Short Transitive Signatures for Directed Trees

Philippe Camacho and Alejandro Hevia
Dept. of Computer Science, University of Chile,
Blanco Encalada 2120, 3er piso, Santiago, Chile.
{pcamacho,ahevia}@dcc.uchile.cl

August 17, 2011

Abstract

A transitive signature scheme allows to sign a graph in such a way that, given the signatures of edges (a, b) and (b, c), it is possible to compute the signature for the edge (or path) (a, c) without the signer's secret. Constructions for undirected graphs are known but the case of directed graphs remains open. A first solution for the case of directed trees ($DTTS$) was given by Yi at CT-RSA 2007. In Yi's construction, the signature for an edge is $O(n(\log(n \log n)))$ bits long in the worst case. A year later, Neven designed a simpler scheme where the signature size is reduced to $O(n \log n)$ bits. Although Neven's construction is more efficient, $O(n \log n)$-bit long signatures still remains impractical for large n.

In this work, we propose a new $DTTS$ scheme such that, for any value $\lambda \geq 1$ and security parameter κ:

- Signatures for edges are only $O(\kappa\lambda)$ bits long.
- Signing or verifying a signature for an edge requires $O(\lambda)$ cryptographic operations.
- Computing a signature for an edge requires $O(\lambda n^{1/\lambda})$ cryptographic operations.

To the best of our knowledge this is the first construction with such a trade off. In particular, we can achieve $O(\kappa \log(n))$-bit long signatures while taking only $O(\log(n))$ time to generate edge signatures, verify or even compute edge signatures.

Our construction relies on *hashing with common-prefix proofs*, a new variant of collision resistance hashing. A family $\mathcal{H}$ provides hashing with common-prefix proofs if for any $H \in \mathcal{H}$, given two strings X and Y equal up to position i, a *Combiner* can convince a *Verifier* that $X[1..i]$ is a prefix of Y by sending only $H(X), H(Y)$, and a small proof. We believe that this new primitive will lead to other interesting applications.

Keywords: Transitive Signatures, Authenticated Data-structures, Collision-resistant Hashing, Hashing with common-prefix proofs.

1 Introduction

Transitive signatures is a primitive introduced by Micali and Rivest [14] where a signer wants to authenticate a graph. The main property of such scheme is that, given the signatures of edges (a, b) and (b, c), it is possible to compute - *without the knowledge of the secret* - a signature for the edge (a, c). In their work, the authors propose an efficient scheme for undirected graphs based on the difficulty of computing discrete logarithm for large groups. They left the existence of a transitive signature scheme for directed graph (DTS) as a challenging open question. The easier problem of building transitive signatures for directed trees was first addressed by Yi [19]. The construction, based on a special assumption for the

RSA cryptosystem, produces signatures of size $O(n \log(n \log n))$ bits, where n is the number of vertices of the tree. Neven described in [16], a simpler solution based only on the existence of standard digital signatures which also improves the bound on the size of the signature to $O(n \log n)$ bits.

In this work we describe a new construction for $DTTS$ schemes that enjoys much better worst-case complexity. Using the number of cryptographic operations as our notion of time, we obtain for any $\lambda \geq 1$:

- Signing a new edge (or verifying the signature for any edge) can be done in $O(\lambda)$ time.
- The time to compute a signature for any edge is $O(\lambda n^{1/\lambda})$.

Moreover, signature size is also substantially improved: our signatures require only $O(\kappa\lambda)$ bits, where κ is a security parameter. In particular, if $\lambda = \log(n)$ then signatures are only $O(\kappa \log(n))$ bits, while allowing efficient signature computation ($O(\log(n)$ time). Alternatively, by setting for example $\lambda = 2$, we obtain an optimal edge signature size of $O(2 \cdot \kappa) = O(\kappa)$ bits if we are willing to afford $O(\sqrt{n})$ computation time.

Our Approach. There are two main ideas in our construction. First we use the following fact observed by Dietz in [8]: given a tree $\mathcal{T}$, if `Pre` and `Post` are the strings representing the sequences of labels obtained by a pre-order and respectively post-order depth first traversal, then there exists a path from a to b if and only if a appears before b in `Pre` and b appears before a in `Post`. Armed by this result we can reduce the problem of deciding if there is a path between vertices a and b to the one of comparing the position of a and b in $\mathcal{S}$. To do so we consider an *order data structure* – a concept also introduced in [8] – where the idea is to dynamically insert elements into a sequence such that it is efficient to decide whether an element is before or after another. We implement such data structure through a binary search tree $\mathcal{T}'$, where each node of $\mathcal{T}'$ is associated to an element of the sequence $\mathcal{S}$ in the following way: if $a \in \mathcal{S}$ (bound to $a' \in \mathcal{T}'$) appears before $b \in \mathcal{S}$ (bound to $b' \in \mathcal{T}'$), then a' and b' have a common ancestor c and a' belongs to the left (resp. b' belongs to the right) sub-tree of c. We then label each left (resp. right) edge of $\mathcal{T}'$ by 0, (resp. 1). Now we assign to a the string A formed by the concatenation of $0, 1$'s from the root of $\mathcal{T}'$ to the node a' and similarly assign the string B to node b'. From this construction we can define a total order relation on strings $\prec$ such that $A \prec B$, means a appears before b in the sequence $\mathcal{S}$. The advantage of this order data structure is that it allows incremental computations of new order labels: that is, every new string V (associated to an element v of $\mathcal{S}$) will share all bits except one with another already computed label. As shown in section 4, this property is crucial to enable efficient computation of edge signatures. The problem that arises now is that large strings of $O(n)$ bits are bound to the vertices of $\mathcal{T}$, so at first sight we have not won too much: the signature length is now $O(n)$ bits v/s $O(n \log n)$ bits for Neven's construction. That is where our second idea comes into play: we design a new kind of collision resistant hash functions family which enables the following: Given only two hash values $H(A)$, $H(B)$ and a small proof a *Combiner* can convince a *Verifier* that A and B share a common prefix up to a position i. We call this new primitive *collision-resistant hashing with common-prefix proofs*. We can see that this primitive allows also to prove that $A \prec B$ for two strings A, B. Using this tool we can complete our construction for $DTTS$. The last remaining difficulty is that the time to compute a proof for strings of n bits involves a priori $O(n)$ cryptographic operations. To overcome this drawback we show how to balance the work between the *Verifier* and the *Combiner* using the natural idea of hashing consecutive chunks of the initial string to obtain a shorter one, and repeat this operation several times. This technique leads to the trade off $O(\lambda n^{1/\lambda})$ v/s $O(\lambda)$ for $\lambda \geq 1$ between the time to compute a proof versus the time to verify a proof. The security of our primitive is based on the *n-Bilinear Diffie Hellman Inversion* assumption, introduced by Boneh and Boyen in [3].

Related Work. The concept of transitive signatures was introduced by Rivest and Micali [14] who also gave constructions for undirected graphs. Bellare and Neven in [2], as well as Shahandashti et al. in [18], introduced new schemes based on bilinear maps (but still for undirected graphs). Hohenberger [11] showed that the existence of transitive signatures for directed graphs (DTS) implies the existence of abelian groups where inversion is computationally infeasible except when given a trapdoor. Such groups are not

known to exist either. Transitive signatures are a special case of homomorphic signatures, a primitive introduced by Rivest and explored in [12, 4, 5]. We observe that using accumulators techniques like [6, 7] we can improve Neven's construction [16] in order to obtain short signatures. Such a solution, however, does not enable two key properties we seek and achieve on this paper's construction: the computation of edge signatures is paralellizable, and it tolerates unbounded growth for the trees (the construction can increase the bound on the number of nodes by dynamically increasing the setup parameter, see sec. 3). We explore a $DTTS$ construction based on accumulators in appendix C.

Our contributions. Our contribution is twofold: first we introduce a general and practical new primitive, *collision-resistant hashing with common-prefix proofs*, which enables efficient proofs that certain strings share common-prefixes. We believe that this primitive may lead to many applications in the field of authenticated data structures. Our second contribution is a practical $DTTS$ scheme which is, to the best of our knowledge, the most efficient one to the date.

Organization of the paper. In section 2 we introduce the notations, the definitions for $DTTS$ and the complexity assumptions that we use. Section 3 describes in details our new primitive. Then in section 4 we show how to use *collision-resistant hashing with common-prefix proofs* to obtain a practical $DTTS$ scheme.

2 Preliminaries

Notations and Conventions. If $\kappa \in \mathbb{N}$ is the security parameter then 1^κ denotes the unary string with κ ones. A function $\nu : \mathbb{N} \to [0,1]$ is said to be negligible in κ, if for every polynomial $p(\cdot)$ there exists κ_0 such that $\forall \kappa > \kappa_0 : \nu(\kappa) < 1/p(\kappa)$. In the following, neg will denote *some* negligible function in κ. An algorithm is said to be PPT if it is probabilistic and runs in polynomial time in κ. We write $x \xleftarrow{R} X$ to denote an element x chosen at random from the set X. The time complexities expressed in the rest of this work are relative to the number of cryptographic operations (signature, group exponentiation, and application of a bilinear map).

Strings. Let $n \in \mathbb{N}$. A string S of size $|S| = n$ is a sequence of symbols $S[1], S[2], ..., S[n]$ from an alphabet Σ. We assume Σ is totally ordered, and note the order relation $<$. If $n = 0$ then $S = \epsilon$ is the *empty* string. $S[i..j]$ denotes the substring of S starting at position i and ending at position j (both $S[i]$ and $S[j]$ are included). In particular if $A = S[1..j]$ for some $j \geq 0$ then we say that A is a prefix of S (by convention $A[1..0]$ for any string A is the empty string ϵ). We say a string C is a common prefix of A and B if C is prefix of A and also of B. String C is said to be the *maximum common prefix* of A and B if moreover $C||\sigma$ is not a common prefix of A and B for any symbol $\sigma \in \Sigma$. The concatenation operator on strings is denoted as $||$. That is, if A, B are two strings of size n, then $C = A||B$ is the string formed by the sequence $C[1] = A[1], C[2] = A[2], ..., C[n] = A[n], C[n+1] = B[1], ..., C[2n] = B[n]$. A symbol $\sigma \in \Sigma$ refers equivalently to the symbol or the string of length one. If A and B are strings then $A \prec B$ means A appears before B w.r.t. the lexicographical order. \$ is a special (and implicit) symbol that is used only to mark the end of a string. For example the empty string is written \$ and the string with symbols a followed by b is represented by $ab\$$.

