UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PREDICATE-PRESERVING COLLISION-RESISTANT HASHING

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS MENCIÓN COMPUTACIÓN

PHILIPPE CAMACHO CORTINA

PROFESOR GUÍA:
ALEJANDRO HEVIA ANGULO

PROFESOR CO-GUÍA:
JÉRÉMY BARBAY

MIEMBROS DE LA COMISIÓN:
MARCOS KIWI KRAUSKOPF
GONZALO NAVARRO BADINO
GREGORY NEVEN

ESTE TRABAJO HA SIDO FINANCIADO PARCIALMENTE POR CONICYT

SANTIAGO DE CHILE
SEPTIEMBRE 2013

Resumen

Se estudian funciones de hash resistentes a colisiones (FHRC) que permiten validar eficientemente predicados sobre las entradas, usando solamente los valores de hash y certificados cortos. Para los predicados, consideramos conjuntos y cadenas de caracteres.

La idea de computar el valor de hash de un conjunto con el fin de demostrar (no)pertenencia aparece en la literatura bajo el nombre de *acumuladores criptográficos* (Benaloh y De Mare, CRYPTO 1993). En esa tesis se propone primero un acumulador criptográfico que permite manipular conjuntos dinámicos (es decir donde es posible insertar y borrar elementos) y cuya seguridad no depende de ninguna autoridad de confianza. Luego mostramos que *no* existe ningún acumulador criptográfico que permite la actualización de todos los certificados en tiempo constante después de varias modificaciones. Este resultado resuelve un problema abierto propuesto por Nicolisi y Fazio en su estado del arte sobre acumuladores criptográficos (2002). La siguiente contribución de esa tesis es una FHRC que permite la comparación de cadenas largas según el orden lexicográfico. Usamos esa FHRC para construir un esquema de firma digital transitivo que permite autenticar árboles dirigidos. Esa construcción es la más eficiente a la fecha, y mejora de forma sustancial el resultado de Gregory Neven (Theoretical Computer Science 396). Finalmente usamos una FHRC similar para demostrar que una cadena corresponde a la expansión binaria de un cierto valor. Con la ayuda de técnicas de pruebas de nula divulgación usamos esa construcción para implementar un protocolo que permite revelar gradualmente un secreto. Luego este protocolo se usa para poder intercambiar de forma equitativa firmas cortas de Boneh-Boyen (EUROCRYPT 2004) sin la necesidad de recurrir a una autoridad de confianza.

Abstract

We study Collision-Resistant Hash Functions (CRHF) that allow to compute proofs related to some predicate on the values that are hashed. We explore this idea with predicates involving sets (membership) and strings (lexicographical order, binary decomposition). The concept of hashing a set in order to prove (non)membership first appears in the literature under the name of *cryptographic accumulators* (Benaloh and De Mare, CRYPTO 1993). In this thesis we start by introducing a cryptographic accumulator that handles dynamic sets (it is possible to insert and delete elements) and whose security does not involve a trusted third party. Then we show that *no* cryptographic accumulator can have the property of *batch update* (efficient refresh of all the proofs after several updates to the set via a single operation) and thus solve an open problem stated by Nicolisi and Fazio in their survey on cryptographic accumulators (2002).

We then describe a CRHF that enables efficient comparison of large strings through their lexicographical order. We use this CRHF to build a practical transitive signature scheme to authenticate *directed trees*. To the best of our knowledge, this is the first practical construction for signing directed trees. In particular, signatures for paths in the tree are of constant size. This dramatically improves the previous better bound by Gregory Neven (Theoretical Computer Science 396).

Finally we use a similar CRHF to prove that a binary string corresponds to the binary expansion of some other value. Using zero-knowledge techniques we build upon this construction to obtain a protocol for releasing a secret gradually. This tool is then used to fairly exchange Boneh-Boyen short signatures (EUROCRYPT 2004) without relying on a trusted third party.

Acknowledgements

I owe thanks to many people.

I will start with my grandfather, who by his example of hard work, humility and perseverance, taught me the right mindset that lead to a successful and happy life. I would like to thank my mother for her love and for imparting to me the taste for study. My father played a fundamental role in building my self-confidence and my choice to work with computers: He always praised my achievements, especially my small programs when I was a young boy. During high school I was fortunate to be taught by very good professors. I am especially very grateful to professor Georges Lelandais who made me discover that mathematics are not only useful but also fun.

I am indebted to Agustín Villena, my friend and mentor. Agustín taught me numerous useful things and, in particular, helped me to keep an eye on "real" world applications and problems. He also introduced me to the research staff of the *Universidad de Chile* which triggered my application to the PhD program.

I enjoyed a lot my time at the *Universidad de Chile* and I really felt that all the staff was there to help me so that I could develop my research in the best conditions. I would like to thank Angelica Aguirre in particular for reducing all the complexity of administrative tasks down to that of a child's game.

When I arrived in Chile in 2004, professor Marcos Kiwi offered a warm welcome for me to work on my Master's project. I really appreciated his kindness and patience[1]. I am also very grateful to him for introducing me to research. My first contact with the Department of Computer Science of the *Universidad de Chile* occurred through professors Sergio Ochoa and Claudio Gutierrez. Thank you both for giving me this great opportunity of pursuing a PhD! Thank you Claudio also for having your door

[1]I am a bit absent-minded.

always opened and for your inspiring teachings on research methodology. I really enjoyed the courses I took while being a PhD student, especially those by Gonzalo Navarro and Patricio Poblete (algorithms) as well as those of Éric Tanter and Johan Fabry (programming languages). I would like also to thank Jérémy Barbay for his interest in my work, Ren Cerro for helping me to improve the writing up of this thesis and Gregory Neven for his detailed review, in particular the counter example of Figure 4.2.

These long years of work were made more enjoyable thanks to several friends with whom I shared meals, homework or simply had fun: Juan Manuel Barrios, Alex Bórquez, Pablo Burgos, Violeta Chang, Ernesto Castillo, Carlos Gomez, Claudio Gonzalez, Valeria Herskovic, Julio Hurtado, Julio Quinteros, José Manuel Saavedra, Paul Saint Leger, Juan Jose Segura, Rodolfo Toledo, and Carla Vairetti.

I would like to thank my PhD advisor, Alejandro Hevia for helping me become a better researcher. From the very beginning of our relation Alejandro did not treat me as a simple student, but as a colleague, despite my very limited knowledge and experience. He gave me freedom to follow my ideas while at the same time helping me through useful advice, both methodological, and technical. I would like to thank him also for encouraging me to collaborate with other researchers and especially for helping me to improve my writing and public speaking skills. Indeed he spent a lot of time showing me how to design the slides of my presentations and the way in which I could communicate more efficiently. Clearly, there is still a lot of room to improve the way I solve problems and interact with others. However, thanks to Alejandro, I really feel like a part of the "crypto" community.

Finishing my PhD would not have been possible without the support and love of my wife Laura. She gave me the strength to keep going when I thought everything was lost. She helped me taking care of myself, especially encouraging me to sleep enough and exercise. She made huge sacrifices so that I could have enough time and peace of mind to do my research. Thank you my love, I could not have done this without you!

I would like to thank Mrs Sonia Contreras for looking after my son Martín: knowing he was in good hands allowed me to focus on my work without being worried.

And finally thank you Martín, my sweet son, for giving me so much joy and reminding me that every instant in life is an opportunity to learn and grow.

Contents

Chapter 1

Introduction

1.1 Predicate-Preserving Collision-Resistant Hashing

Collision-Resistant Hash Functions (CRHFs) are widely used in the design of cryptographic schemes or protocols to improve efficiency [HW09], avoid interaction [FS86], or build other primitives [BK12]. A common CRHF like SHA-256 has the property of compressing possibly large input strings into small ones. Moreover, because of collision-resistance, in practice, CRHFs are considered injective. This makes them useful constructs to manipulate shorter strings without losing 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) may help to simplify and improve the efficiency of a large class of protocols, especially those where it is required to authenticate structured data.

More concretely, assume that $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^\kappa$ is a CRHF (see Figure 1.1). Let $X = 10001$ be the input and $y = \mathsf{H}(X)$ the output. Assume Bob wants to convince Alice that y is such that the pre-image X starts with the string 100. If H is a common CRHF like the one mentioned above, there are two ways to solve the problem:

- Using recent techniques like the one introduced by Groth at Asiacrypt 2010 [Gro10]: Here the idea is to compute a proof of knowledge of the pre-image X of y and also show that this pre-image satisfies the predicate. This solution is not very practical as it involves reducing the hash function as well as the predicate to boolean circuits which may have millions of gates.

- Giving Alice the pre-image X. In this case, Alice will check that X starts with

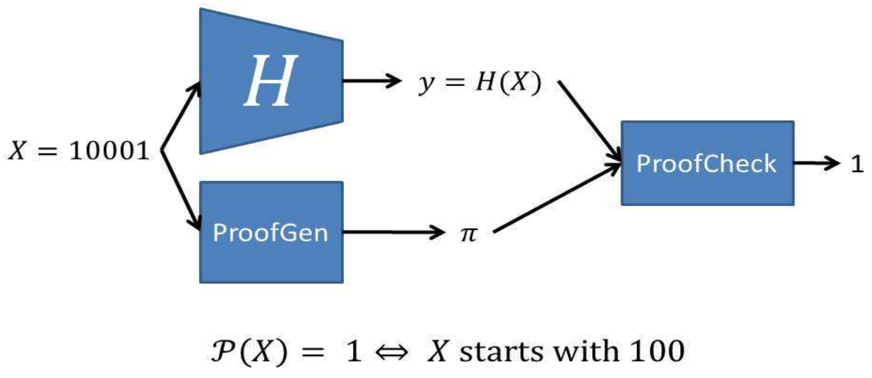


Figure 1.1: **Conceptual view of a predicate-preserving CRHF:** The input X is hashed in order to obtain $y = \mathsf{H}(x)$. The same input is used to compute a proof π through the algorithm $\mathsf{ProofGen}$. Finally one can check that the pre-image X verifies some predicate by running the algorithm $\mathsf{ProofCheck}$ on y and π. Note that the hash function H, and the algorithms $\mathsf{ProofGen}$ and $\mathsf{ProofCheck}$ depend on the predicate $\mathcal{P}$ that must be preserved.

a $z = 100$ and also that $y = \mathsf{H}(X)$. This alternative is not satisfying either, especially when the pre-image is large (several gigabytes for instance) or the rest of the string X must be kept secret.

In this thesis we study CRHF that enable to efficiently prove some predicate about the input X given the output y and some small proof π. In particular, such functions could be such that $|\pi| \ll |X|$. Note however that efficiency is not the only goal: In some cases, as shown in Chapter 7, we may consider hash functions with short inputs (κ bits), especially when we are interested in controlling the information leakage of the proofs.

More generally we consider the following setting:

- $\mathsf{H}_i : \{0,1\}^* \to \{0,1\}^\kappa$ is a CRHF for $1 \leq i \leq q, q \in \mathbb{N}$.
- $\mathcal{P} : (\{0,1\}^*)^q \to \{0,1\}$ is a predicate computable in polynomial-time.

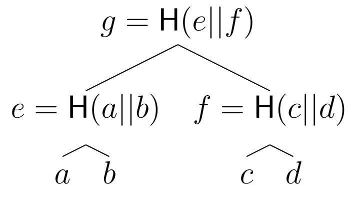


Figure 1.2: **Merkle tree for sequence** (a, b, c, d)**.** The hash function induced by the tree and a CRHF H takes the set (a, b, c, d) as input and returns the root hash value g as output. A proof that a belongs to the set is composed by the nodes containing values (b, f) which are the siblings of the node on the path from a to the root g. Checking the proof consists of computing $e' = \mathsf{H}(a||b)$, then computing $g' = \mathsf{H}(e'||f)$ and finally checking that $g = g'$.

- There exists an efficient (e.g. linear time) algorithm $\mathsf{ProofGen}$ that given $X_1, \ldots, X_q$ such that $\mathsf{H}_i(X_i) = y_i$ outputs a proof π.
- There exists an efficient (e.g. logarithmic or constant time) algorithm $\mathsf{ProofCheck}$ such that if $y_i = \mathsf{H}_i(X_i)$ then $\mathsf{ProofCheck}(y_1, y_2, \ldots, y_q, \pi) = 1$ if, and only if, $\mathcal{P}(X_1, \ldots, X_q) = 1$.

Intuitively, the security of such a scheme has to guarantee that no adversary is able to forge a proof that would invalidate the predicate $\mathcal{P}$ on some inputs $(X_i^*)_{i\in[1..q]}$: That is, it should be hard to compute $\pi^*, X_1^*, \ldots, X_q^*$ such that $y_i = \mathsf{H}_i(X_i^*)$, $\mathsf{ProofCheck}(y_1, \ldots, y_q, \pi^*) = 1$, and $\mathcal{P}(X_1^*, \ldots, X_q^*) = 0$. If for CRHFs H_i there exist algorithms $\mathsf{ProofGen}$ and $\mathsf{ProofCheck}$ associated to a predicate $\mathcal{P}$, then we will say that $(\mathsf{H}_i)_{i\in[1..q]}$ *preserve* the predicate $\mathcal{P}$.

A SIMPLE EXAMPLE: TIME STAMPING SCHEME USING MERKLE TREES. To illustrate the concept previously introduced, let us consider the problem of *time stamping.* The goal of a time stamping scheme is to prove that a document, like a contract between two parties, has been created within a given time round (e.g. a day, an hour). A solution proposed by Benaloh and De Mare [BdM91] (subsequently improved by Bayer and Haber [BH92]) consists of building for each time round a binary tree, also called Merkle tree [Mer87], where the value of a node is computed recursively by taking the hash of both children (see Figure 1.2).

In this scheme the documents are placed at the leaves and the hash value obtained at the root is published. This value stands for the state or representation of the time round when these documents have been submitted. To prove that a document belongs to this time round, one needs to consider the nodes on the path from the leaf to the root and collect the siblings of these nodes. The verification procedure consists of recomputing the root hash value from the authentication path which is formed by the sibling nodes. On a more abstract level, this construction corresponds to a pair of predicates preserving CRHFs:

- The predicate preserved is set membership ("the document belongs to the time round").
- The first hash function for the set is induced by the Merkle tree. In particular, the input is the set of documents (collection of leaves), and the output is the hash value at the root of the tree.
- The second hash function for the element is the identity function.
- The proof is the authentication path that connects the root (hash value of the set) to the leaf (other input of the predicate checking algorithm).

There exist other ways to implement CRHF that preserve the set-membership predicate. Challenging problems arise when we try to perform updates on the set that is hashed. The first part of this thesis (chapter 4 and 5), focuses on the security model and the limitations of such constructions. Then, in chapters 6 and 7, CRHFs that preserve predicates related to strings are described, and it is shown how to apply these hash functions for building an efficient signature scheme for directed trees and a fair exchange protocol for Boneh-Boyen signatures [BB08].

In the following we detail the contributions of this thesis.

1.2 Sets

In chapters 4 and 5 we study CRHFs where the input encodes a set X_1 of values belonging to some universe $U = \{0,1\}^\kappa$. We will denote by 2^U the collection of sets whose elements are in U. These CRHFs are used to preserve the membership and non-membership predicates for an element in $X_2 \in U$. Thus we have $\mathcal{P}_{\in} : 2^U \times U \rightarrow \{0,1\}$ such that for $X_1 \in 2^U, X_2 \in U$, $\mathcal{P}_{\in}(X_1, X_2) = 1 \Leftrightarrow X_2 \in X_1$ and similarly $\mathcal{P}_{\notin}(X_1, X_2) = 1 \Leftrightarrow X_2 \notin X_1$. The hash functions are H_1 (a CRHF) and H_2, the identity function on U. Hash functions $(\mathsf{H}_1, \mathsf{H}_2)$ that preserve such predicates are known in the literature as *cryptographic accumulators* [BdM93]. There exist many applications for this primitive, in particular, accumulators can be used to implement revocation for anonymous-credentials schemes [ATSM09, Ngu05, CKS09, CL02], e-cash protocols [ASM08] and broadcast encryption protocols [GR04]. Cryptographic accumulators can naturally serve to implement authenticated dictionaries [CHKO08, PTT08, GTT08].

STRONG ACCUMULATORS [CHKO08]. In chapter 4, we propose a simple construction based on standard CRHF that handles insertions and deletions of elements and that does not require a trusted third party[1]. Our main contribution is to strengthen the standard security definition for accumulators, by allowing the adversary to corrupt the owner of the set. We show that our construction, though not optimal, is efficient enough to solve a multi-party computational problem of practical relevance, the *e-Invoice Factoring Problem*.

BATCH UPDATE IS IMPOSSIBLE [CH10]. In their survey on cryptographic accumulators, Fazio and Nicolisi asked whether it was possible to refresh all the proofs for each element after several updates to the set using a single short value. Such an accumulator scheme would be said to have the *batch update* property. In Chapter 5, we show that batch update for cryptographic accumulators is impossible.

We then tried to solve the following open problem:

Is it possible to build an optimal authenticated dictionary where the time to compute a proof would be logarithmic in the size of the set, while at the same time keeping proof of (non)membership of constant size?

[1]A similar idea appeared previously in a work by Buldas et al. (CCS '00) [BLL00]. This work, however, focused on static sets, and did not propose a formalization nor a security proof for dynamic operations.

While this question was not answered, by trying to use other techniques (transitive signatures) it became clear that the idea of accumulators (hashing a set and then proving (non)membership) could be generalized to other data structures. Somehow surprisingly, we discovered some connection between transitive signatures and authenticated dictionaries:

1. As a partial result for the problem of optimal data authentication (see section 3.4.3) we showed [Cam11] that optimal authenticated dictionaries can be built using transitive signatures for directed graphs. However, transitive signatures are only known to exist for undirected graphs. The case of directed graph is still open and would imply the existence of groups with infeasible inversion [Hoh03].

2. Using cryptographic accumulators, it is possible to improve the size of the signatures in the transitive signature scheme for *directed trees* proposed by Neven [Nev08]. This straightforward application of accumulators is described in Section 3.4.2.

Our study on transitive signatures lead us to investigate CRHFs that enable to the design of efficient transitive signature schemes for directed trees. These functions are in fact predicate preserving CRHFs where the predicate is the lexicographical order between two strings. Thus, the second class of predicates we analyze in this work is related to strings.

1.3 Strings

Short Signatures for Directed Trees [CH12]. In Chapter 6 we show how to build a practical transitive signature scheme for directed trees improving Neven's result [Nev08].

Our scheme is practical and is the most efficient one to the best of our knowledge. Our scheme also provides a flexible tradeoff between signature computation and verification. More precisely, our construction is such that, for any $\lambda \geq 1$, (a) signing or verifying an edge signature requires $O(\lambda)$ cryptographic operations, and

(b) computing (without the secret key) an edge signature in the transitive closure of the tree requires $O(\lambda(\frac{N}{\kappa})^{1/\lambda})$ cryptographic operations, where κ is the security parameter and N is the size of the tree. To achieve this goal, we design an efficient CRHF that preserves the following predicate: $\mathcal{P}_{\texttt{CommonPrefix}} : \{0,1\}^* \times \{0,1\}^* \times \mathbb{N} \rightarrow \{0,1\}$ with $\mathcal{P}_{\texttt{CommonPrefix}}(X_1, X_2, i) = 1 \Leftrightarrow X_1[1..i] = X_2[1..i]$ where $X[1..i]$ denote the first i symbols of X. This predicate is interesting in itself as it can be used to build other predicates, for example, string comparison with respect to the lexicographical order. In Section 6.2, we show how to build a CRHF that preserves the predicate $\mathcal{P}_{\prec}(X_1, X_2) = 1 \Leftrightarrow X_1 \prec X_2$, where $\prec$ is the lexicographical order for strings.

Fair Exchange of Short Signatures without Trusted Third Party [Cam13]. In Chapter 7 we show how to adapt the previously mentioned CRHF so that the following predicate is preserved: $\mathcal{P}_{\texttt{Equiv}}(X_1, X_2) = 1 \Leftrightarrow X_2 = \sum_{i=1}^{\kappa} X_1[i]2^{i-1}$ where $X[i]$ is the i^{th} bit of the binary string X and κ is the security parameter. This construction enables the design of a fair-exchange protocol for Boneh-Boyen short signatures [BB04b, BB08] that does not rely on a trusted third party. To the best of our knowledge, this protocol is the first of its sort: The number of rounds is $\kappa + 5$, the communication complexity is $16\kappa^2 + 12\kappa$ bits, and the protocol requires a linear number of cryptographic operations.

