*, volume 3017 of *LNCS*, pages 371–388. Springer, 2004.

[73] P. Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *ACM CCS'99, Proceedings*, pages 93–100. ACM, 1999.

[74] J. Rompel. One-way functions are necessary and sufficient for secure signatures. In *22nd ACM STOC, Proceedings*, pages 387–394. ACM, 1990.

[75] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information Systems Security*, 2(2):159–176, 1999.

[76] B. Schoenmakers. Explicit optimal binary pebbling for one-way hash chain reversal. In *FC 2016, Revised Selected Papers*, volume 9603 of *LNCS*, pages 299–320. Springer, 2017.

[77] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

[78] V. Stathopoulos, P. Kotzanikolaou, and E. Magkos. A framework for secure and verifiable logging in public communication networks. In *Proceedings of the First International Conference on Critical Information Infrastructures Security*, pages 273–284. Springer, 2006.

[79] M. Szydlo. Merkle tree traversal in log space and time. In *EUROCRYPT 2004, Proceedings*, volume 3027 of *LNCS*, pages 541–554. Springer, 2004.

[80] A. Truu and H. Ivanov. On using testing-related tasks in the IOI. In *Olympiads in Informatics: The International Conference joint with the XX International Olympiad in Informatics*, *Proceedings*, pages 171–180. Institute of Mathematics and Informatics, Vilnius, Lithuania, 2008.

[81] D. Unruh. Collapse-binding quantum commitments without random oracles. In *ASIACRYPT 2016*, *Proceedings*, *Part II*, volume 10032 of *LNCS*, pages 166–195. Springer, 2016.

[82] D. Unruh. Computationally binding quantum commitments. In *EUROCRYPT 2016*, *Proceedings*, *Part II*, volume 9666 of *LNCS*, pages 497–527. Springer, 2016.

Acknowledgements

Neither this thesis nor the underlying research work would have been possible without the generous support I have received from many parties over the years.

First and foremost, I'm indebted to my advisor Ahto Buldas from whom I have learned most of what I know about cryptographic research.

Next I thank my colleagues in GuardTime; in addition to my co-authors Risto Laanoja, Henri Lakk and Denis Firsov, discussions with Hema Krishnamurthy, Risto Alas, Andres Ojamaa, Andres Kroonmaa and Peeter Omler haven given many useful insights.

I also appreciate the efforts of management of GuardTime in supporting a dedicated research group in a relatively small enterprise.

As the vice-dean of research at IT Faculty of Tallinn University of Technology, Maarja Kruusmaa has done an enormous job at modernizing the whole Ph.D. program and at pushing the students towards graduation.

I'm grateful for the scholarship funded by Estonian Information Technology Foundation and Skype Technologies that I received during my first year of graduate studies and the financial support from the European Regional Development Fund through the Estonian smart specialization program NUTIKAS.

I would also like to thank the reviewers Andreas Hülsing and Elena Andreeva whose comments on the pre-print of this thesis helped me significantly improve the presentation of the material for publication.

Finally, I owe special gratitude to my wife Piret who has not only tolerated my erratic schedule over all these years, but also patiently proof-read most of my writings. Obviously, any remaining mistakes are my own responsibility, as indeed were the ones she did find.

Abstract
Hash-Based Server-Assisted Digital Signature Solutions

Digital signatures are one of the cornerstones of a modern information society. All the signature schemes in wide use today (RSA, DSA, and ECDSA) are known to be vulnerable to quantum attacks, so it is of interest to look for post-quantum secure alternatives.

In this work, we consider signature schemes built from hash functions, which are widely believed to be much more resilient to quantum attacks. We propose several such schemes and analyze their security in terms of security properties of the underlying hash function. A novel design element of the schemes is their reliance on time-stamping as an inherent component.

One of the proposed schemes is particularly suitable for use in server applications that need to produce a lot of signatures. The scheme features efficient signing and verification and small signatures. Only the key generation is relatively expensive, but still tolerable for full-sized computers. A benefit for the scheme is that it's stateless: the key to be used is determined by the signing time and thus the signer does not need to explicitly track spent keys and also access to the private key material does not need to be synchronized in a parallel execution environment.

Another scheme is optimized for personal signing devices that are used only occasionally. In addition to small signatures and efficient signing and verification, also key generation costs are relatively low. The price for this efficiency is the introduction of state on the client side. However, typically personal key management devices, such as smart cards and USB tokens, are not backed up and do not support multi-processing, and thus the risks related to managing key state are significantly reduced.

We also propose a hash-function based aggregation scheme for efficient yet privacy-protecting signing of audit logs. The scheme allows log records to be signed in large batches to reduce the computational and storage overheads, but at the same time the integrity of any record can be proven without leaking any information about the contents of other records in the log.

Kokkuvõte
Räsifunktsioonidel põhinevad serveri toega digitaalse signeerimise lahendused

Turvalised ja usaldusväärsed digiallkirjad on üks tänapäeva infoühiskonna nurgakive. Samas on teada, et kõik praegu laialt kasutatavad digiallkirjade skeemid (RSA, DSA ja ECDSA) muutuvad praktiliste kvantarvutite saabumisel parandamatult ebaturvaliseks. Seega tuleb otsida alternatiivseid lahendusi.

Räsifunktsioonid on praeguste teadmiste kohaselt kvantrünnetele märksa vastupidavamad. Käesolevas töös pakumegi välja mitu räsifunktsioonidel põhinevat digiallkirjade skeemi ja uurime nende turvaomadusi. Uuritavate skeemide ühine uudne element on ajatembelduse kasutamine allkirjaskeemi komponendina.

Üks uutest skeemidest on sobiv kasutamiseks serverirakendustes, kus on vaja anda palju allkirju. Pakutud skeemi tugevused on signeerimise ja verifitseerimise algoritmide efektiivsus ja signatuuride väiksus. Võtmete genereerimine on üsna töömahukas, aga täismõõdulistele arvutitele siiski jõukohane. Skeem on ka olekuvaba: iga uue signatuuri loomiseks kasutatav võtmekomponent on üheselt määratud signeerimise ajaga; seega pole vaja pidada arvet selle üle, millised võtmekomponendid on juba ära kasutatud; samuti pole rööpsüsteemides vaja võtmekomponentide kasutust erinevate protsesside vahel sünkroonida.

Teine pakutud skeem on optimeeritud rakendamiseks personaalsetes signeerimisseadmetes, mida kasutatakse vaid aeg-ajalt. Ka selles skeemis on signatuurid väikesed ning signeerimine ja verifitseerimine efektiivsed. Lisaks on selles skeemis ka võtmete genereerime märksa odavam, kuid see on saavutatud olekuvabadusest loobumise hinnaga. Oleku haldusega seotud riske maandavad selles kontekstis asjaolud, et personaalseid signeerimisseadmeid (näiteks kiipkaarte ja USB krüptopulki) ei varundata ja need ei toeta ka rööptöötlust.

Lisaks pakume välja räsifunktsioonidel põhineva agregeerimisskeemi logide efektiivseks kuid privaatsust säilitavaks signeerimiseks. Arvutus- ja salvestusmahtude kokkuhoiuks signeeritakse logikirjeid suurte pakkidena; samas on võimalik üksikute kirjete terviklust tõestada ilma teiste kirjete sisu lekitamata.

Appendix 1

Publication I
A. Buldas, R. Laanoja, and A. Truu. A server-assisted hash-based signature scheme. In *NordSec 2017*, *Proceedings*, volume 10674 of *LNCS*, pages 3–17. Springer, 2017

A Server-Assisted Hash-Based Signature Scheme

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

[1] Tallinn University of Technology, Akadeemia tee 15a, 12618 Tallinn, Estonia
[2] Guardtime AS, A.H. Tammsaare tee 60, 11316 Tallinn, Estonia
ahto.truu@guardtime.com

Abstract. We present a practical digital signature scheme built from a cryptographic hash function and a hash-then-publish digital time-stamping scheme. We also provide a simple proof of existential unforgeability against adaptive chosen-message attack (EUF-ACM) in the random oracle (RO) model.

1 Introduction

All the digital signature schemes in use today (RSA [42], DSA [22], ECDSA [30]) are known to be vulnerable to quantum attacks by Shor's algorithm [46]. While the best current experimental results are still toy-sized [35], it takes a long time for new cryptographic schemes to be accepted and deployed, so it is of considerable interest to look for post-quantum secure alternatives already now. Error-correcting codes, discrete lattices, and multi-variate polynomials have been used as foundations for proposed replacement schemes [4]. However, these are relatively complex structures and new constructions in cryptography, so require significant additional scrutiny before gaining trust.

Hash functions, on the other hand, have been studied for decades and are widely believed to be quite resistant to quantum attacks. The best currently known quantum results against hash functions are using Grover's algorithm [25] to find a pre-image of a given k-bit value in $2^{k/2}$ queries instead of the 2^k queries needed by a classical attacker, and Brassard et al.'s modification [7] to find a collision in $2^{k/3}$ instead of $2^{k/2}$ queries. To counter these attacks, it would be sufficient to deploy hash functions with correspondingly longer outputs when moving from pre-quantum to post-quantum setting.

2 Related Work

The earliest digital signature scheme constructed from hash functions is due to Lamport [19,31]. Merkle [37] introduced two methods for reducing the key sizes, one proposed to him by Winternitz. The Winternitz scheme has subsequently been more thoroughly analyzed and further refined by Even et al. [23],

This research was supported by the European Regional Development Fund through the Estonian smart specialization program NUTIKAS.

Dods et al. [21], Buchmann et al. [9], and Hülsing [27]. All of these schemes are *one-time*, and require generation of a new key pair and distribution of a new public key for each message to be signed.

Merkle's arguably most important contribution in [37] was the concept of *hash tree*, which enables a large number of public keys to be represented by a single hash value. With the hash value published, any one of the N public keys can be shown to belong to the tree with a proof consisting of $\log_2 N$ hash values, thus combining N instances of a one-time scheme into an N*-time* scheme. Buldas and Saarepera [16] and Coronado García [17] showed the aggregation to be secure if the hash function used to build the tree is collision resistant. Rohatgi [43] used the XOR-tree construct proposed by Bellare and Rogaway [3] to create a variant of hash tree whose security is based on second pre-image resistance of the hash function instead of collision resistance. Dahmen et al. [18] proposed a similar idea with a more complete security proof.

A drawback of the above hash tree constructs is that the whole tree has to be built at once, which also means all the private keys have to be generated at once. Merkle [38] proposed a certification tree that allows just the root node of the tree to be populated initially and the rest of the tree to be grown gradually as needed. However, to authenticate the lower nodes of the tree, a chain of full-blown one-time signatures (as opposed to a chain of sibling hash values) is needed, unless the protocol is used in an interactive environment where the recipient keeps the public keys already delivered as part of earlier signatures. Malkin et al. [34] and Buchmann et al. [8,11] proposed various multi-level schemes where the keys authenticated by higher-level trees are used to sign roots of lower-level trees to enable the key sets to be expanded incrementally.

Buchmann et al. [10] proposed XMSS, a version of the Merkle signature scheme with improved efficiency compared to previous ones. Hülsing et al. [28] introduced a multi-tree version of it. Hülsing et al. [29] described a modification hardened against so-called multi-target attacks where the adversary will succeed when it can find a pre-image for just one of a large number of target output values of a hash function.

A risk with the N-time schemes is that they are *stateful*: as each of the one-time keys may be used only once, the signer will need to keep track of which keys have already been used. If this state information is lost (for example, when a previous state is restored from a backup), keys may be re-used by accident.

Perrig [39] proposed BiBa which has small signatures and fast verification, but rather large public keys and slow signing. Reyzin and Reyzin [41] proposed the HORS scheme that provides much faster signing than BiBa. These two are not strictly one-time, but so-called *few-time* schemes where a private key can be used to sign several messages, but the security level decreases with each additional use. Bernstein et al. [5] proposed SPHINCS, which combines HORS with XMSS trees to create a *stateless* scheme that uses keys based on a pseudo-random schedule that makes the risk of re-use negligible even without tracking the state.

3 Our Contribution

We propose a signature scheme with a hash function as its sole underlying primitive. At the time of writing, XMSS and SPHINCS are the state of the art in the stateful and stateless hash signature schemes, respectively, so these are what new schemes should be measured against.

XMSS has fast signing and verification, and small signatures, but requires careful management of key state [36]. Our scheme has comparable efficiency, but the private key to be used is determined by signing time, which removes the risk of accidental roll-backs. Also, a single private key can be used to sign multiple messages simultaneously, so no synchronization is required when the scheme is deployed in multi-threaded or multi-processor environments.

SPHINCS has small keys and efficient verification, but quite large signatures and rather expensive signing. Our scheme requires orders of magnitude less computations for signing and produces signatures roughly a tenth the size.

A more general feature is that each signature produced by our scheme is inherently time-stamped. Most other schemes require time-stamping as a separate step after signing to handle key expirations, key revocations, and time-limited signing authority. Due to the time-stamping component, our scheme is necessarily server-assisted. While this may look like a disadvantage, it may in fact be beneficial in enforcing various key usage policies and limiting damage in case of a key leakage. For these reasons, even the technically off-line schemes are usually deployed within on-line frameworks in practice.

4 Preliminaries

Hash Trees. Introduced by Merkle [37], a hash tree is a tree-shaped data structure built using a 2-to-1 hash function $h\colon \{0,1\}^{2k} \to \{0,1\}^k$. The nodes of the tree contain k-bit values. Each node is either a leaf with no children or an internal node with two children. The value x of an internal node is computed as $x \leftarrow h(x_l, x_r)$, where x_l and x_r are the values of the left and right child, respectively. There is one root node that is not a child of any node. We will use $r \leftarrow T^h(x_1, \ldots, x_N)$ to denote a hash tree whose N leaves contain the values $x_1, \ldots, x_N$ and whose root node contains r.

Hash Chains. In order to prove that a value x_i participated in the computation of the root hash r, it is sufficient to present values of all the siblings of the nodes on the unique path from x_i to the root in the tree. For example, to claim that x_3 belongs to the tree shown on the left in Fig. 1, one has to present the values x_4 and $x_{1,2}$ to enable the verifier to compute $x_{3,4} \leftarrow h(x_3, x_4)$, $r \leftarrow h(x_{1,2}, x_{3,4})$, essentially re-building a slice of the tree, as shown on the right in Fig. 1. We will use $x \overset{c}{\rightsquigarrow} r$ to denote that the hash chain c links x to r in such a manner.

Intuitively, it seems obvious that if the function h is one-way, the existence of such a chain whose output equals the original r is a strong indication that x was indeed the original input. However, this result was not formally proven until 25 years after the hash tree construct was proposed [16,17].

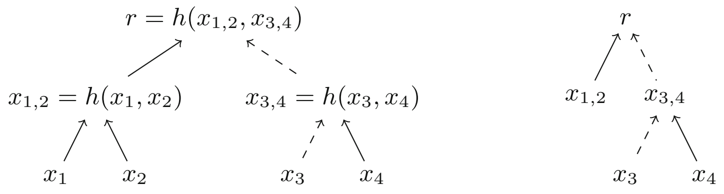

Fig. 1. The hash tree $T^h(x_1, \ldots, x_4)$ and the corresponding hash chain $x_3 \rightsquigarrow r$.

Hash-Then-Publish Time-Stamping. The general idea of time-stamping information by publishing its hash value was used already by Galilei and Hooke in the XVII century. In more modern cryptographic times, Haber and Stornetta [26] were the first to propose time-stamping a sequence of records by having each of them contain the hash of the previous one, in a manner that was later popularized as the *blockchain* structure. Bayer et al. [2] proposed using hash trees to aggregate the inputs in batches and then linking the roots of the trees instead of individual records. The most recent results on security bounds of such schemes are by Buldas et al. [13–15].

5 Description of the Scheme

The principal idea of our signature scheme is to have the signer commit to a sequence of keys such that each key is assigned a time slot when it can be used to sign messages and will transition from signing key to verification key once the time slot has passed.

Signing itself then consists of time-stamping the message-key pair in order to prove that the signing operation was performed at the correct time. For simplicity of presentation, we count time in aggregation rounds of the time-stamping service and use the expression "at time t" to mean "during aggregation round t".

More formally, the classic triple of procedures for key generation, signature generation, and signature verification [24] is as follows:

Key Generation. To prepare to sign messages at times $1, \ldots, N$, the signer:

1. Generates N signing keys: $(z_1, \ldots, z_N) \leftarrow \mathcal{G}(N, k)$.
   We assume the keys are unpredictable values drawn from $\{0, 1\}^k$.
2. Binds each key to its time slot: $x_i \leftarrow h(i, z_i)$ for $i \in \{1, \ldots, N\}$.
3. Computes the public key p by aggregating the key bindings into a hash tree: $p \leftarrow T^h(x_1, \ldots, x_N)$.

The resulting data structure is shown in Fig. 2 and its purpose is to be able to extract hash chains $c_i \leftarrow h(i, z_i) \rightsquigarrow p$ for $i \in \{1, \ldots, N\}$.

$$\begin{tikzcd}
 & & & p & & & & \\
 & x_{1,2} \arrow[urr] & & & & x_{3,4} \arrow[ull] & & \\
x_1 \arrow[ur] & & x_2 \arrow[ul] & & x_3 \arrow[ur] & & x_4 \arrow[ul] & \\
1 \arrow[u] & z_1 \arrow[ul] & 2 \arrow[u] & z_2 \arrow[ul] & 3 \arrow[u] & z_3 \arrow[ul] & 4 \arrow[u] & z_4 \arrow[ul]
\end{tikzcd}$$

Fig. 2. Computation of public key for $N = 4$.

Signing. To sign message m at time t, the signer:

1. Uses the appropriate key to authenticate the message: $y \leftarrow h(m, z_t)$.
2. Time-stamps the authenticator: $a_t \leftarrow y \rightsquigarrow r_t$.
   Here r_t is the root hash of the aggregation tree built by the time-stamping service for the aggregation round t. We assume the root is committed to in some reliable way, such as broadcasting it to all interested parties, but place no other trust in the service.
3. Outputs the tuple (t, z_t, a_t, c_t), where t is the signing time, z_t is the signing key for time slot t, a_t is the hash chain from the time-stamping service linking the key usage to r_t, and c_t is the hash chain linking the binding of z_t and time slot t to the signer's public key p.

Note that the signature is composed and emitted after the time-stamping step, which makes it safe for the signer to release the key z_t as part of the signature: the aggregation round t has ended and any future uses of the key z_t can no longer be stamped with time t.

Verification. To verify that the message m and the signature $s = (t, z, a, c)$ match the public key p, the verifier:

1. Checks that z was committed as signing key for time t: $h(t, z) \overset{c}{\rightsquigarrow} p$.
2. Checks that m was authenticated with key z at time t: $h(m, z) \overset{a}{\rightsquigarrow} r_t$.

6 Security Proof

Goldwasser et al. [24] proposed a framework for studying security of signature schemes where the attackers have various levels of access to signing oracles and various requirements on what they need to achieve for the attack to be considered successful (and the scheme broken).

As the highest security level, they defined the concept of *existential unforgeability* (EUF) where an attacker should be unable to forge signatures on any messages, even nonsensical ones.

They also defined the *chosen-message attack* where the attacker can submit a number of messages to be signed by the oracle before having to come up with a forged signature on a new message, and in particular, as the one giving the attacker the most power, the *adaptive chosen-message attack* (ACM) where the attacker will receive each signature immediately after submitting the message and can use any information gained from previous signatures to form subsequent messages.

Luby [33] defined the *time-success ratio* as a way to express the resilience of a cryptographic scheme against attacks as the relationship of the probability that the attack will succeed to the computation time the attacker is allowed to spend.

We will now combine these notions to define and prove the security of our signature scheme.