Trees. Let $\mathcal{T}$ be a directed tree where each node is identified by a unique *vertex-label* $a \in \mathbb{N}$. Our construction makes use of trees which edges are associated to a symbol $\sigma \in \Sigma$. This means a node can also be identified by a string (or *path-label*) A which is the concatenation of the symbols present on the path from the root to this node. Moreover, if α is a vertex label, we write $\alpha \in \mathcal{T}$ to mean the node with label α belongs to $\mathcal{T}$. If we assume that each path-label A is unique then $node(A)$ refers to the node in $\mathcal{T}$ with path-label A. We say a node $a \in \mathcal{T}$ is a descendant of c if a belongs to the sub-tree rooted at c or equivalently if there is a path from c to a. The lowest common ancestor of two nodes a, b of $\mathcal{T}$ is the node

c such that a and b belong to the sub-tree rooted at c, and for any child d of c, a or b is not a descendant of d. If we consider a depth-first traversal of a tree, We denote by $\texttt{Pre}$ and $\texttt{Post}$ the strings formed by the successive labels of the nodes that are visited in a pre-order (the node is append to the string when it is visited for the first time) respectively post-order (the node is append to the string when it is visited for the last time). The transitive closure of $\mathcal{T}$ is $\mathcal{T}^* = \{(a,b) : a,b \in \mathcal{T}$ and there is a path from a to $b\}$.

Collision Resistance and Standard Signatures Schemes. Let $\mathcal{H}$ be a family of functions and $H\colon \{0,1\}^* \to \{0,1\}^\kappa$ an element of $\mathcal{H}$. The family $\mathcal{H}$ is said to be collision-resistant ($CRHF$) if, for H randomly chosen in $\mathcal{H}$, any computationally bounded adversary can not find two different messages M and M' such that $H(M) = H(M')$, except with negligible probability. Let $Alg_H(\cdot)$ be a PPT algorithm that computes H, then if $Alg_H(\cdot)$ is fed with input X and returns y, we write $X = H^{-1}(y)$. We denote by $\texttt{SSig} = (\mathsf{SKG}, \mathsf{SSig}, \mathsf{SVf})$ a standard signature scheme. $(sk, pk) \leftarrow \mathsf{SKG}(1^\kappa)$ is the pair of private/public keys created in the setup phase. Then for a message $M \in \{0,1\}^*$ its associated signature is $\sigma_M = \mathsf{SSig}(sk, M)$. The validation of a signature σ on M is done by running $\mathsf{SVf}(pk, M, \sigma)$ which returns valid if σ is a valid signature for M under pk and $\perp$ otherwise. For the security of digital signatures, we use the standard notion of existential unforgeability under chosen message attack [9].

Transitive Signatures. In a *transitive* signature scheme, the *Signer* can sign the vertices of some graph but also the edges. Then without the secret, given two signed edges (a,b) and (b,c) it is possible to compute the signature of the path (or edge) (a,c). We can see that this property enables to compute the signature for any path in the tree.

Definition 1. *(Transitive Signature Scheme, [14, 16]) A transitive signature scheme (for directed trees) is a tuple* $\texttt{DTTS} = (\mathsf{TSKG}, \mathsf{TSign}, \mathsf{TSComp}, \mathsf{TSVf})$ *where:*

- $\mathsf{TSKG}(1^\kappa)$: *returns a pair of private/public keys* (tsk, tpk).
- $\mathsf{TSign}(tsk, a, b)$: *returns the signature* $\tau_{(a,b)}$ *of edge* (a,b).
- $\mathsf{TSComp}((a,b), \tau_{(a,b)}, (b,c), \tau_{(b,c)}, tpk)$*: returns the signature* $\tau_{(a,c)}$ *of path* (a,c)*. Note that the secret key is not required.*
- $\mathsf{TSVf}((a,b), \tau, tpk)$: *returns* valid *if the* τ *is a valid signature for the path* (a,b) *and* $\perp$ *otherwise.*

Intuitively, a transitive signature scheme is secure if, for any PPT adversary it is infeasible to compute a signature for a path that is outside the transitive closure of $\mathcal{T}$.

Definition 2. *(Security of Transitive Signature Schemes, [14, 16]) Let* $\texttt{DTTS}$ *be a transitive signature scheme. Consider the following experiment. The PPT adversary* $\mathcal{A}$ *is given the public key of the scheme* tpk*.* $\mathcal{A}$ *asks for a polynomial number of edge signatures to the oracle* $\mathcal{O}_{\mathsf{TSign}}(\cdot)$*. Finally* $\mathcal{A}$ *outputs* (a,b) *and* τ *where* a, b *are nodes of the tree* $\mathcal{T}$ *formed by the successive edge insertions. The advantage of* $\mathcal{A}$ *is defined by:*

$$Adv^{tuf\text{-}cma}(\mathcal{A}, \kappa) = Pr\left[\begin{array}{c}(a,b) \notin \mathcal{T}^* \wedge \\ \mathsf{TSVf}((a,b), \tau, tpk) = \mathsf{valid}\end{array}\right]$$

The scheme is said to be secure if for any PPT adversary $\mathcal{A}$ *we have* $Adv^{tuf\text{-}cma}(\mathcal{A}, \kappa) = \mathsf{neg}(\kappa)$.

A trivial solution for $DTTS$ can be implemented by simply concatenating standard edge signatures, in which case the size of a signature grows linearly with the size of the path and may reach $O(n\kappa)$ bits. Yi's solution [19] with $O(n \log(n \log n))$ bits signatures is clearly better than the trivial construction. Neven's $DTTS$ scheme [16] reduces the size of signature to $O(n \log n)$ bits. We note that both solutions need to maintain the state of the tree to enable new edge signature computations. As mentioned in [16]

the initial definition of transitive signatures is stateless in the sense that signing a new edge should only require the knowledge of the two vertices. Our solution is also stateful and thus we make it explicit by introducing a third participant in addition to the *Signer* and the *Verifier* that we call the *Combiner*. His role is to compute, *without the help of the* Signer, signatures for any path in the tree.

BILINEAR MAPS. Our construction for *collision resistant hashing with common-prefix proofs* requires the use of bilinear maps. Let $\mathbb{G}, \mathbb{G}_T$, be cyclic groups of prime order p. We consider a map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ which is

- *bilinear*: $\forall a, b \in \mathbb{G}, x, y \in \mathbb{Z}_p : e(a^x, b^y) = e(a, b)^{xy}$.
- *non-degenerate*: let g be a generator of $\mathbb{G}$ then $e(g, g)$ also generates $\mathbb{G}_T$.
- *efficiently computable*: there exists a polynomial time algorithm BMGen with parameter 1^κ that outputs $(p, \hat{\mathbb{G}}, \hat{\mathbb{G}_T}, \hat{e}, g)$ where $\hat{\mathbb{G}}, \hat{\mathbb{G}_T}$ refer to the representation of both groups of size p (p being a prime number of κ bits), g is a generator of $\mathbb{G}$ and $\hat{e}$ is an efficient algorithm to compute the map. For the sake of simplicity, in the following we will not distinguish between $\mathbb{G}, \mathbb{G}_T, e$ and $\hat{\mathbb{G}}, \hat{\mathbb{G}_T}, \hat{e}$.

The security of our construction relies on the n-Billinear Diffie-Hellman Inversion (n-$BDHI$) assumption which was introduced by Boneh and Boyen [3].

Definition 3. *($n - BDHI$ assumption [3]) Let $P = (p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BMGen}(1^\kappa)$, $s \xleftarrow{R} \mathbb{Z}_p$, and $T = (g, g^s, g^{s^2}, ..., g^{s^n})$. The n-Bilinear Diffie-Hellman Inversion (n-$BDHI$) problem consists in computing $e(g, g)^{\frac{1}{s}}$, given P and T. We say the $n - BDHI$ assumption holds if for any PPT adversary $\mathcal{A}$ we have:*

$$Adv^{n\text{-}BDHI}(\mathcal{A}, \kappa, n) = \Pr\left[e(g, g)^{\frac{1}{s}} \leftarrow \mathcal{A}(1^\kappa, P, T) \right] = \mathsf{neg}(\kappa)$$

3 Collision-Resistant Hashing with Common-Prefix Proofs

Standard collision-resistant hash functions have the property of compressing possibly large inputs strings to small ones. In practice, hash functions are considered injective due to the collision-resistance property. This makes them useful constructs to manipulate shorter strings without loosing much security. In that context, proving some relations or predicates on pre-images using only the corresponding hash values (and perhaps an additional short proof) is certainly very useful. For example, given two hash values $H(A), H(B)$, proving efficiently predicates like $|A - B| \geq 10$ or $A \prec B$ may help to simplify some protocols or make them more efficient.

With the above goal in mind, in this paper we consider the *common prefix* predicate for strings, `CommonPrefix`: given $A, B \in \Sigma^n$, $\texttt{CommonPrefix}(A, B, i) = true$ if and only if A and B share a common prefix up to position i. We seek collision-resistant hash function families $\mathcal{H}$ with the following property: given $H(A)$ and $H(B)$ where A and B share a common prefix until position i, it should be possible to produce a certificate π such that running some verification algorithm on inputs $i, \pi, H(A), H(B)$ one can be convinced that $\texttt{CommonPrefix}(A, B, i) = true$. Any such scheme should be secure in the sense that if $\texttt{CommonPrefix}(A, B, i) = false$ then producing a forged π^* that makes the verification algorithm accept should be computationally infeasible. Clearly, there exists a trivial instantiation of this primitive: just consider H a standard (collision-resistant) hash function and $\pi = (A, B)$. Of course, this is not really useful as the size of the certificate is proportional to the size of the longest string. Thus, interesting implementations should have short certificates. Additionally, we want hash function H to be easily updatable: given $H(A)$ one should be able to compute $H(A||\sigma)$, without knowing A (this concept is also known as *incremental hashing* [1]).

