Key Evolving Signature Scheme - Formal Specification

Aaron Schutza

November 2021

1 Introduction

This document specifies a series of protocols that each realize a Key Evolving Signature (KES) scheme. The cryptographic routines considered here are designed to be used with proof-of-stake blockchian protocols that require protection against long range bribing attacks. The goal is to produce a key evolving protocol that executes in tandem with the round based synchronously driven execution of the staking procedure carried out in the blockchain protocol.

KES schemes are cryptographic procedures for validating signatures with a public piece of information that stays constant in time. They are set up in the usual public-private key-pair configuration but with the added notion of an evolving private key with a time-step and procedural deterministic updates. The time step in these constructions corresponds to an indexed state of the private key that increments in tandem with a trapdoor operation for deriving the next key configuration. These constructions provide *forward security* preventing past signatures from being reforged upon exposure of the private key that's in a future time step configuration. A necessary and sufficient condition for forward security is that the private key evolves in a way that past keys cannot be recovered from the present key configuration.

KES schemes may be composed from one or more time series of public-private key-pairs in an ordered derivation process where the time step can be parameterized at the protocol level. Some care must be taken in specifying these compositions since a past key must be non-recoverable from any future key configuration to ensure past signatures cannot be reforged. The KES public-private keys considered here are designed for use in the individual rounds in a blockchain protocol to sign block headers. The proposed Operation Composition is a coupling between a linear-secret-key scheme first put forth by Anderson [1], Bellare, and Miner [4] with the MMM construction [8]. The goal is to retain a high number of time steps per registration, while having the shortest signature size possible. The linear-secret-key composition enables a high number of time steps while retaining succinct proofs with the trade-off being that the secret key becomes large. The size of the linear-secret key is significantly reduced if the rounds to be signed are known in advance. This construction is designed for use

with round based proof-of-stake blockchain protocols where forward security is required and succinct signatures and public keys are desired for the headers and ledger. Test vectors are realized in this construction with an Ed25519 signing routine [5], and an evolving key and accumulator scheme that builds on the product composition from the MMM construction [8]. Finally the format of public-private keys and certificates in this KES construction is specified with test vectors.

2 Motivation

The use of a key evolving scheme is crucial for forward security of a proof-of-stake platform that does not have a checkpoint mechanism. The KES portion of this functionality prevents long range attacks in an environment where node operators may be coerced to reveal old private staking information. Secure erasure is performed in the KES setup and erases the key required to sign in a given time step. This means that honest activity will never be retroactively corrupted because the required private key cannot be recovered from the present evolved key configuration. We wish to maintain forward security in the shortest possible timescale of the protocol, meaning that each round update is accompanied by a secure erasure step preventing the signing of a block in that round. The key time steps should evolve in tandem with the individual rounds to maintain this level of forward security. This prevents producing blocks in rounds that have already transpired at some later time.

The linear-secret-key scheme (linear scheme) provides a protocol that authenticates a time series of KES public-private key-pairs that sign individual rounds. Key derivation occurs in an initialization step, where one master private key is used to authenticate the verification part of a cache of randomly selected public-private key-pairs (one for each time step). To ensure forward security, the master private key is discarded immediately while the master public key is recorded and registered with the blockchain protocol. As time goes on, the cache of private keys are used to sign block headers, while the private key cache is cleared of any keys past keys. The signatures that sign the KES keys are verifiable with the initial master public key included in the registration because each signature includes a proof that committed to the verification key of the intended time step. When composed together with multiple key sets (forming a multi-dimensional array of keys), the number of time steps can be increased drastically, while retaining reasonably sized private keys (on the order of kilobytes varying in time). This allows years worth of keys to be cached and securely erased with no intervention from the node operator.

The MMM construction has more succinct private keys but larger signatures. In any given time step, the private key is derivable from the parent configuration, leading to a simpler update procedure without any need to cache a large set of information. This yields a much more succinct private key, but the signatures are significantly larger when compared to the linear scheme. The initialization step in this construction is similar in spirit to that of the linear scheme, but a

deterministic hierarchical seeding of signing key-pairs is performed in a binary tree. The verification part of each of these keys are then Merkleized and witness paths are included in signatures that verify with the root of the Merkle tree, which is the public key in this setup. The witness path constrains the time step and the verification key of the signing routine, while a single signature commits to any message. Key updates are performed by deriving the next step from the parent configuration, where seeds of the initial binary tree are used to derive the next set of keys and then the previous configuration is securely erased.

We wish to compose these schemes together in a way that is ideal for the use case pattern of proof-of-stake blockchains. The MMM construction provides a limited number of time steps at the cost of kilobytes of space in block headers. This means that the key time step must evolve slower than individual blockchain rounds to keep the signatures small. The protocol must specify how many rounds correspond to a key time step and forward security is not guaranteed in that time period. This severely limits the number of rounds that a key can be valid for, as this time interval cannot exceed limits ultimately set by the desired confirmation depth of the blockchain protocol.

3 Ideal Functionality of a Key Evolving Scheme

Any KES scheme protocol must realize the ideal functionality of a key evolving signature scheme shown in Figure 3.1. For a more through treatment for realizing a KES ideal functionality, see [7] [3] [6].

Functionality $\mathcal{F}_{\mathsf{KES}}$

The functionality $\mathcal{F}_{\mathsf{KES}}$ is parameterized by the total number of signature updates T, interacting with a signer U_S and a registered set of parties $\mathcal{P}$ denoted by $U_i \in \mathcal{P}$ as follows:

Key Generation. Upon receiving a message $(\mathsf{KeyGen}, sid, U_S)$ from party U_S, send $(\mathsf{KeyGen}, sid, U_S)$ to the adversary. Upon receiving $(\mathsf{VerificationKey}, sid, U_S, v)$ from the adversary, send $(\mathsf{VerificationKey}, sid, v)$ to U_S, record the triple (sid, U_S, v) and set counter $\mathsf{k_{ctr}} = 1$.

Sign and Update. Upon receiving a message $(\mathsf{USign}, sid, U_S, m, j)$ from U_S, verify that (sid, U_S, v) is recorded for some sid and that $\mathsf{k_{ctr}} \leq j \leq T$. If not, then ignore the request. Else, set $\mathsf{k_{ctr}} = j + 1$ and send $(\mathsf{Respond}, (\mathsf{Sign}, sid, U_S, m, j))$ to the adversary. Upon receiving $(\mathsf{Signature}, sid, U_S, m, j, \sigma)$ from the adversary, verify that no entry $(m, j, \sigma, v, 0)$ is recorded. If it is, then output an error message to U_S and halt. Else, send $(\mathsf{Signature}, sid, m, j, \sigma)$ to U_S, and record the entry $(m, j, \sigma, v, 1)$.