Definition 1. *A signature scheme is S-secure existentially unforgeable against adaptive chosen-message attacks (EUF-ACM), if any T-time adversary, having access to a signer's public key p and to a signing oracle $\mathcal{S}$ to obtain signatures $s_1 \leftarrow \mathcal{S}(m_1)$, ..., $s_n \leftarrow \mathcal{S}(m_n)$ on adaptively chosen messages m_1, ..., m_n, can produce a new message-signature pair (m, s) such that $m \notin \{m_1, \ldots, m_n\}$, but s is a valid signature on m, with probability at most T/S.*

```
Oracle $\mathcal{S}$ (signing oracle)
Query Sig$(m, t)$:
    return $h(m, z_t)$
Query Get$(t)$:
    If $c \geq t$ then:
        return $(z_t, x_t \rightsquigarrow p)$
    else:
        return $\perp$
```

```
Oracle $\mathcal{R}$ (repository)
Initialize:
    $c \leftarrow 0$
Query Put$(r)$:
    $c \leftarrow c + 1$
    $r_c \leftarrow r$
Query Get$(t)$:
    If $c \geq t$ then:
        return $r_t$
    else:
        return $\perp$
```

Fig. 3. The oracles used in the security condition.

To formalize our security assumptions, we introduce three oracles:

We model the publishing of the root hashes of the time-stamping aggregation trees as the oracle $\mathcal{R}$ (Fig. 3, right) that allows each r_t to be published just once.

The signing oracle $\mathcal{S}$ (Fig. 3, left) will compute the message authenticators at any time, but will release only the keys that have already expired for signing (transitioned to verification keys).

We model the hash function h as a random oracle using the *lazy sampling* technique: every time h is queried with a previously unseen input, a new return value is generated by uniform random sampling from $\{0, 1\}^k$; when h is queried with a previously seen input, the same value is returned as last time.

The adversary $\mathcal{A}$ will be interacting with the oracles as shown in Fig. 4 with the goal of producing a forgery.

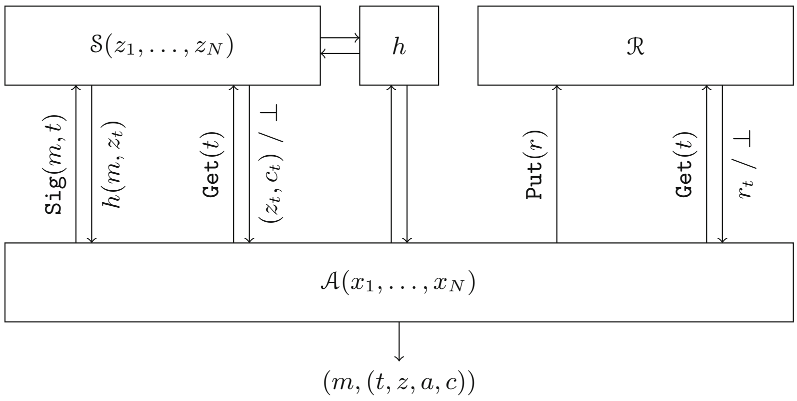

Fig. 4. The adversary's interactions with the oracles.

To model the fact that the signer needs to keep secret only the keys $z_1,\ldots,z_N$, we explicitly initialize the adversary with $x_1,\ldots,x_N$. Note that the verification rule still assumes that the verifier has access only to the signer's public key p, which means the adversary is not limited to presenting hash chains that were actually extracted from $T^h(x_1,\ldots,x_N)$.

Also note that we leave the aggregation process of the time-stamping service fully under the adversary's control; only the repository $\mathcal{R}$ needs to be trusted to operate correctly.

As normally signing message m involves first calling $\mathcal{S}.\texttt{Sig}(m,t)$, then committing to $\mathcal{R}$ the root of a hash tree that includes the return value, and then calling $\mathcal{S}.\texttt{Get}(t)$, we formalize the forgery condition by demanding that the adversary can't make the two $\mathcal{S}$ calls in that order:

Definition 2. *The pair (m,s) produced by an adversary is a successful forgery if s is a valid signature on m, but the adversary did not make the calls $\mathcal{S}.\texttt{Sig}(m,t)$, $\mathcal{S}.\texttt{Get}(t)$, in that order, for any $t\in\{1,\ldots,N\}$.*

Theorem 1. *Our signature scheme, when instantiated with a hash function $h\colon\{0,1\}^{2k}\to\{0,1\}^k$ indistinguishable from a random oracle, is at least $2^{(k-2)/2}$-secure existentially unforgeable against adaptive chosen-message attacks by any T-time adversary.*

Proof. We will directly show an upper bound on the success probability of the adversary in the forgery game $\mathcal{F}$ (Fig. 5).

Assume that the adversary does not call $\mathcal{S}.\texttt{Get}(t)$. To win the game $\mathcal{F}$, he must produce t, z, c such that $h(t,z)\overset{c}{\rightsquigarrow}p$. For that, the output of the last step of the chain computation must equal the root of the tree $T^h(x_1,\ldots,x_N)$. Let's now consider the inputs to that step. If they equal the corresponding children of the root of the tree, we can repeat the reasoning for the second last step and the corresponding node of the tree, and so on. As we walk a finite chain and

```
Game $\mathcal{F}$ (forgery)
$(z_1, \ldots, z_N) \leftarrow \mathcal{G}(N)$
$x_i \leftarrow h(i, z_i)$ for $i \in \{1, \ldots, N\}$
$p \leftarrow T^h(x_1, \ldots, x_N)$
$\boxed{(m, (t, z, a, c)) \leftarrow \mathcal{A}^{h,\mathcal{S},\mathcal{R}}(x_1, \ldots, x_N)}$
If $\mathcal{A}$ did not call $\mathcal{S}.\mathtt{Sig}(m, t)$, $\mathcal{S}.\mathtt{Get}(t)$,
    but $h(t, z) \overset{c}{\rightsquigarrow} p$ and $h(m, z) \overset{a}{\rightsquigarrow} r_t$
then:
    return 1
else:
    return 0
```

Fig. 5. The forgery game.

simultaneously traverse a finite tree from the root towards leaves, one of the following events must eventually happen:

1. We run out of the chain at the same time we run out of the tree. This means the adversary has found t and z such that $x_i = h(t, z)$ for some $i \in \{1, \ldots, N\}$. If $i \neq t$, then the adversary has found a second pre-image for the x_i originally computed as $h(i, z_i)$. With h being a random oracle, the probability of a T-time adversary achieving that for any given i is $\leq T/2^k$. If $i = t$, then the adversary may have found a second pre-image for x_t, with probability $\leq T/2^k$, or may have guessed z_t, also with probability $\leq T/2^k$. Thus the total probability of $h(t, z)$ matching a leaf of the tree is $\pi_{A,1} \leq (N + 1)T/2^k$.
2. We run out of the chain before we run out of the tree. This means $h(t, z)$ matches one of the internal nodes of the tree, say x. This can be the case in two ways:
   (a) The left child of x contains t and the adversary uses the right child of x as z. The probability of any given node having the given value t is $1/2^k$. As there are $N - 1$ candidate nodes and N possible values of t, the total probability is $\leq (N - 1)N/2^k$.
   (b) The adversary has found a second pre-image for x. The probability of a T-time adversary achieving that for any given node is $\leq T/2^k$. As the adversary has $N - 1$ nodes as potential targets for such a hit, the total probability is $\leq (N - 1)T/2^k$.

   Thus the total probability of $h(t, z)$ matching an internal node of the tree is $\pi_{A,2} \leq (N - 1)(N + T)/2^k$.
3. We run out of the tree before we run out of the chain. This means that the adversary has found a pre-image for one of the $2N$ values $\{1, z_1, \ldots, N, z_N\}$. The probability of that is $\pi_{A,3} \leq 2NT/2^k$.
4. We encounter a hash step where the output of the step equals an internal node in the tree, say x, but the inputs of the step do not match the children of x. This means the adversary has found a second pre-image for x. The probability of that is $\pi_{A,4} \leq (N - 1)T/2^k$.

So, the total success probability of a T-time adversary who does not call $\mathcal{S}.\mathtt{Get}(t)$ is $\pi_A \le \pi_{A,1}+\pi_{A,2}+\pi_{A,3}+\pi_{A,4} \le (N+1)T/2^k+(N-1)(N+T)/2^k+2NT/2^k+(N-1)T/2^k < (N^2+5NT)/2^k$.

Assume now that the adversary does call $\mathcal{S}.\mathtt{Get}(t)$. Then we can, without loss of generality, also assume that

– he calls $\mathcal{S}.\mathtt{Get}(t)$ only after committing r_t, as before that $\mathcal{S}.\mathtt{Get}(t)$ would always return $\perp$, which would provide no useful information;
– he calls $\mathcal{S}.\mathtt{Get}(t)$ only once, as all additional calls to $\mathcal{S}.\mathtt{Get}(t)$ would return the same result, which would provide no new information;
– he never calls $\mathcal{S}.\mathtt{Sig}(m,t)$, as he is not allowed to call $\mathcal{S}.\mathtt{Sig}(m,t)$ before calling $\mathcal{S}.\mathtt{Get}(t)$ according to the security condition, but after calling $\mathcal{S}.\mathtt{Get}(t)$ he already has z_t and can compute $h(m,z_t)$ directly with no need to call the signing oracle any more.

Finally, we can also assume that in order to win the game $\mathcal{F}$, the adversary must produce m and a such that $h(m,z_t) \overset{a}{\rightsquigarrow} r_t$. Indeed, if the adversary wins the game with $h(m,z) \overset{a}{\rightsquigarrow} r_t$ where $z \ne z_t$, then he has not used the information gained from the $\mathcal{S}.\mathtt{Get}(t)$ call and thus could not have done any better than without the call, a case we have already analyzed.

Let H be the set of h-calls $y \leftarrow h(x_1,x_2)$ the adversary made before committing r_t. As the adversary is T-time, we have $|H| \le T$. Consider now the h-calls to be made during the computation of $h(m,z_t) \overset{a}{\rightsquigarrow} r_t$:

1. If all the calls are in H, then the adversary must have called $h(m,z_t)$ before committing r_t and thus also before learning z_t from the call to $\mathcal{S}.\mathtt{Get}(t)$. This means that the adversary guessed z_t. The probability of a T-time adversary achieving that is $\pi_{B,1} \le T/2^k$.
2. If none of the calls are in H, then there are two possibilities:
   (a) The value r_t was not returned from any of the calls in H. This means the adversary was able to find a pre-image of r_t after committing it, the probability of which is $\le T/2^k$.
   (b) The value r_t was returned by some call in H. Since the chain a is computed entirely using calls not in H, the inputs of the final step of the computation represent a second pre-image of r_t. The probability of a T-time adversary achieving that is also $\le T/2^k$.

   Thus the total probability of the adversary finding a chain entirely outside of H is $\pi_{B,2} \le 2T/2^k$.
3. Some, but not all of the calls are in H. Let's examine, among the calls that are not in H, the one made last during the computation of the chain. Let it be $y \leftarrow h(x_1,x_2)$. Again, there are two possibilities:
   (a) The value y was not returned from any of the calls in H. However, the next step in a is already a call in H. This means that y is among the inputs of calls in H and the adversary was able to find a pre-image of it. The probability of the adversary achieving that for any given y is $\le T/2^k$. As there are $2|H|$ possible values of y, the total probability is $\le 2|H|T/2^k$.
   (b) The value y was returned by some call in H. Since the call $y \leftarrow h(x_1,x_2)$ is not in H, the adversary must have found a second pre-image of y. The total probability of that over all available values of y is $\le |H|T/2^k$.

Thus the probability of the adversary finding a chain entering into H is $\pi_{B,3} \leq 3|H|T/2^k \leq 3T^2/2^k$.

Hence the total success probability of a T-time adversary who calls $\mathcal{S}.\mathtt{Get}(t)$ is $\pi_B \leq \pi_{B,1} + \pi_{B,2} + \pi_{B,3} \leq T/2^k + 2T/2^k + 3T^2/2^k = (3T + 3T^2)/2^k$.

Summary. If the adversary does not call $\mathcal{S}.\mathtt{Get}(t)$, he can win the forgery game $\mathcal{F}$ with probability $\pi_A < (N^2 + 5NT)/2^k$. If he does call $\mathcal{S}.\mathtt{Get}(t)$, he can win with probability $\pi_B \leq (3T + 3T^2)/2^k$. Overall, he can win with probability $\pi = \max(\pi_A, \pi_B)$.

Since generating the N keys $z_1, \ldots, z_N$ and making the $2N - 1$ calls to h to compute $x_1, \ldots, x_N$ and $T^h(x_1, \ldots, x_N)$ is something the signers are expected to do routinely, we can assume that $N \ll T$. Already with $N < T/10$, we have $\pi_A < (N^2 + 5NT)/2^k < (T^2/100 + T^2/2)/2^k < T^2/2^k$. With $T > 10N \geq 10$, we have $T^2 > 3T$ and thus $\pi_B \leq (3T + 3T^2)/2^k < 4T^2/2^k$.

Therefore, $\pi = \max(\pi_A, \pi_B) < 4T^2/2^k$, or $T^2/\pi > 2^{k-2}$. As $\pi \leq 1$, we also have $(T/\pi)^2 \geq T^2/\pi$, which yields the claim $T/\pi > 2^{(k-2)/2}$.

7 Practical Considerations

Key Generation. In the description of the scheme we assumed that the signing keys $z_1, \ldots, z_N$ are unpredictable values drawn from $\{0, 1\}^k$, but left unspecified how they might be generated in practice. Obviously they could be generated as independent truly random values, but this would be rather expensive and also would necessitate keeping a large number of secret values over a long time. It would be more practical to generate them pseudo-randomly from a single random seed s. There are several known ways of doing that:

– Iterated hashing: $z_N \leftarrow s$, $z_{i-1} \leftarrow h(z_i)$ for $i \in \{2, \ldots, N\}$.
  This idea of generating a sequence of one-time keys from a single seed is due to Lamport [32] and has also been used in the TESLA protocol by Perrin et al. [40]. Implemented this way, our scheme would also bear some resemblance to the Guy Fakes protocol by Anderson et al. [1]. Note that the keys have to be generated in reverse order, otherwise the earlier keys released as signature components could be used to derive the later ones that are still valid for signing. To be able to use the keys in the direct order, the signer would have to either remember them all, re-compute half of the sequence on average, or implement a traversal algorithm such as the one proposed by Schoenmakers [45].
– Counter hashing: $z_i \leftarrow h(s, i)$.
  With a hash function behaving as a random oracle, this scheme would generate keys indistinguishable from truly random values, but there does not appear to be much research on the security of practical hash functions when used in this mode.
– Counter encryption: $z_i \leftarrow E_s(i)$.
  The signing keys are generated by encrypting their indices with a symmetric block cipher using the seed as the encryption key. This is equivalent to

using the block cipher in the counter mode as first proposed by Diffie and Hellman [20]. The security of this mode is extensively studied and well understood for all common block ciphers. Another benefit of this approach is that it can be implemented using standard hardware security modules where the seed is kept in a protected storage and the encryption operations are performed in a security-hardened environment.

Time-Stamping. As already mentioned, we side-step the key state management problems [36] common for most N-time signing schemes by making the signing keys not one-time, but *time-bound* instead. This in turn raises the issue of clock synchronization.

We first note that even when the signer's local clock is running fast, premature key release is easy to prevent by having the signer verify the time-stamp on $h(m, z_t)$ before releasing z_t. This is how the condition $c \geq t$ of the signing oracle $\mathcal{S}$ in Fig. 3 should be implemented in practice.

The next issue is that the signer needs to select the key z_t before computing $h(m, z_t)$ and submitting it to time-stamping. If, due to clock drift or network latency, the time in the time-stamp received does not match t, the signature can't be composed. To counter clock drift and stable latency, the signer can first time-stamp a dummy value and use the result to compare its local clock to that of the time-stamping service.

To counter network jitter, the signer can compute the message authenticators $h(m, z_{t'})$ for several consecutive values of t', submit all of them in parallel, and compose the signature using the components whose t' matches the time t in the time-stamps received. Buldas et al. [12] have shown that with careful scheduling the latency can be made stable enough for this strategy even in an aggregation network with world-wide scale.

Finally, we note that time-stamping services operating in discrete aggregation rounds are particularly well suited for use in our scheme, as they only return time-stamps once the round is closed, thus eliminating the risk that a fast adversary could still manage to acquire a suitable time-stamp after the signer has released a key.

Efficiency. In the following estimates, we assume the use of SHA-256, a common 256-bit hash function. On small inputs, a moderate laptop can perform about a million SHA-256 evaluations per second. We also assume a signing key sequence containing one key per second for a year, or a total of a bit less than 32 million, or roughly 2^{25} keys.

Using the techniques described above, generation of N signing keys takes N applications of either a hash function or a symmetric block cipher. Binding them into a public key takes $2N - 1$ hashing operations. Thus, the key generation in our example takes about 100 seconds.

The resulting public key consists of just one hash value. In the private key, only the seed s has to be kept secret. The signing keys $z_1, \ldots, z_N$ can be erased once the public key has been computed, an then re-generated as needed for signing. The hash tree $T^h(x_1, \ldots, x_N)$ presents a space-time trade-off. It may be kept (in regular unprotected storage, as it contains no sensitive information), taking up $2N - 1$ nodes, or about 1 GB, and then the key authentication hash chains can be just read from the tree with no additional computations needed. Alternatively, one can use a hash tree traversal algorithm, such as the one proposed by Szydlo [47], to keep only $3 \log_2 N$ nodes of the tree and spend $2 \log_2 N$ hash function evaluations per chain extraction, assuming all chains are extracted consecutively.

The size of the signature (t, z_t, a_t, c_t) is dominated by the two hash chains. The key authentication chain consists of $\log_2 N$ hash values, for a total of about 800 B for our 1-year key sequence. The time-stamping chain consists of $\log_2 M$ hash values, where M is the number of requests received by the time-stamping service in the round t. Assuming the use of the KSI service described in [12] under its theoretical maximum load of 2^{50} requests, this adds about 1 600 B. Thus we can expect signatures of less than 3 kB.

As the verification means re-computing the hash chains, it amounts to less than a hundred hash function evaluations.

8 Conclusions and Outlook

We have presented a simple and efficient digital signature scheme built from a hash function and a hash-then-publish time-stamping scheme. Considering that the existence of hash functions is a necessary pre-condition for the existence of digital signatures [44], one could argue our scheme is based on minimal assumptions. However, there is still much room for improvement in both theoretical and practical aspects.

Current security proofs are given in the random oracle model and in the classical setting. It would be desirable to prove the security also in the standard model and in the quantum setting, in particular taking into account the effects of quantum-oracle access to the hash function [6] and possible quantum interactions between the aggregation and the hash chain extraction phases of time-stamping, as these are all under the adversary's control.

It would also be good to reduce, or at least defer, the key generation costs, perhaps by adopting some of the incremental tree generation approaches, and to develop a version of the scheme suitable for personal signing devices like smart cards and USB dongles. These devices, in addition to having significantly less memory and computational power, also lack several functional qualities of the full-sized computers: they are powered on only intermittently, and do not have on-board real-time clocks or independent network communication capabilities.