Given a `CommonPrefix` predicate we can now implement more interesting predicates over strings such as `Compare`, where $\texttt{Compare}(A, B) = true$ if $A \prec B$: If $A \prec B$ it follows that there exists a (possibly

empty) common prefix C for A and B, such that (1) $C||\sigma$ is a prefix of A, (2) $C||\sigma'$ is a prefix of B, and (3) $\sigma < \sigma'$. In summary, once we know how to do proofs for `Prefix`, we can compare any two strings only using their hash values and a short proof.

Collision-Resistant Hashing with common-prefix proofs ($CRHwCPP$). Let $\kappa \in \{0,1\}^*$ be the security parameter, $PK \in \{0,1\}^\kappa$ some public key, and $n \in \mathbb{N}$ a bound on the size of the input which is a polynomial in κ. We denote by $\mathcal{H} = \{\mathcal{H}_{PK,n,\kappa}\}$ a hash function family.

Definition 4. *(Collision-Resistant Hashing with common-prefix proofs - Syntax) A function family $\mathcal{H}$ of collision-resistant hashing with common-prefix proofs ($CRHwCPP$) is defined by the tuple of algorithms* $(\mathsf{PHGen}, \mathsf{PHEval}, \mathsf{PHProofGen}, \mathsf{PHCheck})$ *where:*

- $\mathsf{PHGen}(1^\kappa, n)$*: given a bound n on the length of the strings to hash, this probabilistic algorithm returns a public parameter PK. Value PK implicitly defines a hash function $H = H_{PK,n,\kappa} \in \mathcal{H}$ where $H : \{0,1\}^n \to \{0,1\}^\kappa$.*
- $\mathsf{PHEval}(M, PK)$*: given $M \in \{0,1\}^n$, this* deterministic *algorithm returns a string $H(M) \in \{0,1\}^\kappa$.*
- $\mathsf{PHProofGen}(A, B, i, PK)$*: given two messages A, $B \in \{0,1\}^n$, and an index $1 \leq i \leq n$, this deterministic algorithm computes a proof $\pi \in \{0,1\}^\kappa$ that will be used by the* $\mathsf{PHCheck}$ *algorithm.*
- $\mathsf{PHCheck}(H_A, H_B, \pi, i, PK)$*: a deterministic algorithm that, given $H_A, H_B \in \{0,1\}^\kappa$, two hash values, and a proof $\pi \in \{0,1\}^\kappa$, returns either* valid *or $\perp$.*

The scheme is said to be correct if for any strings A, B and $i \in \mathbb{N}$ such that $\texttt{CommonPrefix}(A, B, i) = \mathit{true}$, and $\pi = \mathsf{PHProofGen}(A, B, i, PK)$, we have that $\mathsf{PHCheck}$ on inputs $(H(A), H(B), \pi, i, PK)$ returns valid.

The notion of security is also rather natural: for any PPT adversary $\mathcal{A}$ it should be difficult to compute two n-bit strings A, B, an index $i \in \{1, \ldots, n\}$, and a proof $\pi \in \{0,1\}^\kappa$ such that $\mathsf{PHCheck}(H(A), H(B), \pi, i, PK)$ returns valid but $A[1..i] \neq B[1..i]$. Note that the adversary is required to output pre-images A and B to win, which assures that the hash values $H(A)$ and $H(B)$ have been correctly computed.

Definition 5. *(Collision-Resistant Hashing with Common-Prefix Proofs - Security) Let $\mathcal{H}$ be a family of collision-resistant hash functions with common-prefix proofs and $\mathcal{A}$ a PPT adversary. The $CRHwCPP$ advantage of $\mathcal{A}$ is*

$$Adv_{\mathcal{H}}^{CRHwCPP}(\mathcal{A}, \kappa, n) = Pr\left[\begin{array}{c} PK \leftarrow \mathsf{PHGen}(1^\kappa, n); A, B, \pi, i \leftarrow \mathcal{A}(1^\kappa, n, PK) : \\ A[1..i] \neq B[1..i] \wedge H_A = H(A) \wedge H_B = H(B) \wedge \\ \mathsf{PHCheck}(H_A, H_B, \pi, i, PK) = \mathsf{valid} \end{array}\right]$$

We say $\mathcal{H}$ is a secure prefix collision resistant hash function family if for every PPT $\mathcal{A}$, we have $Adv_{\mathcal{H}}^{CRHwCPP}(\mathcal{A}, \kappa, n) = \mathsf{neg}(\kappa)$.

The following proposition shows that *collision resistance hashing with common-prefix proofs* imply (standard) *collision resistance*. We omit the proof.

Proposition 1. *Let $\mathcal{H}$ be a family of collision-resistant hashing with common-prefix proofs. Then $\mathcal{H}$ is a collision-resistant hash function family (in the standard sense).*

The Construction. We assume that the description of the hash function H - which is the tuple $(g^s, g^{s^2}, ..., g^{s^n})$ of the $n-BDHI$ problem - has been computed securely by a trusted third party or using multi-party computations techniques. The idea is to represent a binary string M by $H(M) \stackrel{\text{def}}{=} g^{M[1]s} \cdot g^{M[2]s^2} \cdots g^{M[n]s^n}$. Now if some message M' is equal to M up to position i then the value $\Delta = \frac{H(M)}{H(M')}$ will be

a product of g^{s^j} factors for $1 \leq j \leq n$ where for all $j \leq i$ the exponents are equal to 0. Hence, $\Delta = \prod_{j=i+1}^{n} g^{c_j s^j}$ where $c_j \in \{-1, 0, 1\}$. So with the knowledge of M,M' and H we can compute a proof $\pi = \prod_{j=i+1}^{n} g^{c_j s^{j-(i+1)}}$. The intuition behind this is that as M and M' are equal up to position i then we can represent the difference of M and M' using only $n-i$ positions. Verifying that proof π is valid then consists in using the bilinear map to "shift forward" the exponents in the proof by i positions, in order to get back the value $H(M)/H(M')$. More precisely, π will be a valid proof for $H(M), H(M')$ if and only if $e(\frac{H(M)}{H(M')}, g) = e(\pi, g^{s^{i+1}})$. Details follow.

Construction 1. *(Collision-Resistant Hashing with Common-Prefix Proofs - Construction) Let* PH *be the scheme defined by the following algorithms:*

- $\mathsf{PHGen}(1^\kappa, n)$*: run* $\mathsf{BMGen}(1^\kappa)$ *to obtain* $P = (p, \mathbb{G}, \mathbb{G}_T, e, g)$*. Let* $s \xleftarrow{R} \mathbb{Z}_p$*, and* $T = (g, g^s, g^{s^2}, ..., g^{s^n})$*. Return* $PK = (P, T)$*.*
- $\mathsf{PHEval}(M, PK)$*:* $M \in \{0,1\}^n$*. Compute* $H(M) = \prod_{j=1}^{n} g^{b_j s^j}$ *where* $b_j = M[j]$ *for* $j \in \{1, \ldots, n\}$*. Return* $H(M)$*.*
- $\mathsf{PHProofGen}(A, B, i, PK)$*: given n-bits strings* A, B*, let* C *be the array such that* $\forall j \in \{1, \ldots, n\}$: $C[j] = A[j] - B[j]$*. Return* $\pi = \prod_{j=i+1}^{n} g^{C[j] s^{j-(i+1)}}$*.*
- $\mathsf{PHCheck}(H_A, H_B, \pi, i, PK)$*: compute* $\Delta = \frac{H_A}{H_B}$*, then return* valid *if* $e(\Delta, g) = e(\pi, g^{s^{i+1}})$*, otherwise return* $\perp$*.*

Proposition 2. *Under the n-BDHI assumption the hash functions family defined by the scheme* PH *is a secure CRHwCPP.*

Proof. Given an adversary $\mathcal{A}$ that breaks the security of PH, we construct an adversary $\mathcal{B}$ that breaks the $n-BDHI$ assumption as follows. Once $\mathcal{B}$ receives as input the parameters (P, T) where $T = (g, g^s, g^{s^2}, \ldots, g^{s^n})$, it forwards them to $\mathcal{A}$. Eventually, $\mathcal{A}$ will output values A, B, π, i such that $\mathsf{PHCheck}(H(A), H(B), \pi,\ i,\ PK) = \mathsf{valid}$, that is,

$$e(\Delta, g) \quad = \quad e(\pi, g^{s^{i+1}}) \tag{1}$$

Then, $\mathcal{B}$ computes the array C defined as $C[j] = A[j] - B[j] = c_j$ for $j \in \{1, \ldots, n\}$. Since $A[1..i] \neq B[1..i]$, let $k < i$, be the smallest index such that $c_k \neq 0$. Clearly $i - k > 0$. From (1) it follows that $\pi = \Delta^{\frac{1}{s^{i+1}}}$, and then:

$$\begin{aligned} E = e(\pi, g^{s^{i-k}}) &= e(\Delta^{\frac{1}{s^{i+1}}}, g^{s^{i-k}}) \\ &= \prod_{j=k}^{n} e(g,g)^{c_j s^{j-k-1}} \\ &= e(g,g)^{\frac{c_k}{s}} \prod_{j=k+1}^{n} e(g,g)^{c_j s^{j-k-1}} \\ &= e(g,g)^{\frac{c_k}{s}} D \end{aligned}$$

As all the c_j are known, and $c_k = \pm 1$, $\mathcal{B}$ can compute $(\frac{E}{D})^{1/c_k} = e(g,g)^{\frac{1}{s}}$. □

Additional properties. Our construction for $CRHwCPP$ functions family $\mathcal{H}$ is homomorphic in the following sense: for any $H \in \mathcal{H}$, any bit b, $H(M||b) = H(M) \cdot H(0^{|M|}||b)$. Moreover, since $H(0^{|M|}||b) = g^{bs^{|M|+1}}$ can be computed in constant time w.r.t $|M|$, our construction yields in fact to an incremental hash function [1]. Furthermore, its computation can be easily parallelizable as obtaining a proof only involves group *multiplications*. In particular, with $O(n)$ processors, we can compute a proof using only $O(\log n)$ (sequential) group multiplications. Finally note that, handling strings of length $m > n$ can be done dynamically, *without having to recompute any proof*, by simply extending the public parameter $T = (g, g^s, g^{s^2}, \ldots, g^{s^n})$ say by invoking the distributed procedure (or calling the trusted generator) to compute $g^{s^{n+1}}, \ldots, g^{s^m}$. Finally, let Σ be a non binary alphabet with an efficient mapping to $\mathbb{Z}_p$. We observe that we can adapt our construction to handle such alphabets by defining H as $H(S) = \prod_{i=1}^{|S|} g^{S[i] s^i}$.