Moreover, we introduce a new non-interactive zero-knowledge (NIZK) argument to prove that a commitment encodes a bit vector. We combine this argument with another NIZK argument that allows proving a commitment to a bit vector corresponds to the binary decomposition of some value θ which is hidden as the discrete logarithm of some other group element. We think these arguments may lead to other useful applications. We also revisit the notion of partial fairness [GK10, GMPY06] by introducing a new and simple definition that captures the fact that an adversary while having more computing power than the honest participant will not have a significant advantage in obtaining the expected signature.

Publications. The results presented in this thesis yielded the following publications:

- Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong

Accumulators from Collision-Resistant Hashing. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security Conference*, volume 5222 of *LNCS*, pages 471–486. Springer Berlin / Heidelberg, September 2008.

- Philippe Camacho and Alejandro Hevia. On the Impossibility of Batch Update for Cryptographic Accumulators. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT*, volume 6212 of *LNCS*, pages 178–188. Springer Berlin / Heidelberg, 2010.
- Philippe Camacho. Optimal Data Authentication from Directed Transitive Signatures. `http://eprint.iacr.org/2011/439` (last access 5/12/2012), 2011.
- Philippe Camacho and Alejandro Hevia. Short Transitive Signatures for Directed Trees. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *LNCS*, pages 35–50. Springer Berlin / Heidelberg, 2012.
- Philippe Camacho. Fair Exchange of Short Signatures without Trusted Third Party. In Ed Dawson, editor, *CT-RSA*, volume 7779 of *LNCS*, pages 34–49. Springer Berlin / Heidelberg, 2013.

Overview. We introduce notations and fundamental cryptographic concepts in Chapter 2. In Chapter 3 we introduce the problem of designing a secure and efficient authenticated dictionary and review related techniques. Our results related to cryptographic accumulators are introduced in chapters 4 and 5. Chapter 6 is dedicated to our scheme for transitive signatures for directed trees. We show how to apply the CRHFs that preserve the predicate $\mathcal{P}_{\texttt{Equiv}}$ to the problem of fair exchange of digital signatures in Chapter 7.

Chapter 2

Preliminaries

Contents

2.1 Notations

Let $\kappa \in \mathbb{N}$ be the security parameter. We denote by 1^κ the unary string with κ ones. An algorithm is called PPT if it is probabilistic and runs in polynomial time in κ.

Let $\mathsf{neg} : \mathbb{N} \to \mathbb{N}$ denote a negligible function, that is, for every polynomial $q(\cdot)$ and any large enough integer κ, $\mathsf{neg}(\kappa) < 1/q(\kappa)$.

We denote $x \leftarrow v$ the assignation of value v to variable x. We write $x \overset{R}{\leftarrow} X$ for an element x chosen uniformly at random from a set X. When R is a randomized algorithm, $a \overset{R}{\leftarrow} R()$ denotes the process of choosing a according to the probability distribution induced by R. We also denote by $\langle R() \rangle$ the set of all possible values a returned by R with positive probability. Let $Alg_{\mathsf{H}}(\cdot)$ be a (PPT) algorithm that computes a function H, then when $Alg_{\mathsf{H}}(\cdot)$ is fed with input x and returns y, we write $x = \mathsf{H}^{-1}(y)$.

For $m, n \in \mathbb{N}$ with $m < n$, $[m..n]$ denotes the set of integers $\{m, m+1, ..., n-1, n\}$ and $[n]$ stands for the set of integers $\{1, ..., n\}$. A vector of n components and values v_i is denoted $\vec{v} = (v_i)_{i\in[n]}$. If the vector contains elements of $\mathbb{Z}_p$ we may also represent its components as an array $B[\cdot] = (B[1], B[2], ..., B[n])$. For any $\theta \in \mathbb{Z}_p$, where p is a prime of κ bits, we denote by $\theta[\cdot]$ the binary decomposition (vector) of θ. That is $\theta[\cdot] = (\theta[1], ..., \theta[\kappa])$ and in particular $\theta = \sum_{i\in[\kappa]} \theta[i]2^{i-1}$. When $P(\cdot)$ is a polynomial with coefficients in $\mathbb{Z}_p$, $P[]$ will denote the vector of its coefficients. This yields the expression $P(X) = \sum_{i \in [d+1]} P[i]X^{i-1}$ where $d = deg(P(\cdot))$ is the degree of polynomial $P(\cdot)$.

Let $x \in \mathbb{N}$ be an integer such that $0 \leq x < 2^\kappa$, we denote by $\langle x \rangle_\kappa$ its encoding as a κ-bit string. In this thesis log refers to the logarithm in base two function.

Strings. A string S of size $m = |S|$ is a sequence of symbols $S[1], S[2], ..., S[m]$ from some alphabet Σ. We assume there is a total order relation $<$ over Σ. If $m = 0$ 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). The concatenation operator on strings is denoted as $||$. 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 *longest common prefix* of A and B if C is a common-prefix of A and B, but $C||\sigma$ is not a common prefix of A and B for any symbol $\sigma \in \Sigma$. We denote by $<$ the (standard) lexicographical order on $\{0,1\}^*$. We single out \$ as a special symbol that is used only to mark the end of a string, and satisfies $0 < \$ < 1$. We define the *extended* lexicographical order $\prec$ on $\{0,1\}^*$ as following: Let $X, Y \in \{0,1\}^*$ and $X' = X||\$, Y' = Y||\$$ strings obtained by appending the end marker to X, Y. We say that $X \prec Y$ if, and only if, $X' < Y'$.

Graphs. $G = (V, E)$ denotes a graph G with set of vertices V and set of edges E. If $A \in V$ is a vertex of G, then A^* is the transitive closure of A, that is the set of vertices: $\{Q \in V :$ there exists a path from A to Q $\}$. Similarly, the transitive closure of a graph G, denoted G^*, is the union of the transitive closures of every vertex of G. A path from vertex R to vertex H is written $R \rightarrow H$.

Trees. In the case where $\mathcal{T}$ is a *binary tree*, we consider the following conventions. The root node of a tree $\mathcal{T}$ will be denoted *root*$(\mathcal{T})$. The left subtree (respectively right subtree) rooted at the left (respectively right) child node of $\mathcal{T}$ will be denoted *Left*$(\mathcal{T})$ (respectively *Right*$(\mathcal{T})$). The node *root*$(\mathcal{T})$ is said to be the *parent* of *Left*$(\mathcal{T})$ and *Right*$(\mathcal{T})$. Each node N of $\mathcal{T}$ will be labeled by a string henceforth denoted $\ell(\mathsf{N})$. Sometimes we identify the tree $\mathcal{T}$ with its root $\mathsf{N} = root(\mathcal{T})$ and we write $\ell(\mathcal{T})$

to denote $\ell(root(\mathcal{T}))$. We denote by Nil an empty node. As usual, a leaf of $\mathcal{T}$ corresponds to a node of $\mathcal{T}$ that has no children (or two empty nodes as children). If $\mathcal{T}$ consists in only one node, then we say that $\mathcal{T}$ has depth 0 and denote it as $d(\mathcal{T}) = 0$. Otherwise, let $d(\mathcal{T}) = 1 + \max\{d(Left(\mathcal{T})), d(Right(\mathcal{T}))\}$. A tree $\mathcal{T}$ is *balanced* if, and only if, for every node N we have $|d(Left(\mathsf{N})) - d(Right(\mathsf{N}))| \leq 1$. It is a well known fact that a balanced tree with N nodes has maximum depth $O(\log(N))$.

Let R be the root of the tree. If $b_i \in \{0,1\}$, and $0 \leq i \leq l \leq d-1$, then $R_{b_0 b_1 \ldots b_l}$ denotes the node reachable from R by taking the path $b_0 b_1 \ldots b_l$, where 0 means left child and 1 means right child. By $R_{[b_0 b_1 \ldots b_l]}$ we denote the path from R to $R_{b_0 b_1 \ldots b_l}$ formed by the nodes $R, R_{b_0}, R_{b_0 b_1}, \ldots, R_{b_0 b_1 \ldots b_{l-1}}, R_{b_0 b_1 \ldots b_l}$. If we index the leaves from left to right starting from 0 until $N - 1 = 2^d - 1$ we note that L_i, the i^{th} leaf of the tree with root R, is such that $L_i = R_{b_0 b_1 \ldots b_{d-1}}$ where $i = \Sigma_{j=0}^{d-1} b_{d-1-j} 2^j$. In other words, $(b_0, \ldots, b_{d-1})$ is the binary decomposition of i where b_0 is the bit of strongest weight.

Our construction for transitive signatures (see Chapter 6) makes use of binary *tries* [Fre60], a type of binary tree, which associates labels to each node as follows. First, for each node, the edge going to the left (resp. right) child is tagged 0 (resp. 1). Then, the label for the node is obtained by concatenating the tags on the edges in a path from the root to the node. This way, any node x in the trie $\mathcal{B}$ can be identified by its associated label $X \in \{0,1\}^*$. Given some label X, we denote by $node(X)$ the corresponding node in $\mathcal{B}$ if it exists. We say a node $x' \in \mathcal{B}$ is a *descendant* of x if x' belongs to the subtree rooted at x or equivalently if there is a path from x to x'. The *lowest common ancestor* of two nodes x, y of $\mathcal{B}$ is the node z such that x and y belong to the subtree rooted at z, and for any child z' of z, x or y is not a descendant of z'. In Chapter 6) we will consider two types of depth-first tree traversals:

- Pre-order: recursively (1) visit the root, (2) traverse the left subtree, and (3) traverse the right subtree.

- Post-order: recursively (1) traverse the left subtree, (2) traverse the right subtree, and (3) visit the root.

2.2 Collision-Resistant Hashing

We recall here some standard definitions for collision-resistance.

Definition 1 *Let κ be the security parameter. A hash-function family is a function $\mathcal{H}_\kappa : K \times U \to Y$ where $K = \{0,1\}^\kappa$, $Y = \{0,1\}^\kappa$ and $U = \{0,1\}^*$.*

The family of functions $\mathcal{H}_\kappa$ is said to be collision-resistant ($CRHF$) if, for $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\kappa$ uniformly chosen at random in $\mathcal{H}_\kappa$, any computationally bounded adversary can not find two different messages m and m' such that $\mathsf{H}(m) = \mathsf{H}(m)'$ except with negligible probability.

Definition 2 *(Collision-Resistance) Let $\mathcal{H}_\kappa : K \times X \to Y$ be a hash-function family. Then $\mathcal{H}_\kappa$ is collision-resistant if, and only if, for any PPT algorithm $\mathcal{A}$ we have:*

$$\Pr\left[k \overset{R}{\leftarrow} K; m, m' \leftarrow \mathcal{A}(1^\kappa, k) : m \neq m' \wedge \mathsf{H}(m) = \mathsf{H}(m')\right] = \mathsf{neg}(\kappa)$$

where $\mathsf{H} := \mathcal{H}_\kappa(k, \cdot)$ and the probability is taken over K and the random coins of $\mathcal{A}$.

When the context is clear we will write $\mathcal{H}$ instead of $\mathcal{H}_\kappa$.

2.3 Bilinear Maps

In his seminal paper [Jou00] Joux showed that the Weil and Tate pairing (bilinear maps) could be used not only for cryptanalytic purposes but also to design efficient cryptosystems. Since this breakthrough, many works followed. The main reason is that the bilinearity property of these functions enables the construction of cryptographic schemes or protocols that are more efficient when compared to their implementation with more standard techniques like RSA, or simply cannot be built (considering actual knowledge) without this new primitive.

In this section we introduce the abstract definition for bilinear maps, and the related computational assumptions we rely on for our constructions described in Chapter 6 and Chapter 7.

2.3.1 Definition

Let $\mathbb{G}$ and $\mathbb{G}_T$, 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 $\hat{\mathbb{G}}, \hat{\mathbb{G}_T}$ is the representation of the corresponding 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, we will not distinguish between $\mathbb{G}, \mathbb{G}_T, e$, and $\hat{\mathbb{G}}, \hat{\mathbb{G}_T}, \hat{e}$.

2.3.2 Computational assumptions

Let $N \in \mathbb{N}$. For the following computational assumptions, the common public parameter is $\mathsf{PP} = \langle (p, \mathbb{G}, \mathbb{G}_T, e, g), (g_0, g_1, g_2, \cdots, g_N) \rangle$ where s is chosen randomly in $\mathbb{Z}_p$ and $g_i = g^{s^i}$ for $i \in [0..N]$.

Definition 3 ***N-Diffie-Hellman Inversion (N-DHI) assumption****, [MSK02]. The N-Diffie-Hellman Inversion problem consists in computing $g^{\frac{1}{s}}$ given PP. We say the N-DHI assumption holds if for any PPT adversary $\mathcal{A}$ we have*

$$Adv^{N\text{-}DHI}(\mathcal{A}, \kappa, N) = \Pr\left[g^{\frac{1}{s}} \leftarrow \mathcal{A}(1^\kappa, \mathsf{PP})\right] = \mathsf{neg}(\kappa)$$

The following bilinear variant of the previous assumption was introduced in [BB04a]:

Definition 4 ***N-Bilinear Diffie-Hellman Inversion assumption (N-BDHI), [BB04a].*** *The N- Bilinear Diffie-Hellman Inversion problem consists in computing $e(g, g)^{\frac{1}{s}}$ given PP. We say the N-BDHI assumption holds if for any PPT adversary $\mathcal{A}$ we have*

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

Definition 5 ***N*-Strong Diffie-Hellman assumption (*N*-SDH)**, *[BB08]. The N-Strong Diffie-Hellman (N-SDH) problem consists in computing $(c, g^{\frac{1}{s+c}})$ given PP. We say the N-SDH assumption holds if for any PPT adversary $\mathcal{A}$ we have*

$$Adv^{N\text{-}SDH}(\mathcal{A}, \kappa, N) = \Pr\left[\,(c, g^{\frac{1}{s+c}}) \leftarrow \mathcal{A}(1^\kappa, \mathsf{PP})\,\right] = \mathsf{neg}(\kappa)$$

As mentioned in [BB08], the N-SDH assumption is equivalent to the N-DHI assumption when c is fixed.

The following assumption can be considered as a particular case of the *poly*-Diffie-Hellman assumption [KZG10], or a generalization of the N+1-Exponent assumption introduced in [ZSNS04].

Definition 6 ***N*+*i*-Diffie-Hellman Exponent(*N*+*i*-DHE) assumption.** *The N+i-Diffie-Hellman Exponent problem consists in computing $g^{s^{N+i}}$, for $1 \leq i \leq N$ given PP. We say the N+i-DHE assumption holds if for any PPT adversary $\mathcal{A}$ we have*

$$Adv^{N+i\text{-}DHE}(\mathcal{A}, \kappa, N) = \Pr\left[\,g^{s^{N+i}} \leftarrow \mathcal{A}(1^\kappa, \mathsf{PP})\,\right] = \mathsf{neg}(\kappa)$$

In [ZSNS04], the N-DHI was shown to be equivalent to the N+1-Exponent assumption (N+1-DHE). We prove here the following implication following the idea of [ZSNS04].

Proposition 1 *N-BDHI $\Rightarrow$ N+i-DHE.*

Proof. Let $\mathcal{A}$ be a PPT adversary that breaks the N+i-DHE assumption. We build the following adversary $\mathcal{B}$. Algorithm $\mathcal{B}$ receives the challenge tuple $g, g^s, g^{s^2}, ...,$ g^{s^N} and then it sets $h = g^{s^N}$. Then if we consider $t = \frac{1}{s}$, we have: $(h, h^t, h^{t^2}, \cdots, h^{t^N}) = (g^{s^N}, g^{s^{N-1}}, g^{s^{N-2}}, \cdots, g)$. $\mathcal{B}$ sends the tuple $(h, h^t, h^{t^2}, \cdots, h^{t^N})$ to $\mathcal{A}$ who outputs $h' = h^{t^{N+i}}$ where $1 \leq i \leq N$. We have that $h' = h^{t^{N+i}} = g^{s^{N-N-i}} = g^{\frac{1}{s^i}}$. Finally, $\mathcal{B}$ outputs $e(h', g^{s^{i-1}}) = e(g,g)^{s^{-i+i-1}} = e(g,g)^{\frac{1}{s}}$ ■

2.4 Digital Signatures

2.4.1 Standard Digital Signatures

We denote by $\texttt{SSig} = (\mathsf{SKG}, \mathsf{SSig}, \mathsf{SVf})$ a standard signature scheme. A pair of private/public keys is created by running $\mathsf{SKG}(1^\kappa)$. Given a message $m \in \{0,1\}^*$, a signature on m under pk is $\sigma_m = \mathsf{SSig}(sk, m)$. A signature σ on m is deemed valid if and only if $\mathsf{SVf}(pk, m, \sigma)$ returns valid. Regarding security, we use the standard notion of existential unforgeability under chosen message attack [GMR88].

2.4.2 Boneh and Boyen Signature Scheme [BB08]

We recall here the short signature scheme [BB08] introduced by Boneh and Boyen. The setup algorithm $\mathsf{BMGen}(1^\kappa)$ generates the public parameters of the scheme $(p, \mathbb{G}, \mathbb{G}_T, e, g)$[1]. The key generation algorithm $\mathsf{SKG}(1^\kappa)$ selects random integers $x, y \overset{R}{\leftarrow} \mathbb{Z}_p$ and sets $u = g^x$ and $v = g^y$. The secret key is $sk = (g, x, y)$ and the public key is $pk = (g, u, v)$. Given a message m and sk, the signing algorithm $\mathsf{SSig}(sk, m)$ works as follows. It selects $r_\sigma \overset{R}{\leftarrow} \mathbb{Z}_p$ such that $r_\sigma - (x+m)/y \neq 0 \bmod p$ and return the (randomized) signature $\sigma = (g^{\frac{1}{x+m+yr_\sigma}}, r_\sigma) = (\sigma', r_\sigma)$. Finally, in order to verify a signature σ on message m, relative to the public key pk, the algorithm $\mathsf{SVf}(pk, m, \sigma)$ consists in checking that $e(\sigma', ug^m v^{r_\sigma}) = e(g, g)$.

The scheme is secure in the standard model under the N-SDH assumption [BB08].

2.5 Trapdoor commitments

Let $\mathcal{R}$ be the space of randomness, $\mathcal{C}$ the set where commitments belong and $\mathcal{M}$, the space for messages. A trapdoor commitment scheme is composed of the following algorithms: $\mathcal{K}$, Commit, Verify, $\mathsf{TCommit}$, TOpen. Here $\mathcal{K}(1^\kappa)$ is a randomized algorithm that generates the common reference string CRS and an associated trapdoor τ. We denote by $\mathsf{Commit}(\mathsf{CRS}, m, r)$ a deterministic algorithm that computes

[1] We use symmetric bilinear map for the sake of exposition.

a commitment C to value $m \in \mathcal{M}$ using $r \in \mathcal{R}$. Algorithm $\mathsf{Verify}(\mathsf{CRS}, C, m, r)$ returns valid if and only if $C = \mathsf{Commit}(\mathsf{CRS}, m, r)$, otherwise it returns $\perp$. We will sometime use the notation $\mathtt{open}$ to denote the opening of the commitment C, that is $\mathtt{open} = (m, r)$. $\mathsf{TCommit}(\tau)$ is a randomized algorithm that returns an equivocal commitment C along with an equivocation key ek given the trapdoor τ. $\mathsf{TOpen}(ek, C, m)$ is a deterministic algorithm that returns the randomness $r \in \mathcal{R}$ of C with respect to message $m \in \mathcal{M}$. In order to simplify the notation, in the following the common reference string CRS will be an implicit argument of algorithms Commit and Verify.