References

1. Anderson, R.J., Bergadano, F., Crispo, B., Lee, J.-H., Manifavas, C., Needham, R.M.: A new family of authentication protocols. Oper. Syst. Rev. **32**(4), 9–20 (1998)

2. Bayer, D., Haber, S., Stornetta, W.S.: Improving the efficiency and reliability of digital time-stamping. In: Capocelli, R., De Santis, A., Vaccaro, U. (eds.) Sequences II, Proceedings. LNCS, vol. 9056, pp. 329–334. Springer, Heidelberg (1992). doi:10.1007/978-1-4613-9323-8_24
3. Bellare, M., Rogaway, P.: Random oracles are practical: a paradigm for designing efficient protocols. In: ACM CCS 1993, Proceedings, pp. 62–73. ACM (1993)
4. Bernstein, D.J., Buchmann, J.A., Dahmen, E. (eds.): Post-Quantum Cryptography. Springer, Heidelberg (2009). doi:10.1007/978-3-540-88702-7
5. Bernstein, D.J., et al.: SPHINCS: practical stateless hash-based signatures. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 368–397. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46800-5_15
6. Boneh, D., Dagdelen, Ö., Fischlin, M., Lehmann, A., Schaffner, C., Zhandry, M.: Random oracles in a quantum world. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 41–69. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25385-0_3
7. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. In: Lucchesi, C.L., Moura, A.V. (eds.) LATIN 1998. LNCS, vol. 1380, pp. 163–169. Springer, Heidelberg (1998). doi:10.1007/BFb0054319
8. Buchmann, J.A., Coronado García, L.C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS – an improved Merkle signature scheme. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 349–363. Springer, Heidelberg (2006). doi:10.1007/11941378_25
9. Buchmann, J.A., Dahmen, E., Ereth, S., Hülsing, A., Rückert, M.: On the security of the Winternitz one-time signature scheme. IJACT **3**(1), 84–96 (2013)
10. Buchmann, J.A., Dahmen, E., Hülsing, A.: XMSS - a practical forward secure signature scheme based on minimal security assumptions. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 117–129. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25405-5_8
11. Buchmann, J.A., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle signatures with virtually unlimited signature capacity. In: Katz, J., Yung, M. (eds.) ACNS 2007. LNCS, vol. 4521, pp. 31–45. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72738-5_3
12. Buldas, A., Kroonmaa, A., Laanoja, R.: Keyless signatures' infrastructure: how to build global distributed hash-trees. In: Nielson, H.R., Gollmann, D. (eds.) NordSec 2013. LNCS, vol. 8208, pp. 313–320. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41488-6_21
13. 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, vol. 7959, pp. 235–250. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39059-3_16
14. Buldas, A., Laanoja, R., Laud, P., Truu, A.: Bounded pre-image awareness and the security of hash-tree keyless signatures. In: Chow, S.S.M., Liu, J.K., Hui, L.C.K., Yiu, S.M. (eds.) ProvSec 2014. LNCS, vol. 8782, pp. 130–145. Springer, Cham (2014). doi:10.1007/978-3-319-12475-9_10
15. Buldas, A., Niitsoo, M.: Optimally tight security proofs for hash-then-publish time-stamping. In: Steinfeld, R., Hawkes, P. (eds.) ACISP 2010. LNCS, vol. 6168, pp. 318–335. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14081-5_20
16. Buldas, A., Saarepera, M.: On provably secure time-stamping schemes. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 500–514. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30539-2_35

17. Coronado García, L.C.: Provably secure and practical signature schemes. Ph.D. thesis, Darmstadt University of Technology, Germany (2005)
18. Dahmen, E., Okeya, K., Takagi, T., Vuillaume, C.: Digital signatures out of second-preimage resistant hash functions. In: Buchmann, J.A., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 109–123. Springer, Heidelberg (2008). doi:10.1007/978-3-540-88403-3_8
19. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inf. Theor. **22**(6), 644–654 (1976)
20. Diffie, W., Hellman, M.E.: Privacy and authentication: an introduction to cryptography. Proc. IEEE **67**(3), 397–427 (1979)
21. Dods, C., Smart, N.P., Stam, M.: Hash based digital signature schemes. In: Smart, N.P. (ed.) Cryptography and Coding 2005. LNCS, vol. 3796, pp. 96–115. Springer, Heidelberg (2005). doi:10.1007/11586821_8
22. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Inf. Theor. **31**(4), 469–472 (1985)
23. Even, S., Goldreich, O., Micali, S.: On-line/Off-line digital signatures. J. Cryptol. **9**(1), 35–67 (1996)
24. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM J. Comput. **17**(2), 281–308 (1988)
25. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: 28th ACM STOC, Proceedings, pp. 212–219. ACM (1996)
26. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. J. Cryptol. **3**(2), 99–111 (1991)
27. Hülsing, A.: W-OTS+ – shorter signatures for hash-based signature schemes. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 173–188. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38553-7_10
28. Hülsing, A., Rausch, L., Buchmann, J.A.: Optimal parameters for XMSS^{MT}. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E., Xu, L. (eds.) CD-ARES 2013. LNCS, vol. 8128, pp. 194–208. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40588-4_14
29. Hülsing, A., Rijneveld, J., Song, F.: Mitigating multi-target attacks in hash-based signatures. In: Cheng, C.-M., Chung, K.-M., Persiano, G., Yang, B.-Y. (eds.) PKC 2016. LNCS, vol. 9614, pp. 387–416. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49384-7_15
30. Johnson, D., Menezes, A., Vanstone, S.A.: The elliptic curve digital signature algorithm (ECDSA). Int. J. Inf. Secur. **1**(1), 36–63 (2001)
31. Lamport, L.: Constructing digital signatures from a one way function. Technical report, SRI International, Computer Science Laboratory (1979)
32. Lamport, L.: Password authentification with insecure communication. Commun. ACM **24**(11), 770–772 (1981)
33. Luby, M.: Pseudorandomness and Cryptographic Applications. Princeton University Press, Princeton (1996)
34. Malkin, T., Micciancio, D., Miner, S.: Efficient generic forward-secure signatures with an unbounded number of time periods. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 400–417. Springer, Heidelberg (2002). doi:10.1007/3-540-46035-7_27
35. Martín-López, E., Laing, A., Lawson, T., Alvarez, R., Zhou, X.-Q., O'Brien, J.L.: Experimental realization of Shor's quantum factoring algorithm using qubit recycling. Nat. Photonics **6**(11), 773–776 (2012)

36. McGrew, D., Kampanakis, P., Fluhrer, S., Gazdag, S.-L., Butin, D., Buchmann, J.A.: State management for hash-based signatures. In: Chen, L., McGrew, D., Mitchell, C. (eds.) SSR 2016. LNCS, vol. 10074, pp. 244–260. Springer, Cham (2016). doi:10.1007/978-3-319-49100-4_11
37. Merkle, R.C.: Secrecy, authentication and public key systems. Ph.D. thesis, Stanford University (1979)
38. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988). doi:10.1007/3-540-48184-2_32
39. Perrig, A.: The BiBa one-time signature and broadcast authentication protocol. In: ACM CCS 2001, Proceedings, pp. 28–37. ACM (2001)
40. Perrig, A., Canetti, R., Tygar, J.D., Song, D.: The TESLA broadcast authentication protocol. CryptoBytes **5**(2), 2–13 (2002)
41. Reyzin, L., Reyzin, N.: Better than BiBa: short one-time signatures with fast signing and verifying. In: Batten, L., Seberry, J. (eds.) ACISP 2002. LNCS, vol. 2384, pp. 144–153. Springer, Heidelberg (2002). doi:10.1007/3-540-45450-0_11
42. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
43. Rohatgi, P.: A compact and fast hybrid signature scheme for multicast packet authentication. In: ACM CCS 1999, Proceedings, pp. 93–100. ACM (1999)
44. Rompel, J.: One-way functions are necessary and sufficient for secure signatures. In: 22nd ACM STOC, Proceedings, pp. 387–394. ACM (1990)
45. Schoenmakers, B.: Explicit optimal binary pebbling for one-way hash chain reversal. In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 299–320. Springer, Heidelberg (2017). doi:10.1007/978-3-662-54970-4_18
46. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Rev. **41**(2), 303–332 (1999)
47. Szydlo, M.: Merkle tree traversal in log space and time. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 541–554. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24676-3_32

Appendix 2

Publication II
A. Buldas, R. Laanoja, and A. Truu. A blockchain-assisted hash-based signature scheme. In *NordSec 2018*, *Proceedings*, volume 11252 of *LNCS*, pages 138–153. Springer, 2018

A Blockchain-Assisted Hash-Based Signature Scheme

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

[1] Tallinn University of Technology, Akadeemia tee 15a, 12618 Tallinn, Estonia
[2] Guardtime AS, A. H. Tammsaare tee 60, 11316 Tallinn, Estonia
ahto.truu@guardtime.com

Abstract. We present a server-supported, hash-based digital signature scheme. To achieve greater efficiency than current state of the art, we relax the security model somewhat. We postulate a set of design requirements, discuss some approaches and their practicality, and finally reach a forward-secure scheme with only modest trust assumptions, achieved by employing the concepts of *authenticated data structures* and *blockchains*. The concepts of blockchain authenticated data structures and the presented blockchain design could have independent value and are worth further research.

1 Introduction

Buldas, Laanoja, and Truu [14] recently proposed a new type of signature scheme (which we will refer to as the *BLT scheme* in the following) based on the idea of combining one-time time-bound keys with a cryptographic time-stamping service. The scheme is post-quantum secure against known attacks and the integrity of the signatures does not depend on the secrecy of any keys. However, the keys have to be pre-generated for every possible signing time slot and this creates some implementation challenges. In particular, key generation on smart-cards would be prohibitively slow in real-world parameters.

In order to avoid the inherent inefficiency of pre-assigning individual keys to every time slot, we propose ways to spend such keys sequentially, one-by-one, as needed. This approach is particularly useful for real-world use-cases by human end-users where signing is performed in infrequent batches, e.g. paying monthly bills, and vast majority of the time-bound keys would go unused.

Sequential key use needs more elaborate support from the server. In particular, it is necessary to keep track of spent keys by both the signer and the server, and to avoid successful re-use of the spent keys. We will observe some ways to manage these keys sequentially and finally reach a solution where the server does not have to be trusted.

The proposed signature scheme can be considered practical. It provides forward security, non-repudiation of the origin via efficient revocation; there are no

This research was supported by the European Regional Development Fund through the Estonian smart specialization program NUTIKAS.

known attacks by quantum computers; it comes with "free" cryptographic time-stamping. Key and signature sizes and computational efficiency are comparable with state-of-the-art hash-based signature schemes. The scheme is stateful and maximum number of signatures created using a set of keys is determined at the key-generation time. Like other non-hierarchical hash-based signature schemes, the key generation time becomes noticeable when more than $\sim 2^{20}$ signatures have to be created using a set of keys.

The rest of the paper is organized as follows. In Sect. 2 we survey the state of the art in hash-based signature schemes, server-assisted signature schemes, and authenticated data structures. In Sect. 3 we define the design goals and outline the reasoning that led us to the new scheme. In Sect. 4 we specify the design of the new scheme and in Sect. 5 provide some notes on implementation. We wrap up with conclusions in Sect. 6.

2 Related Work

2.1 Hash-Based Signatures

The earliest signature scheme constructed from hash functions is due to Lamport [22,30]. His scheme, as well as the refinements proposed in [7,23,24,27,33], are *one-time*: they require generation of a new key pair and distribution of a new public key for each message to be signed.

Merkle [33] introduced the concept of *hash tree* which aggregates a large number of inputs into a single root hash value so that any of the N inputs can be linked to it with a proof consisting of $\log_2 N$ hash values. This allowed combining N instances of a one-time signature scheme into an N*-time* scheme. This approach has been further studied in [17,19,21,39]. A common drawback of these constructs is that the whole tree has to be built at once.

Merkle [34] proposed a method to grow the tree gradually as needed. However, to authenticate the lower nodes of the tree, a chain of one-time signatures (rather than a sequence of sibling hash values) is needed, unless the scheme is used in an interactive environment and the recipient keeps the public keys already delivered as part of earlier signatures. This multi-level approach has subsequently been refined in [6,8,9,28,29,32].

A complication with the N-time schemes is that they are *stateful*: as each of the one-time keys may be used only once, the signer has to keep track of them. If this information is lost (for example, when a previous state is restored from a backup), key re-use may result in a catastrophic security loss. Perrig [35] proposed a *few-time* scheme where a private key can be used to sign several messages, and the security level decreases gradually with each additional use.

Bernstein *et al.* [3] combined the optimized few-time scheme of [38] with the multi-level tree of [8] to create SPHINCS, a *stateless* scheme that uses keys based on a pseudo-random schedule, making the risk of re-use negligible even without tracking the state.

2.2 Server-Assisted Signatures

In server-assisted schemes the signer has to co-operate with a server to produce a signature. The two main motivations for such schemes are: (a) performance: costly computations can be offloaded from an underpowered signing device (such as a smart-card) to a more capable computer; and (b) security: risks of key misuse can be reduced by either keeping the keys in a server environment (which can presumably be managed better than an end-user's personal computer) or by having the server perform additional checks as part of the signature generation protocol.

An obvious solution is to just have the server handle all asymmetric-key operations based on requests from the signers [37]. In this case the server has to be completely trusted, but it's not clear whether that is in fact less secure than letting end-users manage their own keys [16].

To reduce the need to trust the server, Asokan *et al.* [2] proposed and others in [4,25] improved methods where asymmetric-key operations are performed by a server, but a user can prove the server's misbehavior when presented with a signature that the server created without the user's request. However, such signatures appear to be valid to a verifier until challenged by the user. Thus, these protocols are usable in contexts where a dispute resolution process exists, but unsuitable for applications with immediate and irrevocable effects, such as authentication for access control purposes.

Several methods have been proposed for outsourcing the more expensive computation steps of specific signature algorithms, notably RSA, but most early schemes have subsequently been shown to be insecure. In recent years, probably due to increasing computational power of handheld devices and wider availability of hardware-accelerated implementations, attention has shifted to splitting keys between end-user devices and back-end servers to improve the security of the private keys [10,18].

2.3 Interactive Signature Protocols

Interactive signature protocols, either by interaction between parties or with an external time-stamping service, were considered by Anderson *et al.* [1]. They proposed the "Guy Fawkes Protocol", where, once bootstrapped, a message is preceded by publishing the hash of the message and each message is authenticated by accompanying it with a secret whose hash was published together with an earlier message. Although the verification is limited to a single party, the protocol is shown to be a signature scheme according to several definitions. The broadcast commitment step is critical for providing non-repudiation of origin. Similar concept was first used in the TESLA protocol [36], designed to authenticate parties who are constantly communicating with each other. Due to this, it has the same inflexibility of not supporting multiple independent verifiers.

Buldas *et al.* [14] presented a generic hash-based signature scheme which depends on interaction with a time-stamping service. In the following we call this scheme the *BLT scheme*. The principal idea of the scheme is to have the signer commit to a sequence of secret keys so that each key is assigned a time slot when it can be used to sign messages and will transition from signing key to verification key at the end of the time slot. In order to prove timely usage of the keys, a cryptographic time-stamping service is used. It is possible to provide suitable time-stamping service [11] with no trust in the service provider [12,13], using hash-linking and hash-then-publish schemes [26]. Signing then comprises of time-stamping the message-key commitment in order to prove that the signing operation was performed at the correct time.

2.4 Authenticated Data Structures

An authenticated data structure is a data structure whose operations can be performed by an untrusted prover (server) and the integrity of the results can be verified efficiently by a verifier. We do not follow the less general 3-party model where trusted clients modify data on an untrusted server, and the query responses are accompanied with proof of correct operation based on server's data structure [40].

Authenticated data structures in the sense used here were first proposed for checking the correctness of computer memory [5]. Thorough analysis of applications in the context of tamper-evident logging was performed in [20]. The concept found its practical use-case in PKI certificate management: first proposed as "undeniable attesters" [15], where PKI users receive attestations of their certificates' inclusion in or removal from the database of valid certificates, and then the "certificate transparency" framework [31], which facilitates public auditing of certification authority operations.

3 Approach

3.1 Preliminaries

Hash Trees. Introduced by Merkle [33], a hash tree is a tree-shaped data structure built using a 2-to-1 hash function $h\colon \{0,1\}^{2k} \to \{0,1\}^k$. The nodes of the tree contain k-bit values. Each node is either a leaf with no children or an internal node with two children. The value x of an internal node is computed as $x \leftarrow h(x_l, x_r)$, where x_l and x_r are the values of the left and right child, respectively. There is one root node that is not a child of any node. We will use $r \leftarrow T^h(x_1, \ldots, x_N)$ to denote a hash tree whose N leaves contain the values $x_1, \ldots, x_N$ and whose root node contains r.

Hash Chains. In order to prove that a value x_i participated in the computation of the root hash r, it is sufficient to present values of all the siblings of the nodes on the unique path from x_i to the root in the tree. For example, to claim that x_3 belongs to the tree shown on the left in Fig. 1, one has to present the values x_4 and $x_{1,2}$. This enables the verifier to compute $x_{3,4} \leftarrow h(x_3, x_4)$, $r \leftarrow h(x_{1,2}, x_{3,4})$, essentially re-building a slice of the tree, as shown on the right in Fig. 1. We will use $x \overset{c}{\leadsto} r$ to denote that the hash chain c links x to r in such a manner.

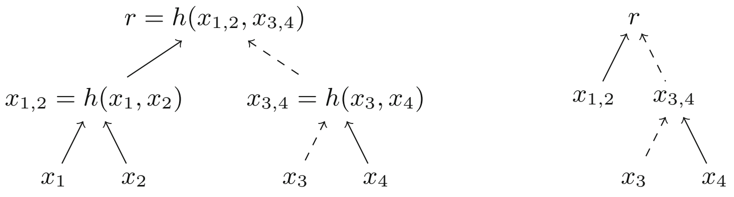

Fig. 1. The hash tree $T^h(x_1, \ldots, x_4)$ and the corresponding hash chain $x_3 \rightsquigarrow r$.

3.2 The BLT Signature Scheme

We start from the BLT scheme [14] with the following parties (Fig. 2):

– The signer who uses trusted functionality in secure device D to manage private keys.
– Server S that aggregates key usage events from multiple signers in fixed-length rounds and posts the summaries to append-only repository R.
– Verifier V who can verify signatures against the signer's public key p and the round summaries r_t obtained from the repository.

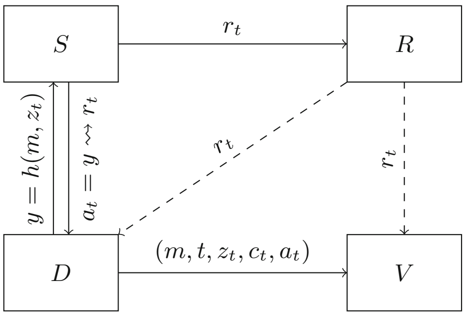

Fig. 2. Components of the BLT signature scheme.

Note that S and R together implement a hash-and-publish time-stamping service where neither the signer nor the verifier needs to trust S; only R has to operate correctly for the scheme to be secure.

Key Generation. To prepare to sign messages at times $1, \ldots, T$, the signer:

1. Generates T unpredictable k-bit signing keys: $(z_1, \ldots, z_T) \leftarrow \mathscr{G}(T, k)$.
2. Binds each key to its time slot: $x_t \leftarrow h(t, z_t)$ for $t \in \{1, \ldots, T\}$.
3. Computes the public key p by aggregating the key bindings into a hash tree: $p \leftarrow T^h(x_1, \ldots, x_T)$.

The purpose of the resulting data structure (Fig. 3) is to be able to extract the hash chains c_t linking the private key bindings to the public key: $h(t, z_t) \overset{c_t}{\rightsquigarrow} p$ for $t \in \{1, \ldots, T\}$.

$$\begin{tikzcd}
 & & & p & & & & \\
 & x_{1,2} \arrow[urr] & & & & x_{3,4} \arrow[ull] & & \\
x_1 \arrow[ur] & & x_2 \arrow[ul] & & x_3 \arrow[ur] & & x_4 \arrow[ul] & \\
1 \arrow[u] & z_1 \arrow[ul] & 2 \arrow[u] & z_2 \arrow[ul] & 3 \arrow[u] & z_3 \arrow[ul] & 4 \arrow[u] & z_4 \arrow[ul]
\end{tikzcd}$$

Fig. 3. Computation of public key for $N = 4$.