4 Short Transitive Signature for Directed Trees

Our construction for DTTS is based on the following idea: handling a growing tree can be reduced to maintaining two ordered sequences, one corresponding to the pre-order traversal and another to the post-order traversal in a depth first search. This was first observed by Dietz [8].

Proposition 3. *([8]) Let $\mathcal{T}$ be a tree and consider a depth-first traversal. Let* `Pre` *and* `Post` *be the strings formed by the nodes that are visited in pre-order and post-order respectively, then for any pair of nodes a, b in $\mathcal{T}$, b is descendant of a if and only if $\exists i, j : 0 < i < j$ and $\exists i', j' : 0 < j' < i'$ such that:*

$$(\texttt{Pre}[i] = a \wedge \texttt{Pre}[j] = b) \wedge (\texttt{Post}[i'] = a \wedge \texttt{Post}[j'] = b)$$

For example if we look at the last tree in the first column of figure 1, we get `Pre` $= acdbe$ and `Post` $= dcbea$. We can check that as there is a path from c to d, c appears before d in `Pre` and D appears before c in `Post`. Also note that if there is no path between two nodes x and y (recall that the tree is directed from top to bottom) then y may appear before x in `Pre` or x may appear before y in `Post`. See for example pairs $(c, b), (e, d)$ or (b, a).

The challenge to use this result is that the ordered sequences are dynamic (new elements can be inserted between any two existent elements). Such problem is addressed by the so called *order data structure* [8, 15]. In such a data structure we want to compare any pair of elements and also compute a new element such that it lies between two existing ones. A naive way - as mentioned in [8] - to implement the proposed data structure, would be to consider the interval $[0..2^n - 1]$ for the indexes; to insert an element between X and Y one would use index $\lfloor \frac{X+Y}{2} \rfloor$. This way we can always find an element between two others and the comparison algorithm consists in comparing the numbers. Unfortunately, in this case the *Signer* would have to handle indexes of length n for each new edge to sign, because the string representation of the new index cannot be easily obtained from already computed values. So our first improvement is a way to design the order data structure such that, if we want to insert an element between X and Y, the new resulting string for index Z (where $X < Z < Y$) will share all bits except one with the string representing X or or the one representing Y.

Before describing our construction we introduce the formal definition of an order data structure [15].

Definition 6. *Let $\mathcal{U}$ be a totally ordered set. An order data structure consists of three algorithms:*

- ODSetup() : *initializes the data structure.*
- ODInsert(X, Y) : *compute an element Z that will lie between the two consecutive elements $X, Y \in \mathcal{U}$.*
- ODCompare(X, Y)*: returns true if X precedes Y in the total order.*

Our data structure uses a binary search tree [13] to insert elements. Then comparing two elements in the data structure reduces to finding their lowest common ancestor c and checking whether one of them is descendant of c's left child or whether one element is an ancestor of the other. The values (strings) from the total order correspond to the path from the root to a node in this tree, where each edge is labelled by 0 (left child) or 1 (right child).

Construction 2. *. Let* `OrderDS` *be the data structure defined by the following operations.*

- ODSetup()*: create a tree with two nodes, a root $-\infty$ and its right child ∞. The label of the root is ϵ and the label of the right child is 1. Intuitively $-\infty$ represents the lowest element of the universe and ∞ the greatest.*
- ODInsert(X, Y) : *let X, Y be to consecutive elements, i.e. in particular $X \prec Y$. Search node(X) and node(Y) in the tree. If node(Y) belongs to the right sub-tree of node(X) then add node(Z) as the left child of node(Y). If node(X) belongs to the left sub-tree of node(Y) then add node(Z) as the right child of node(X). Return $Z||\$$, the label of node(Z) concatenated with the end marker $\$$.*

- $\mathsf{ODCompare}(X, Y)$: *If X or Y does not end with \$ return $false$. If $X = Y$ return $false$. Obtain the index i such that C is the maximum common prefix of X and Y. If $X[i+1] < Y[i+1]$, return true else return $false$.*

We observe that in the worst case the largest path of the tree may be of size $n+1$, and thus the largest label will contain $n+1 = O(n)$ bits. Note that now the elements X, Y, Z are strings and not integers as in the naive order data-structure. This order data-structure has an important property: for a pair of consecutive elements (strings) X, Y the string Z returned by $\mathsf{ODInsert}(X, Y)$ is equal to $X||b$ or $Y||b$ where $b \in \{0, 1\}$. This turns out to be crucial as these strings will be hashed using a hash function with common-prefix proofs $\mathcal{H}$, introduced in the previous section. As a consequence of the homomorphic property of $\mathcal{H}$ it will require only a constant number of cryptographic operations in order to compute $H(Z)$ from $H(X)$ or $H(Y)$. We remark that as we append the symbol \$ at the end of all strings (i.e. we consider the alphabet $\Sigma = \{0, 1, \$\}$ where $0 < \$ < 1$ and \$ is the end of string marker), $\mathsf{ODCompare}(X, Y)$ consists simply in comparing the strings X,Y w.r.t. the lexicographic order.

Basic Construction. Our first construction is based only on standard digital signatures – as Neven's construction – but where the size of each path signature is $O(n)$ bits instead of $O(n \log n)$ bits. The construction works as follows. Each time an edge (and thus a vertex) is added to the tree $\mathcal{T}$ the pre/post-order lists are updated with the new vertex label. We also update two order data-structures, one for the pre-order and one for the post-order list. More precisely, to each vertex label v we associate a value α_v (resp. β_v) computed by the order data-structure for pre-order (resp. post-order) such that if v appears before some w in $\mathtt{Pre}$ (resp. $\mathtt{Post}$) then $\mathsf{ODCompare}(\alpha_v, \alpha_w) = true$ (resp. $\mathsf{ODCompare}(\beta_v, \beta_w) = true$).

Construction 3. *(DTTS from Standard Digital Signatures) Let $\mathtt{SSig} = (\mathsf{SKG}, \mathsf{SSig}, \mathsf{SVf})$ be a standard digital signature scheme, and let $\mathtt{BasicDTTS}$ be the scheme consisting of the following algorithms.*

- $\mathsf{TSKG}(1^\kappa)$: *use the SKG to generate a pair of keys (sk, pk). Set $tsk = sk$ and $tpk = pk$. Initialize two order data structures $\mathtt{OrderDS}_{\mathtt{Pre}}$ and $\mathtt{OrderDS}_{\mathtt{Post}}$ that will be used to maintain the sequence for pre-order and post-order traversal respectively. Return (tsk, tpk).*
- $\mathsf{TSign}(tsk, a, b)$:
  - *Add the vertex a or b to the graph if it does not exists. Let v (either a or b) be the vertex that was inserted.*
  - *Update $\mathtt{OrderDS}_{\mathtt{Pre}}, \mathtt{OrderDS}_{\mathtt{Post}}$ data structures to reflect the pre-order and post-order traversal sequence of the new tree: let $x_{\mathtt{Pre}}$ and $y_{\mathtt{Pre}}$ be the elements in $\mathtt{Pre}$ such that v comes just after $x_{\mathtt{Pre}}$ and lies just before $y_{\mathtt{Pre}}$. Compute using $\mathtt{OrderDS}_{\mathtt{Pre}}$ the label α_v (that is we have $\alpha_x \prec \alpha_v \prec \alpha_y$). Do the same for the post-order list $\mathtt{Post}$ and obtain β_v.*
  - *Sign using tsk the message $M_v = v||\alpha_v||\beta_v$. We obtain the signature $\sigma_v = \mathsf{SSig}(tsk, M_v)$.*
  - *Return $\tau_{(a,b)} = (M_a, \sigma_a, M_b, \sigma_b)$.*
- $\mathsf{TSComp}((a, b), \tau_{(a,b)}, (b, c), \tau_{(b,c)}, tpk)$*: parse $\tau_{(a,b)}$ as $(M_a, \sigma_a, M_b, \sigma_b)$ and $\tau_{(b,c)}$ as $(M_b, \sigma_b, M_c, \sigma_c)$. Return $\tau_{(a,c)} = (M_a, \sigma_a, M_c, \sigma_c)$.*
- $\mathsf{TSVf}((a, b), \tau, tpk)$: *parse τ as $(M_a, \sigma_b, M_a, \sigma_b)$. Verify the signatures; if any of them is invalid or M_a (resp. M_b) is not of the form $a||\alpha_a||\beta_a$ (resp. $b||\alpha_b||\beta_b$) then return $\perp$. If not, extract from M_a, M_b the values $\alpha_a, \beta_a, \alpha_b, \beta_b$. Verify that $\alpha_a \prec \alpha_b$ and $\beta_b \prec \beta_a$, using the algorithm $\mathsf{ODCompare}$. If the verification succeeds return valid else return $\perp$.*

Theorem 1. *If $\mathtt{SSig}$ is a secure digital signature scheme under chosen message attack, then $\mathtt{BasicDTTS}$ a is secure transitive signature scheme for directed trees where (a) the size of path signature is $O(n)$ bits, (b) the* Signer *generates an edge signature in time $O(n/\kappa)$, and (c) the time to compute a path signature for the* Combiner *is $O(1)$.*

Proof. Direct from proposition 3. □

Combining the basic construction and collision-resistant hashing with common-prefix proofs we can shrink the size of a path signature to $O(\kappa)$ bits (see theorem 3 in appendix A.1). Using the trade off technique introduced in appendix B for our hashing family we obtain the following result. (The full construction with trade off is outlined in section A.2.)