We say the commitment scheme is *computationally binding* if for all non-uniform stateful PPT adversary $\mathcal{A}$ we have

$$\Pr\left[\begin{array}{c}(\mathsf{CRS}, \tau) \leftarrow \mathcal{K}(1^\kappa); (m_0, m_1, r_0, r_1) \leftarrow \mathcal{A}(\mathsf{CRS}) : \\ m_0 \neq m_1 \wedge \mathsf{Commit}(m_0, r_0) = \mathsf{Commit}(m_1, r_1)\end{array}\right] = \mathsf{neg}(\kappa)$$

The scheme is said to be *perfectly hiding* if for all non-uniform stateful adversaries $\mathcal{A}$ we have

$$\begin{aligned}&\Pr\left[\,(\mathsf{CRS}, \tau) \leftarrow \mathcal{K}(1^\kappa); (m_0, m_1) \leftarrow \mathcal{A}(\mathsf{CRS}); C \leftarrow \mathsf{Commit}(m_0, r_0) : \mathcal{A}(C) = 1\,\right]\\ =\ &\Pr\left[\,(\mathsf{CRS}, \tau) \leftarrow \mathcal{K}(1^\kappa); (m_0, m_1) \leftarrow \mathcal{A}(\mathsf{CRS}); C \leftarrow \mathsf{Commit}(m_1, r_1) : \mathcal{A}(C) = 1\,\right]\end{aligned}$$

A commitment scheme is *perfectly trapdoor* if, for any stateful PPT adversary, we have

$$\Pr\left[\begin{array}{c}(\mathsf{CRS}, \tau) \leftarrow \mathcal{K}(1^\kappa); \\ m \leftarrow \mathcal{A}(\mathsf{CRS}); \\ r \overset{R}{\leftarrow} \mathcal{R}; \\ C \leftarrow \mathsf{Commit}(m, r) : \\ \mathcal{A}(m, r) = 1\end{array}\right] = \Pr\left[\begin{array}{c}(\mathsf{CRS}, \tau) \leftarrow \mathcal{K}(1^\kappa); \\ m \leftarrow \mathcal{A}(\mathsf{CRS}); \\ (C, ek) \leftarrow \mathsf{TCommit}(\tau); \\ r \leftarrow \mathsf{TOpen}(ek, C, m) : \\ \mathcal{A}(m, r) = 1\end{array}\right]$$

As a commitment is perfectly indistinguishable from an equivocal commitment, a perfect trapdoor commitment scheme is also perfectly hiding.

Our construction relies on a slight variant of the Pedersen[2] commitment scheme [Ped91] which we recall here. Let $\mathbb{G}$ be a cyclic group of prime order $p \in \mathbb{N}$. We

[2]Note that this kind commitment was introduced earlier in [CDvdG87] (see page 98).

consider the common reference string composed by $g \in \mathbb{G}$ and $h \in \mathbb{G}$ where g, h are chosen randomly and the discrete logarithm s of h in base g remains secret. To commit to a message $m \in \mathbb{Z}_p$ with randomness $r \in \mathbb{Z}_p$ we compute[3] $\mathsf{Commit}(m, r) = g^r h^m$. We denote by $\mathtt{open} = (m, r)$ the opening of the commitment. As shown in [Ped91], this scheme is perfectly hiding and computationally binding, under the assumption that computing the discrete logarithm in $\mathbb{G}$ is hard.

2.6 Non-interactive Zero-Knowledge

2.6.1 Arguments

In this section we follow the notation of [Gro10]. We are interested in statements that are efficiently verifiable. Let R be a NP relation such that $(C, w) \in R$ means the statement is true and this can be verified with the witness w. We will consider R_N, the subset of R where the statements are of size $N = \kappa^{O(1)}$. For relation R we define a non-interactive argument in the following way. An algorithm $\mathsf{KeyGen}(1^\kappa, N)$ generates the common reference string CRS. Then the prover $\mathcal{P}$ given as input (CRS, C, w), checks first that $(C, w) \in R_N$. If this is not the case it outputs $\perp$. Otherwise it outputs an argument π. The verifier $\mathcal{V}$ using CRS, C and π returns valid in case it accepts the argument and $\perp$ otherwise.

In our case, C will be a commitment and w its opening (the message and the randomness). We will consider non-interactive zero-knowledge (NIZK) argument (proof) systems $(\mathsf{KeyGen}, \mathcal{P}, \mathcal{V})$ for the relation R_N with the following properties.

Perfect Completeness. The argument is perfectly complete if an honest prover can convince a honest verifier with probability 1 in case the statement is true. For any PPT adversary $\mathcal{A}$ we have

$$\Pr\left[\begin{array}{c} \mathsf{CRS} \leftarrow \mathsf{KeyGen}(1^\kappa, N); (C, w) \leftarrow \mathcal{A}(\mathsf{CRS}); \pi \leftarrow \mathcal{P}(\mathsf{CRS}, C, w) : \\ \mathcal{V}(\mathsf{CRS}, C, \pi) = \mathsf{valid} \land (C, w) \in R_N \end{array}\right] = 1$$

[3]Note that we change a bit the convention as the message is "encoded" into the exponent of h, instead that of g.

Computational Soundness. The argument is said to be sound if no adversary can convince a verifier of a false statement. For any PPT adversary $\mathcal{A}$ we have

$$\Pr\left[\begin{array}{c}\mathsf{CRS} \leftarrow \mathsf{KeyGen}(1^\kappa, N); (C, \pi) \leftarrow \mathcal{A}(\mathsf{CRS}) : \\ \mathcal{V}(\mathsf{CRS}, C, \pi) = \mathsf{valid} \wedge \nexists w : (C, w) \in R_N\end{array}\right] = \mathsf{neg}(\kappa)$$

Perfect witness-indistinguishability. The argument is said to be perfectly witness-indistinguishable if the verifier does not learn which witness was used by the prover in order to produce the proof. For all stateful interactive PPT adversaries $\mathcal{A}$ we have

$$\begin{aligned} & \Pr\left[\begin{array}{c}\mathsf{CRS} \leftarrow \mathsf{KeyGen}(1^\kappa, N); (C, w_1, w_2) \leftarrow \mathcal{A}(\mathsf{CRS}); \\ \pi \leftarrow \mathcal{P}(\mathsf{CRS}, C, w_1) : \\ ((C, w_1), (C, w_2)) \in R_N^2 \wedge \mathcal{A}(\pi) = 1\end{array}\right] \\ = \; & \Pr\left[\begin{array}{c}\mathsf{CRS} \leftarrow \mathsf{KeyGen}(1^\kappa, N); (C, w_1, w_2) \leftarrow \mathcal{A}(\mathsf{CRS}); \\ \pi \leftarrow \mathcal{P}(\mathsf{CRS}, C, w_2) : \\ ((C, w_1), (C, w_2)) \in R_N^2 \wedge \mathcal{A}(\pi) = 1\end{array}\right]\end{aligned}$$

Note that in case there is only one valid witness w for some statement C, then the argument becomes perfectly witness-indistinguishable.

Perfect Zero-Knowledge. We say an argument is zero-knowledge if the verifier learns nothing but the truth of the statement. To formalize this idea we consider two simulators S_1, S_2 such that S_1 generates the CRS and the trapdoor τ. The simulator S_2 uses the common reference string CRS, the statement C and the trapdoor τ to output a simulated argument π. The argument is said to be perfect zero-knowledge if for any stateful interactive PPT adversary $\mathcal{A}$ we have

$$\Pr\left[\begin{array}{c}\mathsf{CRS} \leftarrow \mathsf{KeyGen}(1^\kappa, N); \\ (C, w) \leftarrow \mathcal{A}(\mathsf{CRS}); \\ \pi \leftarrow \mathcal{P}(\mathsf{CRS}, C, w) : \\ (C, w) \in R_N \wedge \mathcal{A}(\pi) = 1\end{array}\right] = \Pr\left[\begin{array}{c}(\mathsf{CRS}, \tau) \leftarrow S_1(1^\kappa, N); \\ (C, w) \leftarrow \mathcal{A}(\mathsf{CRS}); \\ \pi \leftarrow S_2(\mathsf{CRS}, C, \tau) : \\ (C, w) \in R_N \wedge \mathcal{A}(\pi) = 1\end{array}\right]$$

2.6.2 Proofs of Knowledge

Our protocol for fair exchange introduced in Chapter 7 uses zero-knowledge proofs of knowledge (ZKPoK) relative to bit commitments. In order to simplify the description of the fair exchange protocol we will use non-interactive zero-knowlege proofs of knowledge. We note however that interactive ZKPoK would work as well, though adding two rounds to our protocol. The most popular way to implement non-interactive ZKPoK protocols is by using the Fiat-Shamir heuristic [FS86], trading non-interaction for a security proof relying on the random oracle model.

Let $\mathbb{G}$ be a cyclic group of prime order p where the discrete logarithm is hard. Let $\mathsf{H} : \mathbb{G} \to \mathbb{Z}_p$ be a randomly chosen function from a CRHF. Let g, h be two random generators of $\mathbb{G}$ such that the discrete logarithm of h in base g is unknown.

We will need a ZKPoK of the discrete logarithm θ of some public value $D = g^\theta$. Following the notation of [CS97], we have that the proof of knowledge for the discrete logarithm θ of D in base g is $PK\{\theta : g^\theta\} = (c = \mathsf{H}(g^r), z = r - c\theta)$, where $r \overset{R}{\leftarrow} \mathbb{Z}_p$. The verifier checks that $c = \mathsf{H}(D^c g^z)$. We will also use the following ZKPoK that convinces a verifier that the prover knows the representation of a commitment $C = g^\alpha h^\beta$ in base (g, h) where $\alpha, \beta \in \mathbb{Z}_p$. $PK\{(\alpha, \beta) : C = g^\alpha h^\beta\} = (c = \mathsf{H}(g^{r_1} h^{r_2}), z_1 = r_1 - c\alpha, z_2 = r_2 - c\beta)$ where $r_1, r_2 \overset{R}{\leftarrow} \mathbb{Z}_p$. The verifier checks that $c = \mathsf{H}(C^c g^{z_1} h^{z_2})$.

An alternative to the Fiat-Shamir heuristic is the use of non-black-box assumptions that allow to look inside the adversary and extract some value that it “must” already know, in order to perform some specific computation. An example of such assumption is the one proposed by Groth[Gro10] called the N- power knowledge of exponent assumption (N-PKE).

Definition 7 (N-PKE assumption,[Gro10]) *Let* PP *be the public parameter computed as follows:* $\mathsf{PP} = \langle (p, \mathbb{G}, \mathbb{G}_T, e, g), g^s, \cdots, g^{s^N}, g^\alpha, g^{\alpha s}, \cdots, g^{\alpha s^N} \rangle$ *where* $(p, \mathbb{G}, \mathbb{G}_T, e, g)$ *and* s *are computed as described in Section 2.3.2 and* $\alpha \overset{R}{\leftarrow} \mathbb{Z}_p$. *We say the* $N(\kappa)$*-PKE assumption holds if, for every non-uniform PPT adversary* $\mathcal{A}$*, there exists a non-uniform PPT extractor* $\chi_\mathcal{A}$ *such that*

$$\Pr\left[(c, \hat{c}; a_0, \cdots, a_N) \leftarrow (\mathcal{A}||\chi_\mathcal{A})(\mathsf{PP}) : \hat{c} = c^\alpha \wedge c \neq \prod_{i=0}^{N} g^{a_i s^i}\right] = \mathsf{neg}(\kappa)$$

where $(y; z) \leftarrow (\mathcal{A}||\chi_{\mathcal{A}})(x)$ *means that adversary* $\mathcal{A}$ *returns* y *on input* x *and* $\chi_{\mathcal{A}}$ *returns* z *on input* x *and the random tape of* $\mathcal{A}$*.*

The intuition of the definition is as follows: If the adversary is able to compute c^α without knowing α and s, then it *must* know the representation of c in base $(g, g^s, \cdots, g^{s^N})$, that is the vector $(a_0, \cdots, a_N)$ such that $c = \prod_{i=0}^{N} g^{a_i s^i}$. Note that while this assumption is not falsifiable [Nao03] (i.e. no evidence that it is false can be exhibited, like in the case of standard computational assumptions), this assumption does not rely on the random-oracle heuristic.

Groth proposes in the same work [Gro10] a commitment scheme, called *knowledge commitment* that, as its name suggests, is extractable. The idea to commit to a vector of values $a_1, \cdots, a_N \in \mathbb{Z}_p$ is to compute values $c = g^r \prod_{i=1}^{N} g_i^{a_i}$ and $\hat{c} = \hat{g}^r \prod_{i=1}^{N} \hat{g}_i^{a_i}$ where r is the randomness of the commitment, $\hat{g} = g^\alpha$, and for all $i : 1 \leq i \leq N$, $g_i = g^{s^i}$ and $\hat{g}_i = g_i^\alpha$ with $\alpha \in \mathbb{Z}_p$ random. The extractability of the commitment follows directly from the N-PKE assumption. Moreover, by checking that $e(c, \hat{g}) = e(\hat{c}, g)$, it can be verified that the commitment $(c, \hat{c})$ has been computed correctly. While we do not describe our construction of Chapter 7 using knowledge commitments, we note it could be easily adapted to rely on Groth's scheme.

Chapter 3

Data authentication

Contents

Suppose that some organization needs to replicate its database in order to improve the availability of the data. As the organization has no control over the replicas, there is no guarantee that the data provided by the delegated servers is authentic. We consider the following scenario, also called *three-party model* [GTH02] which is depicted in Figure 3.1. In this model, a *Source* wants to delegate the access to a data structure through a *Replica* which may be corrupted. The final *Client* interacts with the *Replica* to gain access to the data-structure. In order to prevent the *Replica* from fooling the *Client*, the *Source* publishes some short value that represents the state of the data-structure after each update as well as some extra cryptographic data for

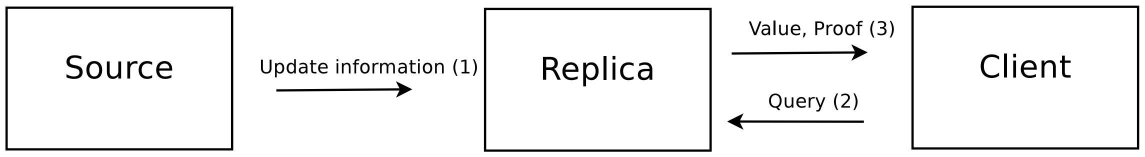


Figure 3.1: **Three-party model**: (1) The *Source* sends the updates of the database to the *Replica* with additional cryptographic values. The *Replica* convinces the *Client* that the answer to the query (2) is valid using a proof (3).

the *Replica*. Then, using this value and some proof computed by the *Replica*, the *Client* gets the guarantee that the answer of the *Replica* is authentic. The challenge in this model is related to efficiency. How can we enable authentic answers while at the same time avoiding too much computation or bandwidth use? In this thesis we are interested in the authentication of simple data structures which are dictionaries and graphs.

Another model, called *two-party model*, considers two participants: a *Client* that has only a small amount of storage and *Server* which is used to extend the memory of the *Client*, but is not trusted. This problem also appears in the literature under the name of *memory checking* [BEG$^+$91, NR05, DNRV09].

3.1 Authenticated dictionaries

In an authenticated dictionary the operations to access and update the data involve some guarantee that the answers are consistent with the actual state of the data structure.

Let $\mathcal{D}$ be a dictionary. The owner of the dictionary or *Source* publishes the data structure with some additional cryptographic information. Moreover, the *Source* also publishes a short value corresponding to the current state of the dictionary. Using this short value, a *Client* can ask a *Replica* for a proof that $\mathcal{D}[i] = v$ for some index i and value v. The security requirement states that the *Replica* should only be able to compute the proof if indeed $\mathcal{D}[i] = v$, otherwise the verification algorithm run by the *Client* should detect the forgery. When the dictionary is updated, the *Source* must publish a new state and also the necessary data to allow proof computations by the *Replica*.

Next we define formally, the syntax, correctness, and security for an authenticated dictionary.

Definition 8 *(Authenticated dictionary in the 3-party model [GTH02])*

Let $\kappa \in \mathbb{N}$ be the security parameter. An authenticated dictionary scheme `AuthDict` *consists of the following algorithms.*

- $\mathsf{KeyGen}(1^\kappa, N)$*: This probabilistic algorithm takes κ in unary as input and N, a bound of the number of elements of the dictionary $\mathcal{D}$. It returns a pair of public and private keys (PK, SK), the initial state of the data structure $\mathfrak{m}_0$ and a short value $\hat{\mathfrak{m}}_0$ that represents $\mathfrak{m}_0$. This short state is public and represents the dictionary $\mathcal{D}$ which set of keys is $\{0, 1, ..., N-1\}$.*

- $\mathsf{Verify}(i, v, \pi, \hat{\mathfrak{m}}, PK)$*: Given an index i, a value v, a proof π, the short state $\hat{\mathfrak{m}}$ and the public key PK, return* valid *meaning that $\mathcal{D}[i] = v$, or $\perp$ otherwise. This algorithm is run by a* Client.

- $\mathsf{ProofGen}(i, \mathfrak{m}, PK)$*: This algorithm returns a proof π associated with the value v and index i such that $\mathcal{D}[i] = v$ where $\mathcal{D}$ is represented by $\mathfrak{m}$. This algorithm is run by the* Replica.

- $\mathsf{Update}(i, v, \mathfrak{m}_{before}, PK, [SK])$*: This algorithm computes the new state of the data structure $\mathfrak{m}_{after}$ where $\mathcal{D}[i] = v$ and the rest of the dictionary remains unchanged. Moreover, the short representation of the state $\hat{\mathfrak{m}}_{before}$ is also updated to $\hat{\mathfrak{m}}_{after}$. The* Source *publishes $\hat{\mathfrak{m}}_{before}$. The argument SK is optional as some schemes, like the one introduced in Chapter 4, do not require the use of any secret . Finally, the* Source *sends some update data to the* Replica *so that the* Replica *can also update the full state of the dictionary and compute new proofs.*

An authenticated dictionary is said to be correct if the *Replica* is always able to compute valid proofs for pairs (i, v) such that $\mathcal{D}[i] = v$.

Definition 9 *(Correctness) Let $\mathfrak{m}_1, \mathfrak{m}_2, ..., \mathfrak{m}_h$ be a sequence of updates, then we say the scheme is correct if for any $0 \leq i \leq N-1$ and any $1 \leq j \leq h$, we have that*

$$\Pr\left[\,\pi = \mathsf{ProofGen}(i, \mathfrak{m}_j, PK) \wedge \mathsf{Verify}(i, v, \pi, \hat{\mathfrak{m}}_j, PK) = \mathsf{valid} \wedge \mathcal{D}[i] = v\,\right] = 1$$

where the probability is taken over the random coins of all the algorithms involved.

The authenticated dictionary is said to be secure if, for any PPT $\mathcal{A}$, the probability to compute a proof π for a pair (i, v') that both passes the verification step and satisfies $\mathcal{D}[i] \neq v'$, is negligible.

Definition 10 *(Security for authenticated data structures [GTH02])*

Let `AuthDict` *be an authenticated dictionary. We consider the notion of security denoted $\mathcal{UF}{-}\mathcal{AD}$ described by the following experiment: On input the security parameter κ, the adversary $\mathcal{A}$ has access to an oracle $\mathcal{O}_{AD}(\cdot)$ that replies to queries by playing the role of the* Source. *Using the oracle, the adversary asks for updates to the dictionary a polynomial number of times. The oracle $\mathcal{O}_{AD}(\cdot)$ replies with the new state of the data structure and also the necessary data in order to compute proofs. Finally, the adversary is required to output a tuple (i, v, π).*

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

$$Adv^{\mathcal{UF}-\mathcal{AD}}_{\texttt{AuthDict}}(\mathcal{A}, \kappa) \quad = \quad \Pr\left[\,\mathsf{Verify}(i, v, \pi, \hat{\mathfrak{m}}, PK) = \mathsf{valid} \wedge \mathcal{D}[i] \neq v\,\right]$$

where PK is the public key generated by KeyGen*, and $\hat{\mathfrak{m}}$ is the state of the dictionary $\mathcal{D}$ at the end of the experiment. The scheme* `AuthDict` *is said to be secure if for every PPT adversary $\mathcal{A}$, $Adv^{\mathcal{UF}-\mathcal{AD}}_{\texttt{AuthDict}}(\mathcal{A}, \kappa)$ is negligible.*

Note that a trivial construction can be obtained using standard digital signatures. The idea is to time-stamp every element of the dictionary after each update and sign these new values. Obviously, this solution is very inefficient for the *Source* and the *Replica* who need to handle $O(N)$ operations for each update. Another idea is to hash all the dictionary using a standard CRHF and exhibit the whole dictionary as the proof π. Here the problem becomes worse as even the *Client* will have to manage the $O(N)$ values to check for a given item of the dictionary. In section 3.2 we will describe a technique called *incremental hashing* that enables efficient updating of the hash value so that in the previous solution the work of the *Source* is reduced to constant time. This only solves a part of the problem of building an efficient authenticated dictionary. However, this technique will be used in chapter 6.