Signature Verification. Upon receiving a message $(\mathsf{Verify}, sid, m, j, \sigma, v')$ from some party U_i do:

1. If $v' = v$ and the entry $(m, j, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
2. Else, if $v' = v$, the signer is not corrupted, and no entry $(m, j, \sigma', v, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. (This condition guarantees unforgeability: If v' is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
3. Else, if there is an entry (m, j, σ, v', f') recorded, then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
4. Else, if $j < \mathsf{k_{ctr}}$, let $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. Otherwise, if $j = \mathsf{k_{ctr}}$, send $(\mathsf{Verify}, sid, m, j, \sigma, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, sid, m, j, \phi)$ from the adversary, let $f = \phi$ and record the entry (m, j, σ, v', ϕ). (This condition guarantees that the adversary is only able to forge signatures under keys belonging to corrupted parties for time periods corresponding to the current or future slots.)

Output $(\mathsf{Verified}, sid, m, j, f)$ to U_i.

Figure 3.1: The Ideal Functionality of a Key Evolving Signature Scheme.

4 Sum Composition

This section specifies algorithms required to implement the sum composition. To realize this construction in a fashion that's independent of the number of time steps we make use of recursive function calls. The total number of time steps T is then parameterized by the height h of the binary tree used in this composition giving $T = 2^h$. We assume an underlying signing routine shown in Figure 4.1. This is an expansion of the original MMM exposition with clearly defined procedures and steps to help implementers understand and test their implementation. The goal is to specify the sum composition as a protocol Π_Σ with steps shown in Figure 4.2.

Definitions. Let $\mathcal{T}$ be the set of extended binary trees with elements $\tau \in \mathcal{T}$, such that τ is a rooted tree in which every node has at most two child nodes. Each node in τ has an associated value and child nodes that are elements of $\mathcal{T}$. Trees are referenced by their root node Node such that where $\tau = \mathsf{Node}[v, l, r]$

is a data structure containing the node's value v with left and right pointers denoted by l and r respectively. If $l \in \mathcal{T}$ and $r \in \mathcal{T}$ then τ is a node with l and r the left and right child nodes respectively. If $l = \text{null}$ and $r \in \mathcal{T}$ or $r = \text{null}$ and $l \in \mathcal{T}$ then τ has only one child. If $l = \text{null}$ and $r = \text{null}$ then τ has no children and is called a leaf.

The following algorithms assume access to a digital signature routine given in Figure 4.1. Binary strings are represented as $\{0,1\}^*$ where the wildcard $*$ denotes any length. Also assume that $H(\cdot) : \{0,1\}^* \rightarrow \{0,1\}^\ell$ is a secure cryptographic hash function where ℓ is the bit-length of its digest output. Cryptographic seeds are assumed to be exactly ℓ-bits long and should be generated from uniform entropy. An ordered data structure List with any number of elements is assumed, with associated functions for returning the length, head, and tail of the list.

A signature Σ in the sum composition is a data structure $\mathsf{SumSignature}$ such that $\Sigma = \mathsf{SumSignature}[vk, \sigma, W]$ where vk is the verification key of the signing routine, σ is a signature of the signing routine,and $W = \mathsf{List}[w_1, \ldots, w_{\ell_W}]$ is a witness path consisting of a list of length ℓ_W where the list elements $w_i \in W$ are hashes such that $w_i \in \{0,1\}^\ell$. The total number of time steps can be inferred from the length of the witness path and is ultimately set by the height of the binary tree upon key generation. All routines are designed to infer time step information from the height of the tree and the length of the witness path. For a tree of height $h > 0$ the witness path length is $\ell_W = h$ and the total number of time steps is $T = 2^h$. The total signature byte-length is then $\ell_\Sigma = \ell_{vk} + \ell_\sigma + h\ell$.

Signing Routine:

The signing routine is assumed to satisfy the properties of a strong digital signature routine with deterministic signatures. The signatures, secret keys, and verification keys are binary strings of length ℓ_σ, ℓ_{vk}, and ℓ_{sk} respectively.

Key Generation. $\mathsf{KeyGen} : s \to (sk, vk)$

Require: $s \in \{0,1\}^\ell$

Ensure: (sk, vk) where $sk \in \{0,1\}^{\ell_{sk}}$ and $vk \in \{0,1\}^{\ell_{vk}}$

From a provided seed s of length ℓ a tuple containing the secret key sk and verification key vk are returned.

Signature Creation. $\mathsf{Sign} : sk, m \to \sigma$

Require: $sk \in \{0,1\}^{\ell_{sk}}, m \in \{0,1\}^*$

Ensure: $\sigma \in \{0,1\}^{\ell_\sigma}$

Produces a signature σ that commits to message m that is signed with sk.

Signature Verification. $\mathsf{Verify} : vk, \sigma, m \to b$

Require: $vk \in \{0,1\}^{\ell_{vk}}, \sigma \in \{0,1\}^{\ell_\sigma}, m \in \{0,1\}^*$

Ensure: $b \in \{\mathbf{true}, \mathbf{false}\}$

Returns true if the provided signature σ verifies with the provided verification key vk and message m, returns false otherwise.

Figure 4.1: Definition of the signing routine used in the following algorithms and protocols.

Sum Composition Protocol Π_Σ:

The protocol is run by a registered set of parties $\mathcal{P}$ denoted by $U_i \in \mathcal{P}$ interacting with a signer U_S as follows:

Key Generation. Upon receiving the message $(\mathsf{KeyGen}, sid, U_S, h)$ from U_S, U_S does the following: pick a random s then compute $\kappa \leftarrow \mathsf{KeyGenSum}(s, h)$ and $R \leftarrow \mathsf{VerificationKeySum}(\kappa)$. Securely erase s and record κ, then send the message $(\mathsf{VerificationKey}, sid, R)$ to U_S.

Key Update. Upon receiving the message $(\mathsf{KeyUpdate}, sid, U_S, t)$ from U_S with a sid for which it has the signing key κ, U_S does the following, otherwise ignore the input: compute $\kappa' \leftarrow \mathsf{KeyUpdateSum}(\kappa, t)$ and securely erase κ. Record κ' and send the message $(\mathsf{Updated}, sid)$ to U_S.