Theorem 2. *Let $\lambda \geq 1$. If* `SSig` *is a secure digital signature scheme under chosen message attack and $\mathcal{H}$ is a family of secure collision-resistant hash functions with common-prefix proofs, then* `PHDTTS` *with trade off is a secure DTTS scheme and (a) the size of the signature of an edge is $O(\lambda\kappa)$ bits, (b) the* Signer *has to perform $O(\lambda)$ cryptographic operations ($O(\lambda)$ hash computations and $O(1)$ signature) per edge insertion, (c) the* Verifier *can check that there is a path between two nodes in time $O(\lambda)$, and (d) the* Combiner *requires $O(\lambda n^{1/\lambda})$ time to compute a path signature.*

5 Conclusion and further work

In this work we introduced a new primitive *collision resistant hashing with common-prefix proofs* and showed it could be used to obtain efficient transitive signatures for directed trees. We recall however that the general problem of building transitive signatures for directed graphs remains open, as the problem of building stateless *DTTS*. We believe that our new hashing primitive may find many useful applications in particular in the design of authenticated data structures.

References

[1] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental Cryptography: The Case of Hashing and Signing. In Yvo G. Desmedt, editor, *CRYPTO 1994*, volume 839 of *LNCS*, pages 216–233–233. Springer-Verlag, July 1994.

[2] Mihir Bellare and Gregory Neven. Transitive Signatures: New Schemes and Proofs. *IEEE Transactions on Information Theory*, 51(6):2133–2151, June 2005.

[3] Dan Boneh and Xavier Boyen. Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles. In Christian Cachin and Jan L. Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 223–238. Springer Berlin / Heidelberg, 2004.

[4] Dan Boneh, David Freeman, Jonathan Katz, and Brent Waters. Signing a Linear Subspace: Signature Schemes for Network Coding. In Stanisaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 68–87–87. Springer / Berlin Heidelberg, 2009.

[5] Dan Boneh and David Mandell Freeman. Homomorphic Signatures for Polynomial Functions, 2010.

[6] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In Stanisaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *Irvine*, pages 481–500. Springer Berlin / Heidelberg, 2009.

[7] Jan Camenisch and Anna Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer-Verlag, 2002.

[8] Paul F. Dietz. Maintaining order in a linked list. In *STOC 1982*, pages 122–127. ACM Press, May 1982.

[9] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, 17(2):281, April 1988.

[10] Michael T. Goodrich, Roberto Tamassia, and Jasminka Hasic. An Efficient Dynamic and Distributed Cryptographic Accumulator. In *ISC '02 Proceedings of the 5th International Conference on Information Security*, pages 372–388, September 2002.

[11] Susan Hohenberger. The Cryptographic Impact of Groups with Infeasible Inversion. http://groups.csail.mit.edu/cis/theses/hohenberger-masters.ps, 2003.

[12] Robert Johnson, David Molnar, Dawn Xiaodong Song, and David Wagner. Homomorphic Signature Schemes. In Bart Preneel, editor, *CT-RSA 2002*, CT-RSA '02, pages 244–262. Springer-Verlag, 2002.

[13] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, third edition, 1997.

[14] Silvio Micali and Ronald Rivest. Transitive Signature Schemes. In Bart Preneel, editor, *CT-RSA 2002*, volume 2271 of *LNCS*, pages 236–243. Springer / Berlin Heidelberg, February 2002.

[15] Rolf Möhring, Rajeev Raman, Michael Bender, Richard Cole, Erik Demaine, Martin Farach-Colton, and Jack Zito. Two Simplified Algorithms for Maintaining Order in a List. *Algorithms - ESA 2002*, 2461:219–223–223, August 2002.

[16] Gregory Neven. A simple transitive signature scheme for directed trees. *Theoretical Computer Science*, 396(1-3):277–282, May 2008.

[17] Lan Nguyen. Accumulators from Bilinear Pairings and Applications. In Alfred Menezes, editor, *Topics in Cryptology CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 275–292–292, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[18] Siamak Fayyaz Shahandashti, Mahmoud Salmasizadeh, and Javad Mohajeri. A Provably Secure Short Transitive Signature Scheme from Bilinear Group Pairs. In Carlo Blundo and Stelvio Cimato, editors, *Security in Communication Networks*, volume 3352 of *LNCS*, pages 60–76. Springer / Berlin Heidelberg, 2005.

[19] Xun Yi. Directed Transitive Signature Scheme. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *LNCS*, pages 129–144. Springer Berlin / Heidelberg, 2007.

A Full construction for $DTTS$ with Hashing with Common-Prefix Proofs.

A.1 Full construction (without trade off)

We extend our basic construction as follows. Using the same alphabet $\Sigma = \{0, \$, 1\}$ where $0 < \$ < 1$, the string comparison is now done through their hash values and corresponding proofs provided by the scheme `PH`. That is, in order to prove that two strings (that correspond to paths in the order data structures) X, Y are such that $X \prec Y$ through their hashes $H(X), H(Y)$ the *Combiner* must compute:

- A maximum common prefix C of X and Y. E.g.: ($X = 10001\$$, $Y = 1001\$, C = 100$) or ($X =$ 10001\$, Y=100011\$, $C = 1000$). Note that C will never contain the symbol \$.
- A proof that C is a prefix of X up to position $i = |C|$.

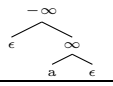
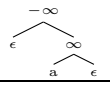
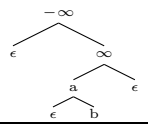
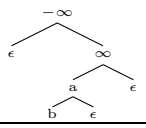
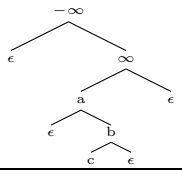
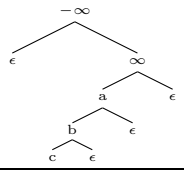
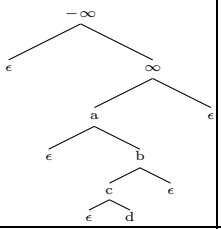
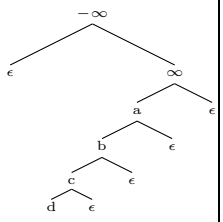
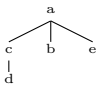
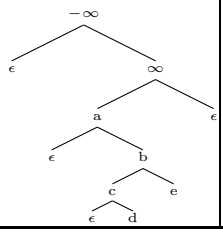
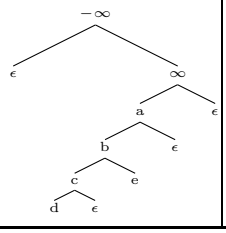
Step	Tree $\mathcal{T}$	Pre/Post order traversal	$\texttt{OrderDS}_{\texttt{Pre}}$	$\texttt{OrderDS}_{\texttt{Post}}$	Labels
0		$\texttt{Pre} = \epsilon$ $\texttt{Post} = \epsilon$			$\alpha_{-\infty} = \$$ $\alpha_{\infty} = 1\$$ $\beta_{-\infty} = \$$ $\beta_{\infty} = 1$
1		$\texttt{Pre} = a$ $\texttt{Post} = a$			$\alpha_a = 10\$$ $\beta_a = 10\$$
2		$\texttt{Pre} = ab$ $\texttt{Post} = ba$			$\alpha_b = 101\$$ $\beta_b = 100\$$
3		$\texttt{Pre} = acb$ $\texttt{Post} = cba$			$\alpha_c = 1010\$$ $\beta_c = 1000\$$
4		$\texttt{Pre} = acdb$ $\texttt{Post} = dcba$			$\alpha_d = 10101\$$ $\beta_d = 10000\$$
5		$\texttt{Pre} = acdbe$ $\texttt{Post} = dcbea$			$\alpha_e = 1011\$$ $\beta_e = 1001\$$

Figure 1: Example of several insertions in a directed tree and their effect on the order data structures.
Step 0: The tree $\mathcal{T}$ to authenticate has no nodes. The sequences `Pre` and `Post` are empty as well. The order data structure $\texttt{OrderDS}_{\texttt{Pre}}$ and $\texttt{OrderDS}_{\texttt{Post}}$ contain two nodes $-\infty$ and ∞ that are the bounds of the ordered universe. Each edge is marked implicitly by 0 (for a left child) and 1 (for a right child).
Step 1: The first node a of $\mathcal{T}$ is created. The pre/post-order lists contain only a. The order data structures are updated in such a way they reflect the order $-\infty \prec a \prec \infty$. In particular we have that labels $\alpha_a = \beta_a = 10\$$. The end marker \$ is appended so it allows direct order label comparison through lexicographical order using that $0 < \$ < 1$.
Step 2: A child b is added to a. Now the pre-order sequence `Pre` is equal to ab and the post-order sequence is ba. As b comes after a in `Pre` we have that b is the right child of a in $\texttt{OrderDS}_{\texttt{Pre}}$. Similarly b is the left child of a in $\texttt{OrderDS}_{\texttt{Post}}$ as it comes before a in `Post`.
Step 3,4 and 5: We follow the same procedure and obtain for each node v its order labels α_v and β_v respectively.
Comparing two node labels: In step 5 we can check easily using the order labels that for example d is a descendant of a. Indeed we have $\alpha_a = 10\$$ and $\alpha_d = 10101\$$ which means $\alpha_a \prec \alpha_d$. Also we can check that $\beta_d = 10000\$ \prec \beta_a = 10\$$. We can also observe that there is no path from b to c for example as $\alpha_c = 1010\$ \prec \alpha_b = 101\$$ and also $\beta_c = 1000 \prec \beta_b = 100\$$.

- A proof that C is a prefix of Y up to position $i = |C|$.
- A proof that $H(C||x)$ is a prefix of X up to position $i + 1$.
- A proof that $H(C||y)$ is a prefix of Y up to position $i + 1$.

Verifying that $X \prec Y$ will then consists basically in checking the proofs and also verifying that $x < y$.