All existing constructions for this model, in particular [CHKO08, CL02, BGG94, Ngu05], involve a tradeoff between the size of the proof and the number of accesses to update the data structure and compute the proof. More precisely, for some $\lambda \geq 1$ for a proof of size $O(\lambda)$, the time to update the data structure or compute the proof will be $O(\lambda N^{1/\lambda})$.

Table 3.1 summarizes the complexities of the current constructions. An important open problem is to know whether we can build optimal authenticated dictionaries (OAD) where the time to update the data structure/compute the proof would involve only $O(\log N)$ cryptographic operations and the size of the proof would remain constant in N. In Section 3.4.3 we give a more precise definition of optimal authenticated dictionaries and establish a relation between optimal authenticated dictionaries and transitive signatures.

3.2 Incremental Hashing

As mentioned earlier, one of the challenges when authenticating data is that this data may be subject to change. Trivial solutions consisting of rehashing/resigning all the data are clearly inefficient, as shown in the previous section. However, changes to the data generally involve a small portion of bits. For example, if we want to authenticate successive video frames, clearly the difference between two consecutive images will be "small".

To solve this problem Bellare, Goldreich and Goldwasser [BGG94] introduced a new kind of CRHF that allows efficient updates to hash values depending only on the changes made to the data. They called this new primitive *incremental hashing.*

Solution	Assumption	T_{Update}	T_{ProofGen}	T_{Verify}	Bandwidth
Timestamps	DSig	$O(N)$	$O(N)$	$O(1)$	$O(1)$
CRHF	CRHF	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Incremental hashing [BGG94]	DLog	$O(1)$	$O(1)$	$O(N)$	$O(N)$
[CHKO08]	CHRF	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$
[CL02]	S-RSA	$O(\lambda \cdot N^{\frac{1}{\lambda}})$	$O(\lambda \cdot N^{\frac{1}{\lambda}})$	$O(\lambda)$	$O(\lambda)$
[Ngu05]	q-SDH	$O(1)$	$O(N)$	$O(1)$	$O(1)$

Table 3.1: Tradeoffs for authenticated dictionaries.
Complexities are relatives to the number of cryptographic operations. The first row is related to the trivial solution which consists in signing elements of the dictionary with their timestamp. *DSig* refers to the existence of standard digital signatures. The second and third row specifies the complexities for the trivial solutions using standard *CRHF* and incremental *CRHF* respectively. T_{Update} is the time to update the data structure, T_{ProofGen} the time to compute a proof and T_{Verify} is the time to check the proof. The abbreviations *DLog, S-RSA, q-SDH* stand respectively for *Discrete Logarithm assumption, Strong RSA Assumption, and q-Strong Diffie-Hellman Assumption.*

The idea is the following: let $\mathbb{G}$ be a cyclic group of order p where the discrete logarithm is assumed to be hard. Then for a message $M = (M_1, ..., M_N)$ where for each $i \in [N] : M_i \in \mathbb{Z}_p$, we define $\mathsf{H}(M) = \prod_{i=1}^{N} g_i^{M_i}$ where g_i are random generators of $\mathbb{G}$. That is H is defined by $(g_1, \ldots, g_N)$.

Authenticated dictionary using incremental hashing. We can observe that by using an incremental hashing scheme we can build an authenticated dictionary with optimal update time. However, the time to verify a proof is still linear in the size of the dictionary.

Construction 1 *(Authenticated dictionary from incremental hashing)*

- $\mathsf{KeyGen}(1^\kappa, N)$: *Let $\mathbb{G}$ a cyclic group of order p where the discrete logarithm is hard. Select $g \overset{R}{\leftarrow} \mathbb{G}$ a generator. Choose random $s_1, s_2, ..., s_N$ in $\mathbb{Z}_p$. Let $g_i = g^{s_i}$ for $i \in [N]$. The $\mathfrak{m}_0$ data structure is simply the dictionary $\mathcal{D}$ which set of keys is $\{0, 1, ..., N-1\}$. That is for each $i \in [0..N-1] : \mathfrak{m}[i] = \mathcal{D}[i-1]$.*

We assume the dictionary has values in $\mathbb{Z}_p$ (we can use a standard CRHF if we need to work in a bigger universe). Return $\hat{\mathfrak{m}}_0 = \prod_{i=1}^{N} g_i^{\mathcal{D}_0[i-1]}$, where $\mathcal{D}_0$ is the initial state of the dictionary (filled with 0 values for example). Here PK is the description of the hash function. That is $PK = (g_i)_{i \in [N]}$. In this scheme there is no need of secret key or trapdoor.

- $\mathsf{Verify}(i, v, \pi, \hat{\mathfrak{m}}, PK)$: *Parse π as $(\alpha_j)_{j \in [0..N-1]}$. Check that $\hat{\mathfrak{m}} = \prod_{j=1}^{N} g_j^{\alpha_{j-1}}$ and that $\alpha_i = v$.*
- $\mathsf{ProofGen}(i, \mathfrak{m}, PK)$: *this algorithm Return $\pi = (\mathcal{D}[j-1])_{j \in [N]}$.*
- $\mathsf{Update}(i, v, \mathfrak{m}_{before}, PK)$: *Update the memory to obtain $\mathfrak{m}_{after}$. Compute and publish $\hat{\mathfrak{m}_{after}} = v(g_i^{\mathfrak{m}_{before}[i]})^{-1} g_i^{\mathfrak{m}_{after}[i]}$.*

Following directly from [BGG94] we have

Proposition 2 *Under the discrete logarithm assumption the authenticated dictionary described above is secure. The time complexities are : $T_{\mathsf{Update}} = O(1)$, $T_{\mathsf{ProofGen}} = O(1)$, $T_{\mathsf{Verify}} = O(N)$.*

3.3 Cryptographic accumulators

The notion of accumulator, first proposed by Benaloh and De Mare in [BdM93], is quite intuitive: Given a set $X = \{x_1, \ldots, x_N\}$, we wish to compute a short value that represents this set, also called accumulated value, such that it is then possible to prove that a given element x belongs to the accumulated set. Informally, an accumulator scheme consists of at least the following polynomial-time algorithms:

- $\mathsf{KeyGen}(1^\kappa)$: Generates the initial parameters, in particular the initial accumulated value that represents the empty set.
- $\mathsf{AccVal}(X)$: Computes the accumulated value Acc_X of the set X.
- $\mathsf{WitGen}(X, x)$: For a given element x that has been accumulated, this algorithm computes a witness that will serve as a parameter for the algorithm Verify.

- $\mathsf{Verify}(Acc, x, \pi)$: From the accumulated value Acc, an element x, and a witness value π, Verify returns valid if, and only if, π is a valid witness that guarantees that x has been accumulated, or equivalently $x \in X$, the set represented by Acc_X.

If we want to give a formalization of accumulators using predicate-preserving hash functions, then we would say that an accumulator is a pair of CRHF $(\mathsf{H}_1, \mathsf{H}_2)$ where H_1 is implemented by the AccVal algorithm and H_2 is the identity function. Moreover the predicate that is preserved by $(\mathsf{H}_1, \mathsf{H}_2)$ is $\mathcal{P}_\in : 2^U \times U \rightarrow \{0, 1\}$, such that $\mathcal{P}_\in(X, x) = 1 \Leftrightarrow x \in X$, where U is some universe and 2^U is the collection of sets with elements in U. A similar definition could be given for non-membership. In this analogy, WitGen stands for $\mathsf{ProofGen}$, Verify stands for $\mathsf{ProofCheck}$ and the proof π is called witness.

From the above definition it is clear that it has to be difficult for the adversary to compute a witness π', for an element $x' \notin X$ such that $\mathsf{Verify}(Acc, x', \pi') = \mathsf{valid}$.

As mentioned in the introduction, one can build a trivial accumulator scheme using a standard CRHF H to implement AccVal and then using the whole set X as a witness. The verification procedure Verify would consist in checking that $x \in X$ and also check that $\mathsf{H}(X) = Acc_X$. Obviously, this construction is of little interest as it is very inefficient (checking for membership requires $O(N)$ computation). To be useful, an accumulator scheme must allow for fast (non)membership checking (i.e.$O(1)$ or $O(\log N)$).

3.3.1 Definitions

One-Way Accumulators. The first definition for cryptographic accumulators [BdM93] focused on the algebraic property of the function used to compute the accumulated value. Basically an accumulator was defined as a one-way hash function with two parameters, one for the new element and another for the current accumulated value that has the property of *quasi-commutativity.*

Definition 11 *(One way hash functions [BdM93, FN02])*

A family of one-way hash functions *is a function $\mathcal{H} : K \times X \times Y \to Z$ where*[1]*:*

1. *For any $\mathsf{H} = \mathcal{H}(k, \cdot, \cdot)$ with $k \in K$, $\mathsf{H}(\cdot, \cdot)$ is computable in polynomial time in κ.*

2. *For any PPT $\mathcal{A}$ we have*

$$\Pr\left[\begin{array}{l} k \overset{R}{\leftarrow} K; x \overset{R}{\leftarrow} X; y, y' \overset{R}{\leftarrow} Y; \\ x' \leftarrow \mathcal{A}(1^\kappa, x, y, y', k) : \\ \mathsf{H}(x, y) = \mathsf{H}(x', y') \end{array}\right] < \mathsf{neg}(\kappa)$$

where the probability is taken over the random choice of k, x, y, y' and random coins of $\mathcal{A}$.

Definition 12 *(Quasi-commutative functions, [BdM93]) A function $f : X \times Y \to X$ is said to be quasi-commutative if for all $x \in X$ and all $y_1, y_2 \in Y$,*

$$f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$$

Definition 13 *(One-way accumulators [BdM93]) A family of one-way accumulators is a family of one-way hash functions each of which is quasi-commutative.*

As mentioned in [FN02] this definition is simple, but restricts accumulator schemes to a class of functions with some specific algebraic property. Moreover, the definition of security does not capture the case where the adversary can choose to be involved in the construction of the set for which it will try to forge an element. A definition for this stronger model was introduced by Nyberg in [Nyb96b].

Definition 14 *(Strongly one-way hash functions [Nyb96a]) A family of strongly one-way hash functions is a family of one-way hash functions $\mathcal{H} : K \times X \times Y \to Z$ where*

[1]Note that the sets K, X, Y and Z depend on κ. We do not make it explicit in the notation for the sake of simplicity.

additionally, for any PPT adversary $\mathcal{A}$:

$$\Pr\left[\begin{array}{l} k \overset{R}{\leftarrow} K; x \overset{R}{\leftarrow} X; y \overset{R}{\leftarrow} Y; \\ (x', y') \leftarrow \mathcal{A}(1^\kappa, x, y, k) : \\ \mathsf{H}(x, y) = \mathsf{H}(x', y') \end{array}\right] < \mathsf{neg}(\kappa)$$

where the probability is taken over the random choices of k, x, y and the random coins of $\mathcal{A}$.

Collision-Freenes [BP97]. An even stronger security notion arises when the adversary is able to choose every element of the set and can choose the specific value on which the forgery will be computed. To take this case into consideration, Barić and Pfitzmann proposed another security definition in [BP97] called *collision-freeness* and introduced the abstract syntax for an accumulator.

Definition 15 *(Syntax for accumulator schemes [BP97])*

An accumulator scheme $\mathtt{Acc}$ *is a 4-tuple of polynomial time algorithms* $\mathtt{Acc} = (\mathsf{KeyGen}, \mathsf{AccVal}, \mathsf{WitGen}, \mathsf{Verify})$*, where:*

- $\mathsf{KeyGen}(\kappa, N)$: *This probabilistic algorithm takes as input a security parameter κ, N, the maximum size of the set and returns a key k from the space K. Intuitively this key represents the hash function for the set, which is implemented by the next algorithm,* AccVal*. Some (private) auxiliary information SK is also returned, which will be used by other algorithms.*

- $\mathsf{AccVal}(k, X)$: *This probabilistic algorithm is used to compute the accumulated value of set $X = \{x_1, \ldots, x_N\}$. k is the accumulator key, that is* $\mathsf{AccVal}(k, \cdot)$ *can be considered as a hash function for its argument X.* AccVal *returns an accumulated (hashed) value $Acc_X \in Z$.*

- $\mathsf{WitGen}(k, y, Acc_X, SK)$: *This algorithm computes a witness for element y in set X, which is such that $Acc_X = \mathsf{AccVal}(k, X)$. If $y \notin X$, the special symbol $\perp$ is returned. The auxiliary (private) value SK can be used to compute the witness.*

- $\mathsf{Verify}(k, y, \pi, Acc_X)$: *This deterministic algorithm returns* valid *if the witness* π *constitutes a valid proof that* y *has been accumulated within* Acc_X. *Otherwise it returns* $\perp$.

Definition 16 *(N-times collision-freenes [BP97])*
An accumulator scheme is said to be N*-times* collision-free *for any PPT* $\mathcal{A}$*:*

$$\Pr\left[\begin{array}{l} (k, SK) \leftarrow \mathsf{KeyGen}(1^\kappa, N); \\ (x_1, \ldots, x_N, y, \pi) \leftarrow \mathcal{A}(1^\kappa, N, k); \\ Acc_X \leftarrow \mathsf{AccVal}(k, x_1, \ldots, x_N) : \\ (y \notin \{x_1, \ldots, x_N\}) \wedge (\mathsf{Verify}(k, y, \pi, Acc_X) = \mathsf{valid}) \end{array}\right] < \mathsf{neg}(\kappa)$$

where the probability is taken over the random coins of KeyGen*,* AccVal *and* $\mathcal{A}$.

Definition 17 *(Collision-Freeness, [BP97]) An accumulator scheme is said to be collision-free if it is* $P(\kappa)$*-times collision-free where* $P(\cdot)$ *is a polynomial.*

Dynamic Accumulators. The previous definitions assume implicitly that the set that is accumulated is static. Although such accumulators can be useful, considering dynamic accumulators where the set can evolve opens a wide range of applications in particular for building efficient authenticated dictionaries like for example [GTH02, GTT08]. Because *sets* is a very general data structure, dynamic accumulators have been used in very diverse contexts like broadcast encryption [GR04], e-cash [AWSM07] and zero-knowledge sets [XLL07, CFM08, LY10]. Dynamic accumulators were introduced in [CL02] with a non trivial application for anonymous credentials with efficient revocation.

In the dynamic scenario we can consider two kinds of participants:

- The *manager* who has access to the set and is able to update the accumulated value with some public or private information.
- The *user* who wants to check for (non)membership in the set. The *user* knows the accumulated value along with the public parameters of the scheme.

The next definition introduces the functionalities involved in a dynamic accumulator.

Definition 18 *(Syntax for dynamic accumulators, adapted from [CL02]) Let $\kappa \in \mathbb{N}$ be the security parameter. An accumulator scheme* `Acc` *consists of the following algorithms.*

- $\mathsf{KeyGen}(1^\kappa)$*: 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 $Acc_\emptyset$.*

- $\mathsf{AccVal}(X, Acc_\emptyset, PK, [SK])$*: Given a finite set of elements X (of at most polynomial size in κ), a public key PK*[2] *and the initial accumulated value $Acc_\emptyset$, this algorithm returns the accumulated value Acc_X corresponding to the set X. Depending on the implementation, the secret key SK may also be given as optional parameter, often to improve the efficiency*[3]*.*

- $\mathsf{Verify}(x, \pi, Acc_X, PK, [SK])$*: Given an element x, a witness π, an accumulated value Acc_X, and a public key PK (and possibly a secret SK as in the construction by Papamanthou et al. [PTT10]), 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*.*

- $\mathsf{WitGen}(x, Acc_X, PK, [SK])$*: This algorithm returns a witness π associated to the element x of the set X represented by Acc_X. If $x \notin X$ and the accumulator scheme provides non-membership proofs then returns π, the witness that proves that $x \notin X$. This algorithm is run by the* manager*.*

- $\mathsf{AddEle}(x, Acc_X, PK, [SK])$*: 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*.*

[2]In previous definitions this public key PK was the index k used to choose a hash function. In other words, PK describes a CRHF for sets.

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

- $\mathsf{DelEle}(x, Acc_X, PK, [SK])$: *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.

- $\mathsf{UpdWitGen}(X, X', PK, [SK])$: *Suppose the set X is transformed into the set X' after several updates (insertions/deletions). The algorithm* $\mathsf{UpdWitGen}$ *returns the information $Upd_{X,X'}$ required to update all the witnesses (using the algorithm* UpdWit*) of the elements of X that are still in X'. This algorithm is run by the* manager.

- $\mathsf{UpdWit}(\pi_x, Acc_X, Acc_{X'}, Upd_{X,X'}, PK)$: *This algorithm recomputes the witness π_x for some element x that remains in the set X'. It takes as parameters an existent witness π_x with respect to set X (represented by the accumulated value Acc_X), some update information $Upd_{X,X'}$, and the public key PK. It returns a new witness π'_x for the element x with respect to the new set X' represented by some accumulated value $Acc_{X'}$. This algorithm is run by the* user.

The above definition is slightly more general than the one proposed by Camenisch and Lysyanskaya [CL02], as it does not depend on how these algorithms are implemented, and it explicitly includes the update algorithms UpdWit and $\mathsf{UpdWitGen}$ in the syntax.

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

Definition 19 *(Correctness) Let X, Y be sets, Acc_X, Acc_Y their respective associated accumulated values, PK a public key, SK the corresponding private key, and $y \in Y$. Let π_y a value (witness) that satisfies either*

- $\pi_y \in \{\mathsf{WitGen}(y, Acc_Y, PK, SK)\}$, *or*

- $\pi_y \in \{\mathsf{UpdWit}(\pi'_y, Upd_{X,Y}, PK)\}$ *with π'_y witness of y with respect to Acc_X, and $Upd_{X,Y} \in \{\mathsf{UpdWitGen}(X, Y, PK, SK)\}$.*

We say that an accumulator scheme `Acc` *is correct if, and only if,*
$\mathsf{Verify}(y, \pi_y, Acc_Y, PK) = \mathsf{valid}$, *for every such y, π_y, X, Y.*

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. find 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 [GMR88].

Definition 20 *(Security for dynamic accumulators [CL02]) Let* $\mathtt{Acc}$ *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* $\mathcal{O}_{\mathtt{Acc}}(\cdot)$ *that replies to queries by playing the role of the* manager*. Using the oracle, the adversary can insert and delete a polynomial number of elements of his choice. The oracle* $\mathcal{O}_{\mathtt{Acc}}(\cdot)$ *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, π)*. The advantage of the adversary* $\mathcal{A}$ *is defined by:*

$$Adv_{\mathtt{Acc}}^{\mathcal{UF}-\mathcal{ACC}}(\mathcal{A}, \kappa) \quad = \quad \Pr\left[\mathsf{Verify}(x, \pi, Acc_X, PK) = \mathsf{valid} \wedge x \notin X\right]$$

where PK *is the public key generated by* KeyGen*, and* Acc_X *is the accumulated value of the resulting accumulated set* X*. The scheme* $\mathtt{Acc}$ *is said to be secure if for every PPT* $\mathcal{A}$ *we have:*

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

AUTHENTICATED DICTIONARIES USING ACCUMULATORS. Accumulators can be used naturally to implement authenticated dictionaries. Here, the *manager* will be split in two parts: (1) The *Source* that will update the accumulated value and possibly compute some additional value to help compute or update the witnesses, and (2) the *Replica* that will compute the witnesses from the state of the data structure and the values sent by the *Source*.

In chapter 4, we show that we can replace the *Source* by a public broadcast channel meaning that the *Replica* can compute all the proofs without any trapdoor. In other words, we formalize the security and instantiate an authenticated dictionary that does not rely on any kind of secret.

3.3.2 Constructions

We introduce here the main constructions for cryptographic accumulators.