Signing. To sign message m at time t, the signer:

1. Uses the appropriate key to authenticate the message: $y \leftarrow h(m, z_t)$.
2. Time-stamps the authenticator by submitting it to S for aggregation and getting back the hash chain a_t linking the authenticator to the published summary: $y \overset{a_t}{\rightsquigarrow} r_t$.
3. Outputs the tuple (t, z_t, c_t, a_t).

Note that the signature is composed and emitted after the time-stamping step, which makes it safe for the signer to release the key z_t as part of the signature: the aggregation round t has ended and any future uses of the key z_t can no longer be stamped with time t.

Verification. To verify the message m and the signature $s = (t, z, a, c)$ with the public key p and the aggregation round summary r_t, the verifier:

1. Checks that z was committed as signing key for time t: $h(t, z) \overset{c}{\rightsquigarrow} p$.
2. Checks that m was authenticated with the key z at time t: $h(m, z) \overset{a}{\rightsquigarrow} r_t$.

3.3 Desired Properties

The components of BLT can in fact be used to create a variety of signing schemes. In the following we draft some of them and explain the necessary compromises compared to the ideal properties:

– Early forgery prevention: it is better to block revoked or expired keys at signing time (so that signature can't be created) than to leave key status detection to verification time; the least desired is a scheme where forgery is detected only eventually during an audit.
– Minimal number and resource requirements of trusted components: these have to be implemented using secure hardware or distributed consensus which are both expensive.
– Minimal globally shared data: authenticated distribution is expensive.
– Well-defined security model: assumptions, root of trust, etc.
– Efficiency: many signers, few servers, single shared root of trust.
– Privacy: signing events should ideally be known only to verifiers.

Note that providing higher-level properties like key revocation and proof of signing time almost certainly requires some server support.

3.4 Design of the Proposed Scheme

One-Time Keys. The signing keys in BLT are really not one-time, but rather time-bound: every key can be used for signing only at a specific point of time. This incurs quite a large overhead as keys must be pre-generated even for time periods when no signatures are created. The schemes discussed below use one-time keys sequentially instead.

As the first idea, we can have the signer time-stamp each signature, just as in the basic BLT scheme; in case of a dispute, the signature with the earlier time-stamp wins and the later one is considered a forgery. This obviously makes verification very difficult and in particular gives the signer a way to deny any signature: before signing a document d with a key z, the signer can use the same key to privately sign some dummy value x; when later demanded to honor the signature on document d, the signer can show the signature on x and declare the signature on d a forgery.

To prevent this, we assign every signer to a designated server which allows each key to be used only once. A trivial solution would be to just trust the server to behave correctly. This would still not achieve non-repudiation, as the server could collect spent keys and create valid-looking signatures on behalf of the signer.

Situation can be improved with trusted logging and auditing. If either the signer or the server published all signing events, including the key index for each one, then the server could not reuse keys and would not have to be treated as a trusted component. This would be quite inefficient, though, because of the amount of data that would have to be distributed and processed during verification, and would also leak information about the signer's behavior.

Validating the Server's Behavior. In this section we discuss some ways to avoid publishing all transactions while still not having to trust the server. As a common feature, we use spent key counters both at the signer and the server side. The server periodically creates hash trees on top of its set of counters and publishes the root hashes to a public repository.

If we could assume no collaboration between the server and any verifier, the server would not learn the keys and thus could not produce valid signatures. This is quite unrealistic, though. We must assume that signatures can be published, and the server may have access to spent keys. So, we must eliminate the attack where the server decrements a spent key counter for a client from k to $i < k$, signs a message using captured z_i, and then increments the counter back to k.

On assumption that the server and (other) signers do not cooperate maliciously, a "neighborhood watch" could be a solution: all signers observe changes in received hash chains and in committed roots and request proofs from the server that all changes were legitimate (i.e. that key counters of signers assigned to neighboring leaves were never decreasing). This approach would only detect forgeries but not block them, and also would not give very strong guarantees: it is not realistic to exclude malicious cooperation between the server and some of its clients.

The concept of authenticated data structures could be used for checking the server. If proofs of correct operation were included in signatures, verifiers could reject signatures without valid proofs. This approach would have quite large overhead, however, as the verifiers would have to be able to validate the counters throughout their entire lifetime. Other parties who could perform such validation are the repository, the signers, or independent auditors. Both signers and auditors could only discover a forgery after the fact, not early enough to avoid creation of forged signatures.

Pre-validation by the Repository. A promising idea is to validate the server's correct operation by the repository itself. We require the server to provide a proof of correctness with each update to the repository. The repository accepts the update only after validating the proof. Accepted root hashes are made immutable using cryptographic techniques and widely distributed. Because signatures are verified based on published root hashes in the repository, forgery by temporarily decrementing key usage counters is prevented.

This solution has most of the desired properties from Sect. 3.3: it is efficient, as the amount of public data (the blockchain) grows linearly in time, independent of the number of signers or their activity; there is reasonably low number of trusted components; the blockchain, including its input validation is *forward secure*; server's forgery attempts will be prevented at signing time; it is not necessary to have a long-term log of private data. The repository can be implemented as a byzantine fault tolerant distributed state machine, so we do not have to trust a single party. We describe this scheme in more detail in the following.

4 New Signature Scheme

4.1 Components

Our proposed scheme (Fig. 4) consists of the following parties:

– The **signer** uses trusted device D to generate keys and then sign data. We assume there is an authenticated way to distribute public keys. We also assume the connection between D and S and the connection between D and R use authenticated channels implemented at another layer of the system (for example, using pre-distributed HMAC keys).
– **Server** S assists signers in generating signatures. S keeps a counter of spent keys for each signer and sends updates to the repository.
– The **repository** performs two tasks. The layer R_v verifies the correctness of each operation of S before accepting it and periodically commits the summary of current state to a public append-only repository R.
– **Verifier** V is a relying party who verifies signatures.

The server maintains a hash tree with a dedicated leaf for each client (Fig. 5). The value of the leaf is computed by hashing the pair (i, y) where i is the spent key counter and y is the last message received from the client (as detailed in Sect. 4.3).

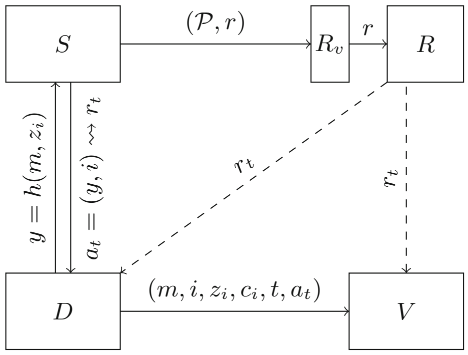

Fig. 4. Components of the new signature scheme.

Each public key must verifiably have just one leaf assigned to it. Otherwise, the server could set up multiple parallel counters for a client, increment only one of them in response to client requests, and use the others for forging signatures with keys the signer has already used and released.

One way to achieve that would be to have the server return the *shape* (that is, the directions to move to either the left or the right child on each step) of the path from the root of the tree to the assigned leaf when the client registers for service, and the client to include that shape when distributing its public key to verifiers. Another option would be to use the bits of the public key itself as the shape. Because most possible bit sequences are not actually used as keys, the hash tree would be a *sparse* one in this case.

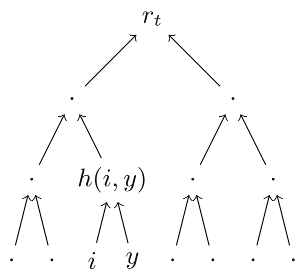

Fig. 5. Server tree for round t, showing key counter and input of the second client only.

4.2 Initialization

Signer. To prepare to sign up to N messages, the signer:

1. Generates N unpredictable k-bit signing keys: $(z_1, \ldots, z_N) \leftarrow \mathscr{G}(N, k)$.
2. Binds each key to its sequence number: $x_i \leftarrow h(i, z_i)$ for $i \in \{1, \ldots, N\}$.
3. Computes the public key p by aggregating the key bindings into a hash tree: $p \leftarrow T^h(x_1, \ldots, x_N)$.
4. Registers with the server S.

The data structure giving the public key is similar to the one in the original BLT scheme (Fig. 3), and also has the same purpose: to be able to extract the hash chains c_i linking the private key bindings to the public key: $h(i, z_i) \overset{c_i}{\rightsquigarrow} p$ for $i \in \{1, \ldots, N\}$.

Server. Upon receiving registration request from a signer, the server dedicates a leaf in its tree and sets i to 0 and y to an arbitrary value in that leaf.

4.3 Signing

Signer. Each signer keeps the index i of the next unused key z_i in its state. To sign message m, the signer:

1. Uses the current key to authenticate the message: $y \leftarrow h(m, z_i)$.
2. Sends the authenticator y to the server.
3. Waits for the server to return the hash chain a_t linking the pair (i, y) to the new published summary r_t: $h(i, y) \overset{a_t}{\rightsquigarrow} r_t$.
4. Checks that the shape of the received hash chain is correct and its output value matches the authentic r_t acquired directly from the repository.
5. If validation succeeds then outputs the tuple (i, z_i, c_i, t, a_t), where i is the key index, z_i is the i-th signing key, c_i is the hash chain linking the binding of the key z_i and its index i to the signer's public key p, and a_t is the hash chain linking (i, y) to the published r_t.
6. Increments its key counter: $i \leftarrow i + 1$.

Server. Upon receiving request y' from a signer, the server:

1. Extracts the hash chain a linking the current state of the client record (i, y) to the current root r of the server tree: $h(i, y) \overset{a}{\rightsquigarrow} r$.
2. Updates the client's record from (i, y) to $(i' \leftarrow i + 1, y')$ and computes the corresponding new root hash r' of the server tree.
3. Submits the tuple (i, y, a, r, y', r') to the repository for validation and publishing.
4. Waits for the repository to end the round and publish r_t.
5. Uses the state of its hash tree corresponding to the published r_t to extract and return to all clients with pending requests the hash chains a_t linking their updated (i', y') records to the published r_t: $h(i', y') \overset{a_t}{\rightsquigarrow} r_t$.

Repository. The validation layer R_v of the repository R keeps as state the current value $r^\star$ of the root hash of the server tree. Upon receiving the update (i, y, a, r, y', r') from S, the validator verifies its correctness:

1. The claimed starting state of the server tree must match the current state of R_v: $r = r^\star$.
2. The claimed starting state of the signer record must agree with the starting state of the server tree: $h(i, y) \overset{a}{\rightsquigarrow} r$.
3. The update of the client record must increment the counter: $i' \leftarrow i + 1$.

4. The new state of the server tree must correspond to changing just this one record: $h(i', y') \overset{a}{\rightsquigarrow} r'$.
5. If all the above checks pass, R_v updates its own state accordingly: $r^\star \leftarrow r'$.

R_v operates in rounds. During a round, it receives updates from the server, validates them, and updates its own state accordingly. At the end of the round, it publishes the current value of its state as the new round commitment r_t in the append-only public repository R.

Note that the hash chain a is the same in the verification of the starting state of the signer record against the starting state of the server tree and in the verification of the new state of the signer record against the new state of the server tree. This ensures no other leaves of the server tree can change with this update.

4.4 Verification

To verify that the message m and the signature $s = (i, z, c, t, a)$ match the public key p, the verifier:

1. Checks that z was committed as the i-th signing key: $h(i, z) \overset{c}{\rightsquigarrow} p$.
2. Retrieves the commitment r_t for the round t from repository R.
3. Checks that the use of the key z to compute the message authenticator $y \leftarrow h(m, z)$ matches the key index i: $h(i, y) \overset{a}{\rightsquigarrow} r_t$.

Note that the signature is composed and sent to verifier only after the verification of r_t, which makes it safe for the signer to release the key z_i as part of the signature: the server has already incremented its counter i so that only z_{i+1} could be used to produce the next valid signature.

5 Discussion

5.1 Server-Supported Signing

The model of server-supported signing is a higher-level protocol and is not directly comparable to traditional signature algorithms like RSA. To justify usefulness of the model, we will nonetheless highlight some distinctive properties:

– It is possible to create a server-side log of all signing operations, so that in the case of either actual or suspected key leak there is a complete record, making damage control and forensics manageable.
– Key revocation is implemented as blocking the access by the server, thus no new signatures can be created after the revocation, making key life-cycle controls much simpler. Note that the server can naturally record the revocation by setting the client's counter to some sentinel "infinite" value, and also return a proof of the update after it has been committed to the repository.
– The server can add custom attributes, and even *trusted attributes* which can't be forged by the server itself: cryptographic time-stamp, address, policy ID, etc.

– The server can perform data-dependent checks, such as transaction validation, before allowing a signing. Note that normally the server receives only a hash value of the data, and the signed data itself does not have to be revealed.

Finally, in scenarios where non-repudiation must be provided, all traditional schemes and algorithms must be supplemented with some server-provided functionalities like cryptographic time-stamping.

5.2 Implementation of the Repository

The proposed scheme dictates that the repository must have the following properties:

– Updates are only accepted if their proof of correctness is valid.
– All commitments are final and immutable.
– Commitments are public, and their immutability is publicly verifiable.

To minimize trust requirements on the repository, we propose to re-use the patterns used for creating *blockchains*. We do not consider proof-of-work, focusing on byzantine fault tolerant state machine replication model.

Instead of full transactions, we record in the blockchain only aggregate hashes representing batches of transactions. This provides two benefits: (1) the size of the blockchain grows linearly in time, in contrast with the usual dependency on the number and storage size of transactions; and (2) recording and publishing only aggregate hashes provides privacy. Such a blockchain design is an interesting research subject by itself.

A blockchain validates all transactions before executing them. An example of such validation is double-spending prevention in crypto-currency specific blockchains. We validate correctness proofs presented by signing servers. Such a model—where authenticated data structures are validated by a blockchain—is another potential research subject of independent interest.

The repository, when implemented as a byzantine fault tolerant blockchain, does not have trusted components.

5.3 Practical Setup

Although presented above as a list of components, envisioned real-life deployment of the scheme is hierarchical, as shown on Fig. 6.
The topmost layer is a distributed cluster of blockchain consensus nodes, each possibly operated by an independent "permissioned" party. The blockchain can accept inputs from multiple signing servers, each of which may in turn serve many clients. Because of this hierarchical nature the scheme scales well performance-wise. In terms of the amount of data, as stated earlier, the size of blocks and the number of blocks does not depend on the number of clients and number of signatures issued.

The system assumes that the certification service assigns each signer a dedicated signing server and a dedicated leaf position in this server's hash tree.

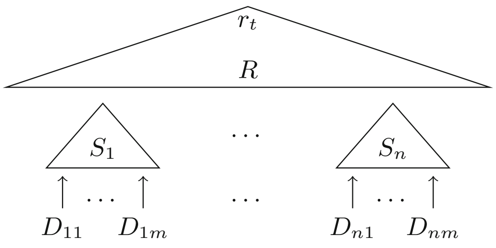

Fig. 6. A scalable deployment architecture for the new scheme.

5.4 Efficiency

The efficiency of our proposed signature scheme for both signers and verifiers is at least on par with the state of the art.

The considerations for key generation and management on the client side are similar to the original BLT scheme [14], except the number of private keys required is much smaller (assuming 10 signing operations per day, just 3 650 keys are needed for a year, compared to the 32 million keys in BLT) and the effort required to generate and manage them, which was the main weakness of BLT, is also correspondingly reduced.

Like in the original BLT scheme, the size of the signature in our scheme is also dominated by the two hash chains. The key sequence membership proof contains $log_2 N$ hash values, which is about 12 for the 3 650-element yearly sequence. The blockchain membership proof has $log_2 K$ hash values, where K is the number of clients the service has. Even when the whole world (8 billion people) signs up, it's still only about 33 hash values. Conservatively assuming the use of 512-bit hash functions, the two hash chains add up to less than 3 kB in total.

Verification of the signature means re-computing the two hash chains and thus amounts to about 45 hash function evaluations.

Admittedly, the above estimates exclude the costs of querying the blockchain to acquire the committed r_t that both the signer and the verifier need. However, that is comparable to the need to access a time-stamping service when signing and an OCSP (Online Certificate Status Protocol) responder when verifying signatures in the traditional PKI setup.

6 Conclusions and Outlook

We have proposed a novel server-assisted signature scheme based on hash functions as the sole underlying cryptographic primitive. The scheme is computationally efficient for both signers and verifiers and produces small signatures with tiny public keys.

Due to the server-assisted and blockchain-backed nature, the scheme provides instant key revocation and perfect forward security without the need to trust the server or any single component in the blockchain.

Formalizing and proving the security properties of the scheme in composition with different implementation architectures of the blockchain consensus is an interesting future research topic.

The concept of a blockchain containing only aggregate hashes of batches of transactions instead of full records and the notion of a blockchain based on pre-validation of correctness proofs of transactions before admitting them to the chain could both be of independent interest.

References

1. Anderson, R.J., Bergadano, F., Crispo, B., Lee, J.-H., Manifavas, C., Needham, R.M.: A new family of authentication protocols. Oper. Syst. Rev. **32**(4), 9–20 (1998)
2. Asokan, N., Tsudik, G., Waidner, M.: Server-supported signatures. J. Comput. Secur. **5**(1), 91–108 (1997)
3. Bernstein, D.J., et al.: SPHINCS: practical stateless hash-based signatures. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 368–397. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_15
4. Bicakci, K., Baykal, N.: Server assisted signatures revisited. In: Okamoto, T. (ed.) CT-RSA 2004. LNCS, vol. 2964, pp. 143–156. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24660-2_12
5. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica **12**(2–3), 225–244 (1994)
6. Buchmann, J., Coronado García, L.C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS – an improved Merkle signature scheme. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 349–363. Springer, Heidelberg (2006). https://doi.org/10.1007/11941378_25
7. Buchmann, J.A., Dahmen, E., Ereth, S., Hülsing, A., Rückert, M.: On the security of the Winternitz one-time signature scheme. IJACT **3**(1), 84–96 (2013)
8. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - a practical forward secure signature scheme based on minimal security assumptions. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 117–129. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25405-5_8
9. Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle signatures with virtually unlimited signature capacity. In: Katz, J., Yung, M. (eds.) ACNS 2007. LNCS, vol. 4521, pp. 31–45. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72738-5_3
10. Buldas, A., Kalu, A., Laud, P., Oruaas, M.: Server-supported RSA signatures for mobile devices. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10492, pp. 315–333. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66402-6_19
11. Buldas, A., Kroonmaa, A., Laanoja, R.: Keyless signatures' infrastructure: how to build global distributed hash-trees. In: Riis Nielson, H., Gollmann, D. (eds.) NordSec 2013. LNCS, vol. 8208, pp. 313–320. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41488-6_21
12. 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, vol. 7959, pp. 235–250. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39059-3_16