Construction 4. *(DTTS from CRHwCPP)*
Let $\mathtt{PH} = (\mathsf{PHGen}, \mathsf{PHEval}, \mathsf{PHProofGen}, \mathsf{PHCheck})$ *be a prefix collision-resistant hash function family, and let* PHDTTS *be the scheme consistent of the following algorithms.*

- $\mathsf{TSKG}(1^\kappa)$:
  First, generate the public parameters for the PH *scheme as well as a pair of private/public keys* (sk, pk) *for the* Signer *running* SKG. *Set* $tsk = sk$ *and* $tpk = tk$. *Initialize two order data structures* $\mathtt{OrderDS}_{\mathtt{Pre}}$ *and* $\mathtt{OrderDS}_{\mathtt{Post}}$. *Return* (tsk, tpk).

- $\mathsf{TSign}(tsk, a, b)$:
  Do the same as in BasicDTTS *except that the message to be signed is now* $M_v = v||H_{\alpha_v}||H_{\beta_v}$, *where* $H_{\alpha_v} = H(\alpha_v)$ *and* $H_{\beta_v} = H(\beta_v)$. *Note that* $H(\alpha_v), H(\beta_v)$ *can be computed incrementally due to the fact that* H *is homomorphic and that the labels* α_v *(resp.* β_v*), have been computed previously.*

- $\mathsf{TSComp}((a, b), \tau_{(a,b)}, (b, c), \tau_{(b,c)}, tpk)$:
  If $\tau_{(a,b)}$ *or* $\tau_{(b,c)}$ *is invalid, then reject. To compute the signature of the edge* (a, c) *the* Combiner *proceeds as follows. Find the lowest common ancestor in* $\mathtt{OrderDS}_{\mathtt{Pre}}$ *for node*(α_a) *and node*(α_c). *Denote it node*(α'_d). *Note that* α'_d *is a string without the terminating symbol* \$ *but such that* $H(\alpha_d) = H(\alpha'||\$)$ *is part of an already signed message* $M_d = d||H(\alpha_d)||H(\beta_d)$. *Let* σ_d *be the signature of* M_d. *We have that* α_d *is the maximum common prefix of* α_a *and* α_c. *Let* $l = |\alpha_d|$. *Compute the following values:*

  - $\pi_1 = \mathsf{PHProofGen}(\alpha'_d, \alpha_a, l - 1, PK)$
  - $\pi_2 = \mathsf{PHProofGen}(\alpha'_d, \alpha_c, l - 1, PK)$
  - $\pi_3 = \mathsf{PHProofGen}(\alpha'_d||x, \alpha_a, l, PK)$
  - $\pi_4 = \mathsf{PHProofGen}(\alpha'_d||y, \alpha_c, l, PK)$

  where (x, y) *may be the pair of symbols* $(0, \$)$ *or* $(0, 1)$ *or* $(\$, 1)$. *(Recall that* $0 < \$ < 1$ *and that all strings are ended by* \$.*) Finally obtain* $\pi_5 = (M_a, \sigma_a, M_c, \sigma_c, H_{\alpha'_d}, \sigma_d, l, x, y)$ *where* $H_{\alpha'_d} = H(\alpha'_d)$. *Set* $\pi_{\mathtt{Pre}} = (\pi_1, \pi_2, \pi_3, \pi_4, \pi_5)$.
  Compute similarly $\pi_{\mathtt{Post}}$ *and return* $\tau_{(a,c)} = (\pi_{\mathtt{Pre}}, \pi_{\mathtt{Post}})$.

- $\mathsf{TSVf}((a, b), \tau, tpk)$:
  Extract $\pi_{\mathtt{Pre}}$ *from* $\tau = (\pi_{\mathtt{Pre}}, \pi_{\mathtt{Post}})$. *Parse* $\pi_{\mathtt{Pre}}$ *as* $(\pi_1, \pi_2, \pi_3, \pi_4, \pi_5)$.
  Parse π_5 *as* $(M_a, \sigma_a, M_c, \sigma_c, M_d, \sigma_d, H_{\alpha'_d}, l, x, y)$ *where* $M_v = v||H_{\alpha_v}||H_{\beta_v}$ *for* $v \in \{a, c, d\}$. *Check that all the pairs of message-signatures are valid under public key tpk. Check that* $H_{\alpha'_d} \cdot H(0^l||\$) = H_{\alpha_d}$ *the second component of* M_d. *Check that* x *and* y *are symbols. If one of the verification fails return* $\perp$. *Verify proofs* $\pi_1, \pi_2, \pi_3, \pi_4$ *using* PHCheck, *namely compute:*

  - $\mathsf{PHCheck}(H_{\alpha'_d}, H_{\alpha_a}, \pi_1, l - 1, PK)$
  - $\mathsf{PHCheck}(H_{\alpha'_d}, H_{\alpha_b}, \pi_2, l - 1, PK)$
  - $\mathsf{PHCheck}(H_{\alpha'_d} \cdot H(0^{l-1}||x), H_{\alpha_a}, \pi_3, l, PK)$
  - $\mathsf{PHCheck}(H_{\alpha'_d} \cdot H(0^{l-1}||y)), H_{\alpha_b}, \pi_4, l, PK)$

If all these verifications pass return valid *otherwise return* $\perp$.

Theorem 3. *If* SSig *is a secure digital signature scheme under chosen message attack and* $\mathcal{H}$ *is a family of secure collision-resistant hash function with common-prefix proofs, then* PHDTTS *is a secure DTTS scheme where (a) the size of the signature is* $O(\kappa)$ *bits, (b) the* Signer *generates an edge signature in* $O(1)$ *time, and (c) the time to compute a path signature for the* Combiner *is* $O(n)$.

Proof. Let $\mathcal{A}$ a PPT adversary that breaks our scheme. We build an adversary $\mathcal{B}$ that breaks the security of SSig or PH. $\mathcal{B}$ has access to a signing oracle $\mathcal{O}_{\mathtt{SSig}}(\cdot)$ and is given the description of the prefix hash function H as described in construction 1. $\mathcal{B}$ forwards all the public parameters to $\mathcal{A}$. $\mathcal{A}$ asks for edges signing to $\mathcal{B}$ who replies using the signing oracle $\mathcal{O}_{\mathtt{SSig}}(\cdot)$ and H. Finally $\mathcal{A}$ outputs a signature τ such that $\mathsf{TSVf}((a,b),\tau,tpk) = \mathsf{valid}$ and there is no path from a to b in $\mathcal{T}$. We first consider the case where signed messages M_a, M_b do not all correspond to edges insertion in $\mathcal{T}$. This means that $\mathcal{B}$ has been able to compute some signature for a message M' not previously requested to the oracle $\mathcal{O}_{\mathtt{SSig}}$. So now we assume that all signed messages reflects the history of edges insertions in the tree.

Let α_a, α_b and β_a, β_b be the order labels associated to vertices a, b in $\mathtt{OrderDS}_{\mathtt{Pre}}$ and $\mathtt{OrderDS}_{\mathtt{Post}}$ respectively. As there is no path from a to b, this means that (i) a appears before b in Pre and also a appears before b in Post or (ii) b appears before a in Pre and also b appears before a in Post or (iii) there is a path, but from b to a. Assume we are in case (i), and note that cases (ii) and (iii) are similar. If indeed a appears before b in Pre then the adversary $\mathcal{A}$ managed to prove that b appears before a in Post although the contrary is true. This means in particular that $\pi_1, \pi_2, \pi_3, \pi_4$ prove that there exists a string C such that $C||x$ is a prefix of β_b and $C||y$ is a prefix of β_a and $x < y$. It is worth noting that, although the proofs $\pi_1, \pi_2, \pi_3, \pi_4$ do not mention explicitly the strings tied to the nodes (only hash values and lengths), these strings are present in the data structures $\mathtt{OrderDS}_{\mathtt{Pre}}$ and $\mathtt{OrderDS}_{\mathtt{Post}}$. If some hash value is linked to two different pre-images then $\mathcal{B}$ has found a collision for $\mathcal{H}$. In particular this means that $\mathcal{B}$ knows C. Now, as indeed $\beta_a \prec \beta_b$, there exists no such string C, so this means that either $C||x$ is not a prefix of β_a or $C||y$ is not a prefix of β_b, therefore $\mathcal{B}$ has been able to break the security of the scheme PH. □

A.2 Sketch of the full construction with trade off

We sketch here the construction. We use a pair of order data structures for each level in the following way. Consider the case of $\mathtt{OrderDS}_{\mathtt{Pre}}$ (the case $\mathtt{OrderDS}_{\mathtt{Post}}$ can be treated similarly.) At level 0 we have the label α that is a path in the binary search tree of $\mathtt{OrderDS}^0_{\mathtt{Pre}}$. When a path and its associated label α of $\mathtt{OrderDS}^0_{\mathtt{Pre}}$ becomes larger than $n^{1/\lambda}\kappa$ then we compute $H(\alpha_0)$ (where α_0 is the $n^{1/\lambda}\kappa$ bits long prefix of α) and insert this new string in $\mathtt{OrderDS}^1_{\mathtt{Pre}}$, a new binary search tree. We store in the node of $\mathtt{OrderDS}^0_{\mathtt{Pre}}$ with label α_0 the pointer to the node of $\mathtt{OrderDS}^1_{\mathtt{Pre}}$ with label $H(\alpha_0)$. Then if the length of the label α continues to grow until reaching length $2n^{1/\lambda}\kappa$ and label $\alpha = \alpha_0||\alpha_1$, we compute $H(\alpha_1)$ and insert this string in $\mathtt{OrderDS}^1_{\mathtt{Pre}}$ starting from node with label $H(\alpha_0)$. We follow the procedure for each chunk of size $n^{1/\lambda}\kappa$, always inserting the new string in $\mathtt{OrderDS}^1_{\mathtt{Pre}}$. This process causes the tree $\mathtt{OrderDS}^1_{\mathtt{Pre}}$ to grow. We apply the same procedure to $\mathtt{OrderDS}^1_{\mathtt{Pre}}$ by creating the tree $\mathtt{OrderDS}^2_{\mathtt{Pre}}$ as described above. We can follow this mechanics until reaching the tree $\mathtt{OrderDS}^{\lambda-1}_{\mathtt{Pre}}$ of maximal height $n^{1/\lambda}\kappa$. We observe that in each tree the common ancestor between two nodes N_1, N_2 will be at maximum distance $n^{1/\lambda}\kappa$ from both nodes. This means that computing the proof for the PH will take time $O(n^{1/\lambda})$. The *Signer* will have to compute himself the hash values for all levels but only has to sign the final string of the last level, so the time to compute a signature is $O(\lambda)$. Finally signature for any path is $O(\lambda\kappa)$ bits long.