RSA ACCUMULATORS. The concept of cryptographic accumulators was first introduced in [BdM93]. The idea of this construction relies on a quasi-commutative one-way hash function f that "hashes" an element of the set to be accumulated with a temporary accumulated value (that corresponds to the elements that have been accumulated at the moment) into a single value. Then the quasi-commutativity of the function f guarantees that the accumulated value does not depend on the order of insertion of the elements. Let us consider for example the set $X = \{x_1, x_2, x_3, x_4\}$. In this case, the accumulated value will be

$$Acc_X = f(f(f(f(v, x_1), \mathbf{x_2}), x_3), \mathbf{x_4})$$

where v is some initial arbitrary value. As the order of insertion of elements does not change the value Acc, given $x_2 \in X$, we can swap the elements x_2 and x_4 and obtain

$$Acc_X = f(f(f(f(v, x_1), \mathbf{x_4}), x_3), \mathbf{x_2})$$

Let $\pi = f(f(f(v, x_1), \mathbf{x_4}), x_3)$, we have $f(\pi, x_2) = Acc_X$. Our witness for x_2 is π and the algorithm Verify consists in verifying that $f(\pi, x_2) = Acc$.

As f is a one-way hash function, given Acc and $x \in X$, it has to be difficult to find π_x such that $f(\pi_x, x) = Acc$ with $x \notin X$ considering that Acc is a random value. However, in many applications, one-way hash functions are often too weak, especially for handling the case of an adversary that would carefully select the values to be accumulated, or equivalently, where Acc is not a random value. To fix this issue, Benaloh and de Mare use the random oracle model[4]: Instead of directly accumulating the values of the set, the accumulated value is computed from the hash values of the elements of X. The concrete instantiation of f proposed is the RSA modular exponentiation: Let p, q be primes, $n = pq$, the function f is defined by: $f(x, y) = x^y \mathsf{\ mod\ } n$. Note that these primes need to be safe, i.e. of the form $p = 2r + 1$ where r is also prime, in order to resist to cryptanalytic attacks like Pollard's $p - 1$ method [Pol74].

In this case, f is clearly quasi-commutative. The one-way property of f relies on the RSA Assumption, although special care must be taken as f is a one-way hash function and does not necessarily imply that f composed many times is still one-way.

It is worth noting that the knowledge of the factorization of n allows a malicious participant to compute inconsistent witnesses. A solution to keep the factorization secret to all parties is to use a generic multi-party secure computation protocol [GMW87]. Alternatively one can use Sander's algorithm ([San99]) which returns an integer n of the form pq, p, q primes with high probability, without revealing the factors. Lipmaa proposed recently [Lip12] a construction that does not require a trusted setup and offers similar efficiency (constant-size witness).

Although the security properties of this construction is insufficient to handle dynamic sets, Benaloh and de Mare's idea is a first important step for building accumulators. Upcoming works will improve on this proposal, by adding new properties and refining the security requirements. That is the case of [BP97] which is described next.

The problem with the previous construction can be seen better with an example. Let $n = pq$ an integer where p and q are safe primes. Let $f(x, y) = x^y \mathsf{\ mod\ } n$ and let $X = \{2, 3, 5\}$. Let $g \overset{R}{\leftarrow} \mathbb{Z}_n$. Then $Acc = g^{2\times3\times5} \mathsf{\ mod\ } n = g^{30} \mathsf{\ mod\ } n$. A witness for 5 is $\pi = g^6 \mathsf{\ mod\ } n$ as $\pi^5 = Acc \mathsf{\ mod\ } n$. However, $\pi' = g^5 \mathsf{\ mod\ } n$ is a witness for 6 ($\pi'^6 = Acc$) and so it can be proven that 6 belongs to X. We found a collision. The fact that elements accumulated are not necessarily primes introduced a trivial flaw in the scheme. Fixing this flaw is indeed quite simple: We need to accumulate only prime

[4]For a discussion on the limits of this model and how it can be "well implemented", the reader can refer to the work by Canetti et al. [CGH04].

numbers. This is the idea of Barić and Pfitzmann who manage to strengthen Benaloh and de Mare's construction. They prove the security of the scheme without relying on the random oracle model. To do so, a new (at that time) security assumption was proposed. This is the Strong RSA Assumption. The Strong RSA Assumption states that given a random $z \in \mathbb{Z}_n^*$, it is hard to find non trivial x, y such that $y^x = z \bmod n$.

Definition 21 *(Strong RSA Assumption) For any random integer $n = pq$, with p, q safe primes such that $|p| = |q| = \frac{1}{2}\kappa$, and for every probabilistic polynomial time algorithm $\mathcal{A}$:*

$$\Pr\left[z \stackrel{R}{\leftarrow} \mathbb{Z}_n^*; (x, y) \leftarrow \mathcal{A}(n, z) : y^x = z \bmod n \wedge 1 < x < n \right] < \mathsf{neg}(\kappa)$$

In the previous definitions the accumulator scheme is static, i.e. the set, once accumulated cannot evolve. Camenisch and Lysyanskaya showed how Barić and Pfitzmann's construction could be used also to build dynamic accumulators. The idea is quite simple too: Assuming that the *manager* is trusted, this one is given the knowledge of the factorization of $n = pq$. Then, if the current accumulated set is $X = \{x_1, x_2, \ldots, x_N\}$ and its corresponding accumulated value is $Acc = g^{\prod_{i=1}^N x_i} \bmod n$ where g is chosen at random, to compute Acc_{new} as to represent $X \cup \{x_{N+1}\}$, we only need to raise Acc to power x_{N+1}: $Acc_{new} = Acc^{x_{N+1}} \bmod n$. Deletion of an element $x_i \in X$ can be implemented efficiently too by the *manager* who knows $\Phi(n) = (p-1)(q-1)$ and can compute $x' = x_j^{-1} \bmod \Phi(n)$ using the extended euclidean algorithm. We get $Acc_{new} = Acc^{x'} \bmod n = g^{(\prod_{i=1}^N x_i)x_j^{-1}} \bmod n = g^{\prod_{i=1, i\neq j}^N x_i} \bmod n$ that represents $X \backslash \{x_j\}$. We can observe that the *manager* indeed needs to be trusted, because it is able to compute inconsistent witnesses for every x: $\pi_x = Acc^{x^{-1} \bmod \Phi(n)} \bmod n \Rightarrow \pi_x^x = Acc^{x^{-1}x} \bmod n = Acc \bmod n$. If $x \in X$, then the previous calculus can be used to obtain a (valid) witness efficiently.

An optimization to [CL02] was presented in [TX03] where instead of accumulating prime numbers, elements to be inserted are composites. We can note that all the operations (updating the accumulated value, the witnesses, and testing membership) can be done in time $O(1)$ in function of N, the size of the set.

The accumulator proposed in [CL02] only implements membership proofs. However, the construction can be improved to handle non-membership proofs as well. This was shown by Lie, Lie and Xue in [LLX07]. Membership witnesses are computed as in [BP97, CL02] and the corresponding Verify algorithm remains the same. The "trick" in proving non-membership is based on the following lemma:

Lemma 1 *([LLX07]) For any integer n, for any $u, v \in \mathbb{Z}_n^*$ and $a, b \in \mathbb{Z}$, such that $u^a = v^b \bmod n$ and $gcd(a, b) = 1$, one can efficiently compute $x \in \mathbb{Z}_n^*$ such that $x^a = v \bmod n$.*

Proof. As $gcd(a, b) = 1$, then using the extended euclidean algorithm, it is possible to efficiently compute $c, d \in \mathbb{Z}$ such that $bd = 1 + ac$. Let $x = u^d v^{-c} \bmod n$. We can verify that $x^a = (u^d v^{-c})^a = u^{ad} v^{-ac} = v^{bd-ac} = v \bmod n$. ■

Then, the non-membership witness is defined as the following:
Let $X = \{x_1, \ldots, x_N\}$ the set to be accumulated and $Acc = g^{\prod_{i=1}^N x_i} \bmod n$ the corresponding accumulated value. The non-membership witness for $x \notin X$ is the pair $w = (a, d)$ such as $Acc^a = d^x g \bmod n$. Computing (a, d) from x and Acc can be done noting that $gcd(u, x) = 1$ with $u = \prod_{i=1}^N x_i$ and thus it is possible to find efficiently a, b such that $au + bx = 1$. Then set $\pi = (a, g^{-b})$. We can verify that $Acc^a = g^{ua} \bmod n = g^{1-bx} \bmod n = (g^{-b})^x g \bmod n$.

As in [BP97, CL02] the security of the construction is based on the Strong RSA Assumption.

Theorem 1 *([LLX07]) Under the Strong RSA Assumption, the construction described above is a secure universal accumulator.*

Proof. Let us assume there exists an adversary $\mathcal{A}$ that can output
$X = \{x_1, \ldots, x_N\}$ along with x, π_1, π_2 where x is an element (that belongs or not to X), π_1 is a witness of membership, and π_2 is a witness of non-membership. Indeed, if the adversary is able to find a witness of membership and a witness of non-membership that works for the same x, that means that one of the witnesses is inconsistent. We will prove then that the Strong RSA Assumption does not hold. That means given

$g \xleftarrow{R} \mathbb{Z}_n^*$, it is possible to build and adversary $\mathcal{B}$ which computes efficiently z, x with $1 < x < n$ such that $z^x = g \bmod n$. Let $u = \prod_{i=1}^{N} x_i$.

Let us suppose $x \notin X$. Then $gcd(x, u) = 1$ (we remind the reader that all the x_i are primes) Moreover, π_1 is a witness of membership, i.e. $\pi_1^x = Acc = g^u \bmod n$. By the lemma, it is possible to efficiently compute z such that $z^x = g \bmod n$.

In the other case, let us consider that $x \in X$. Then $\pi_2 = (a, d)$ in that $Acc^a = d^x g \bmod n$ or equivalently $g^{ua-1} = d^x \bmod n$. As $x \in X \Rightarrow x|u$, then $gcd(au-1, x) = 1$ and by the lemma it is possible to efficiently compute z such that $z^x = g \bmod n$. ■

The next definition summarizes the algorithms involved in a RSA accumulator.

Construction 2 *(RSA Accumulators construction [CL02, LLX07])*

- $\mathsf{KeyGen}(1^\kappa)$*: Generate a safe RSA modulus* $n = pq$ *where* p, q *which binary representation has length* κ. *Choose a random* $g \in \mathbb{Z}_n^*$. *Set* $Acc = g$. *Let* $\mathsf{Prime} : \mathbb{N} \to \mathbb{P}$ *be an efficient map which associates to* $x \in \mathbb{N}$, *the* x^{th} *prime number in the set of primes* $\mathbb{P}$.

  Set $PK = (n, g)$ *and* $SK = (p, q, g)$.

- $\mathsf{AccVal}(X, Acc_\emptyset, PK, [SK])$*: Return* $Acc_X = g^{\prod_{x \in X} \mathsf{Prime}(x)}$.

- $\mathsf{WitGen}(x, Acc_X, PK, SK)$*:*

  - *(Membership)Check that* $x \in X$. *If this is the case return* $w_x = Acc_X^{\frac{1}{y}}$ *where* $y = \mathsf{Prime}(x)$.
  - *(Non-membership) Let* $u = \prod_{v \in X} y_v$ *where* $y_v = \mathsf{Prime}(v)$. *As* $x \notin X$ *we have that* $gcd(u, y) = 1$ *where* $y = \mathsf{Prime}(x)$. *Using the extended euclidean algorithm, find* (a, b) *such that* $au + by = 1$. *Then, return* $\pi = (a, d)$ *where* $d = g^{-b} \bmod n$.

- $\mathsf{Verify}(x, \pi, Acc_X, PK, [SK])$*: Parse* π *as* (a, d). *Let* $y = \mathsf{Prime}(x)$. *Return* valid *if, and only if,* $Acc_X^a = d^y g \bmod n$.

- $\mathsf{AddEle}(x, Acc_X, PK)$*: Compute* $y = \mathsf{Prime}(x)$ *and return* $Acc_{X \cup \{x\}} = Acc_X^y \bmod n$.

- $\mathsf{DelEle}(x, Acc_X, PK, SK)$: *Compute* $y = \mathsf{Prime}(x)$ *and return* $Acc_{X\setminus\{x\}} = Acc_X^{\frac{1}{y}} \bmod n$.

- $\mathsf{UpdWitGen}(X, X', PK, [SK])$: *Let* U_{Add} *be the list of elements that were added between the moment when the set was* X *and the moment it becomes* X'. *Let* U_{Del} *be the analog set corresponding to deletions. Return* $Upd_{X,X'} = (U_{Add}, U_{Del})$.

- $\mathsf{UpdWit}(\pi, Acc_X, Acc_{X'}, Upd_{X,X'}, PK)$: *Parse* $Upd_{X,X'}$ *as* (U_{Add}, U_{Del}). *The algorithm to update the witness* π *relative to element* x *after the addition of an element* x' *consists in raising* π *to* $y' = \mathsf{Prime}(x')$. *That is* $\pi_{new} = \pi^{y'} \bmod n$. *The algorithm to update the witness* π *relative to element* x *after the deletion of an element* x' *consists in finding* a, b *such that* $ay + by' = 1$ *where* $y' = \mathsf{Prime}(x')$ *and* $y = \mathsf{Prime}(x)$. *This is possible as* y, y' *are prime and distinct. Then,* $\pi_{new} = Acc_{X'}^{a} w^b \bmod n$ *is the new witness for* x. *The update is correct since* $\pi_{new}^{y} = Acc_{X'}^{ay} w^{by'} = Acc_{X'}^{ay} Acc_X^{b} = Acc_{X'}^{ay} Acc_{X'}^{by'} = Acc_{X'}$.

As mentioned in [FN02], the time to update a witness after m changes made to the set is proportional to m for the construction introduced in [CL02]. It is natural to ask whether this can be improved, that is, if one could update each witness after several updates using only a value of constant size w.r.t. m. We answer in the negative in chapter 5.

Constructions based on Collision-Resistant Hash Functions. The other main type of accumulator is based on collision-resistant hash functions. In [BdM91], although the concept of accumulator was not yet formulated, Benaloh and de Mare proposed a hash-based linking scheme, also called Merkle trees [Mer87], that allows to prove a given document (that is, an element) belongs to a given time-round (set). This scheme solves the practical problem of time-stamping. The idea is to build a binary tree where the leaves are the documents, and where internal nodes values are computed recursively taking the hash of both children. The root value of the tree represents the time-round value, and can be used to prove a document has been time-stamped to this instant of time by giving a log-size sub-tree of the round tree. This sub-tree clearly takes the role of the witness, and the root value can be considered as the accumulated value. In [LBLV98] an accumulator based on CRHF but with an additional property is presented: It allows comparing the order of insertions of the elements in the set.

In [CHKO08] (see chapter 4), we take this idea of using binary trees and hash functions and we extend it to build an accumulator where the *manager* need not to be trusted.

Pairing based constructions. The first pairing based construction for accumulators was proposed by Nguyen [Ngu05]. This construction relies on the N-SDH assumption. Nguyen's construction has shown to be useful beyond its applications to ring signatures and anonymous credentials. In particular it is a building block for improving the efficiency of zero-knowledge sets [CFM08] or implementing short commitments for polynomials [KZG10]. We use a similar idea in our fair exchange protocol (see chapter 7). The idea of the construction is to represent $X = \{x_1, x_2, \ldots, x_N\} \in \mathbb{Z}_p^*$ by the polynomial $P_X(s) = \prod_{i=1}^{N}(x_i + s)$ where s is the trapdoor of the scheme. In order to "hide" the secret, the polynomial is put in the exponent and the accumulated value is set to $Acc_X = g^{P_X(s)}$. A witness for an element x corresponds to the polynomial $P_X(s)/(x+s)$, that is $\pi_x = Acc_X^{\frac{1}{x+s}}$. To compute such a witness without knowing the trapdoor s, the N-SDH tuple $(g, g^s, g^{s^2}, \ldots, g^{s^N})$ is published. Then, the computation of the formal polynomial $P_X(Y) = \prod_{i=1}^{N}(x_i + Y) = \sum_{i=1}^{N} a_i Y^i$, in the exponent can be performed if, and only if, $(x+s)|P_X(s)$, which means $x \in X$. Checking that an element $x \in X$ is done by taking advantage of the bilinear property of $e(\cdot,\cdot)$ that allows to multiply the exponents once: That is $x \in X$ if, and only if, $e(\pi_x, g^{(x+s)}) = e(Acc_X, g)$.

The non-membership functionality was introduced in [DT08, ATSM09] and is based on the dual idea that if an element $y \notin X$, then there exist polynomials Z and U with $U \neq 0$ such that $P_X(Y) = (y+s)V(s) + U(s)$. Again, such polynomials can be computed with the public parameters. Formally we have:

Construction 3 *(Nguyen's accumulator [Ngu05, DT08, ATSM09])*

- $\mathsf{KeyGen}(1^\kappa, N)$*: Returns a pairing tuple* $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BMGen}(1^\kappa)$*. Let* $s \overset{R}{\leftarrow} \mathbb{Z}$ *be the trapdoor of the scheme from which the public* N*-SDH tuple can be* *derived:* $(g, g^s, g^{s^2}, \ldots, g^{s^N})$. *$N$ is an upper bound on the size of the set. The initial accumulated value is defined simply as* $Acc_\emptyset = g$. *The public key is defined as* $PK =< (p, \mathbb{G}, \mathbb{G}_T, e), (g, g^s, g^{s^2}, \ldots, g^{s^N}) >$.

- $\mathsf{AccVal}(X, Acc_\emptyset, PK)$: *Given a finite set of elements X where $|X| \leq N$, and the public parameter PK return* $Acc_X = g^{P_X(s)} = g^{\prod_{x\in X}(x+s)}$.

- $\mathsf{WitGen}(x, Acc_X, PK)$:

  - *Membership: Compute using the formal polynomial $P_X(Y)$ the formal polynomial* $\bar{W}(Y) = P_X(Y)/(x+Y)$. *Then return* $\pi = g^{\frac{P_X(s)}{x+s}}$.

  - *Non-membership: If* $y \notin X$:

    - $*$ *Compute* $u_y = -P_X(y) \bmod p = -\prod_{x\in X}(x-y)$
    - $*$ *Compute* $w_y = g^{\frac{P_X(s)-P_X(y)}{y+s}}$

  Return $\bar{\pi} = (u_y, w_y)$.

- $\mathsf{Verify}(x, \pi, Acc_X, PK)$:

  - *Membership: Return* valid *if, and only if,* $e(\pi, g^{(x+s)}) = e(Acc_X, g)$.

  - *Non-membership: Parse π as (w_y, u_y) return* valid *if, and only if,* $e(w_y, g^y \cdot g^s) = e(Acc_X \cdot g^{u_y}, g)$ *and* $u_y \neq 0$.

- $\mathsf{AddEle}(x, Acc_X, PK)$: *The new accumulated value* $Acc_{X\cup\{x\}} = Acc_X^{(x+s)}$.

- $\mathsf{DelEle}(x, Acc_X, PK)$: *The new accumulated value* $Acc_{X\setminus\{x\}} = Acc_X^{\frac{1}{(x+s)}}$.

The security of the construction (according to definition 20) relies on the N-SDH assumption.

Theorem 2 *([Ngu05]) If the N-SDH assumption holds, then Nguyen's accumulator is secure.*

Proof. (Membership) Assume the adversary $\mathcal{A}$ finds a set $X = \{x_1, \ldots, x_l\}$, $l \leq N$, an element $y \notin X$ and π such that $e(\pi, y) = e(Acc_X, g)$. As $y \notin X$, we have that $P_X(s) = Q(s) \cdot (y + s) + R(s)$ where Q, R are polynomials and $R(s) \neq 0$. This means $g^{\frac{P_X(s)}{(y+s)}} = \pi = g^{Q(s) + \frac{R(s)}{(y+s)}}$. As Q and R are computable from the public data, the Adversary $\mathcal{A}$ can deduce $g^{\frac{1}{(y+s)}}$.

(Non-membership) Similarly, assume that the adversary $\mathcal{A}$ finds a set
$X = \{x_1, \ldots, x_l\}$, and element $y \in X$ and a witness of non-membership $\bar{\pi} = (w_y, u_y)$ such that $u_y \neq 0$. We have $w_y^{(y+s)} = g^{P_X(s)+u_y}$ (i). Given that $y \in X$ we also have $g^{Q(s)(y+s)} = g^{P_X(s)}$ (ii) for some computable polynomial Q. From (i) and (ii) we can deduce $w_y = g^{Q(s) + \frac{u_y}{(y+s)}}$. From this equation the Adversary can compute $g^{\frac{1}{(y+s)}}$. ■

In [CKS09], Camenisch et al. introduced another dynamic accumulator which like Nguyen's construction, is based on bilinear maps. The advantage of Camenisch et al.'s construction is that the cost to update a witness, even though still linear in the size of the set, now depends on the number of group multiplications instead of the number of group exponentiations. The security of their construction relies on the N-Diffie-Hellman Exponent assumption (N-DHE [BBG05]). Though based on a different assumption and used to solve a different problem, in chapter 6, we introduced a CRHF that has some similarities to Camenisch et al. construction: Indeed in both cases, the bilinear map is used to "shift the data structure by a given number of positions in the exponent".