Signature Creation. Upon receiving the message $(\mathsf{Sign}, sid, U_S, m, t)$ from U_S with a sid for which it has the signing key κ and that $t = \mathsf{KeyTimeSum}(\kappa)$, U_S does the following, otherwise ignore the input: compute $\Sigma_t \leftarrow \mathsf{SignSum}(\kappa, m)$, then send the message $(\mathsf{Signature}, sid, m, t, \Sigma_t)$ to U_S.

Signature Verification. When party U_i receives the message $(\mathsf{Verify}, sid, m, t, \Sigma_t, R)$, U_i computes $b \leftarrow \mathsf{VerifySumSignature}(R, \Sigma_t, t, m)$ and outputs the message $(\mathsf{Verified}, sid, m, t, b)$.

Figure 4.2: The MMM construction in the sum composition as a protocol.

Algorithm 1 $\mathsf{IsLeaf} : n \to \{\mathbf{true}, \mathbf{false}\}$

Require: $n \in \mathcal{T}$

1: $\mathsf{Node}[v, l, r] \leftarrow n$
2: **if** $l = \text{null} \wedge r = \text{null}$ **then**
3: **return true**
4: **else**
5: **return false**
6: **end if**

Algorithm 2 $\mathsf{DoublingPRNG} : s \to (\{0,1\}^\ell, \{0,1\}^\ell)$

Require: $s \in \{0,1\}^\ell$

1: $s_l \leftarrow H(\mathsf{0x00}||s)$
2: $s_r \leftarrow H(\mathsf{0x01}||s)$
3: **return** (s_l, s_r)

Algorithm 3 $\mathsf{SeedTree} : s, h \to \mathsf{Node}$

Require: $s \in \{0,1\}^{\ell}$, $h \in \mathbb{N}_0$

if $h > 0$ **then**
  $(s_l, s_r) \leftarrow \mathsf{DoublingPRNG}(s)$
  return $\mathsf{Node}(s_r, \mathsf{SeedTree}(s_l, h-1), \mathsf{SeedTree}(s_r, h-1))$
else
  $(sk, vk) \leftarrow \mathsf{KeyGen}(s)$
  return $\mathsf{Node}[(sk, vk), \mathrm{null}, \mathrm{null}]$
end if

Algorithm 4 $\mathsf{MerkleVK} : n \to \mathsf{Node}$

Require: $n \in \mathcal{T}$

if $\mathsf{IsLeaf}(n)$ **then**
  return n
else
  $\mathsf{Node}[v, l, r] \leftarrow n$
  $\mathsf{Node}[v_l, l_l, r_l] \leftarrow l$
  $\mathsf{Node}[v_r, l_r, r_r] \leftarrow r$
  $n_l \leftarrow \mathsf{MerkleVK}(l)$
  $n_r \leftarrow \mathsf{MerkleVK}(r)$
  if $\mathsf{IsLeaf}(l) \wedge \mathsf{IsLeaf}(r)$ **then**
    $(sk_l, vk_l) \leftarrow v_l$
    $(sk_r, vk_r) \leftarrow v_r$
    return $\mathsf{Node}[(v, H(vk_l), H(vk_r)), n_l, n_r]$
  else
    $\mathsf{Node}[x, l_{n_l}, r_{n_l}] \leftarrow n_l$
    $\mathsf{Node}[y, l_{n_r}, r_{n_r}] \leftarrow n_r$
    $(s_x, x_l, x_r) \leftarrow x$
    $(s_y, y_l, y_r) \leftarrow y$
    return $\mathsf{Node}[(v, H(x_l||x_r), H(y_l||y_r)), n_l, n_r]$
  end if
end if

Algorithm 5 $\mathsf{ReduceTree} : n \to \mathsf{Node}$

Require: $n \in \mathcal{T}$

if $\mathsf{IsLeaf}(n)$ **then**
  return n
else
  $\mathsf{Node}[v, l, r] \leftarrow n$
  return $\mathsf{Node}[v, \mathsf{ReduceTree}(l), \mathrm{null}]$
end if

Algorithm 6 KeyGenSum : $s, h \rightarrow \mathsf{Node}$

Require: $s \in \{0,1\}^{\ell}$, $h \in \mathbb{N}_0$

$\tau_1 \leftarrow \mathsf{SeedTree}(s, h)$
$\tau_2 \leftarrow \mathsf{MerkleVK}(\tau_1)$
$\tau_3 \leftarrow \mathsf{ReduceTree}(\tau_2)$
return τ_3

Algorithm 7 VerificationKeySum : $\tau \rightarrow \{0,1\}^{\ell}$

Require: $\tau \in \mathcal{T}$

$\mathsf{Node}[v, l, r] \leftarrow \tau$
$(s_r, w_l, w_r) \leftarrow v$
$R \leftarrow H(w_l || w_r)$
return R

Algorithm 8 Height : $n \rightarrow \mathbb{N}_0$

Require: $n \in \mathcal{T} \vee n = \text{null}$

if $n \in \mathcal{T}$ **then**
    if $\mathsf{IsLeaf}(n)$ **then**
        return 0
    else
        $\mathsf{Node}[v, l, r] \leftarrow n$
        return $\max(\mathsf{Height}(l), \mathsf{Height}(r)) + 1$
    end if
else
    return 0
end if

Algorithm 9 KeyTimeSum : $n \rightarrow \mathbb{N}_0$

Require: $n \in \mathcal{T}$

if $\mathsf{IsLeaf}(n)$ **then**
    return 0
else
    $\mathsf{Node}[v, l, r] \leftarrow n$
    if $l = \text{null} \wedge r \in \mathcal{T}$ **then**
        if $\mathsf{IsLeaf}(r)$ **then**
            return 1
        else
            $h \leftarrow \mathsf{Height}(r)$
            return $\mathsf{KeyTimeSum}(r) + 2^h$
        end if
    else
        return $\mathsf{KeyTimeSum}(l)$
    end if
end if

Algorithm 10 KeyUpdateSum : $n, t \to \mathsf{Node}$

Require: $n \in \mathcal{T}, t \in \mathbb{N}_0$

$t_{\mathrm{key}} \leftarrow \mathsf{KeyTimeSum}(n)$
$h \leftarrow \mathsf{Height}(n)$
if $t > t_{\mathrm{key}} \wedge t < 2^h$ **then**
    return $\mathsf{EvolveKey}(n, t)$
else
    return n
end if