B Proof Generation and Verification Tradeoff

First, we can see that a simple optimization can be made to our scheme: instead of working with the binary alphabet $\Sigma = \{0,1\}$ it is possible to encode the string S using $\Sigma = \{0,1,...,2^\kappa - 1\}$ and thus

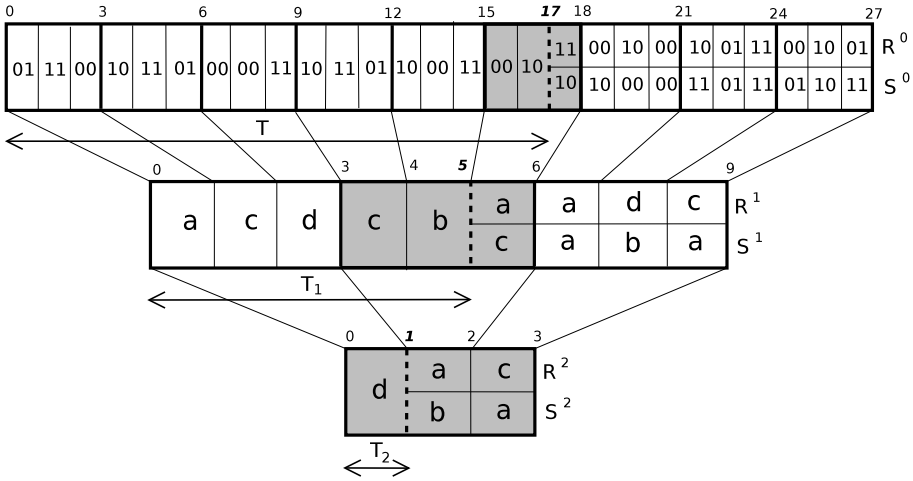


Figure 2: Toy example of trade off data structure.

In this example, let $n = 54, \lambda = 3, \kappa = 2$ and the alphabet $\Sigma = \{a, b, c, d\}$. We have $t = (54/2)^{1/3} = 3$. R and S are two strings that share the same prefix T until position $i = i_0 = 17 \cdot 2 = 34$. We start at level 2. The *Combiner* shows that R^2 and S^2 are equal up to the position $i_2 = 1$. Then he also proves that da is a prefix of R^2 and db is a prefix of S^2. This means we need to find the common prefix of strings $H^{-1}(a) = cba$ and $H^{-1}(b) = cbc$. So we move up to level 1. Now the *Combiner*, using $H^{-1}(a)$ and $H^{-1}(b)$, shows that R^1 and S^1 share a common j_1-symbol prefix up the relative position $j_1 = 2$. This means that $i_1 = 1 \cdot 3 + 2 = i_2 \cdot t + j_1 = 5$. We move at the level 0. The *Combiner* then shows that a,c are the symbols that come just after T^1 in R^1 and S^1 respectively. Now using $H^{-1}(a) = 001011$ and $H^{-1}(c) = 001010$ the *Combiner* shows that $R^0 = R$ and $S^0 = S$ share a common prefix $T^0 = T$ up to the relative position $j_0 = 2$. This means that $i = i_0 = 34 = (5 \cdot 3 + 2) \cdot 2 = (i_1 \cdot t + j_0)\kappa$.

compute $H(S) = \prod_{i=1}^{n/\kappa} g^{\sigma_i s^i}$ where $\forall i : 1 \leq i \leq n/\kappa$, $\sigma_i \in \{0, 1, ..., 2^\kappa - 1\}$. This observation reduces the number of cryptographic operations[1] required to compute a proof from $O(n)$ to $O(\frac{n}{\kappa})$ [2]. Although simple, this observation is crucial for the following.

We now show how to balance the computational work between the *Combiner* – who generates the hashes and the proofs – and the *Verifier*– who checks the proofs – as follows. We obtain a $CRHwCPP$ scheme such that, for any $\lambda \geq 1$, the time it takes to compute a proof is reduced to $O(\lambda n^{1/\lambda})$ while the time it takes to generate a signature or verify a proof is now $O(\lambda)$. Let $t > 0$, and $S = S^0$ be the n-bit string (and thus $\frac{n}{\kappa}$ symbols) to hash. Assume for clarity of the exposition that $\frac{n}{\kappa}$ is a power of t. In order to compute $H(S)$, we first cut S in chunks of size $t\kappa$. For each chunk $S_0, S_1, ..., S_{n/(t\kappa)-1}$ we compute the hash value $H(S_i)$ and obtain a new string S^1 of size $\frac{n}{t\kappa}\kappa = \frac{n}{t}$ bits. We repeat the same procedure with the new string S^1 and obtain a string S^2 of size $\frac{n}{t^2\kappa}\kappa = \frac{n}{t^2}$ bits. We follow the same algorithm until reaching a string $S^{\frac{\log(n/\kappa)}{\log t}-1}$ with at most t symbols (i.e. $t\kappa$ bits) which hash is the final output.

To be more concrete, we set $t = (\frac{n}{\kappa})^{1/\lambda}$ so that the new data structure has λ levels. In order to prove that the n-bit strings R and S have a i-bit common prefix we do the following. Let $R^0, R^1, ..., R^{\lambda-1}$ and $S^0, S^1, ..., S^{\lambda-1}$ be the sequences of strings obtained by following the above hashing algorithm on input $R^0 = R$ and $S^0 = S$, where R^ℓ (resp. S^ℓ) is the string processed at level ℓ. We start at level $\lambda - 1$. At this level there is only one chunk of size $t = (\frac{n}{\kappa})^{1/\lambda}$ (number of symbols). Using PHProofGen, the *Combiner* computes a proof $\pi_{\lambda-1}$ showing that $R^{\lambda-1}$ and $S^{\lambda-1}$ share a common prefix $T^{\lambda-1}$ until position $i_{\lambda-1}$.

[1] For the sake of clarity we do not describe the case where the alphabet contains the special symbol \$. The asymptotic efficiency remains the same however.

[2] The proof of security remains the same, except that the symbols lie in a larger alphabet.

Then the *Combiner* computes two additional proofs: one proof $\pi_{\lambda-1,R}$, showing that $T^{\lambda-1}||\sigma_{R,\lambda-1}$ is a prefix of $R^{\lambda-1}$, and another one (say $\pi_{\lambda-1,S}$), showing that $T^{\lambda-1}||\sigma_{S,\lambda-1}$ is a prefix of $S^{\lambda-1}$. Notice that since the *Combiner* has previously computed the hash values for each level, he knows the pre-images of $\sigma_{R,\lambda-1}$ and $\sigma_{S,\lambda-1}$ under H: the $t\kappa$-bit long strings $R^{\lambda-2}_{i_{\lambda-1}t} = H^{-1}(\sigma_{R,\lambda-1})$ (that is the chunk number $i_{\lambda-1}t$ of $R^{\lambda-2}$) and similarly $S^{\lambda-2}_{i_{\lambda-1}t} = H^{-1}(\sigma_{S,\lambda-1})$. The *Combiner* then moves up one level and repeats the procedure at level $\lambda-2$ now working on the strings $H^{-1}(\sigma_{R,\lambda-1}), H^{-1}(\sigma_{S,\lambda-1})$ and generating a proof (say $\pi_{\lambda-2}$) that they have some $j_{\lambda-2}$-symbol common prefix. We can see that, up to this point, the *Combiner* has proven that strings $R^{\lambda-2}$ and $S^{\lambda-2}$ share a common prefix of length $i_{\lambda-2} = i_{\lambda-1}t + j_{\lambda-2}$. The procedure continues iteratively going up at the levels until it reaches level 0 (see example in figure 2) where $i_0 = (i_1 t + j_0)\kappa$. The total size of the proof is $O(\lambda\kappa)$ bits.

The verification step at each level consists in verifying that (1) the proofs computed by the *Combiner* are valid, and (2) for each two consecutive levels $\ell-1$ and ℓ the proofs for level ℓ are relative to the pre-images $H^{-1}(\sigma_{R,\ell-1})$ and $H^{-1}(\sigma_{S,\ell-1})$. These considerations lead to the following result.

Theorem 4. *Let $\lambda \geq 1$. Under the n-BDHI assumption we can build a secure CRHwCPP function family where (a) the time to compute a hash value is $O(\lambda)$, (b) the time to compute a proof is $O(\lambda n^{1/\lambda})$, and (c) the time to verify a proof is $O(\lambda)$.*

Proof. First we observe that the mapping between each level is a collision resistant hash function family.

Assume an adversary $\mathcal{A}$ manages to break the trade off scheme. Then we build an adversary $\mathcal{B}$ that breaks the (simple) PH by computing a forged proof or finding a collision for $\mathcal{H}$. Adversary $\mathcal{B}$ sends the public parameters of the scheme to $\mathcal{A}$ who answers with a forgery for the trade off scheme. More precisely $\mathcal{A}$ returns two strings R, S and valid proofs for each level that lead to the claim that $R = R^0$ and $S = S^0$ are equal up to position i although there exists some index $k < i$ such that $R[k] \neq S[k]$.

Let $i = (i_1 \cdot t + j_0) \cdot \kappa$ the decomposition of i for the first level. If $(i_1 \cdot t)\kappa \leq k < (i_1 \cdot t + j_0)\kappa$ this means that $R_{i_1 \cdot t}$ and $S_{i_1 \cdot t}$ will not share a common prefix until relative position j_1, thus $\mathcal{B}$ has found a forgery for PH. If $k < i_1 \cdot t$ we are reduced to the case where $k' = (\lceil k/\kappa \rceil) - (i_1 \cdot t)$ is such that $R^1_{i_2 \cdot t}[k'] \neq S^1_{i_2 \cdot t}[k']$ as otherwise $\mathcal{B}$ would have found a collision for $\mathcal{H}$. Now if $i_2 \cdot t \leq k' < i_2 \cdot t + j_1$ then again, $\mathcal{B}$ has broken the security of PH. If $k' < i_2 \cdot t$, we need to analyse similarly the case for the next level. Eventually we will reach a level where $\mathcal{B}$ manages to break the security of PH because $k < i$ and the decomposition of i in base t is unique. □

C Short DTTS using Cryptographic Accumulators

We first recall Neven's signature scheme for directed trees.