Definition 22 *(N-DHE assumption [BBG05]) Let $\mathcal{A}$ be any PPT algorithm, let $s \xleftarrow{R} \mathbb{Z}_p^*$ and let $g_i = g^{s^i}, i \in [0..2N]$. Assume the adversary is given $(g, g_1, \ldots, g_N, g_{N+2}, \ldots, g_{2N})$. Then we have:*

$$\Pr\left[\,\mathcal{A}(1^\kappa, (g, g_1, \ldots, g_N, g_{N+2}, \ldots, g_{2N})) = g_{N+1}\,\right] < \mathsf{neg}(\kappa)$$

There are two main ideas in Camenisch et al.'s construction:

- First all elements of the set are bound to points in $\{g_i, 1 \leq i \leq N+1\}$ using signed messages of the form $x||i$. Note that this implies that N is a bound on the number of possible insertions made to the set which may be bigger than the size of the set itself.

- The *manager* keeps track of the set of indexes $i \in [N]$ that are *turned on*. That is, if x is bent to some g_i, then i has to be considered. In this case the values of the accumulated set are in a small universe $\{1, \ldots, N\}$.

Construction 4 *(Camenisch et al.'s construction [CKS09])*

- $\mathsf{KeyGen}(1^\kappa, N)$*: Generate a pairing tuple* $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BMGen}(1^\kappa)$*.* $s \overset{R}{\leftarrow} \mathbb{Z}$ *is the trapdoor of the scheme from which the public N-DHE tuple can be derived:* $(g, g_1, g_2, \ldots, g_{N-1},\ g_{N+1}, \ldots, q_{2N})$ *with* $\forall i \in \{0, \ldots, 2N\}\backslash\{N\} : g_i = g^{s^i}$ *where* N *is an upper bound on number of insertions into the set. The initial accumulated value is defined simply as* $Acc_\emptyset = 1_\mathbb{G}$*. Generate a pair of public/private keys* (pk, sk) *for a standard secure signature scheme. Set* $z = e(g_{N+1}, g) = e(g_N, g_1)$*. The public parameter* PK *is defined as*

$$PK = ((p, \mathbb{G}, \mathbb{G}_T, e), (g, g_1, g_2, \ldots, g_q, g_{N+2}, \ldots, q_{2N}), z, pk)$$

  Here the universe is the set $\{1, \ldots, N\}$*. That is the accumulated set* X *will be such that* $X \subseteq \{1, \ldots, N\}$*.*

- $\mathsf{AccVal}(X, Acc_\emptyset, PK)$*: Given a finite set of elements* X *where* $|X| \leq N$ *and the public parameter* PK *return* $Acc_X = \prod_{v \in X} g_{N+1-v}$*.*

- $\mathsf{WitGen}(x, Acc_X, PK)$*:*

  (Membership)[5] *Compute* $w_i = \prod_{j \in X}^{j \neq i} g_{N+1-j+i}$ *where* i *is the index associated with* x*. Then return* $\pi_x = (w_i, \sigma_x)$ *where* $\sigma_x = \mathsf{SSig}(sk, M_x)$ *with* $M_x = x||i$*.*

- $\mathsf{Verify}(x, \pi, Acc_X, PK)$*: (Membership): Parse* π *as* (w_i, σ_x) *Check if* σ_x *is a valid signature that binds* x *to the position* i*. Then check if* $\frac{e(g_i, Acc_X)}{e(g, w_i)} = z$*.*

- $\mathsf{AddEle}(x, Acc_X, PK)$*: Choose a slot* i *that is free, compute the signature of the message that binds* x *and position is that is* $\sigma_x = \mathsf{SSig}(sk, x||i)$*, Update the accumulated value:* $Acc_{X \cup \{i\}} = Acc_X \cdot g_{N+1-i}$ *Compute the current witness relative to* i*:* $w_i = \prod_{j \in X}^{j \neq i} g_{N+1-j+i}$

[5]Non-membership proofs are not described in this work.

- $\mathsf{DelEle}(x, Acc_X, PK)$: [6] *The new accumulated value is* $Acc_{X\setminus\{i\}} = Acc_X / g_{N+1-i}$ *where* i *is the index associated to* x.

Correctness: Let $Acc_X = \prod_{j\in X} g_{N+1-j}$ be the accumulated value. For each $i \in X$, the (updated) witness) is $\pi_i = (w_i, \sigma_x, x)$ with $w_i = \prod_{j\in X}^{j\neq i} g_{N+1-j+i}$. Then,

$$\frac{e(g_i, Acc_X)}{e(g, w_i)} = \frac{e(g,g)^{\sum_{j\in X} s^{N+1-j+i}}}{e(g,g)^{\sum_{j\in X}^{j\neq i} s^{N+1-j+i}}} = e(g,g)^{s^{N+1}} = z$$

Theorem 3 *([CKS09]) Under the* N*-DHE assumption the Camenisch et al.'s accumulator is secure.*

Proof. (Sketch) Let $\mathcal{A}$ be the adversary that breaks the security of the accumulator. Let X be the set of accumulated indexes and $\pi = (w_i, \sigma_i, \hat{g}_i)$ be the witness of the forgery. If $\hat{g}_i \neq g_i$ then we obtained a forgery for the signature scheme. In the other case we have $e(g, Acc_X) = e(g, w_i)z = e(g, w_i)\cdot e(g, g_{N+1})$ from which we can deduce $e(g, \prod_{j\in X} g_{n+1-j+i}) = e(g, w_i \cdot g_{N+1})$. Finally

$$g_{N+1} = \frac{\prod_{i\in X} g_{N+1-j+1}}{w_i}$$

■

COMBINATORIAL ACCUMULATORS [NYB96B, YSL08]. Another technique to design cryptographic accumulators was proposed by Nyberg in [Nyb96b]. This construction considers an accumulator that does not require witness (that is the element, along with the accumulated value is enough for (non)membership testing), and does not rely on trapdoor either. On the flip side, the accumulated value has linear size in $|X|$, the size of the accumulated set and it is not possible to perform deletions (only insertions).

The idea is the following. Let $\oplus$ denote the *xor* operator for bit-strings, that is $x \oplus y = z$ where $\forall i, i \in [|x|] : z_i = x_i \oplus y_i$. To insert x a value into the accumulator, compute its hash value $y = \mathsf{H}(x)$. Then consider y as bit string containing r blocks

[6]The authors of [CKS09] do not mention a DelEle algorithm, but instead they refer to an update algorithm for the witnesses. The formulation presented here, however, is equivalent.

of d bits. Compute a new hash value b of length r such that if the i^{th} block of y is equal to 0^d then $b[i] = 0$ else $b[i] = 1$. Then compute $Acc_X \oplus b$ to obtain the new accumulated value $Acc_{X \cup b}$. To check that $x \in X$, we compute $b = f(\mathsf{H}(x))$ and check against the accumulated value that if $b[i] = 0$, then also that $Acc_X[i] = 0$ to accept $(x \in X)$ If not, it is rejected.

Construction 5 *(Nyberg's accumulator [Nyb96b])*

- $\mathsf{KeyGen}(1^\kappa)$*: Choose a random function* $\mathsf{H} : \{0,1\}^* \to \{0,1\}^l$*, from a collision resistant hash function family* $\mathcal{H}$*. Assume* $N = 2^d$ *is an upper bound on the size of the set* X*. W.l.o.g. we consider that* $l = rd$ *for some* $r \in \mathbb{N}$*.*

  Let $f : \{0,1\}^l \to \{0,1\}^r$ *be the function that given a bit-string* $y[0..l-1]$ *computes the bit string* $b[0..r-1]$ *where* $\forall i : 0 \leq i < r$*:*

  - *If* $y[id..(id+d-1)] = 0^d$ *then* $b[i] = 0$*,*
  - *otherwise* $b[i] = 1$*.*

- $\mathsf{AccVal}(X)$*: Return* $\oplus_{x \in X} f(\mathsf{H}(x))$*.*

- $\mathsf{Verify}(x, Acc_X)$*: Compute* $b_x = f(\mathsf{H}(x))$*.*
  Return valid *if, and only if,* $\forall i : b_x[i] = 0 \Rightarrow Acc_X[i] = 0$*.*

  Note that if $x \in X$*, then we always have* $\mathsf{Verify}(x, Acc_X) = \mathsf{valid}$*. If* $x \notin X$*, there is some probability that* $\mathsf{Verify}(x, Acc_X) = \mathsf{valid}$ *also (false positive).*

- $\mathsf{AddEle}(x, Acc_X, PK, [SK])$*: Return* $Acc_{X \cup \{x\}} = Acc_X \oplus f(\mathsf{H}(x))$*.*

This construction brings some similarities with Bloom filters [Blo70] where membership can be efficiently tested, allowing some false positives for which their rate is related to the size of the index. In Bloom filters, there is a tradeoff between error rate and space occupied by the index: the bigger the space, the lower the error rate. We can see that the security of Nyberg's construction also relies on this tradeoff between space and error rate.

Theorem 4 *([Nyb96b]) Let b_{ij} and c_j be independent binary random variables such that* $\Pr[b_{ij} = 0] = \Pr[(c_j = 0)] = 2^{-d}$, *for* $1 \leq i \leq m$ *and* $1 \leq j \leq r$. *Let* $a = (a_1, \ldots, a_r) = \oplus_{i=1}^{m} b_i$ *where* $b_i = (b_{i1}, \ldots, b_{ir})$. *Then we have that* $\forall j : 1 \leq j \leq r$*:*

$$\Pr[c_j = 0 \Leftrightarrow a_j = 0] = (1 - 2^{-d}(1 - 2^{-d})^m)^r$$

Proof. For each $j, 1 \leq j \leq r$ we have $\Pr[c_j = 0 \wedge a_j = 1] = 2^{-d}(1 - 2^d)^m$. ■

Assuming that H behaves as a random oracle, if we set $m = N$, the probability of producing a forgery is such that $P_{forgery} = \Pr[x \notin X \wedge \mathsf{Verify}(x, Acc_X) = \mathsf{valid}] \leq (1 - 2^{-d}(1 - 2^{-d})^N)^r$, and because $N = 2^d$, it can be rewritten as

$$P_{forgery} \leq (1 - \frac{1}{N}(1 - \frac{1}{N})^N)^r \approx (1 - \frac{1}{eN})^r \approx e^{\frac{-r}{eN}}$$

If we want to bound the probability of forgery by $2^{-\kappa}$, then we must set $l = \frac{N \log N \kappa e}{\log e}$ as $l = rd$ and $N = 2^d$. Thus, the main problem of Nyberg's construction is that the size of the output of the hash function H is $O(\kappa N \log N)$ and yields to an accumulated value of size linear in κN. Note however that the trivial solution to authenticate the set by signing each element would require $\kappa N \log N$ bits, as pointers would be needed to map the signatures to their respective elements.

If we replace e by its numeric value we obtain $P_{forgery} \approx 0.69^{\frac{-r}{N}}$. In [YSL08], the authors improve this bound to $P_{forgery} \approx 0.62^{\frac{-r}{N}}$ by using the following idea: Instead of setting $b_x[i] = 0$ with probability 2^{-d} they choose some random i and set $b_x[i] = 0$ with some probability α. They repeat the operation k times. So now the function f is parametrized by k and α. Finally, by setting suitable values for α and k, one can minimize the probability of forgery. The intuition behind this construction is that in Nyberg's construction, if an element has low hamming weight (many zeros), then the probability of forgery increases. In [YSL08], the minimum Hamming weight of the outputs of f can be controlled.

3.4 Transitive Signatures

In this section we consider a graph that is also signed by a *Source* and we need to enable an efficient mechanism to convince a *Client* that there is a path between two nodes. Note that a trivial solution can be implemented by simply concatenating standard signatures where the message is an edge. In this case the size of a signature grows linearly with the length of the path and may reach $O(N\kappa)$ bits. Another trivial solution consists in signing every path in the graph. This leads to small signatures, but the work of the *Source* may become prohibitive for large N: $O(N^2)$ signature computations.

3.4.1 Undirected Graphs

Transitive signatures is a primitive introduced by Micali and Rivest [MR02] which aims to solve the problem mentioned above. 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). The operational model for transitive signatures is similar to the one of authenticated dictionaries. There is a *signer* who signs edges of the graph. The *signer* is the equivalent of the *Source.* Then a *combiner* (playing a similar role as the *Replica*) will compute signatures for paths without the help of the *signer* (that is only using public information). Finally, a *verifier* (who would be the *Client*) will try to check the validity of path signatures computed by the *combiner.* In their work [MR02], the authors propose an efficient scheme to sign undirected graphs based on the difficulty of computing discrete logarithm for large groups. Here we sketch out the construction. We use a group $\mathbb{G}$ of order p and g, h, two random generators where the discrete logarithm of h in base g is only known to the *signer.* To create a new vertex v_i, the *signer* selects $x_i, y_i \overset{R}{\leftarrow} \mathbb{Z}_p$ and sets $v_i = g^{x_i} h^{y_i}$. Then a message "v_i represents the i^{th} vertex of the graph" is signed using a standard digital signature scheme. To sign an edge (i, j) that is between the i^{th} and j^{th} vertex, the *signer* publishes (without signing) the following quadruple: $\tau_{(i,j)} = (i, j, \alpha_{ij}, \beta_{ij})$ where $\alpha_{ij} = x_i - x_j \bmod p$ and $\beta_{ij} = y_i - y_j \bmod p$. In order to verify that there is an edge between vertices i, j the *verifier* will check that

$v_i = v_j g^{\alpha_{ij}} h^{\beta_{ij}}$. Now, given a signature for edge (i,j) and a signature for edge (j,k), the *combiner* can obtain the signature for the (composed) edge (i,k) as follows:

1. From $\tau_{(i,j)} = (i, j, \alpha_{ij}, \beta_{ij})$ and $\tau_{(j,k)} = (j, k, \alpha_{jk}, \beta_{jk})$, compute $\alpha_{ik} = \alpha_{ij} - \alpha_{jk} = x_i - x_k \mathsf{~mod~} p$ and $\beta_{ik} = \beta_{ij} - \beta_{jk} = y_i - y_k \mathsf{~mod~} p$.

2. Return $\tau_{(i,k)} = (i, k, \alpha_{ik}, \beta_{ik})$.

We can observe that this scheme is such that given a signature for an edge (a,b), it is easy to obtain a signature for the edge (b,a), as $\alpha_{ba} = -\alpha_{ab}$ and $\beta_{ba} = -\beta_{ab}$. This means that this construction only works for *undirected graphs*. The authors left the existence of a transitive signature scheme for directed graph (DTS) as a challenging open question.

We next offer the syntax of a transitive signature scheme.

Definition 23 *(Transitive Signature Scheme, [MR02, Nev08])*
A transitive signature scheme is a tuple TS $= (\mathsf{TSKG}, \mathsf{TSign}, \mathsf{TSComp}, \mathsf{TSVf})$ *where:*

- $\mathsf{TSKG}(1^\kappa)$: *returns a pair of private and 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 a combined signature* $\tau_{(a,c)}$ *on edge* (a,c). *Note that the secret key is not required.*

- $\mathsf{TSVf}((a,b), \tau, tpk)$: *returns* valid *if* τ *is a valid signature for the path* (a,b) *and* $\perp$ *otherwise.*

A transitive signature scheme is correct if both original signatures (those generated honestly with TSign) and combined signatures (those generated honestly with TSComp) do verify correctly with TSVf. Intuitively, a transitive signature scheme is secure if, for any PPT adversary, it is infeasible to compute a signature for a path outside the transitive closure of the graph G.

Definition 24 *(Security of Transitive Signature Schemes, [MR02, Nev08]) Let* TS *be a transitive signature scheme. Consider the following experiment. A PPT adversary*

$\mathcal{A}$ is given the public key tpk of the scheme. $\mathcal{A}$ may ask 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 in the graph G formed by the successive validly signed edges. The advantage of $\mathcal{A}$ is defined by:

$$Adv_{\mathsf{TS}}^{\mathcal{UF}-\mathcal{TS}}(\mathcal{A}, \kappa) = \Pr\left[\begin{array}{c} (a,b) \notin G^{*} \wedge \\ \mathsf{TSVf}((a,b), \tau, tpk) = \mathsf{valid} \end{array}\right]$$

The scheme is said to be secure if we have $Adv_{\mathsf{TS}}^{\mathcal{UF}-\mathcal{TS}}(\mathcal{A}, \kappa) = \mathsf{neg}(\kappa)$ for any PPT adversary $\mathcal{A}$.

Bellare and Neven in [BN05], as well as Shahandashti et al. in [SSM05], introduced new schemes based on bilinear maps (but still for undirected graphs). Hohenberger [Hoh03], showed that the existence of `DTS` implies the existence of abelian groups, for which 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 [JMSW02, BFKW09, BF11].

3.4.2 Directed Trees

The easier problem of transitive signatures for *directed trees*, `TSDT`, was first addressed by Yi [Yi07]. Solutions for this case, even though it is a special kind of directed graph, are still interesting in practice. For example they allow to implement efficient *military chains of command* where the presence of a path between a and b means b must execute orders of a. Yi's construction, based on a special assumption for the RSA cryptosystem, yields signatures of size roughly $N \log(N \log N)$ bits, where N is the number of nodes of the tree. Neven [Nev08], described a simpler solution based only on the existence of standard digital signatures which reduces signature size to $N \log N$ bits.

Note that Yi's [Yi07] as well as Neven's solutions are clearly better than the trivial construction. We remark that both solutions need to maintain the state of the tree to compute new edge signatures (as opposed to the initial definition [MR02]). In

[Xu09], a stateless `TSDT` scheme with constant size signature is proposed, but without security proof. In the following we show how we can improve Neven's construction using cryptographic accumulators in order to reduce the size of a signature from $N \log N$ bits to $O(\kappa)$ bits.

Transitive signature for directed trees using accumulators. We first recall Neven's signature scheme for directed trees. In [Nev08], Neven observes that if the tree is such that the root r remains unchanged, then a trivial solution to authenticate a path in the tree can be implemented simply by storing (and signing, using a standard signature scheme) in each node j the nodes on the path from r to j. Each of these lists will be at most $N \log N$ bits long. Thus, the signature for a path will be $N \log N + \kappa$ bits long, κ being number of bits of the standard signature. Verifying there is a path from i to j will consist of checking that i appears in the list of ancestors of j and that the signature of this list is valid. In order to enable the tree to grow, not only at the bottom, but also at the top (that is allowing changing the root), Neven introduces two lists for each node: one list for the ancestors and one list for the descendants. Using only a constant number of verifications for these lists, this improvement is enough to let the tree grow in both directions while keeping signatures of size $O(N \log N)$ bits.

Construction 6 *(Neven's scheme [Nev08])*

- $\mathsf{TSKG}(1^\kappa)$: *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. $V = \emptyset$:

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

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 $C_a \leftarrow (a, down[a])$ and $C_b \leftarrow (b, up[b])$, 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 $\tau_{(a,b)} \leftarrow (C_a, \sigma_a, C_b, \sigma_b)$.*

- $\mathsf{TSComp}((a,b), \tau_{(a,b)}, (b,c), \tau_{(b,c)}, tpk)$: *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. Otherwise, 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 $\perp$.*

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 schemes introduced in [CL02, Ngu05, CKS09] the edge signature will have constant size. Note that, compared to Definition 18, we modified the parameters for the WitGen algorithm: The combiner has no access to the private key SK of the accumulator scheme and thus must rely on the list to compute the witness. In Chapter 4, we explore more in details this case where the witness is computed by a non-trusted participant like the combiner. Without the private key SK the witness can be computed in linear time in the size of the list. We describe Neven's scheme with accumulators.

Construction 7 *(Neven's scheme [Nev08] using accumulators)*

- $\mathsf{TSKG}(1^\kappa)$: *Returns a pair of private/public keys* (tsk, tpk) *for a standard digital signature scheme* $\mathtt{SSig}$*, and a pair of public and private keys* (PK, SK) *for the accumulator scheme* $\mathtt{Acc}$ *as well. The accumulator value for the empty set is denoted by* $Acc_\emptyset$.