Algorithm 11 EvolveKey : $n, t \to \mathsf{Node}$

Require: $n \in \mathcal{T}$

if $\mathsf{IsLeaf}(n)$ **then**
    return n
else
    $\mathsf{Node}[v, l, r] \leftarrow n$
    $h \leftarrow \mathsf{Height}(n)$
    $t' \leftarrow t \bmod 2^{h-1}$
    if $t \geq 2^{h-1}$ **then**
        if $l \in \mathcal{T} \wedge r = \mathrm{null}$ **then**
            if $\mathsf{IsLeaf}(l)$ **then**
                $(s_r, u_l, u_r) \leftarrow v$
                $(sk, vk) \leftarrow \mathsf{KeyGen}(s_r)$
                $n_r \leftarrow \mathsf{Node}[(sk, vk), \mathrm{null}, \mathrm{null}]$
                return $\mathsf{Node}[(\mathrm{null}, w_l, w_r), \mathrm{null}, n_r]$
            else
                $(s_r, w_l, w_r) \leftarrow v$
                $n_r \leftarrow \mathsf{KeyGenSum}(s_r, h-1)$
                $s'_r \leftarrow \{0\}^\ell$
                return $\mathsf{Node}[(s'_r, w_l, w_r), \mathrm{null}, \mathsf{EvolveKey}(n_r, t')]$
            end if
        else
            return $\mathsf{Node}[v, \mathrm{null}, \mathsf{EvolveKey}(r, t')]$
        end if
    else
        if $l = \mathrm{null} \wedge r \in \mathcal{T}$ **then**
            return $\mathsf{Node}[v, \mathrm{null}, \mathsf{EvolveKey}(r, t')]$
        else
            return $\mathsf{Node}[v, \mathsf{EvolveKey}(l, t'), \mathrm{null}]$
        end if
    end if
end if

Algorithm 12 SignSum : $n, m \to$ SumSignature

Require: $n \in \mathcal{T}, m \in \{0,1\}^*$

$W \leftarrow \mathsf{List}[\cdot]$
$\mathsf{Node}[v, l, r] \leftarrow n$
while $l \neq \text{null} \wedge r \neq \text{null}$ **do**
    $(s_r, w_l, w_r) \leftarrow v$
    if $l = \text{null} \wedge r \in \mathcal{T}$ **then**
        $W \leftarrow \mathsf{List}[w_l] \,||\, W$
        $\mathsf{Node}[v, l, r] \leftarrow r$
    else
        $W \leftarrow \mathsf{List}[w_r] \,||\, W$
        $\mathsf{Node}[v, l, r] \leftarrow l$
    end if
end while
$(sk, vk) \leftarrow v$
$\sigma \leftarrow \mathsf{Sign}(sk, m)$
return $\mathsf{SumSignature}[vk, \sigma, W]$

5 Product Composition

This section specifies the product composition, a way of expanding the number of time steps while retaining succinct proofs. This scheme first put forth in the MMM construction posits two key-evolving signature schemes composed together as a parent and child scheme. The parent scheme seeds and authenticates the child scheme. The underlying child scheme signs individual time steps that commits to the message to be signed. The parent scheme doesn't depend on the message and instead signs the public verification part of child keys and increments once each child key lifetime while deterministically seeding the next child key. Figure 7.1 shows a protocol executing the product composition in a symmetric configuration, where two identical signing routines in the sum composition are used. The product private key consists of two evolving signing keys, and its time steps are the Cartesian product of the ordered sets of keys. Keys in the product composition protocol $\Pi_\Sigma^\otimes = \Pi_\Sigma \otimes \Pi_\Sigma$ may be thought of as a set of signing routines

$$\{\Pi_\Sigma^\otimes(t) : 0 \leq t < T\} = \{\Pi_\Sigma(i) : 0 \leq i < T_i\} \otimes \{\Pi_\Sigma(j) : 0 \leq j < T_j\} \quad (1)$$

where $\Pi_\Sigma(j)$ and j increments until T_j for each increment of t, and if $j+1 = T_j$ then $j \to 0$ and $i \to i+1$ until $i+1 = T_i$. Thus $t = iT_j + j$ and the total number of time steps is then $T = T_i T_j$. This scheme utilizes two evolving keys, where a parent scheme is used to authenticate a child scheme in a chain of signatures. This style of composition may be generalized to chain together any number of evolving keys forming a higher order product composition. For example, a triple

Algorithm 13 $\mathsf{VerifySumSignature} : R, \Sigma, t, m \rightarrow \{\mathbf{true}, \mathbf{false}\}$

Require: $R \in \{0,1\}^{\ell}, \Sigma \in \mathsf{SumSignature}, t \in \mathbb{N}_0, m \in \{0,1\}^*$

$\mathsf{SumSignature}[vk, \sigma, W] \leftarrow \Sigma$
$b_\sigma \leftarrow \mathsf{Verify}(vk, \sigma, m)$
$b_w \leftarrow \mathbf{true}$
if $\text{length}(W) > 0$ **then**
    $w_l \leftarrow \text{null}$
    $w_r \leftarrow \text{null}$
    $h \leftarrow \text{length}(W)$
    if $t \bmod 2 = 0$ **then**
        $w_l \leftarrow H(vk)$
        $w_r \leftarrow \text{head}(W)$
    else
        $w_l \leftarrow \text{head}(W)$
        $w_r \leftarrow H(vk)$
    end if
    $W \leftarrow \text{tail}(W)$
    while $\text{length}(W) > 0$ **do**
        $h' \leftarrow h - \text{length}(W)$
        if $(t/2^{h'}) \bmod 2 = 0$ **then**
            $w_l \leftarrow H(w_l || w_r)$
            $w_r \leftarrow \text{head}(W)$
        else
            $w_r \leftarrow H(w_l || w_r)$
            $w_l \leftarrow \text{head}(W)$
        end if
        $W \leftarrow \text{tail}(W)$
    end while
    $b_w \leftarrow b_w \wedge R = H(w_l || w_r)$
else
    $b_w \leftarrow b_w \wedge R = H(vk)$
end if
return $b_\sigma \wedge b_w$

product composition may be written

$$\begin{aligned}&\{\Pi_{\Sigma}^{\otimes}(t) : 0 \leq t < T\} = \\ &\{\Pi_{\Sigma}(i) : 0 \leq i < T_i\} \otimes \{\Pi_{\Sigma}(j) : 0 \leq j < T_j\} \otimes \{\Pi_{\Sigma}(k) : 0 \leq k < T_k\}\end{aligned} \tag{2}$$

that uses 3 evolving schemes, where the time step is $t = iT_jT_k + jT_k + k$ and the total number of time steps is $T = T_iT_jT_k$. The ordering of the schemes imply that the left most scheme authenticates the verification part of the its child on the right, in turn that scheme authenticates the scheme to its right, where the right most scheme authenticates the message being signed. Figure 5.2 show a diagram of different product compositions.