13. Buldas, A., Laanoja, R., Laud, P., Truu, A.: Bounded pre-image awareness and the security of hash-tree keyless signatures. In: Chow, S.S.M., Liu, J.K., Hui, L.C.K., Yiu, S.M. (eds.) ProvSec 2014. LNCS, vol. 8782, pp. 130–145. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12475-9_10
14. Buldas, A., Laanoja, R., Truu, A.: A server-assisted hash-based signature scheme. In: Lipmaa, H., Mitrokotsa, A., Matulevičius, R. (eds.) NordSec 2017. LNCS, vol. 10674, pp. 3–17. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70290-2_1
15. Buldas, A., Laud, P., Lipmaa, H.: Accountable certificate management using undeniable attestations. In: Proceedings of the 7th ACM Conference on Computer and Communications Security, pp. 9–17. ACM (2000)
16. Buldas, A., Saarepera, M.: Electronic signature system with small number of private keys. In: 2nd Annual PKI Research Workshop, Proceedings, pp. 96–108. NIST (2003)
17. Buldas, A., Saarepera, M.: On provably secure time-stamping schemes. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 500–514. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30539-2_35
18. Camenisch, J., Lehmann, A., Neven, G., Samelin, K.: Virtual smart cards: how to sign with a password and a server. In: Zikas, V., De Prisco, R. (eds.) SCN 2016. LNCS, vol. 9841, pp. 353–371. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44618-9_19
19. Coronado García, L.C.: Provably secure and practical signature schemes. Ph.D. thesis, Darmstadt University of Technology, Germany (2005)
20. Crosby, S.A., Wallach, D.S.: Efficient data structures for tamper-evident logging. In: Proceedings of the 18th USENIX Security Symposium, pp. 317–334. USENIX (2009)
21. Dahmen, E., Okeya, K., Takagi, T., Vuillaume, C.: Digital signatures out of second-preimage resistant hash functions. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 109–123. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88403-3_8
22. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inf. Theory **22**(6), 644–654 (1976)
23. Dods, C., Smart, N.P., Stam, M.: Hash based digital signature schemes. In: Smart, N.P. (ed.) Cryptography and Coding 2005. LNCS, vol. 3796, pp. 96–115. Springer, Heidelberg (2005). https://doi.org/10.1007/11586821_8
24. Even, S., Goldreich, O., Micali, S.: On-line/off-line digital signatures. J. Cryptol. **9**(1), 35–67 (1996)
25. Goyal, V.: More efficient server assisted one time signatures. Cryptology ePrint Archive, Report 2004/135 (2004). https://eprint.iacr.org/2004/135
26. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. J. Cryptol. **3**(2), 99–111 (1991)
27. Hülsing, A.: W-OTS+ – shorter signatures for hash-based signature schemes. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 173–188. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38553-7_10
28. Hülsing, A., Rausch, L., Buchmann, J.: Optimal parameters for XMSS^{MT}. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E., Xu, L. (eds.) CD-ARES 2013. LNCS, vol. 8128, pp. 194–208. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40588-4_14

29. Hülsing, A., Rijneveld, J., Song, F.: Mitigating multi-target attacks in hash-based signatures. In: Cheng, C.-M., Chung, K.-M., Persiano, G., Yang, B.-Y. (eds.) PKC 2016. LNCS, vol. 9614, pp. 387–416. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49384-7_15
30. Lamport, L.: Constructing digital signatures from a one way function. Technical report, SRI International, Computer Science Laboratory (1979)
31. Laurie, B., Langley, A., Kasper, E.: Certificate transparency. RFC 6962, RFC Editor, June 2013
32. Malkin, T., Micciancio, D., Miner, S.: Efficient generic forward-secure signatures with an unbounded number of time periods. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 400–417. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46035-7_27
33. Merkle, R.C.: Secrecy, authentication and public key systems. Ph.D. thesis, Stanford University (1979)
34. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-48184-2_32
35. Perrig, A.: The BiBa one-time signature and broadcast authentication protocol. In: Proceedings of the ACM CCS 2001, pp. 28–37. ACM (2001)
36. Perrig, A., Canetti, R., Tygar, J.D., Song, D.: The TESLA broadcast authentication protocol. CryptoBytes **5**(2), 2–13 (2002)
37. Perrin, T., Bruns, L., Moreh, J., Olkin, T.: Delegated cryptography, online trusted third parties, and PKI. In: Proceedings of the 1st Annual PKI Research Workshop, pp. 97–116. NIST (2002)
38. Reyzin, L., Reyzin, N.: Better than BiBa: short one-time signatures with fast signing and verifying. In: Batten, L., Seberry, J. (eds.) ACISP 2002. LNCS, vol. 2384, pp. 144–153. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45450-0_11
39. Rohatgi, P.: A compact and fast hybrid signature scheme for multicast packet authentication. In: Proceedings of the ACM CCS 1999, pp. 93–100. ACM (1999)
40. Tamassia, R.: Authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39658-1_2

Appendix 3

Publication III
A. Buldas, D. Firsov, R. Laanoja, H. Lakk, and A. Truu. A new approach to constructing digital signature schemes (short paper). In *IWSEC 2019*, *Proceedings*, volume 11689 of *LNCS*, pages 363–373. Springer, 2019

An extended version of this paper is available as
A. Buldas, D. Firsov, R. Laanoja, H. Lakk, and A. Truu. A new approach to constructing digital signature schemes (extended paper). Cryptology ePrint Archive, Report 2019/673, 2019. `https://eprint.iacr.org/2019/673`

A New Approach to Constructing Digital Signature Schemes (Short Paper)

Ahto Buldas[1], Denis Firsov[1,2], Risto Laanoja[1,2], Henri Lakk[2], and Ahto Truu[1,2]( )

[1] Tallinn University of Technology, Akadeemia tee 15a, 12618 Tallinn, Estonia
[2] Guardtime AS, A. H. Tammsaare tee 60, 11316 Tallinn, Estonia
ahto.truu@guardtime.com

Abstract. A new hash-based, server-supported digital signature scheme was proposed recently in [7]. We decompose the concept into *forward-resistant tags* and a generic *cryptographic time-stamping* service. Based on the decomposition, we propose more tag constructions which allow efficient digital signature schemes with interesting properties to be built. In particular, the new schemes are more suitable for use in personal signing devices, such as smart cards, which are used infrequently. We define the forward-resistant tags formally and prove that (1) the discussed constructs are indeed tags and (2) combining such tags with time-stamping services gives us signature schemes.

1 Introduction

Recently, Buldas, Laanoja, and Truu [7] proposed a new type of digital signature scheme (which we will refer to as the *BLT scheme* in the following) based on the idea of combining one-time time-bound keys with a time-stamping service. A limitation of the BLT scheme is the fact that keys are pre-generated and have to be used at their designated time-slots only. On practical parameters the number of keys is rather large, which would make key generation on resource-constrained platforms prohibitively slow.

BLT scheme prevents other parties from misusing keys by making each key expire immediately after a legitimate use. First, each key is explicitly bound to a time slot at the key-generation time, and keys would automatically expire when their designated time-slots passed. Second, the legitimate use of a key is proven by time-stamping the message-key pair. Back-dating a new pair (a new message with an already used key) would allow a signature to be forged. Therefore, the hash-then-publish time-stamping [12] that avoids key-based cryptography and trusted third parties is particularly suitable for the scheme.

This research was supported by the European Regional Development Fund through the Estonian smart specialization program NUTIKAS and by the research measure of the Estonian IT Academy programme.

N. Attrapadung and T. Yagi (Eds.): IWSEC 2019, LNCS 11689, pp. 363–373, 2019.
https://doi.org/10.1007/978-3-030-26834-3_21

Based on this observation, we generalize and decompose the scheme into two functional components: *forward-resistant tags* and a *cryptographic time-stamping* service. As the forward-resistant tag is a novel construct, we define it formally and prove that the BLT scheme is indeed an instance of forward-resistant tag-based schemes. We then propose other forward-resistant tag systems, prove their security, and observe that the resulting new signature schemes are efficient and have some interesting properties.

2 Related Work and Background

Due to space constraints, we refer the reader to either [7] or the extended e-print [4] for overview of related work on **hash-based signatures**, **server-assisted signatures**, and **interactive signature protocols**.

Non-Repudiation. An important property of digital signatures (as an alternative to hand-written ones) [10] is non-repudiation, i.e. the possibility to use the signature as evidence against the signer. Solutions where trusted third parties are (technically) able to sign on behalf of their clients are not desirable for non-repudiation, because clients may use that argument to fraudulently call their signatures into question. Therefore, solutions where clients have personal signature devices are preferable to those relying entirely on trusted parties.

Another real-world complexity is key revocation. Without such capability clients may (fraudulently) claim that their private keys were stolen and someone else may have created signatures in their name. With revocation tracking, signatures created before a key revocation event can be treated as valid, whereas signatures created afterwards can be considered invalid. Usually this is implemented using cryptographic time-stamping and certificate status distribution services. No matter the implementation details, this can not be done without online services, which means that most practical deployments of digital signatures are actually server-supported.

Cryptographic Time-Stamping. Cryptographic time-stamps prove that data existed before a particular time. The proof can be a statement that the data hash existed at a given time, cryptographically signed by a trusted third party. Haber and Stornetta [12] made the first steps towards trustless time-stamping by proposing a scheme where each time-stamp would include some information from the immediately preceding one and a reference to the immediately succeeding one. Benaloh and de Mare [1] proposed to increase the efficiency of hash-linked time-stamping by operating in rounds, where messages to be time-stamped within one round would be combined into a hierarchical structure from which a compact proof of participation could be extracted for each message. The aggregation structures would then be linked into a linear chain. The security of linking-based hash-then-publish schemes has been proven in a very strong model where even the time-stamping service provider does not have to be trusted [6,8], making them particularly suitable for our use-case. It is possible to provide such service efficiently and in global scale [5].

3 Forward-Resistant Tags

Definition 1 (Tag system). *By a tag system we mean a triple* (Gen, Tag, Ver) *of algorithms, where:*

- Gen *is a probabilistic key-generation algorithm that, given as input the tag range* T*, produces a secret key* sk *and a public key* pk*.*
- Tag *is a tag-generation algorithm that, given as input the secret key* sk *and an integer* $t \in \{1, \ldots, T\}$*, produces a tag* $\tau \leftarrow \mathrm{Tag}(\mathsf{sk}, t)$*.*
- Ver *is a verification algorithm that, given as input a tag* τ*, an integer* t*, and the public key* pk*, returns either* 0 *or* 1*, such that* $\mathrm{Ver}(\mathrm{Tag}(\mathsf{sk}, t), t, \mathsf{pk}) = 1$ *whenever* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathrm{Gen}(T)$ *and* $1 \leq t \leq T$*.*

The above definition of a tag system is somewhat similar to that of a signature scheme consisting of procedures for key generation, signature generation, and signature verification [11]. The fundamental difference is that a signature binds the use of the secret key to a message, while a tag binds the use of the secret key to a time.

Definition 2 (Forward-resistant tag system). *A tag system* (Gen, Tag, Ver) *is* S*-forward-resistant if every tag-forging adversary* A *using computational resources* ρ *has success probability*

$$\mathsf{Pr}\Big[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen}(T), (\tau, t) \leftarrow A^{\mathrm{Tag}(\mathsf{sk}, \cdot)}(\mathsf{pk}) : \mathrm{Ver}(\tau, t, \mathsf{pk}) = 1\Big] < \frac{\rho}{S},$$

where A *makes one oracle call* $\mathrm{Tag}(\mathsf{sk}, t')$ *with* $1 \leq t' < t$*.*

The restriction for A to make just one oracle call stems from the fact that the very purpose of a tag system is to bind the use of the secret key to a specific time. Informally, in order to implement a forward resistant tag system, we have to bind each tag to a time t so that the tag can't be re-bound to a later time. This notion could be seen as dual to time-stamping that prevents back-dating.

The resources represented by ρ are computation time and memory. The total resource budget of the adversary is $\rho = \alpha \cdot \mathrm{time} + \beta \cdot \mathrm{memory}$, where α and β are the costs of a unit of computation time and a unit of memory, respectively.

Security proofs of the proposed tag systems will be based on the following definitions of basic cryptographic properties of functions:

Definition 3 (One-way function). *A function* $f : D \to R$ *is* S*-secure one-way (*S*-OW in short) if every* f*-inverting adversary* A *using computational resources* ρ *has success probability*

$$\mathsf{Pr}\Big[x \leftarrow D, x' \leftarrow A^{f(\cdot)}(f(x)) : f(x') = f(x)\Big] < \frac{\rho}{S}.$$

Definition 4 (Collision resistant function). *A function* $f : D \to R$ *is* S*-secure collision resistant (*S*-CR) if for every collision-finding adversary* A *using computational resources* ρ*:*

$$\mathsf{Pr}\Big[x_1, x_2 \leftarrow A^{f(\cdot)} : x_1 \neq x_2,\ f(x_1) = f(x_2)\Big] < \frac{\rho}{S}.$$

Definition 5 (Undetectable function). *A function $f : D \to D$ is S-secure undetectable (S-UD) if for every detecting adversary A using computational resources ρ:*

$$\left|\mathsf{Pr}\Big[x \leftarrow \mathcal{U} : A(x) = 1\Big] - \mathsf{Pr}\Big[x \leftarrow \mathcal{U} : A(f(x)) = 1\Big]\right| < \frac{\rho}{S},$$

where $\mathcal{U}$ generates random values uniformly from D.

Lemma 1. *If $f : D \to D$ is S-UD, then f^n is $\frac{S}{n}$-UD. (Proof in the e-print [4].)*

In the following, we will consider general hash functions $f : \{0,1\}^\star \to \{0,1\}^k$ mapping arbitrary-length inputs to fixed-length outputs. We will write $f(x_1, x_2)$ or $f(x_1, x_2, \ldots, x_n)$ to mean the result of applying f to a bit-string encoding the pair (x_1, x_2) or the tuple $(x_1, x_2, \ldots, x_n)$, respectively.

Cryptographic Time-Stamping. We model the ideal time-stamping service as a trusted repository R that works as follows:

- The time t is initialized to 1, and all the cells R_i to $\perp$.
- The query $R.\mathtt{time}$ is answered with the current value of t.
- The query $R.\mathtt{get}(t)$ is answered with R_t.
- On the request $R.\mathtt{put}(x)$, first $R_t \leftarrow x$ is assigned and then the value of t is incremented by 1.

This is done for the sake of simplicity. It turns out that refining the model of the time-stamping service would make the proofs really complex. For example, even for a seemingly trivial change, where R publishes a hash $h(m, \tau)$ instead of just (m, τ), one needs non-standard security assumptions about h such as non-malleability. In this paper, we try to avoid these technical difficulties and focus on the basic logic of the security argument of the tag-based signature scheme.

Definition 6 (Induced signature scheme). *A tag system* $(\mathrm{Gen}, \mathrm{Tag}, \mathrm{Ver})$ *and a time-stamping repository R induce a one-time signature scheme as follows:*

The signer $S^R(m)$ queries $t \leftarrow R.\mathtt{time}$, then creates $\tau \leftarrow \mathrm{Tag}(\mathsf{sk}, t)$, stores $R.\mathtt{put}((m, \tau))$, and returns $\sigma = (\tau, t)$.

The verifier $V^R(m, (\tau, t), \mathsf{pk})$ queries $x \leftarrow R.\mathtt{get}(t)$, and checks that $x = (m, \tau)$ and $\mathrm{Ver}(\mathsf{pk}, t, \tau) = 1$.

Definition 7 (Existential unforgeability). *A one-time signature scheme is S-secure existentially unforgeable (S-EUF) if every forging adversary A using computational resources ρ has success probability*

$$\mathsf{Pr}\Big[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen}(T), (m, \sigma) \leftarrow A^{S^R, R}(\mathsf{pk}) : V^R(m, \sigma, \mathsf{pk}) = 1\Big] < \frac{\rho}{S},$$

where A makes only one S-query and not with m.

Theorem 1. *If the tag system is S-secure forward-resistant then the induced one-time signature scheme is (almost) S-secure existentially unforgeable.*

Proof. Having a ρ-adversary $A^{S^R,R}$ for the signature scheme, we construct an adversary $B^{\mathrm{Tag}(\mathsf{sk},\cdot)}$ for the tag scheme as follows. The adversary B simulates the adversary A by creating a simulated R of its own. A signing query $S(m)$ is simulated by making an oracle query $\tau \leftarrow \mathrm{Tag}(\mathsf{sk}, t)$, where t is the time value in the simulated R, and then assigning $R_t \leftarrow (m, \tau)$.

Every time the simulated A makes (a direct) query $R.\mathtt{put}(x)$, B checks whether x is in the form (m, τ) and $\mathrm{Ver}(\mathsf{pk}, \tau, t) = 1$, where t is the current time in the simulated R, and then returns (τ, t) if either: (a) A has never made any S-calls, or (b) A has made an S-call with $m' \neq m$.

It is easy to see that one of these events must occur whenever A is successful. In the first case, B is also successful, because it outputs a correct tag without making any $\mathrm{Tag}(\mathsf{sk}, \cdot)$-calls. In the second case, the $S(m')$-query was made at $t' < t$ (as every S-query makes one $R.\mathtt{put}(\cdot)$-query which advances t) and then also the $\mathrm{Tag}(\mathsf{sk}, t')$-query was made at $t' < t$ and hence B is successful again.

If the overhead of B in simulating the environment for A is small, the reduction is tight and thus the signature scheme must indeed be almost as secure as the underlying tag scheme. □

3.1 The BLT Scheme as a Tag System (BLT-TB)

Ignoring the aggregation of individual time-bound keys into a hash tree, the essence of the BLT signature scheme proposed in [7] can be modeled as a tag system as follows:

- The secret key sk is a list $(z_1, z_2, \ldots, z_T)$ of T unpredictable values and the public key pk the list $(f(z_1), f(z_2), \ldots, f(z_T))$, where f is a one-way function.
- The tagging algorithm $\mathrm{Tag}(z_1, z_2, \ldots, z_T; t)$ outputs z_t.
- The verification algorithm Ver, given as input a tag τ, an integer t, and the public key $(x_1, x_2, \ldots, x_T)$, checks that $1 \le t \le T$ and $f(\tau) = x_t$.

We will refer to this model as the BLT-TB tag system.

Theorem 2. *If f is S-OW, then BLT-TB is an $\frac{S}{T}$-forward-resistant tag system.*

Proof. We assume there's a tag-forging adversary A and construct an f-inverting adversary B based on oracle access to A. Since f is S-OW, irrespective of B's construction, its success probability $\delta' < \frac{\rho}{S}$ (Definition 3). We construct B to process the input $x = f(z)$ as follows:

- generate the secret key components $z_i \leftarrow \{0,1\}^k$ and compute the corresponding public key components $x_i = f(z_i)$ for $1 \le i \le T$;
- uniformly randomly pick an index $j \leftarrow \{1, \ldots, T\}$;
- call A on a modified public key to produce a forged tag and its index

$$(\tau, t) \leftarrow A^{\mathrm{Tag}(\mathsf{sk},\cdot)}(x_1, \ldots, x_{j-1}, x, x_{j+1}, \ldots, x_T);$$

- if A succeeded and $t = j$ then return τ, else return $\perp$.

By construction, B's success probability $\delta_j = \mathsf{Pr}\big[A \text{ succeeded} \wedge\ t = j\big]$. Since the distribution of x is identical to the distribution of x_i, the events "A succeeded" and "$t = j$" are independent and thus we have $\delta_j = \mathsf{Pr}\big[A \text{ succeeded}\big] \cdot \mathsf{Pr}\big[t = j\big]$. Since j was drawn uniformly from $\{1, \ldots, T\}$, we further have $\delta_j = \delta \cdot \frac{1}{T}$, where δ is A's success probability (Definition 2).

From f being S-OW, we have $\frac{\delta}{T} = \delta_j \le \delta' < \frac{\rho}{S}$. Thus, $\delta < \frac{\rho}{S/T}$, and BLT-TB is indeed an $\frac{S}{T}$-forward-resistant tag system. □

3.2 The BLT-OT Tag System

We now define the BLT-OT tag system (inspired by Lamport's one-time signatures [9]) as follows:

- The secret key sk is a list $(z_0, z_1, \ldots, z_{\ell-1})$ of $\ell = \lceil \log_2(T+1) \rceil$ unpredictable values and the public key pk the list $(f(z_0), f(z_1), \ldots, f(z_{\ell-1}))$, where f is a one-way function.
- The tagging algorithm $\text{Tag}(z_0, \ldots, z_{\ell-1}; t)$ outputs an ordered subset $(z_{j_1}, z_{j_2}, \ldots, z_{j_m})$ of components of the secret key such that $0 \le j_1 < j_2 < \ldots < j_m \le \ell - 1$ and $2^{j_1} + 2^{j_2} + \ldots + 2^{j_m} = t$.
- The verification algorithm Ver, given as input a sequence $(z_{j_1}, z_{j_2}, \ldots, z_{j_m})$, an integer t, and the public key $(x_0, x_1, \ldots, x_{\ell-1})$, checks that:
  1. $f(z_{j_1}) = x_{j_1}, \ \ldots, \ f(z_{j_m}) = x_{j_m}$; and
  2. $0 \le j_1 < j_2 < \ldots < j_m \le \ell - 1$; and
  3. $2^{j_1} + 2^{j_2} + \ldots + 2^{j_m} = t$; and
  4. $1 \le t \le T$.