- $\mathsf{TSign}(tsk, a, b)$: *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* $down[\cdot]$*. In the following,* $Acc_{\mathsf{N},dir}$ *where* N *is a node and* $dir \in \{up, down\}$*, will denote the accumulated value associated to the list of descendants (dir = down) and ancestors dir = up for a node* N.

  To sign a new edge we distinguish between the following cases:

  1. $V = \emptyset$:
  $$r \leftarrow a; V \leftarrow V \cup \{a, b\}; E \leftarrow E \cup \{(a, b)\}$$
  $$up[a] \leftarrow \epsilon; down[b] \leftarrow \epsilon; down[a] = b; up[b] \leftarrow a$$
     The accumulated values are computed as follows:
     - $Acc_{a,up} \leftarrow Acc_\emptyset$
     - $Acc_{a,down} \leftarrow \mathsf{AccVal}(\{b\}, Acc_\emptyset, PK)$
     - $Acc_{b,up} \leftarrow \mathsf{AccVal}(\{a\}, Acc_\emptyset, PK)$
     - $Acc_{b,down} \leftarrow Acc_\emptyset$
  2. $a \in V$ *and* $b \notin V$:
  $$V \leftarrow V \cup \{b\}; E \leftarrow E \cup \{(a, b)\}$$
  $$up[b] \leftarrow up[a] || a; down[b] \leftarrow \epsilon$$
     The accumulator values are computed as follows:

- $\circ$ $Acc_{b,up} \leftarrow \mathsf{AddEle}(a, Acc_{a,up}, PK)$
- $\circ$ $Acc_{b,down} \leftarrow Acc_{\emptyset}$

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

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

The accumulator values are updated as follows:

- $\circ$ $Acc_{a,up} \leftarrow Acc_{\emptyset}$
- $\circ$ $Acc_{a,down} \leftarrow \mathsf{AddEle}(b, Acc_{b,down}, PK)$

In all other cases the signer *rejects because the query does not preserve the tree structure of the graph. The* signer *sets $C_a \leftarrow (a, Acc_{a,down})$ and $C_b \leftarrow (b, Acc_{b,up})$, and computes two standard signatures $\sigma_a = \mathsf{SSig}(tsk, C_a)$ and $\sigma_b \leftarrow \mathsf{SSig}(tsk, C_b)$. Then, one of the following case occurs:*

- $\circ$ *$a \in up[b]$. In this case compute $\pi = (a, \mathsf{WitGen}(a, up[b], Acc_{b,up}, PK), Nil)$.*
- $\circ$ *$b \in down[a]$. In this case compute*
  $\pi = (b, \mathsf{WitGen}(b, down[a], Acc_{a,down}, PK), Nil)$.
- $\circ$ *There exists c such that $c \in down[a]$ and $c \in up[b]$. In this case, compute*
  $\pi = (c, \mathsf{WitGen}(c, down[a], Acc_{a,down}, PK), \mathsf{WitGen}(c, up[b], Acc_{b,up}, PK))$.

The transitive signature for the edge (a,b) is the tuple $\tau_{(a,b)} \leftarrow (C_a, \sigma_a, C_b, \sigma_b, \pi)$.

- $\mathsf{TSComp}((a,b), \tau_{(a,b)}, (b,c), \tau_{(b,c)}, tpk)$: *Parse $\tau_{(a,b)}$ as $(C_a, \sigma_a, C_b, \sigma_b, \pi_1)$ and $\tau_{(b,c)}$ as*
  $(C_{b'}, \sigma_{b'}, C_c, \sigma_c, \pi_2)$. If $b \neq b'$, reject. Otherwise, return the composed signature for edge (a,c) as $\tau_{(a,c)} \leftarrow (C_a, \sigma_a, C_c, \sigma_c, \pi)$, where π is computed as in algorithm TSign [7].

[7]Note that in this scheme the algorithm TSComp, run by the *combiner*, is stateful as algorithm TSign. This means that the *combiner* cannot compose edge signatures without knowing the state contrary to Neven's construction [Nev08]. Despite this strong restriction, the scheme is still useful because the private key of the *signer* is not required by the *combiner*. This means that the work of combining edge signatures can be outsourced without exposing the private key. We will face a similar limitation in Chapter 6.

- $\mathsf{TSVf}((a,b),\tau,tpk)$: *Parse* τ *as* $(C_a,\sigma_a,C_b,\sigma_b,\pi)$, *and parse* C_a *as* $(a, Acc_{a,down})$, C_b *as* $(b, Acc_{b,up})$ *and* π *as* (w_1,w_2,w_3). *If* $\mathsf{SVf}(tpk,\sigma_a)=\perp$ *or* $\mathsf{SVf}(tpk,\sigma_b)=\perp$ *return* $\perp$.

  If $w_3 = Nil$ *then consider the following cases:*

  - $w_1 = a$: *return* valid *if, and only if,* $\mathsf{Verify}(a, w_2, Acc_{b,up}, PK) =$ valid.
  - $w_1 = b$: *return* valid *if, and only if,* $\mathsf{Verify}(b, w_2, Acc_{a,down}, PK) =$ valid.

  If $w_3 \neq Nil$ *then return* valid *if, and only if,* $\mathsf{Verify}(w_1, w_2, Acc_{a,down}, PK) =$ valid
  and $\mathsf{Verify}(w_1, w_3, Acc_{b,up}, PK) =$ valid.

  Return $\perp$ *otherwise.*

Proposition 3 *The transitive signature scheme for directed trees described above is secure assuming that the standard digital signature scheme* `SSig` *and the accumulator scheme* `Acc` *are secure. Moreover the size of an edge signature is* $O(\kappa)$, *and the time to compute a signature for an edge – whether directly or by composition – is* $O(N)$, *where* N *is the size of the tree.*

Proof. Assume there exists an adversary $\mathcal{A}$ that breaks the security of the transitive signature scheme. We build the following adversary $\mathcal{B}$ that breaks the security of `SSig` or `Acc`. Adversary $\mathcal{B}$ has access to the signing oracle $\mathcal{O}_{\texttt{SSig}}(\cdot)$ and the accumulator oracle $\mathcal{O}_{\texttt{Acc}}(\cdot)$. Clearly adversary $\mathcal{B}$ can simulate the signer for the transitive signature scheme.

As adversary $\mathcal{A}$ wins, this means that it is able to output $\tau_{(x,y)}$ that passes successfully TSVf but such that there is no path from x to y in the directed tree. Let $\tau = (C_x, \sigma_x^*, C_y, \sigma_y^*, \pi)$, where $C_x = (x, Acc_x^*)$, $C_y = (y, Acc_y^*)$. As the verification procedure passes this means in particular that σ_x^* and σ_y^* are valid signatures on C_x and C_y respectively. If C_x or C_y was not produced by the simulator $\mathcal{B}$, then $\mathcal{B}$ found a forgery for the signature scheme SSig. Henceforth we assume that C_x and C_y have been produced by $\mathcal{B}$ and thus are consistent with the directed tree T formed by the successive queries of adversary $\mathcal{A}$. Note that C_x and C_y may correspond to some older state of the directed tree. This is not an issue as the graph we consider can only grow. In other words, if (x, y) is not a path in the last state of T, then (x, y) will not be a path in some previous version of the tree either.

Moreover, $\tau_{(x,y)}$ is a forgery, thus these three conditions are true.

- $x \notin up[y]$,
- $y \notin down[x]$, and
- there is no z such that $z \in down[x]$ and $z \in up[y]$.

As the verification passes, this means that $\pi = (w_1, w_2, w_3)$ is a forgery relative to the accumulated value for the lists $down[x]$ or $up[y]$. More precisely we have the following two cases:

- $w_3 = Nil$. Then either w_2 is a valid witness of membership for $w_1 = x$ in the set represented by Acc_y^* or w_2 is a valid witness of membership for $w_1 = y$ in the set represented by Acc_x^*.
- $w_3 \neq Nil$. In this case w_2 is a valid witness of membership for $w_1 = z$ in the set represented by Acc_x^* and w_3 is a valid witness of membership for $w_1 = z$ in the set represented by Acc_y^*.

In each of the previous cases, $\mathcal{B}$ has broken the security of the accumulator scheme `Acc`. The time to compute a signature for an edge is bounded by the time to compute the witness for a list of nodes that has at most size $O(N)$. Thus, computing edge signature requires at most $O(N)$ cryptographic operations. ■

In Chapter 6 we will show that we can obtain signatures of the same size, but with fewer cryptographic operations, that is $O(\frac{N}{\kappa})$ instead of $O(N)$.

3.4.3 Optimal Data Authentication from Directed Transitive Signatures

In this section we show how to build an optimal authenticated dictionary (`OAD`) from a directed transitive signature scheme `DTS`.

Theorem 5 *([Cam11])* `DTS` $\Rightarrow$ `OAD`

Related work. In [PTT10] it is shown that multi-linear forms [BS02] can be used to build an optimal dictionary in the two-party model. As no concrete implementation for multi-linear forms is known to the date, an impossibility result for an `OAD` in the two-party model would invalidate the complexity assumption for multi-linear forms which was proposed in [PTT10].

Optimal Authenticated Dictionaries. We need now to define what the expression *optimal authenticated dictionary* means. The intuition is simply that the time to perform an operation in an optimal authenticated dictionary should not be much larger than the time required to run a non-authenticated dictionary. However, precision is required as the size of an authenticated dictionary is related to the security parameter, that is $N \leq q(\kappa)$, where N is the size of the authenticated dictionary and $q(\cdot)$ is a polynomial.

Our definition for authenticated dictionary differs slightly from the one of [PTT10] because it explicitly mentions the security parameter κ and allows KeyGen algorithm to run in polynomial time in N. However both definitions are, in essence, equivalent.

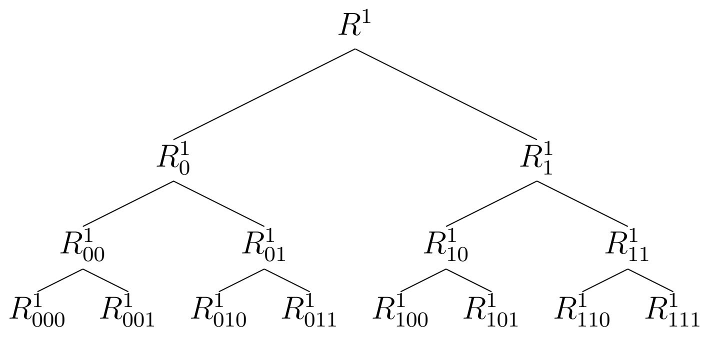
Definition 25 *Let N be the size of the dictionary. An authenticated dictionary* `AuthDict` *scheme is optimal if and only if:*

- *It is correct and secure under definitions 9 and 10 respectively.*
- *There exist two polynomials $m(\cdot), l(\cdot)$ such that:*
    - KeyGen *runs in $O(m(\kappa) \cdot l(N))$ time.*
    - Verify *runs in $O(m(\kappa))$ time and the proof π has size $O(\kappa) = O(1)$ w.r.t N.*
    - ProofGen *runs in $O(m(\kappa) \cdot \log(N))$ time, and involves at most $O(\log N)$ accesses to the dictionary.*
    - Update *runs in $O(m(\kappa) \cdot \log(N))$, and involves at most $O(\log N)$ accesses to the dictionary.*

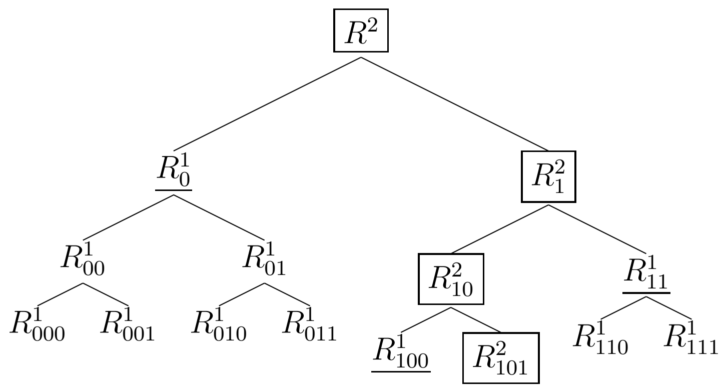
Let us comment and justify the definition. As $N \leq q(\kappa)$, for some polynomial $q(\cdot)$, an authenticated data structure will be slower than the un-authenticated data structure only by a factor of $m(\kappa)$ which is fixed and does not depend on N. Moreover the number of accesses to the dictionary will remain the same. In practice, $m(\kappa)$ is the time to perform some cryptographic operation (hash, signature,...), so it is reasonable to consider that $m(\kappa) \ll N$. Note that the trivial construction like the Merkle tree of [MRK03] does not fit this definition. The reason is that although the algorithms $\mathsf{Verify}, \mathsf{ProofGen}, \mathsf{Update}$ require $O(m(\kappa))$ time where $m(\kappa) = \kappa^2$, the size of the proof is $O(\kappa^2) = \omega(\kappa)$ and the number of accesses to the data structure to compute a proof is $O(\kappa) = \omega(\log(N))$.

Our construction. The idea of our construction (see Figure 3.2) is to build a balanced binary tree of $N = 2^d$ leaves, where each leaf $L_i : 0 \leq i \leq N-1$ will store the value $\mathcal{D}[i]$ of the dictionary. The edges will be signed by the $\mathtt{DTS}$ scheme and the root R^h will "represent" $\mathcal{D}$ at some point of time h. To perform an update for the index j, which corresponds to the leaf L_j^h (that is the leaf at position j reachable from root R^h) a new root R^{h+1} is created. Along with this new root, a parallel path P^{h+1} to the path $P^h = R^h_{[b_0 b_1 \ldots b_{d-1}]}$ (i.e. the path from the old root R^h to the leaf L_j^h) is created. That is $P^{h+1} = R^{h+1}_{[c_0 c_1 \ldots c_{d-1}]}$ where $\forall i : 0 \leq i \leq d-1 : c_i = b_i$.

Each internal node in P^{h+1} is assigned a random label and the leaf $R^{h+1}_{c_0 c_1 \ldots c_d} = L_j^{h+1}$ will contain the new value v such that $\mathcal{D}[j] = v$. Then, this new path will be linked to the other nodes of the previous tree to maintain consistency of the dictionary, in the sense that, except for the index $k = j$, all the leaves L_k^h must remain reachable from the new root R^{h+1}. This is done by connecting each internal node of the new path to the corresponding sibling of the node belonging to the old path. Again, every new edge is signed using a transitive signature scheme. So now, proving that $\mathcal{D}[i] = v$ is equivalent to exhibiting a signature for the path from the current root R^{h+1} to a leaf L_i^{h+1} such that its internal value, $val(L_i^{h+1})$ is equal to v. We can then note that the time to update the balanced binary tree requires $O(\log N)$ signature computations and the proof consists only in a single signature of a path. The security of the construction is based on the fact that the only directed path from a root R^h to



$$\begin{aligned}
\mathcal{D}[0] &= val(R^1_{000})\\
\mathcal{D}[1] &= val(R^1_{001})\\
\mathcal{D}[2] &= val(R^1_{010})\\
\mathcal{D}[3] &= val(R^1_{011})\\
\mathcal{D}[4] &= val(R^1_{100})\\
\mathcal{D}[5] &= val(R^1_{101})\\
\mathcal{D}[6] &= val(R^1_{110})\\
\mathcal{D}[7] &= val(R^1_{111})
\end{aligned}$$



$$\begin{gathered}
\mathcal{D}[0] = val(R^1_{000})\\
\mathcal{D}[1] = val(R^1_{001})\\
\mathcal{D}[2] = val(R^1_{010})\\
\mathcal{D}[3] = val(R^1_{011})\\
\mathcal{D}[4] = val(R^1_{100})\\
\boxed{\mathcal{D}[5] = val(R^2_{101})}\\
\mathcal{D}[6] = val(R^1_{110})\\
\mathcal{D}[7] = val(R^1_{111})
\end{gathered}$$

Figure 3.2: **Update of the dictionary $\mathcal{D}$ using a `DTS` scheme.** The tree at the top represents the state of the dictionary at time $h = 1$. In order to set $\mathcal{D}[5] = v$ we build a new tree (at the bottom) that will represent the state of the dictionary at time $h = 2$. We first create a new root R^2 and then a path (boxed nodes) $(R^2, R^2_1, R^2_{10}, R^2_{101})$ which will replace the old path $(R^1, R^1_1, R^1_{10}, R^1_{101})$. Node R^2_{101} is a leaf that will host the new value of the dictionary in position $5 = (101)_2$. This means that $\mathcal{D}[5] = val(R^2_{101}) = v$. In order to keep all the others leaves reachable from the new root R^2, it is necessary to create (and sign) the edges between the nodes of the new path (in boxes) and its siblings (nodes underlined). Thus the new edges are $(R^2, R^2_1), (R_2, R^1_0), (R^2_1, R^2_{10}), (R^2_1, R^1_{11}), (R^2_{10}, R^2_{101}), (R^2_{10}, R^1_{100})$.

a given leaf L_i^h will always correspond to the current value of the dictionary at time h and index i.

In the following, we consider a graph where each node N is assigned a label $\ell(\mathsf{N})$. This graph is the union of balanced binary trees such that:

- The label of an internal node N is $\ell(\mathsf{N}) = (pos, r)$ where pos refers to the position of the node and r is a random value. The position of the node is encoded by a sequence of bits $b_0 b_1 \dots b_l$ that represents the path from the root to the node N as described in Section 2.1. In the following we will denote the node N by N_{pos}.
- The label of a leaf L is set to $\ell(L) = (i, v, r)$. Value i corresponds to the position of the leaf L at the last level starting from the left. If we consider the position pos of a node mentioned above, we have that $i = (pos)_2$. In the following we will denote the node L by L_i. The value v corresponds to the content of the dictionary we are modeling in position i, that is $\mathcal{D}[i] = v$. The value r is random as for the internal nodes.

Finally, all the random values are κ-bit long, where κ is the security parameter.

We introduce now the construction of an optimal authenticated dictionary that uses directed transitive signatures.

Construction 8 *(Authenticated Dictionary from* `DTS`*) Let* $\texttt{AuthDict}_{\texttt{DTS}}$ *be the authenticated dictionary defined by the following algorithms.*

- $\mathsf{KeyGen}(1^\kappa, N)$*: Generates the parameters* (sk, pk) *for a standard signature scheme* `SSig` *and the parameters* (tsk, tpk) *for the a directed transitive signature scheme* `DTS`*. Then we set* $SK = (sk, tsk)$ *and* $PK = (pk, tpk)$*. Let* N *be the size of the dictionary* $\mathcal{D}$*, and assume w.l.o.g. that* N *is a power of 2. Let* $h \in \mathbb{N}$ *be the number of updates made to the dictionary. Set* $h = 1$*. Build a balanced binary tree* $\mathfrak{m}_0$ *where each leaf* L_i^h *at position* $0 \leq i \leq N-1$ *has the label* $\ell(L_i^h) = (i, v_i^h, r_i^h)$ [8]*. Every internal node* N_{pos}^h *is filled with label*

[8]Indeed it would be sufficient to handle vertices labels of the form (i, v_i^h, c_i^h) where c_i^h is a counter as to guarantee that all labels are distinct. Note that this method would make the scheme stateful and it also leaks some information about the history of updates of the dictionary.

$\ell(\mathsf{N}^h_{pos}) = (pos, r^h_{pos})$. *Sign the labels for every leaf and internal node using algorithm* SSig. *Then, sign the tree with the* `DTS` *scheme using the random values as the identifiers of the nodes. Let* $R = N^h_\epsilon$ *be the root of the tree. Publish the root node label along with the number of updates, that is* $\hat{\mathfrak{m}}_0 = (\ell(N^h_\epsilon), h) = ((\epsilon, r^1_\epsilon), 1)$ *as the short value (state)representing the dictionary.*

- ProofGen$(i, \mathfrak{m}, PK)$: *Let* h *be the current number of updates of the dictionary. Using* $\mathfrak{m}$, *compute the transitive signature* τ *for the path* $R^h \to L^h_i$ *where* R *is the root with label* (ϵ, r^h_ϵ) *of the tree and* $\ell(L^h_i) = (i, v^h_i, r^h_i)$ *is the label at the* i^{th} *leaf. Return* (σ, τ, r^h_i) *where* $\tau =$ TSign$(tsk, r^h_\epsilon, r^h_i)$ *and* $\sigma =$ SSig(sk, M), *a standard signature on message* $M = \ell(L^h_i)$.