This is effectively a multidimensional array of signing keys where the time step parameterization of t in the overall composition can be arbitrarily chosen. This precisely what's done in the MMM construction. In that setup, the product composition child scheme is parameterized with respect to the time step of the parent scheme. The child scheme grows in size with each parent scheme time step. This produces practically unbounded time steps, but leads to degrading performance and growing proof sizes as the time step increases. This limitation makes that specific parameterization of the product composition impractical for use in proof-of-stake blockchain protocols.

The product composition can still be used while producing relatively succinct proofs of constant size while providing a sufficient number of time steps to authenticate individual blockchain protocol rounds. The product composition $\Pi_{\Sigma}^{\otimes}$ uses two binary trees in the sum-composition to represent the time step configuration. The configuration with the most succinct proofs is symmetric in the two sum-signature routines, where the height of either tree is the same. We refer to this setup as a symmetric-product scheme. This gives better performance than a sum-composition scheme on its own with a comparable number of time steps.

Define a data structure $\mathsf{ProductKey}$ such that $\mathsf{ProductKey}[\tau_1, \sigma_1, s, \tau_2]$ corresponds to the private key for $\Pi_{\Sigma}^{\otimes}$ where $\tau_1 \in \mathcal{T}$, $\sigma_1 \in \mathsf{SumSignature}$, $s \in \{0,1\}^{\ell}$, and $\tau_2 \in \mathcal{T}$. Define $\mathsf{ProductSignature}$ as signature data structure $\mathsf{ProductSignature}[\sigma_1, \sigma_2, R_2]$ where $\sigma_1 \in \mathsf{SumSignature}$, $\sigma_2 \in \mathsf{SumSignature}$, and $R_2 \in \{0,1\}^{\ell}$.

Product Composition Protocol $\Pi_{\Sigma}^{\otimes} = \Pi_{\Sigma}^{1} \otimes \Pi_{\Sigma}^{2}$:
The protocol is run by a registered set of parties $\mathcal{P}$ denoted by $U_i \in \mathcal{P}$ interacting with a signer U_S as follows:

Key Generation. Upon receiving the message $(\mathsf{KeyGen}, sid, U_S, h_1, h_2)$ from U_S, U_S does the following: pick a random s then compute $\kappa = \mathsf{KeyGenProduct}(s, h_1, h_2)$ and $R \leftarrow \mathsf{VerificationKeyProduct}(\kappa)$. Securely erase s and record κ, then send the message $(\mathsf{VerificationKey}, sid, R)$ to U_S.

Key Update. Upon receiving the message $(\mathsf{KeyUpdate}, sid, U_S, t)$ from U_S with a sid for which it has the signing key κ, U_S does the following, otherwise ignore the input: compute $\kappa' \leftarrow \mathsf{KeyUpdateProduct}(\kappa, t)$ and record κ'. Securely erase κ and send the message $(\mathsf{Updated}, sid)$ to U_S.

Signature Creation. Upon receiving the message $(\mathsf{Sign}, sid, U_S, m, t)$ from U_S with a sid for which it has the signing key κ and that $t \leftarrow \mathsf{KeyTimeProduct}(\kappa)$, U_S does the following, otherwise ignore the input: compute $\Sigma_t \leftarrow \mathsf{SignProduct}(\kappa, m)$, then send the message $(\mathsf{Signature}, sid, m, t, \Sigma_t)$ to U_S.

Signature Verification. When party U_i receives the message $(\mathsf{Verify}, sid, m, t, \Sigma_t, R)$, U_i computes $b \leftarrow \mathsf{VerifyProductSignature}(R, \Sigma_t, t, m)$ and outputs the message $(\mathsf{Verified}, sid, m, t, b)$.

Figure 5.1: The product signing routine in the product composition of two sum-composition protocols Π_{Σ}^{1} and Π_{Σ}^{2} as a protocol.

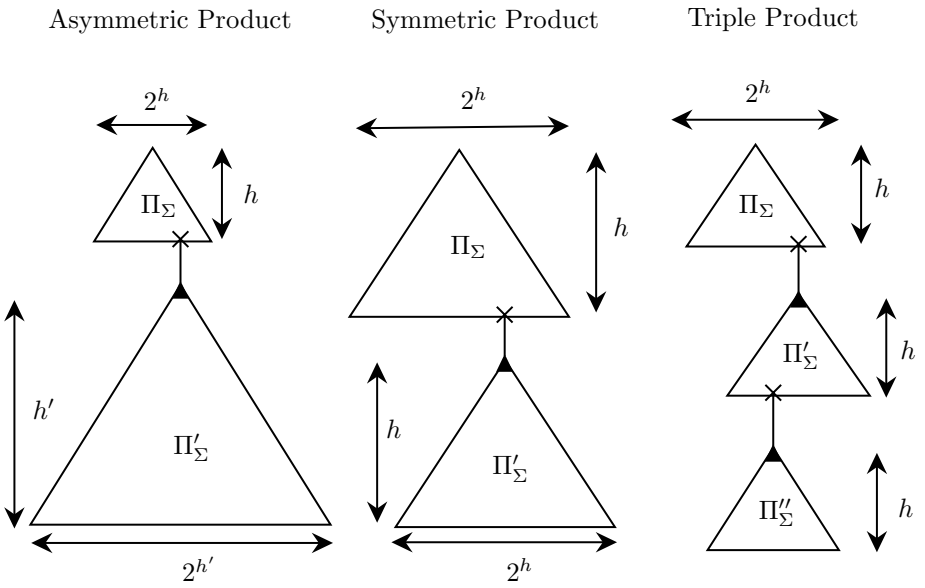

Figure 5.2: A diagram of the different schemes using variations of products of the sum composition Π_Σ represented as triangles (suggestive of the Merkle tree structure where the root is at the top and the leaves are at the bottom). The vertical axis indicates the depth of the trees and the horizontal axis indicates the time steps of each respective sum composition. In each setup the topmost scheme is the parent that authenticates the child scheme underneath it, e.g. the triple product scheme would be written as $\Pi_\Sigma^\otimes = \Pi_\Sigma \otimes \Pi'_\Sigma \otimes \Pi''_\Sigma$. The lowermost child scheme signs the messages in the overall product composition and the other witness signatures authenticate the time step.