Theorem 3. *If f is S-OW, then BLT-OT is an $\frac{S}{\ell}$-forward-resistant tag system. (Proof is very similar to Theorem 2 and available in the e-print [4].)*

3.3 The BLT-W Tag System

We now define the BLT-W tag system (inspired by Winternitz's idea [14] for optimizing the size of Lamport's one-time signatures) as follows:

- The secret key sk is an unpredictable value z and the public key pk is $f^T(z)$, where f is a one-way function.
- The tagging algorithm $\text{Tag}(z; t)$ outputs the value $f^{T-t}(z)$.
- The verification algorithm Ver, given as input a tag τ, an integer t, and the public key x, checks that $1 \le t \le T$ and $f^t(\tau) = x$.

Theorem 4. *If f is S_1-OW and S_2-CR and S_3-UD function, then BLT-W is a $\frac{\min(S_1,S_2,S_3)}{2 \cdot T}$-forward-resistant tag system. (Proof is similar to Theorem 2 and available in the e-print [4].)*

4 BLT-OT One-Time Signature Scheme

The signature scheme induced by the BLT-OT tag system according to Definition 6 would require the signer to know in advance the time when its request reaches the time-stamping service. This is hard to achieve in practice, in particular for devices such as smart cards that lack built-in clocks. To overcome this limitation, we construct the BLT-OT one-time signature scheme as follows.

Key Generation. Let ℓ be the number of bits that can represent any time value t when the signature may be created (e.g. $\ell = 32$ for POSIX time up to year 2106). The private key is generated as $\mathsf{sk} = (z_0, z_1, \ldots, z_{\ell-1})$, where z_i are unpredictable values, and the public key as $\mathsf{pk} = f(X)$, where $X = (x_0, x_1, \ldots, x_{\ell-1})$, $x_i = f(z_i)$, and f is a one-way function.

The **public key certificate** should contain (a) the public key pk, (b) the identity ID_c of the client, and (c) the identity ID_s of the designated time-stamping service. Recording the identity of the designated time-stamping service in certificate enables instant key revocation. Upon receiving a revocation notice, the designated service stops serving the affected client, and thus it is not possible to generate signatures using revoked keys.

Signing. To sign a message m, the client:

- gets a time-stamp S_t on the record (m, X, ID_c) from the time-stamping service designated by ID_s;
- extracts the ℓ-bit time value t from S_t and creates the list $W = (w_0, w_1, \ldots, w_{\ell-1})$, where $w_i = z_i$ if the i-th bit of t is 1, or $w_i = x_i = f(z_i)$ otherwise;
- disposes of the private key $(z_0, z_1, \ldots, z_{\ell-1})$ to prevent its re-use;
- emits (W, S_t) as the signature.

Verification. To verify the signature (W, S_t) on the message m against the certificate $(\mathsf{pk}, ID_c, ID_s)$, the verifier:

- extracts time t from the time-stamp S_t;
- recovers the list $X = (x_0, x_1, \ldots, x_{\ell-1})$ by computing $x_i = f(w_i)$ if the i-th bit of t is 1, or $x_i = w_i$ otherwise;
- checks that the computed X matches the public key: $f(X) = \mathsf{pk}$;
- checks that S_t is a valid time-stamp issued at time t by service ID_s on the record (m, X, ID_c).

Using the reduction techniques from previous sections to formally prove the security of this optimized signature scheme is complicated by both the iterated use of f and the more abstract view of the time-stamping service.

Details of other optimized signature schemes are skipped for brevity. Practical properties are discussed in the following section.

5 Discussion

The BLT-TB scheme proposed in [7] works well for powerful devices that are constantly running and have reliable clocks. These are not reasonable assumptions for personal signing devices such as smart cards, which have very limited capabilities and are not used very often. Generating keys could take hours or even days of non-stop computing on such devices. This is clearly impractical, and also wasteful as most of the keys would go unused.

The BLT-OT scheme proposed in Sect. 4 solves the problems described above at the cost of introducing state on the client side. As the scheme is targeted towards personal signing devices, the statefulness is not a big risk, because these devices are not backed up and also do not support parallel processing. The benefit in addition to improved efficiency is that the device no longer needs to know the current time while preparing a signing request. Instead, it can just use the time from the time-stamp when composing the signature.

Table 1. Efficiency of hash-based one-time signature schemes. We assume 256-bit hash functions, 32-bit time values, and time-stamping hash-tree with 33 levels. Times are in hashing operations and signature sizes in hash values. TS in BLT schemes stands for the time-stamping service call.

Scheme	Key generation	Signing time	Verification time	Signature size
Lamport	1 025	1 024	513	256
Winternitz ($w = 4$)	1 089	1 088	1 021	68
BLT-OT	65	64 + TS	33 + 33	32 + 33
BLT-W ($w = 2$)	65	64 + TS	49 + 33	16 + 33

Efficiency as One-Time Scheme. When implemented as described in Sect. 4, the cost of generating a BLT-OT key pair is ℓ random key generations and $\ell + 1$ hashing operations, the cost of signing ℓ hashing operations and one time-stamping service call, and the cost of signature verification at most $\ell + 1$ hashing operations and one time-stamp verification. In this case the private key would consist of ℓ one-time keys and the public key of one hash value, and the signature would contain ℓ hash values and one time-stamp token. The private storage size can be optimized by generating the one-time keys from one true random seed using a pseudo-random generator. Then the cost of signing increases by ℓ operations, as the one-time keys would have to be re-generated from the seed before signing. This version is listed as BLT-OT in Table 1.

Winternitz's idea [14] for optimizing the size of Lamport's one-time signatures [9] can also be applied to BLT-OT. Instead of using one-step hash chains $z_i \to h(z_i) = x_i$ to encode single bits of t, we can use longer chains $z_i \to h(z_i) \to \ldots \to h^n(z_i) = x_i$ and publish the value $h^{n-j}(z_i)$ in the signature to encode the value j of a group of bits of t. When encoding groups of w bits of

t in this manner, the chains have to be $n = 2^w$ steps long. This version is listed as BLT-W in Table 1. Note that in contrast to applying this idea to Lamport's signatures, in BLT-W no additional countermeasures are needed to prevent an adversary from stepping the hash chains forward: the time in the time-stamp takes that role.

To compare BLT-OT signature sizes and verification times to other schemes, we also need to estimate the size of hash-trees built by the time-stamping service. Even assuming the whole world (8 billion people) will use the time-stamping service in every aggregation round, an aggregation tree of 33 layers will suffice. We also assume that in all schemes one-time private keys will be generated on-demand from a single random seed and public keys will be aggregated into a single hash value. Therefore, the key sizes will be the same for all schemes and are not listed in Table 1.

Table 2. Efficiency of hash-based many-time signature schemes. We assume key supply for at least 3 650 signatures, 256-bit hash functions, 32-bit time values, and time-stamping hash-tree with 33 levels. Times are in hashing operations and signature sizes in hash values. TS in BLT schemes stands for the time-stamping service call.

Scheme	Key generation	Signing time	Verification time	Signature size
XMSS	897 024	8 574	1 151	79
SPHINCS	ca 16 000	ca 250 000	ca 7 000	ca 1 200
BLT-TB	ca 96 000 000	50 + TS	25 + 33	25 + 33
BLT-OT-N	240 900	64 + TS	45 + 33	44 + 33
BLT-W-N ($w = 2$)	240 900	64 + TS	61 + 33	28 + 33

Efficiency as Many-Time Scheme. A one-time signature scheme is not practical by itself. Merkle [13] proposed aggregating multiple public keys of a one-time scheme using a hash tree to produce so-called N-time schemes. Assuming 10 signing operations per day, a set of 3 650 BLT-OT keys would be sufficient for a year. The key generation costs would obviously grow correspondingly. The change in signing time would depend on how the hash tree would be handled. If sufficient memory is available to keep the tree (which does not contain private key material and thus may be stored in regular memory), the authenticating hash chains for individual one-time public keys could be extracted with no extra hash computations. Signature size and verification time would increase by the 12 additional hashing steps linking the one-time public keys to the root of the aggregation tree. This scheme is listed as BLT-OT-N in Table 2, where we compare it with the following schemes:

– XMSS is a stateful scheme, like the N-time scheme built from BLT-OT; the values in Table 2 are computed by taking $N = 2^{12} = 4\,096$ and leaving other parameters as in [3];

– SPHINCS is a stateless scheme and can produce an indefinite number of signatures; the values in Table 2 are inferred from [2] counting invocations of the ChaCha12 cipher on 64-byte inputs as hash function evaluations;
– the values for BLT-TB in Table 2 are from [7].

As can be seen from the table, the performance of BLT-OT as a component in N-time scheme is very competitive when signing and verification time and signature size are concerned. Only SPHINCS has significantly faster key generation, but much slower signing and verification and much larger signatures.

6 Conclusions and Outlook

We have presented a new approach to constructing digital signature schemes from forward-resistant tags and time-stamping services. We observe that this new framework can be used to model an existing signature scheme, and also to construct new ones. The newly derived signature schemes are practical and it would be interesting to further study their security properties, e.g. present security proofs in the standard model. The novel concept of forward-resistant tags has already proven useful, and thus certainly merits further research.

References

1. Benaloh, J., de Mare, M.: Efficient broadcast time-stamping. Technical report, Clarkson University (1991)
2. Bernstein, D.J., et al.: SPHINCS: practical stateless hash-based signatures. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 368–397. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_15
3. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - a practical forward secure signature scheme based on minimal security assumptions. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 117–129. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25405-5_8
4. Buldas, A., Firsov, D., Laanoja, R., Lakk, H., Truu, A.: A new approach to constructing digital signature schemes (extended paper). Cryptology ePrint Archive, Report 2019/673 (2019). https://eprint.iacr.org/2019/673
5. Buldas, A., Kroonmaa, A., Laanoja, R.: Keyless signatures' infrastructure: how to build global distributed hash-trees. In: Riis Nielson, H., Gollmann, D. (eds.) NordSec 2013. LNCS, vol. 8208, pp. 313–320. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41488-6_21
6. Buldas, A., Laanoja, R., Laud, P., Truu, A.: Bounded pre-image awareness and the security of hash-tree keyless signatures. In: Chow, S.S.M., Liu, J.K., Hui, L.C.K., Yiu, S.M. (eds.) ProvSec 2014. LNCS, vol. 8782, pp. 130–145. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12475-9_10
7. Buldas, A., Laanoja, R., Truu, A.: A server-assisted hash-based signature scheme. In: Lipmaa, H., Mitrokotsa, A., Matulevičius, R. (eds.) NordSec 2017. LNCS, vol. 10674, pp. 3–17. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70290-2_1

8. Buldas, A., Saarepera, M.: On provably secure time-stamping schemes. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 500–514. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30539-2_35
9. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inf. Theory **22**(6), 644–654 (1976)
10. European Commission: Regulation no 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/EC (eIDAS regulation). Official Journal of the European Union L 257, 73–114 (2014)
11. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM J. Comput. **17**(2), 281–308 (1988)
12. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. J. Cryptol. **3**(2), 99–111 (1991)
13. Merkle, R.C.: Secrecy, authentication and public key systems. Ph.D. thesis, Stanford University (1979)
14. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-48184-2_32

Appendix 4

Publication IV
A. Buldas, A. Truu, R. Laanoja, and R. Gerhards. Efficient record-level keyless signatures for audit logs. In *NordSec 2014, Proceedings*, volume 8788 of *LNCS*, pages 149–164. Springer, 2014

Efficient Record-Level Keyless Signatures for Audit Logs

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

[1] Guardtime AS, Tallinn, Estonia
{ahto.buldas,ahto.truu,risto.laanoja}@guardtime.com
[2] Adiscon GmbH, Großrinderfeld, Germany
rgerhards@adiscon.com

Abstract. We propose a log signing scheme that enables (a) verification of the integrity of the whole log, and (b) presentation of any record, along with a compact proof that the record has not been altered since the log was signed, without leaking any information about the contents of other records in the log. We give a formal security proof of the scheme, discuss practical considerations, and provide an implementation case study.

Keywords: applied security, secure logging, keyless signatures, cryptographic time-stamps, `syslog`, `rsyslog`.

1 Introduction

Increasingly, logs from various information systems are used as evidence and also the requirements on maintenance and presentation of the log data are growing. Availability is clearly the most important property. If the logs are not available when needed, any other qualities of the log data don't really matter. However, ensuring availability is outside of the scope of the current discussion.

Integrity and authenticity—confidence that the log has not been tampered with or replaced with another—are also quite obvious requirements, especially if the log is to be admitted as evidence in legal proceedings. Signing and time-stamping are standard solutions for proving authenticity and integrity of data.

As information systems usually log all their activities sequentially, often the details of the relevant transactions are interspersed with other information in a log. To protect the confidentiality of the unrelated events, it is desirable to be able to extract records from the signed log and still prove their integrity.

An example of such a case is a dispute between a bank and a customer. On one hand, the bank can't just present the whole log, as the log contains also information about transactions of other customers. On the other hand, the customer involved in the dispute should have a chance to verify the integrity of the relevant records. This feature has been asked for by several European financial institutions. Similar concerns arise in the context of multi-tenant cloud environments. Hence, an ideal log signing scheme should have the following properties:

– The integrity of the whole log can be verified by the owner of the log: no records can be added, removed or altered undetectably.

- The integrity of any record can be proven to a third party without leaking any information about the contents of any other records in the log.
- The signing process is efficient in both time and space. (Ideally, there is a small constant per-record processing overhead and a small constant per-log storage overhead.)
- The extraction process is efficient in both time and space. (Ideally, a small constant-sized proof of integrity can be extracted for any record in time sub-linear in the size of the log.)
- The verification process is efficient in time. (Ideally, it should be running in time linear in the size of the data to be verified—whether verifying the whole log or a single record.)

1.1 Related Work

Schneier and Kelsey [14] proposed a log protection scheme that encrypts the records using one-time keys and links them using cryptographic hash functions. The scheme allows both for verification of the integrity of the whole log and for selective disclosure of the one-time encryption keys. However, it needs a third party trusted by both the logger and the verifier and requires active participation of this trusted party in both phases of the protocol.

Holt [8] replaced the symmetric cryptographic primitives used in [14] and enabled verification without the trusted party. However, Holt's scheme requires public-key signatures on individual records, which adds high computational and storage overhead to the logging process. Also, the size of the information required to prove the integrity of one record is at least proportional to the square root of the distance of the record from the beginning of the log. Other proposed amendments [15,1] to the protocol from [14] have similar weaknesses.

Kelsey *et al* [11] proposed a log signing scheme where records are signed in blocks, by first computing a hash value of each record in a block and then signing the sequence of hash values. This enables efficient verification of the integrity of the whole block, significantly reduces the overhead compared to having a signature per record, and also removes the need to ship the whole log block when a single record is needed as evidence. But still the size of the proof of a record is linear in the size of the block. Also, other records in the same block are not protected from the informed brute-force attack discussed in Sec. 2.1.

Ma and Tsudik [12] utilised the authentication technique they called *FSSA* (*Forward-Secure Sequential Aggregate*) to construct a logging protocol which provides *forward-secure stream integrity*, retaining the provable security of the underlying primitives. They also proposed a possibility to store individual signatures to gain better granularity, at the expense of storage efficiency.

1.2 Our Contribution

We propose a log signing scheme based on Merkle tree aggregation [13]. While using Merkle trees to aggregate data before signing is not new, we are not aware of previous applications in logging context. Aside from application case study, our main scientific contribution is the method for generating multiple blinding masks from a single random value to achieve blinding that is provably as good as using independently generated random masks.

Compared to previous log signing schemes with selective disclosure, our method offers improvements as follows: unlike [14], or scheme does not require a trusted third party; the proof of integrity of a record is $O(\log N)$ in our scheme, compared to $O(\sqrt{N})$ in [8] and $O(N)$ in [11]; while the asymptotic complexities are not directly comparable, based on the performance comparison table provided in [12], where the best-case signer computation cost is 5.55 ms per log record (albeit on slightly weaker hardware than in our experiment), we can estimate that our scheme is two to three orders of magnitude faster.

An extended version of the paper with more technical details is available from the Cryptology ePrint Archive [5].

2 Data Model

We now present the design of a signing scheme that will allow us to achieve almost all of the goals (there will be some trade-offs on the efficiency goals, but no compromises on the security goals).

A computational process producing a log may, in principle, run indefinitely and thus the log as an abstract entity may not have a well-defined beginning and end. In the following, we model the log as an ordered sequence of blocks, where each block in turn is an ordered sequence of a finite number of records. Many practical logging systems work this way, for example in the case of `syslog` output being sent to a log file that is periodically rotated.

Signing each record individually would, of course, have very high overhead in both processing and storage, as signing is quite expensive operation and the size of a signature may easily exceed the size of a typical log record. More importantly, it would also fail to fully ensure the integrity of the log as a whole—deletion of a record along with its signature would not be detected. Signing each log block as a unit would satisfy all the requirements related to processing of the whole block, but would make it impossible to prove the integrity of individual records without exposing everything else in the block.

An improvement over both of the above naive strategies would be to compute a hash value of each record in a log block and then sign the sequence of hash values instead of the records themselves (as proposed by Kelsey *et al* in [11]). This would ensure the integrity of the whole log block, significantly reduce the overhead compared to signing each record separately and also remove the need to ship the whole log block when a single record is needed as evidence. But still the size of the proof of a record would be linear in the size of the block (and the latter could easily run into multiple millions of records for a busy system).

2.1 Merkle Trees with Blinding Masks

To further reduce the size of the evidence for a single record, the records can instead be aggregated using a data structure known as Merkle tree—a binary

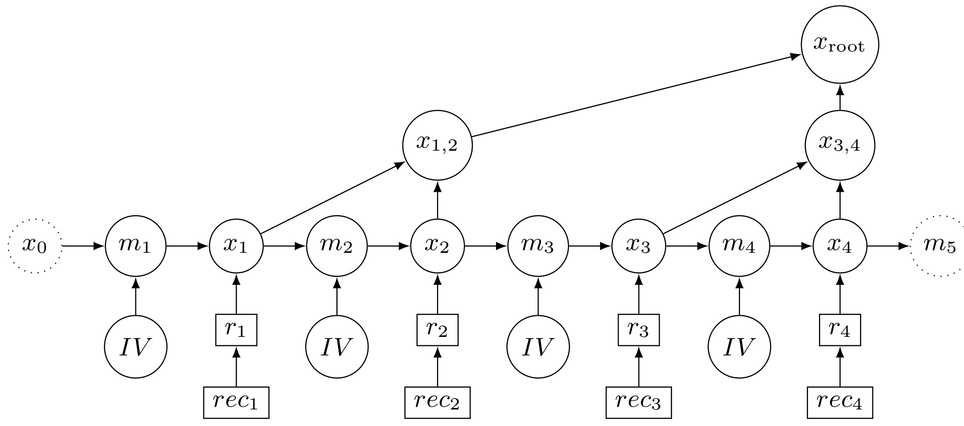

Fig. 1. Log signing using a Merkle tree with interlinks and blinding masks: rec_i are the log records; r_i are the hash values of the records; IV is the random seed; m_i are the blinding masks; x_i are leaves and $x_{a,b}$ are internal nodes of the Merkle tree; x_{root} is the value to be signed

tree whose leaves are the hash values of the records and each non-leaf node is the hash value of the concatenation its child nodes. The hash value in the root node of the tree can then be signed and for each leaf node a compact (logarithmic in the number of leaves) proof extracted showing that the hash value in the leaf participated in the computation that led to the signed root hash value. [13]

There are two complications left to be dealt with. The first one is that the security of such an aggregation scheme against retroactive fabrication of hash chains can in general be proven only if some restrictions are placed on the hash chains allowed as participation proofs. Fortunately, just appending the height of the sub-tree to the concatenated hash values from the child nodes before hashing is sufficient. This limits the length of the hash chains accepted during verification and allows for the security of the scheme to be formally proven. [4]

The second complication is that the hash chain extracted from the Merkle tree for one node contains the values of other nodes. A strong hash function can't be directly reversed to learn the input value from which a hash value in the chain was created. However, a typical log record may contain insufficient entropy to make that argument—an attacker who knows the pattern of the input could exhaustively test all possible variants to find the one that yields the hash value actually in the chain and thus learn the contents of the record. To prevent this kind of informed brute-force attack, a blinding mask with sufficient entropy can be added to each record before aggregating the hash values.