Construction 5. *(Neven's scheme [16])*

- $\mathsf{TSKG}(1^\kappa)$: *returns a pair of private/public keys (tsk, tpk) for a standard digital signature scheme.*
- $\mathsf{TSign}(tsk, a, b)$: *the state of the tree is maintained by its description as a graph $G = (V, E)$, the current root r and two tables $up[\cdot]$ and $down[\cdot]$ To sign a new edge we distinguish between the following cases:*

  1. *$V = \emptyset$:*
  $$r \leftarrow a; V \leftarrow V \cup \{a, b\}; E \leftarrow E \cup \{(a, b)\}$$
  $$up[a] = down[a] = down[b] \leftarrow \epsilon; up[b] \leftarrow a$$

  2. *$a \in V$ and $b \notin V$:*
  $$V \leftarrow V \cup \{b\}; E \leftarrow E \cup \{(a, b)\}$$
  $$up[b] \leftarrow up[a]||a; down[b] \leftarrow \epsilon$$

3. $a \notin V$ *and* $b = r$*:*

$$r \leftarrow a; V \leftarrow V \cup \{b\}; E \leftarrow E \cup \{(a,b)\}$$
$$up[a] \leftarrow \epsilon; down[a] \leftarrow b||down[b]$$

In all other cases the Signer *rejects because the query does not preserve the tree structure of the graph. The* Signer *sets* $C_a \leftarrow (i, down[i])$ *and* $C_j \leftarrow (j, up[j])$*, and computes two standard signatures* $\sigma_a = \mathsf{SSig}(tsk, C_a)$ *and* $\sigma_b \leftarrow \mathsf{SSig}(tsk, C_b)$*. The transitive signature for the edge* (a, b) *is the tuple* $\tau_{(a,b)} \leftarrow (C_a, \sigma_a, C_j, \sigma_j)$*.*

- $\mathsf{TSComp}((a,b), \tau_{(a,b)}, (b,c), \tau_{(b,c)}, tpk)$*: Parse* $\tau_{(a,b)}$ *as* $(C_a, \sigma_a, C_b, \sigma_b)$ *and* $\tau_{(b,c)}$ *as* $(C_{b'}, \sigma_{b'}, C_c, \sigma_c)$*. If* $b \neq b'$ *reject else return the composed signature for edge* (a, c) *as* $\tau_{(a,c)} \leftarrow (C_a, \sigma_a, C_c, \sigma_c)$*.*
- $\mathsf{TSVf}((a,b), \tau, tpk)$ *: Parse* τ *as* $(C_a, \sigma_a, C_b, \sigma_b)$*, and parse* C_a *as* $(a, down)$ *and* C_b *as* (b, up)*. If* $\mathsf{SVf}(tpk, \sigma_a) = \perp$ *or* $\mathsf{SVf}(tpk, \sigma_b) = \perp$ *return* $\perp$*. If* b *occurs in down or* a *occurs in up or there exists some* c *that occurs both in down and up then return* valid *else return* $\perp$*.*

Cryptographic Accumulators. An accumulator is a scheme that enables to represent a set by a short value called accumulated value. Then given an element it is possible to prove that this elements belongs to the set by exhibiting a proof called witness. An accumulator is dynamic if it is possible to update the set. Two kinds of participants are involved in an accumulator scheme: the *Manager* that holds the set, updates it and computes all the related values, and the *User* that can test for membership of a given element. The next definition introduces the functionalities involved in a dynamic accumulator. In our context the *Manager* would be the *Signer* and the *User* takes the role of the *Verifier*. The *Combiner* is also a *User* that only computes witnesses for the accumulator scheme (still without knowing the trapdoor).

Definition 7. *(Syntax for Dynamic Accumulator, [7])*
Let $\kappa \in \mathbb{N}$ *be the security parameter. An accumulator scheme* $\mathfrak{Acc}$ *consists of the following algorithms.*

- $\mathsf{Setup}(1^\kappa)$*: This probabilistic algorithm takes* κ *in unary as input and returns a pair of public and private keys* (PK, SK)*, and the initial accumulated value for the empty set* $Acc_\emptyset$*. This algorithm is run by the* Manager*.*
- $\mathsf{AccVal}(X, Acc_\emptyset, PK, [SK])$*: Given a finite set of elements* X *(of at most polynomial size in* κ*), a public key* PK *and the initial accumulated value* $Acc_\emptyset$*, this algorithm returns the accumulated value* Acc_X *corresponding to the set* X*. This algorithm is run by the* Manager*. Depending on the implementation, the secret key* SK *may also be given as optional parameter, often to improve the efficiency*[3]*.*
- $\mathsf{Verify}(x, w, Acc_X, PK)$*: given an element* x*, a witness* w*, an accumulated value* Acc_X*, and a public key* PK *, this deterministic algorithm returns* valid *if the verification is successful, meaning that* $x \in X$*, or* $\perp$ *otherwise. This algorithm is run by a* User*.*
- $\mathsf{WitGen}(x, Acc_X, PK)$*: this algorithm returns a witness* w *associated to the element* x *of the set* X *represented by* Acc_X*. We consider the case where this algorithm is run by a* User*.*
- $\mathsf{AddEle}(x, Acc_X, PK, [SK])$*: this algorithm computes the new accumulated value* $Acc_{X \cup \{x\}}$ *obtained after the insertion of* x *into set* X*. This algorithm is run by the* Manager*.*
- $\mathsf{DelEle}(x, Acc_X, PK, [SK])$*: this algorithm computes the new accumulated value* $Acc_{X \setminus \{x\}}$ *obtained by removing the element* x *from the accumulated set* X*. This algorithm is run by the* Manager*. In our application however we do not require it.*

[3]The secret key may also be an optional parameter in the algorithms WitGen, AddEle, DelEle.

Naturally, we say the scheme is correct if every valid witness leads to a successful verification.

Definition 8. *(Correctness) Let X be a set and Acc_X its associated accumulated value, PK a public key, SK the corresponding private key, and $y \in Y$. Let w_y a value (witness) that satisfies $w_y \leftarrow \mathsf{WitGen}(y, Acc_Y, PK, SK)$. We say that an accumulator scheme $\mathfrak{Acc}$ is correct if and only if $\mathsf{Verify}(y, w_y, Acc_X, PK) = \mathsf{valid}$, for every such y, w_y, X.*

The security of an accumulator scheme is captured by an experiment where the adversary plays the role of a *User* and attempts to forge a witness (i.e. finding a valid witness for an element that does not belong to the set) while having access to an oracle that implements the operations relative to the *Manager*. Such adversary must succeed with at most negligible probability on the security parameter. This experiment is very similar to the one used to define the security of digital signatures.

Definition 9. *(Security for Dynamic Accumulators, [7])*

Let $\mathfrak{Acc}$ be a dynamic accumulator scheme.

We consider the notion of security denoted $\mathcal{UF}$-$\mathcal{ACC}$ described by the following experiment: on input the security parameter κ, the adversary $\mathcal{A}$ has access to an oracle $\mathcal{O}(\cdot)$ that replies to queries by playing the role of the accumulator Manager*. Using the oracle, the adversary can insert and delete a polynomial number of elements of his choice. The oracle $\mathcal{O}(\cdot)$ replies with the new accumulated value. The adversary can also ask for witness computations or update information. Finally, the adversary is required to output a pair (x, w).*

The advantage of the adversary $\mathcal{A}$ is defined by:

$$Adv_{\mathfrak{Acc}}^{\mathcal{UF}\text{-}\mathcal{ACC}}(\mathcal{A}) \quad = \quad \Pr\left[\,\mathsf{Verify}(x, w, Acc_X, PK) = \mathsf{valid} \wedge x \in X\,\right]$$

where PK is the public key generated by Setup, and Acc_X is the accumulated value of the resulting accumulated set X. The scheme $\mathfrak{Acc}$ is said to be secure if for every probabilistic polynomial time adversary $\mathcal{A}$ we have:

$$Adv_{\mathfrak{Acc}}^{\mathcal{UF}\text{-}\mathcal{ACC}}(\mathcal{A}) \quad = \quad \mathsf{neg}(\kappa)$$

Short signatures for Neven's scheme using Accumulators. The idea to shrink the size of edge signatures for Neven's scheme is simply to maintain an accumulator for each list $up[\cdot]$ and $down[\cdot]$. The accumulated values are signed using a standard signature scheme, and the *Combiner* can convince a *Verifier* that a vertex belongs to some list by computing the appropriate witness. Using for example one of the scheme introduced in [7, 17, 6] the edge signature will have constant size. Note that the accumulated values are related in the following way: if y is a child of x then the $Acc_y = \mathsf{AddEle}(x, Acc_x, PK)$. This means that instead of handling an accumulator for every node which would be costly, we only need to compute the accumulated value on the fly. Then to obtain a witness, the procedure consists in recollecting the values on the path for lists $down[\cdot]$ or $up[\cdot]$.

We can also handle the trade off $(\lambda, \lambda n^{1/\lambda})$ using accumulators with the tree technique presented for example in [10]. Here the idea is to maintain pointers to the previous node that enables to compute a witness / accumulated value.

To the best of our knowledge there is no way to parallelize the computation of witnesses for the RSA accumulator [7] or NGuyen's accumulator [17]. The accumulator presented in [6] allows parallel computation for witnesses but the maximal size of the tree must be known before the scheme is initialized.

To summarize, our solution based on hashing with common prefix proof is the first one that (1) enables the $(\lambda, \lambda n^{1/\lambda})$ trade off between the time to compute a path signature and to verify it, (2) allows the parallelization of the computation of a path signature and (3) does not force the size of the tree to be known beforehand.