Algorithm 14 KeyGenProduct : $s, h_1, h_2 \to$ ProductKey

Require: $s \in \{0,1\}^\ell$, $h_2 \in \mathbb{N}_0$, $h_2 \in \mathbb{N}_0$

1: $(s_1, s_2) \leftarrow \mathsf{DoublingPRNG}(s)$
2: $(s_3, s_4) \leftarrow \mathsf{DoublingPRNG}(s_2)$
3: $\tau_1 \leftarrow \mathsf{KeyGenSum}(s_1, h_1)$
4: $\tau_2 \leftarrow \mathsf{KeyGenSum}(s_3, h_2)$
5: $R_2 \leftarrow \mathsf{VerificationKeySum}(\tau_2)$
6: $\sigma_1 \leftarrow \mathsf{SignSum}(\tau_1, R_2)$
7: $\tau'_1 \leftarrow \mathsf{EraseLeafSK}(\tau_1)$
8: **return** $\mathsf{ProductKey}(\tau'_1, \sigma_1, s_4, \tau_2)$

Algorithm 15 VerificationKeyProduct : $\kappa \to \{0,1\}^\ell$

Require: $\kappa \in$ ProductKey

1: ProductKey$[\tau_1, \Sigma, s, \tau_2] \leftarrow \kappa$
2: **return** VerificationKeySum(τ_1)

Algorithm 16 KeyTimeProduct : $\kappa \to \mathbb{N}_0$

Require: $\kappa \in$ ProductKey

1: ProductKey$[\tau_1, \Sigma, s, \tau_2] \leftarrow \kappa$
2: $h_2 \leftarrow$ Height(τ_2)
3: $t_1 \leftarrow$ KeyTimeSum(τ_1)
4: $t_2 \leftarrow$ KeyTimeSum(τ_2)
5: **return** $t_1 2^{h_2} + t_2$

Algorithm 17 SignProduct : $\kappa, m \to$ ProductSignature

Require: $\kappa \in$ ProductKey, $m \in \{0,1\}^*$

1: ProductKey$[\tau_1, \sigma_1, s, \tau_2] \leftarrow \kappa$
2: $\sigma_2 \leftarrow$ SignSum(τ_2, m)
3: $R_2 \leftarrow$ VerificationKeySum(τ_2)
4: **return** ProductSignature$[\sigma_1, \sigma_2, R_2]$

Algorithm 18 VerifyProductSignature : $R, \Sigma, t, m \to \{\mathbf{true}, \mathbf{false}\}$

Require: $R \in \{0,1\}^\ell$, $\Sigma \in$ ProductSignature, $t \in \mathbb{N}_0$, $m \in \{0,1\}^*$

1: ProductSignature$[\sigma_1, \sigma_2, R_2] \leftarrow \Sigma$
2: SumSignature$[vk, \sigma, W] \leftarrow \sigma_2$
3: $h_2 \leftarrow \text{length}(W)$
4: $t_1 \leftarrow t/2^{h_2}$
5: $t_2 \leftarrow t \bmod 2^{h_2}$
6: $b_1 \leftarrow$ VerifySumSignature(R, σ_1, t_1, R_2)
7: $b_2 \leftarrow$ VerifySumSignature(R_2, σ_2, t_2, m)
8: **return** $b_1 \wedge b_2$

Algorithm 19 $\mathsf{EraseLeafSK} : n \to \mathsf{Node}$

Require: $n \in \mathcal{T}$

1: $\mathsf{Node}[v, l, r] \leftarrow n$
2: **if** $\mathsf{IsLeaf}(n)$ **then**
3: $(sk, vk) \leftarrow v$
4: $sk' \leftarrow \{0\}^{\ell_{sk}}$
5: **return** $\mathsf{Node}[(sk', vk), \text{null}, \text{null}]$
6: **else**
7: **if** $l = \text{null} \wedge r \in \mathcal{T}$ **then**
8: **return** $\mathsf{Node}[v, \text{null}, \mathsf{EraseLeafSK}(r)]$
9: **else**
10: **return** $\mathsf{Node}[v, \mathsf{EraseLeafSK}(l), \text{null}]$
11: **end if**
12: **end if**

6 Linear KES Scheme

This section specifies the linear scheme in a key-evolving setup. Figure 6.1 shows a schematic of the key cache and set of verification keys. The indexing scheme may be designed in a way that provides practically any number of time steps, but to maintain forward security secret keys have to be erased after they are used to make signatures.

Algorithm 20 KeyUpdateProduct : $\kappa, t \to$ ProductKey

Require: $\kappa \in$ ProductKey, $t \in \mathbb{N}_0$

ProductKey$[\tau_1, \sigma_1, s, \tau_2] \leftarrow \kappa$
$t_{\text{key}} \leftarrow$ KeyTimeProduct(κ)
$h_1 \leftarrow$ Height(τ_1)
$h_2 \leftarrow$ Height(τ_2)
if $t > t_{\text{key}} \wedge t < 2^{h_1+h_2}$ **then**
    $i \leftarrow$ KeyTimeSum(τ_1)
    $t_1 \leftarrow t/2^{h_2}$
    $t_2 \leftarrow t \bmod 2^{h_2}$
    if $i < t_1$ **then**
        $s_1 \leftarrow \text{null}$
        $s_2 \leftarrow s$
        while $i < t_1$ **do**
            $(s_1, s_2) \leftarrow$ DoublingPRNG(s_2)
            $i \leftarrow i + 1$
        end while
        $\tau_1' \leftarrow$ EvolveKey(τ_1, t_1)
        $\tau_2' \leftarrow$ KeyGenSum(s_1, h_2)
        $R_2' \leftarrow$ VerificationKeySum(τ_2')
        $\sigma_1' \leftarrow$ SignSum(τ_1', R_2')
        $\tau_2' \leftarrow$ EvolveKey(τ_2', t_2)
        $\tau_1'' \leftarrow$ EraseLeafSK(τ_1')
        return ProductKey$[\tau_1'', \sigma_1', s_2, \tau_2']$
    else
        $\tau_2' \leftarrow$ KeyUpdateSum(τ_2, t_2)
        return ProductKey$[\tau_1, \sigma_1, s, \tau_2']$
    end if
else
    return κ
end if