Generating cryptographically secure random values is expensive. Also, when an independent random mask would be used for each record, all these values would have to be stored for later verification. It is therefore much more efficient to derive all the blinding masks from a single random seed, as in the data structure shown on Fig. 1, where each node with incoming arrows contains the hash value of the concatenation of the contents of the respective source nodes.

3 Security Proof

We now show that the integrity proof for any record does not leak any information about the contents of any other records. The security of the scheme against modification of the log data has already been shown in [4].

We give the proof under the PRF assumption. Informally, the PRF assumption means that a 2-to-1 hash function $h : \{0,1\}^{2n} \to \{0,1\}^n$ can be assumed to behave like a random function $\Omega : \{0,1\}^n \to \{0,1\}^n$ when the first half of the input is a randomly chosen secret value $r \leftarrow \{0,1\}^n$.

Definition 1 (PRF, Pseudo-Random Function Family). *By an S-secure* pseudo-random function family *we mean an efficiently computable two-argument function h, such that if the first argument r is randomly chosen then the one-argument function $h(r,\cdot)$ (given to the distinguisher as a black box without direct access to r) is S-indistinguishable from the true random oracle Ω of the same type, i.e. for any t-time distinguisher D:*

$$\mathsf{Adv}(D) = \left| \Pr\left[1 \leftarrow D^{h(r,\cdot)}\right] - \Pr\left[1 \leftarrow D^{\Omega(\cdot)}\right] \right| \le \frac{t}{S} \ ,$$

where $r \leftarrow \{0,1\}^n$, $h : \{0,1\}^n \times \{0,1\}^p \to \{0,1\}^m$ and $\Omega : \{0,1\}^p \to \{0,1\}^m$.

Note that the PRF assumption is very natural when h is a 2-to-1 hash function, considering the design principles of hash functions, especially of those constructed from block ciphers.

Definition 2 (IND-CPA, Indistinguishability under Chosen-Plaintext Attack). *The log signing scheme is said to be S-secure* IND-CPA content concealing, *if any t-time adversary $A = (A_1, A_2)$ has success probability $\delta \le \frac{t}{S}$ in the following attack scenario (Fig. 2, left):*

1. *The first stage A_1 of the adversary chooses the position i and a list of records $rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell$, as well as two test records rec_i^0, rec_i^1 and an advice string a.*
2. *The environment picks randomly $x_0 \leftarrow 0^n$, $IV \leftarrow \{0,1\}^n$ and $b \leftarrow \{0,1\}$, assigns $rec_i \leftarrow rec_i^b$ and for every $j \in \{1, \ldots, \ell\}$ computes: $m_j \leftarrow h(IV, x_{j-1})$, $r_j \leftarrow \mathsf{H}(rec_j)$, and $x_j \leftarrow h(m_j, r_j)$.*
3. *The second stage A_2 of the adversary, given as input the advice string a and the lists of hash values $x_1, \ldots, x_\ell$, and $m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell$, tries to guess the value of b by outputting the guessed value $\hat{b}$.*

The advantage of A is defined as $\delta = 2\left|\Pr\left[\hat{b} = b\right] - \frac{1}{2}\right|$.

Theorem 1. *If $h(IV,\cdot)$ is an S-secure pseudorandom function family, then the log signing scheme is $\frac{S}{4}$-secure IND-CPA content concealing.*

```
Game_0 or the original attack game:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell, rec_i^0, rec_i^1, a \leftarrow A_1$
2. $x_0 \leftarrow 0^n$, $IV \leftarrow \{0,1\}^n$, $b \leftarrow \{0,1\}$
3. $rec_i \leftarrow rec_i^b$
4. for each $j \in \{1, \ldots, \ell\}$:
   (a) $m_j \leftarrow h(IV, x_{j-1})$
   (b) $r_j \leftarrow \mathsf{H}(rec_j)$
   (c) $x_j \leftarrow h(m_j, r_j)$
5. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell, m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$
6. if $\hat{b} = b$ then output 1
   else output 0
```

```
Game_2 or the Simulator $\mathcal{S}$:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell, rec_i^0, rec_i^1, a \leftarrow A_1$
2. $x_0 \leftarrow 0^n$, $b \leftarrow \{0,1\}$
3. $rec_i \leftarrow rec_i^b$
4. for each $j \in \{1, \ldots, \ell\}$:
   (a) if $\exists k < j : x_{j-1} = x_{k-1}$
       then $m_j \leftarrow m_k$
       else $m_j \leftarrow \{0,1\}^n$
   (b) $r_j \leftarrow \mathsf{H}(rec_j)$
   (c) if $j = i$
       then $x_j \leftarrow \{0,1\}^n$
       else $x_j \leftarrow h(m_j, r_j)$
5. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell, m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$
6. if $\hat{b} = b$ then output 1
   else output 0
```

Fig. 2. The original attack game and the simulator

Proof. Let A be a t-time adversary with success δ. We define three games Game_0, Game_1, and Game_2, where Game_0 is the original attack game, Game_2 is a simulator in which the input of A_2 does not depend on b and hence $\Pr\left[\hat{b} = b\right] = \frac{1}{2}$ in this game (Fig. 2), and Game_1 is an intermediate game, where A tries to break the scheme with independent random masks (Fig. 3), i.e. the mask generation steps $m_j \leftarrow h(IV, x_{j-1})$ are replaced with independent uniform random choices $m_j \leftarrow \{0,1\}^n$. However, as m_j is a function of x_{j-1}, we will use the same value for m_j and m_k in case $x_{j-1} = x_{k-1}$. This allows us to view the numbers m_j as outputs of a random oracle Ω and perfect simulation is possible.

```
Game_1 or the intermediate game:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell, rec_i^0, rec_i^1, a \leftarrow A_1$
2. $x_0 \leftarrow 0^n$, $b \leftarrow \{0,1\}$
3. $rec_i \leftarrow rec_i^b$
4. for each $j \in \{1, \ldots, \ell\}$:
   (a) if $\exists k < j : x_{j-1} = x_{k-1}$
       then $m_j \leftarrow m_k$
       else $m_j \leftarrow \{0,1\}^n$
   (b) $r_j \leftarrow \mathsf{H}(rec_j)$
   (c) $x_j \leftarrow h(m_j, r_j)$
5. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell, m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$
6. if $\hat{b} = b$ then output 1 else output 0
```

Fig. 3. The intermediate game for the security proof

```
Distinguisher $D_{01}^{\Phi}$:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell,$
   $rec_i^0, rec_i^1, a \leftarrow A_1$
2. $x_0 \leftarrow 0^n$, $b \leftarrow \{0,1\}$
3. $rec_i \leftarrow rec_i^b$
4. for each $j \in \{1, \ldots, \ell\}$:
   (a) $m_j \leftarrow \Phi(x_{j-1})$
   (b) $r_j \leftarrow \mathsf{H}(rec_j)$
   (c) $x_j \leftarrow h(m_j, r_j)$
5. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell,$
   $m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$
6. if $\hat{b} = b$ then output 1
   else output 0
```

```
Distinguisher $D_{12}^{\Phi}$:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell,$
   $rec_i^0, rec_i^1, a \leftarrow A_1$
2. $x_0 \leftarrow 0^n$, $b \leftarrow \{0,1\}$
3. $rec_i \leftarrow rec_i^b$
4. for each $j \in \{1, \ldots, \ell\}$:
   (a) if $\exists k < j : x_{j-1} = x_{k-1}$
           then $m_j \leftarrow m_k$
           else $m_j \leftarrow \{0,1\}^n$
   (b) $r_j \leftarrow \mathsf{H}(rec_j)$
   (c) if $j = i$
           then $x_j \leftarrow \Phi(r_i)$
           else $x_j \leftarrow h(m_j, r_j)$
5. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell,$
   $m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$
6. if $\hat{b} = b$ then output 1
   else output 0
```

Fig. 4. The distinguishers

Let δ_i $(i = 0, 1, 2)$ denote the probability that the adversary correctly guessed the value of b (i.e. $\hat{b} = b$) in the game Game_i. Hence, $\delta_2 = \frac{1}{2}$ and $\delta = 2\,|\delta_0 - \delta_2|$. We will now show that the games are negligibly close in terms of the adversary's advantage, and hence the adversary's success cannot be considerably higher in the original attacking game Game_0 compared to the simulator $\mathcal{S}$. To show that Game_0 is close to Game_1, we define a distinguisher D_{01}^{Φ} (with running time $t_1 \approx t$) between $\Phi = \Omega$ and $\Phi = h(r, \cdot)$ with success at least $|\delta_0 - \delta_1|$. Similarly, to show that Game_1 is close to Game_2, we define a distinguisher D_{12}^{Φ} (with running time $t_2 \approx t$) between $\Phi = \Omega$ and $\Phi = h(r, \cdot)$ with success at least $|\delta_1 - \delta_2|$.

The distinguisher D_{01}^{Φ} is constructed (Fig. 4, left) so that in case of the oracle $h(IV, \cdot)$ it perfectly simulates Game_0 (the original attacking game), and in case of the random oracle $\Omega(\cdot)$ it perfectly simulates Game_1 (where random masks are used). Hence, $\mathsf{Adv}(D_{01}^{\Phi}) = |\delta_0 - \delta_1|$.

The other distinguisher D_{12}^{Φ} is constructed (Fig. 4, right) so that in case of the oracle $h(IV, \cdot)$ it perfectly simulates Game_1, whereas in case of the random oracle Ω it simulates Game_2 (the simulator $\mathcal{S}$). Hence, $\mathsf{Adv}(D_{12}^{\Phi}) = |\delta_1 - \delta_2|$, and

$$\begin{aligned}\delta &= 2\,|\delta_0 - \delta_2| \leq 2\,(|\delta_0 - \delta_1| + |\delta_1 - \delta_2|) = 2\left(\mathsf{Adv}(D_{01}^{\Phi}) + \mathsf{Adv}(D_{12}^{\Phi})\right) \\ &\leq 2\left(\frac{t_1}{S} + \frac{t_2}{S}\right) = 4\frac{t}{S}\ ,\end{aligned}$$

and hence the log signing scheme is $\frac{S}{4}$-secure IND-CPA content concealing. $\square$

4 Reference Algorithms

We now present reference algorithms for aggregating a log block, extracting an integrity proof for an individual record, and verifying a record based on such proof. We also discuss some potential trade-offs where additional security benefits or runtime reductions could be gained at the cost of increased storage overhead.

4.1 Canonical Binary Trees

So far we have not specified the shape of the Merkle tree. If the number of leaves is an even power of two, building a perfect binary tree seems natural, but in other cases the appropriate shape is not necessarily obvious.

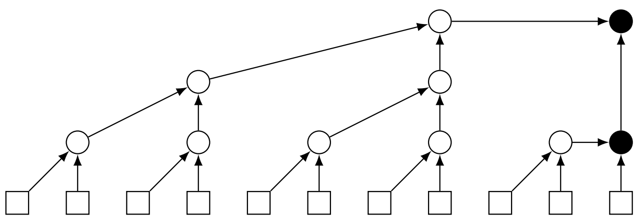

Fig. 5. A canonical binary tree: 11 leaves (squares) grouped into three perfect trees (white circles) and merged into a single tree with minimal height (black circles)

Of course, it is crucial to build the tree in a deterministic manner so that the verifier would be able to construct the exact same tree as the signer did. Another consideration is that to achieve the logarithmic size of the integrity proofs of the individual records, the tree should not be overly unbalanced. Thus, we define the *canonical binary tree* with n leaf nodes (shown for $n = 11$ on Fig. 5) to be built as follows:

1. The leaf nodes are laid out *from left to right* (square nodes on the figure).
2. The leaf nodes are collected into perfect binary trees *from left to right*, making each tree as big as possible using the leaves still available (adding the white circles on the figure).
3. The perfect trees are merged into a single tree *from right to left* which means joining the two smallest trees on each step (adding the black circles on the figure).

A useful property of canonical trees is that they can be built on-line, as the leaf nodes arrive, without knowing in advance the eventual size of the tree, and keeping in memory only logarithmic number of nodes (the root nodes of the complete binary trees constructed so far).

4.2 Aggregation of Log Records

Algorithm AGGR (Fig. 6) aggregates a block of records into a canonical Merkle tree for signing or verification. The input description numbers the records $1 \ldots N$, but the value of N is not used and the algorithm can easily be implemented for processing the records on-line.

The algorithm also enforces the hash chain length limiting as mentioned in Sec. 2.1. The level of a leaf node is defined to be 1 and the level of a non-leaf node to be 1 more than the maximum of the levels of its child nodes.

An amortized constant number of hashing operations is needed per record and the worst-case actual processing time per record is logarithmic in the number of records in the block, as is the size of the auxiliary working memory needed.
To sign a log block, AGGR could be used in the following manner:

1. A fresh random value is generated for IV.
2. The IV, the last leaf hash value from the previous block, and the log records of the current block are fed into AGGR.
3. The resulting root hash value is signed and the last leaf hash value from this block passed on to aggregation of the next block.
4. At the very least the IV and the signature on the root hash value must be saved for later verification.

Given the above, the way to verify a signed log block is quite obvious:

1. The IV saved during signing, the last leaf hash value from the previous block, and the log records are fed into AGGR.
2. The re-computed root hash value is verified against the saved signature.

A placeholder value filled with zeros is used for the last leaf hash value of the previous block in the very first block of the log (when there is no previous block) or when there has been a discontinuity in the log (for example, when the logging service has been down).

Although not strictly required in theory, in practice the last leaf hash value of the previous log block should also be saved along with the IV and the signature. Otherwise the verification of the current block would need to re-hash the previous block to obtain the required input, which in turn would need to re-hash the next previous block, etc. While this would obviously be inefficient, an even more dangerous consequence would be that any damage to any log block would make it impossible to verify any following log blocks, as one of the required inputs for verification would no longer be available.

Considering the negative scenarios in more detail, the only conclusion that can be derived from a failed verification in the minimal case above would be that something has been changed in either the log block or the authentication data. If it is desirable to be able to detect the changes more precisely, either the record hash values r_i or the leaf hash values x_i computed by AGGR could be saved along with the other authentication data. Then the sequence of hash values could be authenticated against the signature and each record checked against its hash value, at the expense of small per-record storage overhead.

Algorithm AGGR:

Aggregates a block of records for signing or verification.

```
inputs
    $rec_{1...N}$: input records
    $IV$: initial value for the blinding masks
    $x_0$: last leaf hash of previous block (zero for first block)
do
    {Initialize block:}
    {create empty roots list}
    $R :=$ empty list
    {Process records:}
    {add to Merkle forest in order}
    for $i := 1$ to $N$ do
        $r_i := \text{hash}(rec_i)$
        $m_i := \text{hash}(x_{i-1}, IV)$
        $x_i := \text{hash}(m_i, r_i, 1)$
        {Add $x_i$ to the forest as new leaf}
        {and update roots list}
        $t := x_i$
        for $j := 1$ to length($R$) do
            if $R_j =$ none then
                $R_j := t$; $t :=$ none
            else if $t \neq$ none then
                $t := \text{hash}(R_j, t, j + 1)$
                $R_j :=$ none
        if $t \neq$ none then
            $R := R \mid\mid t$; $t :=$ none
    {Finalize block:}
    {merge forest into a single tree}
    $root :=$ none
    for $j := 1$ to length($R$) do
        if $root =$ none then
            $root := R_j$; $R_j :=$ none
        else if $R_j \neq$ none then
            $root := \text{hash}(R_j, root, j + 1)$
            $R_j :=$ none
outputs
    $root$: root hash of this block (to be signed or verified)
    $x_N$: last leaf hash of this block (for linking next block)
```

Algorithm EXTR:

Extracts the hash chain for proving or verifying an individual record.

```
inputs
    $rec_{1...N}$: input records
    $pos$: position of the object record within the block $(1 \ldots N)$
    $IV$: initial value for the blinding masks
    $x_0$: last leaf hash of previous block (zero for first block)
do
    {Initialize block}
    $R :=$ empty list; $C :=$ empty list
    {object record not in any level yet}
    $\ell :=$ none
    {Process records,}
    {keeping track of the object}
    for $i := 1$ to $N$ do
        $r_i := \text{hash}(rec_i)$
        $m_i := \text{hash}(x_{i-1}, IV)$
        $x_i := \text{hash}(m_i, r_i, 1)$
        if $i = pos$ then
            $C := C \mid\mid (\textbf{right}, m_i, 0)$ {compute $x_i$}

            {add $x_i$ as a right leaf}
            $\ell := 1$; $d :=$ right
        {Add $x_i$ to the forest as new leaf}
        $t := x_i$
        for $j := 1$ to length($R$) do
            if $R_j =$ none then
                if $j = \ell$ then $d :=$ left
                $R_j := t$; $t :=$ none
            else if $t \neq$ none then
                if $j = \ell$ then
                    if $d =$ right then
                        $S := R_j$
                    else
                        $S := t$
                    $C := C \mid\mid (d, S, 0)$
                    $\ell := j + 1$; $d :=$ right
                $t := \text{hash}(R_j, t, j + 1)$
                $R_j :=$ none
        if $t \neq$ none then
            if length($R$) $< \ell$ then $d :=$ left
            $R := R \mid\mid t$; $t :=$ none
    {Finalize block}
    $root :=$ none
    for $j := 1$ to length($R$) do
        if $root =$ none then
            if $j = \ell$ then $d :=$ right
            $root := R_j$; $R_j :=$ none
        else if $R_j \neq$ none then
            if $j \geq \ell$ then
                if $d =$ right then
                    $S := R_j$
                else
                    $S := root$
                $C := C \mid\mid (d, S, j - \ell)$
                $\ell := j + 1$; $d :=$ right
            $root := \text{hash}(R_j, root, j + 1)$
            $R_j :=$ none
outputs
    $C$: hash chain from the object to the root of the block
```

Fig. 6. Algorithms AGGR and EXTR

It should also be noted that when the record hashes are saved, they should be kept with the same confidentiality as the log data itself, to prevent the informed brute-force attack discussed in Sec. 2.1.

4.3 Extraction of Hash Chains

Algorithm EXTR (Fig. 6) extracts the hash chain needed to prove or verify the integrity of an individual record. The core is similar to AGGR, with additional tracking of the hash values that depend on the object record and collecting a hash chain based on that tracking.

The output value is a sequence of (*direction*, *sibling hash*, *level correction*) triples. The direction means the order of concatenation of the incoming hash value and the sibling hash value. The level correction value is needed to account for cases when two sub-trees of unequal height are merged and the node level value increases by more than 1 on the step from the root of the lower sub-tree to the root of the merged tree. (The step from the lower black node to the higher one on Fig. 5 is an example.)

