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.

[⋆] This research was supported by the European Regional Development Fund through the Estonian smart specialization program NUTIKAS. The final publication is available at Springer via https://doi.org/10.1007/978-3-030-03638-6_9

The proposed signature scheme can be considered practical. It provides forward security, non-repudiation of the origin via efficient revocation; there are no 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 Sec. 2 we survey the state of the art in hash-based signature schemes, server-assisted signature schemes, and authenticated data structures. In Sec. 3 we define the design goals and outline the reasoning that led us to the new scheme. In Sec. 4 we specify the design of the new scheme and in Sec. 5 provide some notes on implementation. We wrap up with conclusions in Sec. 6.

2 Related Work

2.1 Hash-Based Signatures

The earliest signature scheme constructed from hash functions is due to Lamport [30, 22]. His scheme, as well as the refinements proposed in [33, 24, 23, 7, 27], 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, 39, 21]. 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 [32, 6, 9, 8, 28, 29].

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 [18, 10].

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.

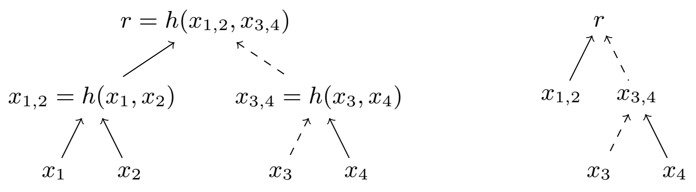

Fig. 1. The hash tree $T^h(x_1, \ldots, x_4)$ and the corresponding hash chain $x_3 \rightsquigarrow r$.

3.2 The BLT Signature Scheme

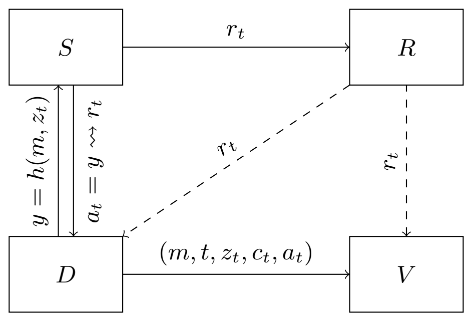

Fig. 2. Components of the BLT signature scheme.

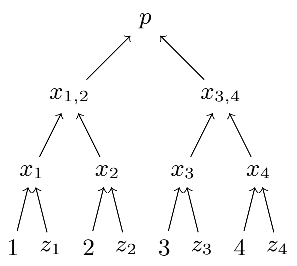

Fig. 3. Computation of public key for $N = 4$.

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.

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: $(z_1, \ldots, z_T) \leftarrow \mathcal{G}(T, k)$.
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 $t \in \{1, \ldots, T\}$.

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 Sec. 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 Sec. 4.3).

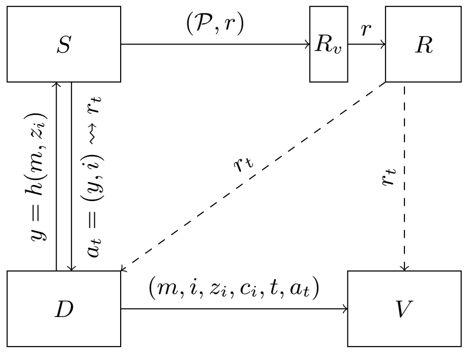

Fig. 4. Components of the new signature scheme.

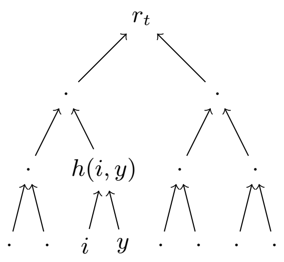

Fig. 5. Server tree for round t, showing key counter and input of the second client only.

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.

4.2 Setup

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

1. Generates N unpredictable k-bit signing keys: $(z_1, \ldots, z_N) \leftarrow \mathcal{G}(N, k)$.
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.

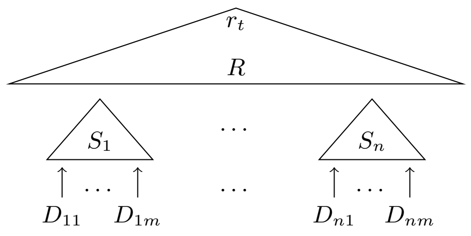

Fig. 6. A scalable deployment architecture for the new scheme.

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.

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. 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.
2. Asokan, G. Tsudik, and M. Waidner. Server-supported signatures. *Journal of Computer Security*, 5(1):91–108, 1997.
3. 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.
4. K. Bicakci and N. Baykal. Server assisted signatures revisited. In *CT-RSA 2004, Proceedings*, volume 2964 of *LNCS*, pages 143–156. Springer, 2004.
5. M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2-3):225–244, 1994.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 9–17. ACM, 2000.

16. 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.
17. A. Buldas and M. Saarepera. On provably secure time-stamping schemes. In *ASIACRYPT 2004, Proceedings*, volume 3329 of *LNCS*, pages 500–514. Springer, 2004.
18. 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.
19. L. C. Coronado García. *Provably Secure and Practical Signature Schemes.* PhD thesis, Darmstadt University of Technology, Germany, 2005.
20. S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symposium*, pages 317–334. USENIX, 2009.
21. 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.
22. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.
23. 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.
24. S. Even, O. Goldreich, and S. Micali. On-line/off-line digital signatures. *J. Cryptology*, 9(1):35–67, 1996.
25. V. Goyal. More efficient server assisted one time signatures. Cryptology ePrint Archive, Report 2004/135, 2004. https://eprint.iacr.org/2004/135.
26. S. Haber and W. S. Stornetta. How to time-stamp a digital document. *J. Cryptology*, 3(2):99–111, 1991.
27. A. Hülsing. W-OTS+—Shorter signatures for hash-based signature schemes. In *AFRICACRYPT 2013, Proceedings*, volume 7918 of *LNCS*, pages 173–188. Springer, 2013.
28. 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.
29. 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.
30. L. Lamport. Constructing digital signatures from a one way function. Technical report, SRI International, Computer Science Laboratory, 1979.
31. B. Laurie, A. Langley, and E. Kasper. Certificate transparency. RFC 6962, RFC Editor, June 2013.
32. 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.
33. R. C. Merkle. *Secrecy, Authentication and Public Key Systems.* PhD thesis, Stanford University, 1979.
34. 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.
35. A. Perrig. The BiBa one-time signature and broadcast authentication protocol. In *ACM CCS 2001, Proceedings*, pages 28–37. ACM, 2001.

36. A. Perrig, R. Canetti, J. D. Tygar, and D. Song. The TESLA broadcast authentication protocol. *CryptoBytes*, 5(2):2–13, 2002.
37. 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.
38. 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.
39. P. Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *ACM CCS'99, Proceedings*, pages 93–100. ACM, 1999.
40. R. Tamassia. Authenticated data structures. In *European Symposium on Algorithms, ESA 2003, Proceedings*, volume 2832 of *LNCS*, pages 2–5. Springer, 2003.