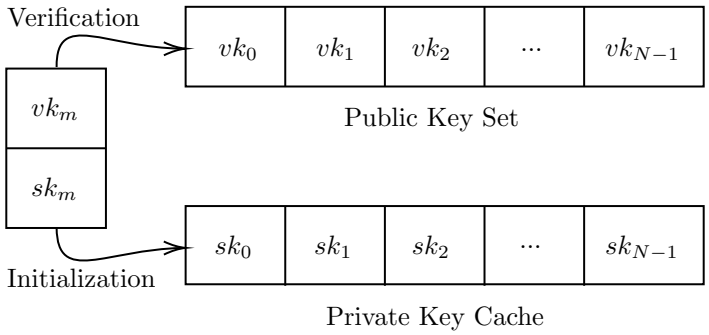

Figure 6.1: A diagram showing the key sets used with the linear scheme in a setup where there are N total time steps. Upon initialization, a key pair sk_m and vk_m is chosen at random. sk_m is used to sign the key cache during initialization and then sk_m is securely erased. Signatures are produced from the private key cache $\{sk_i, \text{Sign}(sk_m, i||vk_i) : 0 \leq i < N\}$ and each time step corresponds to an index i where each sk_i is chosen randomly. As time increments, secure erasure in time step t corresponds to setting the private key cache to $\{sk_i, \text{Sign}(sk_m, i||vk_i) : t \leq i < N\}$ and erasing $\{sk_i, \text{Sign}(sk_m, i||vk_i) : 0 \leq i < t\}$. Upon verification of time step t with a given message and signature $(\text{Sign}(sk_m, t||vk_t), vk_t, \text{Sign}(sk_t, \text{message}))$, vk_m is used to authenticate $\text{Sign}(sk_m, t||vk_t)$ and vk_t is used to authenticate $\text{Sign}(sk_t, \text{message})$.

Linear KES Protocol Π_{L}:

The protocol is run by a registered set of parties $\mathcal{P}$ denoted by $U_i \in \mathcal{P}$ interacting with a signer U_S as follows:

Key Generation. Upon receiving the message $(\mathsf{KeyGen}, sid, U_S, T)$ from U_S if $T > 0$, U_S does the following, otherwise ignore the input: pick a random $s_m \in \{0,1\}^{\ell}$ then compute $(sk_m, vk_m) \leftarrow \mathsf{KeyGen}(s_m)$. For each $i \in \{j : 0 \leq j < T\}$ pick a random s_i and compute $(sk_i, vk_i) \leftarrow \mathsf{KeyGen}(s_i)$ then add (sk_i, vk_i) to set $\mathbb{K}$ and securely erase s_i. For each key pair $(sk_i, vk_i) \in \mathbb{K}$ compute $\sigma_i \leftarrow \mathsf{Sign}(sk_m, i||vk_i)$, then add the tuple $(i, \sigma_i, sk_i, vk_i)$ to the set $\mathbb{S}_m$. Record $\mathbb{S}_m$, securely erase sk_m, s_m and $\mathbb{K}$. Send the message $(\mathsf{VerificationKey}, sid, vk_m)$ to U_S.

Key Update. Upon receiving the message $(\mathsf{KeyUpdate}, sid, U_S, t)$ from U_S with a sid for which it has the signing key set $\mathbb{S}_m$, U_S does the following, otherwise ignore the input: $\forall (i, \sigma_i, sk_i, vk_i) \in \mathbb{S}_m$ such that $i \geq t$, add $(i, \sigma_i, sk_i, vk_i)$ to the set $\mathbb{S}'_m$. Record $\mathbb{S}'_m$ and securely erase $\mathbb{S}_m$. Send the message $(\mathsf{Updated}, sid)$ to U_S.

Signature Creation. Upon receiving the message $(\mathsf{Sign}, sid, U_S, m, t)$ from U_S with a sid for which it has the signing key set $\mathbb{S}_m$ and if $i \geq t$ $\forall (i, \sigma_i, sk_i, vk_i) \in \mathbb{S}_m$ and $\exists (i, \sigma_i, sk_i, vk_i) \in \mathbb{S}_m$ such that $i = t$, U_S does the following, otherwise ignore the input: find the entry $(t, \sigma_t, sk_t, vk_t) \in \mathbb{S}_m$ and compute $\Sigma_t \leftarrow (\sigma_t, vk_t, \mathsf{Sign}(sk_t, m))$, then send the message $(\mathsf{Signature}, sid, m, t, \Sigma_t)$ to U_S.

Signature Verification. When party U_i receives the message $(\mathsf{Verify}, sid, m, t, \Sigma_t, vk_m)$, U_i performs the following: Parse the signature as $(\sigma_t, vk_t, \sigma_m) \leftarrow \Sigma_t$ and compute $b \leftarrow \mathsf{Verify}(vk_m, \sigma_t, t||vk_t) \wedge \mathsf{Verify}(vk_t, \sigma_m, m)$. U_i outputs the message $(\mathsf{Verified}, sid, m, t, b)$.

Figure 6.2: The linear KES scheme as a protocol assuming the underlying signing routine in Figure 4.1.

7 Operational Composition

This section defines the operational composition, a combination of the linear scheme and the product scheme. The root of the scheme's public-private keypair construction is a registration credential R. Registering R with a blockchain protocol that has locally predictable leadership eligibility permits the staking party U_S to significantly reduce the number of rounds to sign in the linear scheme. Let the set $\mathbb{O}$ contain all rounds determined eligible for staking in the consensus protocol. Assume that some subset of all rounds will be eligible for block production. In practice, the majority of rounds will be ineligible and

$\mathbb{O}$ is a small subset of all possible rounds. The protocol uses a product key construction to commit to a set of linearly indexed keys. $\mathbb{O}$ contains only the rounds that a block can be created. This significantly reduces the memory footprint and computational overhead in generating the private key cache in the linear scheme because unused rounds aren't cached. Given N rounds per operational period, this reduces the average cache size by a factor of $\langle o \rangle / N$ where $\langle o \rangle = \langle |\mathbb{O}| \rangle_N$ is the expectation of eligible rounds in one operational period.

Operational Composition Protocol $\Pi_{\Sigma}^{\otimes} \otimes \Pi_{\mathrm{L}}$:
The protocol is run by a registered set of parties $\mathcal{P}$ denoted by $U_i \in \mathcal{P}$ interacting with a signer U_S. Assume that party U_S and parties $\{U_i\}$ have access to constants h_1, h_2 and N. Assume that party U_S provides a locally generated set of operational rounds $\mathbb{O} \subseteq \mathbb{N}_0$ wherein a message is to be signed in round i iff $i \in \mathbb{O}$. The protocol proceeds as follows:

Key Generation. Upon receiving the message $(\mathsf{KeyGen}, sid, U_S)$ from U_S, U_S does the following: pick a random s then compute $\kappa \leftarrow \mathsf{KeyGenProduct}(s, h_1, h_2)$ and $R \leftarrow \mathsf{VerificationKeyProduct}(\kappa)$. Securely erase s, record the signing key (INIT, κ), then send the message $(\mathsf{VerificationKey}, sid, R)$ to U_S.