Because EXTR is closely based on AGGR, its performance is also similar and thus it falls somewhat short of our ideal of sub-linear runtime for hash chain extraction. We do not expect this to be a real issue in practice, as locating the records to be presented as evidence is typically already a linear-time task and thus reducing the proof extraction time would not bring a significant improvement in the total time. Also note that the need to access the full log file in this algorithm is not a compromise of our confidentiality goals, as the extraction process is executed by the owner of the log file and only the relevant log records and the hash chains computed for them by EXTR are shipped to outside parties.

```
Algorithm Comp:
Re-computes the root hash of a block for verifying a record
  inputs
    $rec$: input record
    $C$: hash chain from the record to the root of block
  do
    $root := \text{hash}(rec)$; $\ell := 0$
    for $i := 1$ to length($C$) do
      {direction, sibling, level correction}
      $(d, S, L) := C_i$
      $\ell := \ell + L + 1$
      if $d$ = left then
        $root := \text{hash}(root, S, \ell)$
      else
        $root := \text{hash}(S, root, \ell)$
  outputs
    $root$: root hash of the block (for verification)
```

Fig. 7. Algorithm COMP re-computes the root hash of a block for verifying a record

However, it would be possible to trade space for time and extra confidentiality, if desired. At the cost of storing two extra hash values per record, logarithmic runtime could be achieved and the need to look at any actual records during the hash chain extraction could be removed. Indeed, if the hash values from all the Merkle tree nodes (x_i on Fig. 1) were kept, the whole hash chain could be extracted without any new hash computations. If the values (all of the same fixed size) would be stored in the order in which they are computed as x_i and R_j in AGGR, each of them could be seeked to in constant time.

4.4 Computation of Hash Chains

Algorithm COMP (Fig. 7) computes the root hash value of the Merkle tree from which the input hash chain was extracted. The hash chain produced by EXTR and the corresponding log record should be fed into COMP, and the output hash value verified against the signature to prove the integrity of the record.

5 Implementation

In this section we outline some practical concerns regarding the implementation of the proposed scheme for signing `syslog` messages. Out of the many possible deployment scenarios we concentrate on signing the output directed to a text file on a log collector device. (See [7], Sec. 4.1 for more details.)

5.1 General Technical Considerations

Log File Rotation, Block Size. In Sec. 2 we modeled the log as an ordered sequence of blocks, where each block in turn is an ordered sequence of a finite number of records, and noted that the case of `syslog` output being sent to a periodically rotated log file could be viewed as an instantiation of this model.

We now refine the model to distinguish the logical blocks (implied by signing) from the physical blocks (the rotated files), because it is often desirable to sign the records in a finer granularity than the frequency of file rotation. A log file could contain several signed blocks, but for file management reasons, a signed block should be confined to a single file. This means that when log files are rotated, the current signature block should always be closed and a new one started from the beginning of the new file. The hash links from the last record of previous block to the first record of the next block should span the file boundaries, though, to enable verification of the integrity of the whole log, however the files may have been rotated.

Implementations could support limiting the block sizes both by number of records and time duration. When a limit on the number of records is set and a block reaches that many records, it should be signed and a new block started. Likewise, when a duration limit is set and the oldest record in a block reaches the given age, the block should be signed and a new one started. When both limits are set, reaching either one should cause the block to be closed and also reset both counters. Applying both limits could then be useful for systems with uneven activity. In this case the size limit would prevent the blocks growing too big during busy times and the time limit would prevent the earlier records in a block staying unsigned for too long during quiet times. When neither limit is given, each block would cover a whole log file, as rotating the output files should close the current block and start a new one in any case.

Record Canonicalization. When log records are individually hashed for signing before they are saved to the output file, it is critical that the file could be unambiguously split back into records for verification. End-of-line markers are used as separators in text-based `syslog` files and then multi-line records could not be recovered correctly unless the line breaks within the records are escaped.

Signature Technologies. Once the log blocks are aggregated, the root hash values have to be protected from future modifications. While it could seem natural to sign them using a standard public-key signature such as an OpenPGP [6] or PKCS#7 [10,9] one, these off-line signing technologies do not provide good forward security in case the logging server is compromised or privileged insiders abuse their access. An attacker could modify the logs and then re-hash and re-sign all the blocks starting from the earliest modified one, as the signing keys would be available on the log collector host.

A cryptographic time-stamping service [2] could be used as a mitigation. Note that a time-stamp generally only provides evidence of the time and integrity of the time-stamped datum, but not the identity of the requesting party, so a time-stamp alone would not be sufficient to prevent a log file from another system being submitted instead of the original one. Therefore, time-stamps should be used in addition to, not in place of, signatures.

An alternative would be to use the OpenKSI keyless signatures [3] that combine the hash value of the signed datum, the identity of the requesting system, and the signing time into one independently verifiable cryptographic token. As verification of keyless signatures does not rely on secrecy of keys and does not need trusted third parties, they are well-suited when logs are signed for evidence.

5.2 Case Study: `rsyslog` Integration

The proposed log signing scheme has been implemented in `rsyslog`, a popular logging server implementing the `syslog` protocol and included as a standard component in several major Linux distributions.

Architecture and Configuration. The `rsyslog` server has a modular architecture, with a number of input and output modules available to receive log messages from various sources and store logs using different formats and storage engines. Log signing has been implemented as a new optional functionality in the `omfile` output module that stores log data in plain text files. When signing is enabled, the signatures and related helper data items are stored in a binary file next to the log file. As the data is a sequence of non-human-readable binary tokens (hash values and signatures), there would be no benefit in keeping it in a text format. Both files are needed for verification of log integrity. It is the responsibility of the log maintenance processes to ensure that the files do not get separated when the log files are archived or otherwise managed. The `rsyslog` configuration is specified as a number of sets of rules applied to incoming messages. The following minimal example configures the server to listen for incoming messages on TCP port 514 and apply the `perftest` rules to all of them:

```
module(load="imtcp")
input(type="imtcp" port="514" ruleset="perftest")
ruleset(name="perftest"){
  action(type="omfile" file="/var/log/signed.log" sig.provider="gt")
}
```

The rules in turn just write all the records to the specified text file with no filtering or transformations.

Performance. We tested the performance of the implementation on a quad-core Intel Xeon E5606 CPU. We used 64-bit CentOS 6.4 operating system and installed `rsyslog` version 7.3.15-2 from the Adiscon package repository. There was excess of memory and I/O resources, in all tests the performance was CPU-bound. Load was generated using the `loggen` utility which is part of the `syslog-ng 3.5.0 alpha0` package, another `syslog` server implementation. We used TCP socket for logging in order to avoid potential message loss and mimic a real-life scenario with central logging server. We note that UDP and kernel device interface gave comparable results. The `rsyslog` configuration was as shown above: the simplest possible, without any message processing.

Without signing we achieved sustained logging rate of ≈400,000 messages per second. At this point the `rsyslog` input thread saturated one CPU core. Multiple input threads and multiple main queue worker threads allowed us to achieve slightly better performance. Here and below the average message size was 256 bytes, and the default SHA2-256 hash was used when signing was enabled.

Signed logging rate was constantly higher than 100,000 messages per second. The limiting factor was the main queue worker thread which saturated one CPU core. For one signed output file the building of the hash tree can't be parallelized in an efficient way, because the order of the log messages must be preserved. Although possible to configure, multiple parallel worker queues would spend most of their time waiting for synchronization and total signing performance would be inferior. Storage of the record hash values and intermediate Merkle tree hash values did not affect the signing performance significantly. Also, using different hash algorithms did not have a significant impact.

Aggregating a log message incurs approximately three hash algorithm invocations. Considering that one CPU core can perform roughly one million hash calculations on small inputs, the log signing performance achieved is reasonably close to optimal. It should also be noted that the four-fold decrease of throughput from 400,000 messages per second to 100,000 messages per second is extreme, as in the baseline scenario no CPU power was spent on filtering the records.

Storage overhead depends on whether the record and tree hashes are stored, hash algorithm output size, and signature block size. In case of 256-byte log records and 32-byte hash values, the storage overhead is about 12% for keeping the record hashes and about 25% for keeping the tree hashes. The storage overhead caused by signatures themselves is negligible in practical scenarios.

Table 1. Storage, runtime and verification feature trade-offs (N is log block size)

Characteristic	**No hashes kept**	**Record hashes**	**Tree hashes**
	Signing		
Per-record storage	none	1 hash value	2 hash values
Per-record computation	3 hashings	3 hashings	3 hashings
Per-block storage	1 signature value	1 signature value	1 signature value
Per-block computation	1 signing	1 signing	1 signing
Memory	$O(\log N)$	$O(\log N)$	$O(\log N)$
	Whole log verification		
Report granularity	block	record	record
Time	$O(N)$	$O(N)$	$O(N)$
Memory	$O(\log N)$	$O(\log N)$	$O(\log N)$
	Record proof extraction		
Per-record storage	$O(\log N)$*	$O(\log N)$*	$O(\log N)$*
Time	$O(N)$	$O(N)$	$O(\log N)$
Memory	$O(\log N)$	$O(\log N)$	$O(1)$
	Record proof verification		
Report granularity	record	record	record
Time	$O(\log N)$	$O(\log N)$	$O(\log N)$
Memory	$O(1)$	$O(1)$	$O(1)$

* Asymptotically it's $O(\log N)$, but in practice, the $O(1)$ signature size dominates over the $O(\log N)$ hash chain size. For example, for 3600-byte signatures and 32-byte hash values used in our case study, the signature size exceeds the hash chain size for all $N < 2^{100}$.

6 Conclusions

We have proposed a log signing scheme with good security properties and low overhead in both computational and storage resources. The integrity of either the whole log or any record can be verified, and in the latter case also a compact proof produced without leaking any information about the contents of other records. The scheme also allows for some flexibility in providing additional verification features and proof extraction performance gains at the cost of increased storage overhead, as listed in Table 1.

An interesting direction for future development would be to extend the scheme for use in configurations where the log is signed at a source device, transported over one or more relay devices and then stored at a collector device. The current scheme would be usable only if the relay devices do not perform any filtering of the records, which is rarely the case in practice.

References

1. Accorsi, R.: BBox: A distributed secure log architecture. In: Camenisch, J., Lambrinoudakis, C. (eds.) EuroPKI 2010. LNCS, vol. 6711, pp. 109–124. Springer, Heidelberg (2011)
2. Adams, C., Cain, P., Pinkas, D., Zuccherato, R.: Internet X.509 public key infrastructure time-stamp protocol (TSP). IETF RFC 3161 (2001)
3. Buldas, A., Kroonmaa, A., Park, A.: OpenKSI digital signature format (2012)
4. Buldas, A., Saarepera, M.: On provably secure time-stamping schemes. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 500–514. Springer, Heidelberg (2004)
5. Buldas, A., Truu, A., Laanoja, R., Gerhards, R.: Efficient record-level keyless signatures for audit logs. Cryptology ePrint Archive, Report 2014/552 (2014)
6. Callas, J., Donnerhacke, L., Finney, H., Thayer, R.: OpenPGP message format. IETF RFC 4880 (2007)
7. Gerhards, R.: The syslog protocol. IETF RFC 5424 (2009)
8. Holt, J.E.: Logcrypt: Forward security and public verification for secure audit logs. In: Buyya, R., Ma, T., Safavi-Naini, R., Steketee, C., Susilo, W. (eds.) AISW 2006, pp. 203–211. Australian Computer Society (2006)
9. Housley, R.: Cryptographic message syntax (CMS). IETF RFC 5652 (2009)
10. Kaliski, B.: PKCS#7: Cryptographic message syntax v 1.5. IETF RFC 2315 (1998)
11. Kelsey, J., Callas, J., Clemm, A.: Signed syslog messages. IETF RFC 5848 (2010)
12. Ma, D., Tsudik, G.: A new approach to secure logging. ACM Transactions on Storage 5(1), 2:1–2:21 (2009)
13. Merkle, R.C.: Protocols for public key cryptosystems. In: IEEE Symposium on Security and Privacy, pp. 122–134. IEEE Computer Society (1980)
14. Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. ACM Transactions on Information Systems Security 2(2), 159–176 (1999)
15. Stathopoulos, V., Kotzanikolaou, P., Magkos, E.: A framework for secure and verifiable logging in public communication networks. In: López, J. (ed.) CRITIS 2006. LNCS, vol. 4347, pp. 273–284. Springer, Heidelberg (2006)

Curriculum Vitae

1. Personal data

Name	Ahto Truu
Date and place of birth	27 September 1972, Estonia
Nationality	Estonian

2. Contact information

Address	GuardTime, Küüni 5b, 51004 Tartu, Estonia
Phone	+372 5175876
E-mail	ahto.truu@guardtime.com

3. Education

2015–. . .	Tallinn University of Technology, School of Information Technologies Information and Communication Technology, PhD studies
1994–1997	University of Tartu, Faculty of Mathematics, Computer Science, MSc studies
1990–1994	University of Tartu, Faculty of Mathematics, Computer Science, BSc studies

4. Language competence

Estonian	native
English	fluent
Russian	intermediate

5. Professional employment

2008–. . .	GuardTime AS, software architect
2017–2019	Tallinn University of Technology, Department of Software Science, research engineer
1997–2008	AS Aprote / AS WM-data / AS Logica, software engineer / systems analyst
1995	Intergraph Corp (USA), software engineering intern
1993–1994	AS Magnum Medical, software developer

6. Voluntary work

1997–. . .	Estonian Olympiad in Informatics, member of jury

7. Defended theses

- 2010, Standards for Hash-Linking Based Time-Stamping Systems, M.Sc.
  Supervisor Prof. Ahto Buldas, Institute of Computer Science, University of Tartu
- 2007, Computational Geometry Puzzles, B.Sc.
  Supervisor Ass. Prof. Rein Prank, Institute of Computer Science, University of Tartu

8. Field of research

- Hash function based cryptography
- Authenticated data structures
- Provable security

9. Scientific work
Papers

1. M. Keren, A. Kirshin, J. Rubin, and A. Truu. MDA approach for maintenance of business applications. In *ECMDA-FA 2006, Proceedings*, volume 4066 of *LNCS*, pages 40–51. Springer, 2006
2. M. Opmanis, V. Dagienė, and A. Truu. Task types at Beaver contests. In *ISSEP 2006, Proceedings*, pages 509–519. Institute of Mathematics and Informatics, Vilnius, Lithuania, 2006
3. A. Truu and H. Ivanov. On using testing-related tasks in the IOI. In *Olympiads in Informatics: The International Conference joint with the XX International Olympiad in Informatics, Proceedings*, pages 171–180. Institute of Mathematics and Informatics, Vilnius, Lithuania, 2008
4. T. Poranen, V. Dagienė, Åsmund Eldhuset, H. Hyyrö, M. Kubica, A. Laaksonen, M. Opmanis, W. Pohl, J. Skūpienė, P. Söderhjelm, and A. Truu. Baltic olympiads in informatics: Challenges for training together. In *Olympiads in Informatics: The International Conference joint with the XXI International Olympiad in Informatics, Proceedings*, pages 112–131. Institute of Mathematics and Informatics, Vilnius, Lithuania, 2009
5. A. Buldas, R. Laanoja, P. Laud, and A. Truu. Bounded pre-image awareness and the security of hash-tree keyless signatures. In *ProvSec 2014, Proceedings*, volume 8782 of *LNCS*, pages 130–145. Springer, 2014
6. A. Buldas, A. Truu, R. Laanoja, and R. Gerhards. Efficient record-level keyless signatures for audit logs. In *NordSec 2014, Proceedings*, volume 8788 of *LNCS*, pages 149–164. Springer, 2014
7. A. Buldas, R. Laanoja, and A. Truu. Keyless signature infrastructure and PKI: Hash-tree signatures in pre- and post-quantum world. *IJSTM*, 23(1/2):117–130, 2017
8. A. Buldas, R. Laanoja, and A. Truu. A server-assisted hash-based signature scheme. In *NordSec 2017, Proceedings*, volume 10674 of *LNCS*, pages 3–17. Springer, 2017
9. A. Buldas, R. Laanoja, and A. Truu. A blockchain-assisted hash-based signature scheme. In *NordSec 2018, Proceedings*, volume 11252 of *LNCS*, pages 138–153. Springer, 2018
10. A. Buldas, D. Firsov, R. Laanoja, H. Lakk, and A. Truu. A new approach to constructing digital signature schemes (short paper). In *IWSEC 2019, Proceedings*, volume 11689 of *LNCS*, pages 363–373. Springer, 2019
11. A. Buldas, D. Firsov, R. Laanoja, and A. Truu. Verified security of BLT signature scheme. In *ACM SIGPLAN CPP 2020, Proceedings*, pages 244–257. ACM, 2020

Conference presentations

1. A. Truu, M. Keren. Deploying MDD for Business Application Maintenance. Second European Workshop ‘From Code Centric to Model Centric Software Engineering: Practices, Implications and Return on Investment’ (C2M), 11 July 2006, Bilbao, Spain
2. A. Truu, H. Ivanov. On Using Testing-Related Tasks in the IOI, Olympiads in Informatics: The International Conference Joint with the XX International Olympiad in Informatics, 16–23 August 2008, Cairo, Egypt

3. A. Buldas, A. Truu, R. Laanoja, R. Gerhards. Efficient Record-Level Keyless Signatures for Logs, NordSec 2014, 19th Nordic Conference on Secure IT Systems, 15–17 October 2014, Tromsø, Norway

4. A. Buldas, R. Laanoja, A. Truu. A Server-Assisted Hash-Based Signature Scheme, NordSec 2017, 22nd Nordic Conference on Secure IT Systems, 8–10 November 2017, Tartu, Estonia

5. A. Buldas, D. Firsov, R. Laanoja, H. Lakk, A. Truu. A New Approach to Constructing Digital Signature Schemes, IWSEC 2019, 14th International Workshop on Security, 28–30 August, 2019, Tokyo, Japan

6. A. Buldas, D. Firsov, R. Laanoja, A. Truu. Verified Security of BLT Signature Scheme, ACM SIGPLAN CPP 2020, 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, 20–21 January, 2020, New Orleans, USA

Elulookirjeldus

1. Isikuandmed

Nimi	Ahto Truu
Sünniaeg ja -koht	27.09.1972, Eesti
Kodakondsus	Eesti

2. Kontaktandmed

Aadress	GuardTime, Küüni 5b, 51004 Tartu
Telefon	+372 5175876
E-post	ahto.truu@guardtime.com

3. Haridus

2015–...	Tallinna Tehnikaülikool, infotehnoloogia teaduskond, informatsiooni- ja kommunikatsioonitehnoloogiad, doktoriõpe
1994–1997	Tartu Ülikool, matemaatikateaduskond, informaatika, magistriõpe
1990–1994	Tartu Ülikool, matemaatikateaduskond, informaatika, bakalaureuseõpe

4. Keelteoskus

eesti keel	emakeel
inglise keel	kõrgtase
vene keel	kesktase

5. Teenistuskäik

2008–...	GuardTime AS, tarkvaraarhitekt
2017–2019	Tallinna Tehnikaülikool, tarkvarateaduse instituut, insener
1997–2008	AS Aprote / AS WM-data / AS Logica, tarkvaraarendaja / süsteemianalüütik
1995	Intergraph Corp (USA), tarkvaraarenduse praktikant
1993–1994	AS Magnum Medical, tarkvaraarendaja

6. Vabatahtlik töö

1997–...	Eesti informaatikaolümpiaadi žürii liige

7. Kaitstud lõputööd

- 2010, Räsiahelatel põhinevate ajatemplisüsteemide standardid, M.Sc.
  Juhendaja prof. Ahto Buldas, arvutiteaduse instituut, Tartu Ülikool
- 2007, Arvutusgeomeetria ülesanded “Nupunurgas”, B.Sc.
  Juhendaja dots. Rein Prank, arvutiteaduse instituut, Tartu Ülikool

8. Teadustöö põhisuunad

- Räsifunktsioonidel põhinev krüptograafia
- Autentivad andmestruktuurid
- Tõestatav turvalisus

9. Teadustegevus

Teadusartiklite, konverentsiteeside ja konverentsiettekannete loetelu on toodud ingliskeelse elulookirjelduse juures.