Key Update. Upon receiving the message $(\mathsf{KeyUpdate}, sid, U_S, t, \mathbb{O})$ from U_S with a sid for which it has the signing key $(\kappa, \mathbb{S}_k)$ or (INIT, κ), U_S does the following, otherwise ignore the input: compute $k' \leftarrow t/N$. If $k' > \mathsf{KeyTimeProduct}(\kappa)$ or the signing key is (INIT, κ) then compute $\kappa' \leftarrow \mathsf{KeyUpdateProduct}(\kappa, k')$ and proceed to Case 1, otherwise proceed to Case 2.
Case 1. For each $i \in \{j : t \leq j < N(k'+1)\} \cap \mathbb{O}$, do the following: pick a random s_i, compute $(sk_i, vk_i) \leftarrow \mathsf{KeyGen}(s_i)$, compute $\Sigma_i \leftarrow \mathsf{SignProduct}(\kappa', i||vk_i)$, set $\mathbb{S}_{k'} \leftarrow (i, \Sigma_i, sk_i, vk_i) \cup \mathbb{S}_{k'}$ and finally securely erase s_i. Compute $\kappa'_E \leftarrow \mathsf{EraseProductLeafSK}(\kappa')$. Securely erase $\mathbb{S}_k$, κ and κ'. Record the signing key $(\kappa'_E, \mathbb{S}_{k'})$, then send the message $(\mathsf{Updated}, sid)$ to U_S.
Case 2. $\forall (i, \Sigma_i, sk_i, vk_i) \in \mathbb{S}_k$ such that $i \geq t$, set $\mathbb{S}'_k \leftarrow (i, \Sigma_i, sk_i, vk_i) \cup \mathbb{S}'_k$. Securely erase $\mathbb{S}_k$. Record $(\kappa, \mathbb{S}'_k)$, then send the message $(\mathsf{Updated}, sid)$ to U_S.

Signature Creation. Upon receiving the message $(\mathsf{Sign}, sid, U_S, m, t)$ from U_S with a sid for which it has the signing key $(\kappa, \mathbb{S}_k)$ and that $t/N = \mathsf{KeyTimeProduct}(\kappa)$ and $\exists (i, \Sigma_i, sk_i, vk_i) \in \mathbb{S}_k$ such that $i = t$, U_S does the following, otherwise ignore the input: find the entry $(i, \Sigma_i, sk_i, vk_i) \in \mathbb{S}_k$ for which $i = t$ and compute the signature $\Sigma \leftarrow (\mathsf{Sign}(sk_i, m), vk_i, \Sigma_i)$, then send the message $(\mathsf{Signature}, sid, m, t, \Sigma)$ to U_S.

Signature Verification. When party U_i receives the message $(\mathsf{Verify}, sid, m, t, \Sigma, R)$, U_i does the following: parse the signature Σ as $(\sigma, vk_t, \Sigma_t) \leftarrow \Sigma$ then compute $k \leftarrow t/N$, $b_k \leftarrow \mathsf{VerifyProductSignature}(R, \Sigma_t, k, t||vk_t)$ and $b_t \leftarrow \mathsf{Verify}(vk_t, \sigma, m)$. Output the message $(\mathsf{Verified}, sid, m, t, b_t \wedge b_k)$.

Figure 7.1: The operational protocol expressed as a composition between a product signature scheme and a linear signature scheme.

Algorithm 21 $\mathsf{EraseProductLeafSK} : \kappa \rightarrow \mathsf{ProductKey}$

Require: $\kappa \in \mathsf{ProductKey}$
1: $\mathsf{ProductKey}[\tau_1, \sigma_1, s, \tau_2] \leftarrow \kappa$
2: $\tau'_2 \leftarrow \mathsf{EraseLeafSK}(\tau_2)$
3: **return** $\mathsf{ProductKey}[\tau_1, \sigma_1, s, \tau'_2]$

8 Test Vectors

The following test vectors have been produced using an Ed25519 signing routine [5] and the Blake-2b-256 hash function [2]. The reference Python scripts for the sum composition and product composition are available at `https://github.com/Topl/reference_crypto/tree/main/specs/crypto/signing/KES-Ed25519-Blake2b256-SC` and `https://github.com/Topl/reference_crypto/tree/main/specs/crypto/signing/KES-Ed25519-Blake2b256-PC2` with test vectors provided in the readme files.

References

[1] R. Anderson. "Two Remarks on Public Key Cryptology (Invited Lecture)". In: *Proceedings of the ACM Conference on Computer and Communications*. Zurich, Switzerland: ACM Press, pp. 135–147.

[2] Jean-Philippe Aumasson et al. "BLAKE2: Simpler, Smaller, Fast as MD5". In: *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*. ACNS'13. Banff, AB, Canada: Springer-Verlag, 2013, pp. 119–135. ISBN: 9783642389795. DOI: 10.1007/978-3-642-38980-1_8. URL: https://doi.org/10.1007/978-3-642-38980-1_8.

[3] Christian Badertscher et al. "Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability". In: Oct. 2018, pp. 913–930. DOI: 10.1145/3243734.3243848.

[4] Mihir Bellare and Sara K. Miner. "A Forward-Secure Digital Signature Scheme". In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 431–448. ISBN: 3540663479.

[5] Daniel J. Bernstein et al. "High-Speed High-Security Signatures". In: *CHES*. Vol. 6917. Lecture Notes in Computer Science. Springer, 2011, pp. 124–142. DOI: 10.1007/978-3-642-23951-9_9. URL: https://www.iacr.org/archive/ches2011/69170125/69170125.pdf.

[6] Ran Canetti. *Universally Composable Signatures, Certification and Authentication*. An extended abstract of this work appears in the proceedings of CSFW 2004. The current version contains some corrections and updates of the CSFW 2004 paper. canetti@watson.ibm.com 12645 received 17 Nov 2003, last revised 15 Aug 2004. 2003. URL: http://eprint.iacr.org/2003/239.

[7] Bernardo David et al. "Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain". In: *Advances in Cryptology – EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Cham: Springer International Publishing, 2018, pp. 66–98. ISBN: 978-3-319-78375-8.

[8] Tal Malkin, Daniele Micciancio, and Sara Miner. "Efficient Generic Forward-Secure Signatures with an Unbounded Number of Time Periods". In: *Advances in Cryptology — EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 400–417. ISBN: 978-3-540-46035-0.