- Verify$(i, v, \pi, \hat{\mathfrak{m}}, PK)$: *Parse* π *as* (σ, τ, r), *then extract from* $\hat{\mathfrak{m}}$ *the number of updates* h*,and the label of root node* R, $\ell(R) = (\epsilon, r^h_\epsilon)$.

  Verify that σ *is the signature of message* $M = (i, v, r)$ *with public key* pk. *Then check that* τ *is a valid signature for the path* $R \to L_i$ *with public key* tpk, *where* R *and* L_i *are identified by* r^h_ϵ *and* r *respectively, that is check that* TSVf$((r^h_\epsilon, r), \tau, tpk)$ *returns* valid.

- Update$(i, v, \mathfrak{m}, PK, SK)$: *Let* h *be the current number of updates.*

  - *Create a new leaf* L^{h+1}_i *with label* $\ell(L^{h+1}_i) = (i, v, r)$ *for a random value* r. *Compute the signature* $\sigma_{L^{h+1}_i} =$ SSig$(sk, \ell(L^{h+1}_i))$.
  - *Let* $R_{[b_1 b_2 \ldots b_d]}$ *be the path from the root* $R_h \in G$ *(the graph corresponding to* $\mathfrak{m}$*) to leaf* L^h_i. *Create a new root node* R^{h+1} *and new intermediate nodes* $R^{h+1}_{b_0}, R^{h+1}_{b_0 b_1}, \ldots, R^{h+1}_{b_0 \ldots b_d - 1} = L^{h+1}_i$.
    The edges $(R^{h+1}, R^{h+1}_{b_0})$, $(R^{h+1}_{b_0}, R^{h+1}_{b_0 b_1})$, $(R^{h+1}_{b_0 b_1}, R^{h+1}_{b_0 b_1 b_2})$, $\ldots$, $(R^{h+1}_{b_0 b_1 \ldots b_{d-2}}, R^{h+1}_{b_0 b_1 b_2 \ldots b_{d-1}})$ *are signed too using the algorithm* TSign.
  - *Then the nodes of this path are connected (by signing edges) to the siblings of the old path that has been replaced. That is, for every* $0 \le i \le d-1$ *sign edges* $(R^{h+1}_{b_0 \ldots b_i}, R^h_{b_0 \ldots b_i (1 - b_{i+1})})$ *and also* $(R^{h+1}, R^h_{1 - b_0})$.

◦ *Publish the new value* $\hat{\mathfrak{m}}_{after} = (\ell(R^{h+1}), h+1)$ *as the new state of the dictionary.*

To prove correctness and the security of our construction we need the following lemma.

Lemma 2 *Let* $\{R^h : h \in \mathbb{N}\}$ *denote the sequence of root nodes generated by the consecutive executions of the* Update *algorithm. Then, with probability at least* $P(q) = 1 - \frac{q^2}{2^{\kappa+1}}$*, the two following properties hold:*

1. $\forall h \in \mathbb{N}$, $(R^h)^*$ *is a directed balanced binary tree with* N *leaves* $(L_0^h, L_1^h, ..., L_{N-1}^h)$,
2. *if* D^h *denotes the state of the dictionary after the same sequence of updates, then* $\forall i : 0 \le i \le N-1 : \mathcal{D}^h[i] = val(L_i^h)$,

where q *is the number of graph nodes created during the successive updates.*

Proof. We assume first, that every node is filled with different values. The proof is by induction on h. For $h = 1$ the claim is verified. Assume that for any $h \in \mathbb{N}$, R_h^* is a directed balanced binary tree with N leaves and that $\forall i : 0 \le i \le N-1 : \mathcal{D}^h[i] = val(L_i^h)$. After the update $(R^{h+1})^*$ is also a directed binary tree because it is formed by the new nodes of the path $R^{h+1}_{[b_1...b_d]}$ that are connected to the nodes of $(R^h)^*$ which are by induction roots of balanced directed binary trees of depth $d-1, d-2, ..., 0$ respectively. Let i be the index of the dictionary that is updated. Then we can also see that $\forall j : 0 \le j \le N-1 \wedge j \ne i : \mathcal{D}^h[j] = val(L_j^{h+1})$ as the only new nodes that are reachable from R^{h+1} are those on the new path $R^{h+1}_{b_1...b_d}$ where $i = \Sigma_{j=0}^{d-1} b_{d-j} 2^j$. The other nodes, including the leaves (except L_i^{h+1}) are still reachable from R^h. So we have that the set of leaves that are reachable from R^{h+1} is formed by the leaves reachable from R^h except for L_i^h that is replaced by the new leaf L_i^{h+1}. Thus, if $(R^h)^*$ was the tree such that $\forall i, 0 \le i \le N-1 : \mathcal{D}^h[i] = val(L_i^h)$, we can deduce that the leaves of $(R^{h+1})^*$ represent $\mathcal{D}^{h+1}$.

These two properties hold as long as all values in each node of the graph $G = G_1 \cup G_2 \cup \cdots \cup G_{h+1}$ are different. As by construction these values are chosen randomly in a universe of size 2^κ, we have that $P(q)$, the probability that every two values are different after q nodes creation, is such that $1 - P(q) \leq \sum_{i=1}^{q} \frac{1}{2^\kappa} \leq \frac{q^2}{2^{\kappa+1}}$. Thus both properties are true with probability $P(q) \geq 1 - \frac{q^2}{2^{\kappa+1}}$.

Note finally that the fact that the graph $G = G_1 \cup G_2 \cup \cdots G_{h+1}$ is directed is essential as otherwise every node would be reachable from any root R^j, thus invalidating the lemma. ■

From the lemma we can directly deduce that

Proposition 4 *The authenticated dictionary* $\mathtt{AuthDict}_{\mathtt{DTS}}$ *is correct.*

Next we prove the security of our scheme.

Proposition 5 *Let $\mathcal{A}$ be the adversary for the* `AuthDict`*, q_{AD} the number of queries made to the oracle $\mathcal{O}_{AD}(\cdot)$ by $\mathcal{A}$, and N the size of the dictionary. The advantage of adversary $\mathcal{A}$ is bounded by*

$$\begin{aligned} Adv_{\mathtt{AuthDict}}^{\mathcal{UF-AD}}(\mathcal{A}, \kappa) \leq\ & Adv_{\mathtt{SSig}}^{\mathcal{UF-CMA}}(\mathcal{B}_1, \kappa) + Adv_{\mathtt{DTS}}^{\mathcal{UF-TS}}(\mathcal{B}_2, \kappa) \\ & + \frac{(2N-1+q_{AD}(\log(N)+1))^2}{2^{\kappa+1}} \end{aligned}$$

where $\mathcal{B}_1$ and $\mathcal{B}_2$ are the best adversaries for the $\mathcal{UF-CMA}$ and the $\mathcal{UF-TS}$ security games respectively.

Proof. Assume that there exists a PPT adversary $\mathcal{A}$ that breaks our scheme, then we consider the following sequence of games.

Game 0. This is the attack game of definition 24. Let $\hat{\mathfrak{m}}$ be the short state of the dictionary at the end of the game, and let (i, v, π) the values computed by $\mathcal{A}$ such that $\mathsf{Verify}(i, v, \pi, \hat{\mathfrak{m}}, PK)$ returns valid. Let h be the number of updates performed on the dictionary and $G = G_1 \cup G_2 \cup \cdots G_h$ the union of directed trees that correspond to the sequence of states of the dictionary. We define S_0 to be the event that adversary wins.

Game 1. This game is identical to the previous one except that adversary $\mathcal{A}$ is not allowed to output a standard signature σ on a message that does not correspond to a node label (computed by the signer).

Let S_1 be the event that adversary adversary $\mathcal{A}$ wins. Applying the difference lemma [Sho04], we get that $|\Pr[S_0] - \Pr[S_1]| \leq \epsilon_1$ where $\epsilon_1 = Adv_{\mathtt{SSig}}^{\mathcal{UF-CMA}}(\mathcal{B}_1, \kappa)$ is an upper bound on the probability to break the standard signature scheme SSig for any adversary $\mathcal{B}_1$.

Game 2. In this game $\mathcal{A}$ is only allowed to compute signature for edges that are in the graph G. Let S_2 be the event that adversary $\mathcal{A}$ wins. Applying lemma 2, we have that $\Pr[S_2] \leq \frac{q^2}{2^{\kappa+1}}$ where q is the size of G. Moreover we have that $|\Pr[S_1] - \Pr[S_2]| \leq \epsilon_2$ where $\epsilon_2 = Adv_{\mathtt{DTS}}^{\mathcal{UF}-\mathcal{TS}}(\mathcal{B}_2, \kappa)$ is an upper bound on the probability to break security of the transitive signature $\mathtt{DTS}$ for any adversary $\mathcal{B}_2$.

Combining the three games we can bound the advantage of adversary $\mathcal{A}$ as follows. First we have that $Adv_{\mathtt{AuthDict}}^{\mathcal{UF}-\mathcal{AD}}(\mathcal{A}, \kappa) = \Pr[S_0]$. Secondly we have that:

$$\begin{array}{rcl} |\Pr[S_0] - \Pr[S_2]| & \leq & |\Pr[S_0] - \Pr[S_1]| + |\Pr[S_1] - \Pr[S_2]| \\ |\Pr[S_0] - \Pr[S_2]| & \leq & Adv_{\mathtt{SSig}}^{\mathcal{UF}-\mathcal{CMA}}(\mathcal{B}_1, \kappa) + Adv_{\mathtt{DTS}}^{\mathcal{UF}-\mathcal{TS}}(\mathcal{B}_2, \kappa) \end{array}$$

As $\Pr[S_0] - \Pr[S_2] \leq |\Pr[S_0] - \Pr[S_2]|$, we can deduce that:

$$\begin{array}{rcl} \Pr[S_0] & \leq & |\Pr[S_0] - \Pr[S_2]| + \Pr[S_2] \\ Adv_{\mathtt{AuthDict}}^{\mathcal{UF}-\mathcal{AD}}(\mathcal{A}, \kappa) & \leq & Adv_{\mathtt{SSig}}^{\mathcal{UF}-\mathcal{CMA}}(\mathcal{B}_1, \kappa) + Adv_{\mathtt{DTS}}^{\mathcal{UF}-\mathcal{TS}}(\mathcal{B}_2, \kappa) + \frac{q^2}{2^{\kappa+1}} \end{array}$$

Finally we have that $q = 2N - 1 + q_{AD}(\log(N) + 1)$, as the initial tree contains $2N - 1$ nodes and each update of the dictionary triggers the creation of $\log(N) + 1$ nodes. By replacing q with $2N - 1 + q_{AD}(\log(N) + 1)$ we obtain the desired conclusion. $\blacksquare$

With respect to the complexity, we can observe that the update algorithm requires $O(\log N)$ signature computations of the $\mathtt{DTS}$ scheme, the proof computation consists in combining $O(\log N)$ $\mathtt{DTS}$ signatures and verifying the value of a dictionary at some index requires a $\mathtt{DTS}$ signature verification. Thus, if $\mathtt{DTS}$ is a transitive signature scheme for directed graphs, the authenticated dictionary is optimal and theorem 5 is proved.

3.5 Summary

In this chapter we introduced the problem of dictionary authentication and several techniques to implement such cryptographic data structures.

The main open problem related to authenticated dictionaries involves the following computational tradeoffs (see table 3.1): For any $\lambda \geq 1$ we have that the time (in cryptographic operations) required by the *Replica* to compute a proof is $O(\lambda N^{1/\lambda})$ while the time to verify a witness is $O(\lambda)$. Improving such tradeoff[9] is an open question. In particular, no concrete[10] construction exists for implementing optimal authenticated dictionaries where the time to compute (or update) the proofs for the *Replica* would be $O(\log N)$ and the time to verify a proof for the *Client* would be $O(1)$.

We explored then useful techniques to build authenticated dictionaries. First, with incremental hashing [BGG94] we observe that the time to update the state can be made constant, however proofs are linear in the size of the dictionary.

Cryptographic accumulators invented by Benaloh and De Mare [BdM93] are a powerful primitive to improve the efficiency of a variety of protocols and schemes. Initial security definitions only considered static sets [BdM93, BP97] then dynamic updates [CL02] have been proposed. We note that these definitions do not handle the case (at least explicitly) where the *manager* is corrupted. In Chapter 4 we propose a security definition and a practical construction based on (standard) CRHF that prevent a malicious *manager* to forge witnesses. Moreover cryptographic accumulators are a natural tool to design authenticated dictionaries. In particular, Camenisch and Lysyanskaya's construction allows the design of an authenticated dictionary which enables the update of the proofs. In their survey on accumulators [FN02], Fazio and Nicolosi pointed out that, in Camenisch and Lysyanskaya's construction, the time to recompute these proofs (witnesses) once the accumulated set has been modified was proportional to m, the number of changes of the accumulated set. This raised a natural question: *"Is it possible to construct dynamic accumulators in which the update of several witnesses can be performed in constant (independently of m) time?"* We show in Chapter 5 that accumulator with batch update do *not* exist.

[9]Papamanthou et al.[PTT08] manage to reduce the time to compute a proof by a factor of $\log^{\alpha\lambda-1} N$ for some $\alpha > 1/\lambda$, but the complexity analysis considers *expected* time.

[10]Papamanthou et al. [PTT10] propose some partial solution, but the construction relies on the existence of multi-near forms for which no instantiation is known to the date, and the construction only works in the two-party model. Our construction based on `DTS` works in the three-party model and could also be adapted to work in the two-party model.

Chapter 4

Strong Accumulators from Collision-Resistant Hashing

Contents

4.1 Introduction

In this chapter we revisit the security model for dynamic accumulators: Our construction is secure even in the presence of a corrupted manager. The security of our scheme relies only on the existence of CRHF and the availability of a secure public broadcast channel [CGS97] to ensure that every participant “sees” the same public values.

Our Approach. Our new accumulator scheme is based on hash trees similar to those used in the design of digital time-stamping systems [BH92, BdM91]. Recall that in hash trees, values are associated to leaves of a binary tree. The values of sibling nodes are hashed in order to compute the value associated to their parent node, and so on and so forth, until a value for the root of the tree is obtained. The tree's root value is defined as the accumulator of the set of values associated to the leaves of the tree. We cannot directly use hash trees to obtain the functionality of universal and dynamic accumulators. Indeed, we need to add and delete elements from the accumulated set (tree node values if using hash trees) while at the same time be able to produce non-membership proofs. We solve this last issue using a trick due to Kocher [Koc98]; instead of associating values to the tree's leaves, we associate a pair of consecutive accumulated set elements. To prove that an element x is not in the accumulated set now amounts to showing that a pair (x_α, x_β), where $x_\alpha < x < x_\beta$, belongs to the tree.

A drawback of using a hash tree based scheme is that the size of witnesses and the update time is logarithmic in the number of values accumulated. In contrast, witnesses and updates can be computed in constant time in RSA modular exponentiation based schemes [CL02, BdM93, BP97, LLX07]. We believe, nonetheless, that this problem may in fact not exist for reasonable set sizes – a claim that we will later support.

Overall, the main advantages of our scheme in comparison to the one by Li et al. [LLX07] are: (1) the accumulator manager need not be trusted, and (2) since we only assume the existence of cryptographic hash functions[1] as opposed to the Strong RSA Assumption, the underlying security assumption is (arguably) weaker (Indeed, collision-resistance can be based on the intractability of factoring or computing discrete logarithms [Dam88] while Strong-RSA is likely to be a stronger assumption than factoring [BV98].).

[1]Note that using standard collision-resistant hash functions implies better performance for cryptographic operations compared to RSA modular exponentiation. However, hash-based linking schemes do not allow witnesses of constant size [TT02]. Thus, depending on the context, using constructions based on a stronger assumption than the existence of collision-resistant hash functions may be justified.

Our contributions. Our contribution is threefold. First, we strengthen the basic definition of universal accumulators by allowing an adversary to corrupt the accumulator manager. This gives rise to the notion of *strong universal accumulators*. Second, we show how to build strong universal accumulators using only collision-resistant hash functions and a variant of Merkle trees where the size of the set that is hashed is not bounded. We call this new data-structure *unbounded Merkle trees*. Our construction has interesting properties of its own. As in [CL02, LLX07], we use auxiliary information to compute the (non)membership witness, but this information (called *memory*) need not to be kept private, and does not allow an adversary to prove inconsistent statements about the accumulated set. Indeed, the construction provides almost the same functionality as the (dynamic) universal accumulators described in [LLX07], namely:

- All the elements of the set are accumulated in one short value.
- It is possible to add and remove elements from the accumulated set.
- For every element of the input space there exists a witness that proves whether the element has been accumulated or not.

Our last contribution is showing how to apply strong universal accumulators to solve a multi-party computational problem of practical relevance which we name the *e-Invoice Factoring Problem*. Solving this problem was indeed the original motivation that gave rise to this work.

4.2 Definitions and Notations

Syntax. We formally define the syntax of a strong universal accumulator scheme (with memory). Our definition differs from that of Li et al. [LLX07] as we consider an algorithm to verify if the accumulator value has been updated correctly (by adding or deleting a certain element), and we are not interested in hiding the order in which the elements are inserted into the accumulated set.

Definition 26 (Strong Universal Accumulators with Memory) *Let U be a set of values. A strong universal accumulator scheme (with memory) for universe U is a tuple* $\texttt{Acc} = (\mathsf{KeyGen}, \mathsf{WitGen}, \mathsf{Verify}, \mathsf{Update}, \mathsf{CheckUpdate})$ *where:*

- $\mathsf{KeyGen}(\zeta)$ *is a randomized algorithm which on input some initialization parameter ζ, outputs a public data structure $\mathfrak{m}_0$ (also called the memory), a creation witness w, and an initial accumulator value Acc_0 which is in the set $Y = \{0,1\}^\kappa$. Value ζ is assumed to include at least a security parameter $\kappa \in \mathbb{N}$ in unary, but it may also include some optional system-wide parameters possibly generated by a trusted initialization process. An empty set $X \subseteq U$ is associated to the execution of the scheme, and in particular, to Acc_0. Both the accumulator value Acc_0 and the memory $\mathfrak{m}_0$ will be typically held and updated by the accumulator manager.*

- $\mathsf{Update}_{\texttt{op}}(\zeta, x, Acc_{before}, \mathfrak{m}_{before})$ *is a randomized algorithm that updates the accumulator value by either adding an element ($\texttt{op} = \texttt{add}$) to or removing an element ($\texttt{op} = \texttt{del}$) from the accumulated set. The algorithm takes an element $x \in U$, an accumulator and memory pair $(Acc_{before}, \mathfrak{m}_{before})$, and outputs an updated accumulator, a memory pair $(Acc_{after}, \mathfrak{m}_{after})$, and an update witness $w_{\texttt{op}} = (w, \texttt{op})$.*

- $\mathsf{CheckUpdate}(\zeta, x, Acc_{before}, Acc_{after}, w_{\texttt{op}})$ *is a randomized algorithm that takes as input a pair of accumulator values $(Acc_{before}, Acc_{after})$, a value $x \in U \cup \{\perp\}$, and an update witness $w_{\texttt{op}} = (w, \texttt{op})$ where $\texttt{op} \in \{\texttt{add}, \texttt{del}, \texttt{crt}\}$, and returns a bit b. Typically, this algorithm will be executed by parties other than the accumulator manager in order to verify the correct update of the accumulator by the manager. If $x = \perp$, $\texttt{op} = \texttt{crt}$, and $b = 1$, then $w_{\texttt{op}}$ is deemed a valid creation witness of the accumulated set $X = \emptyset$. If $b = 1$, $w_{\texttt{op}}$ is deemed a valid witness that the update operation (for $\texttt{op} \in \{\texttt{add}, \texttt{del}\}$) which replaced Acc_{before} with Acc_{after} as the accumulator value, was valid. Otherwise, $w_{\texttt{op}}$ is deemed invalid for the given accumulator pair.*

- $\mathsf{WitGen}(\zeta, x, \mathfrak{m})$ *is a randomized algorithm that takes as input* $x \in U$ *and memory* $\mathfrak{m}$*, and outputs a witness of membership* w *if* $x \in X$ *(*x *has been accumulated) or a witness of non-membership* w' *if* $x \notin X$*.*

- $\mathsf{Verify}(\zeta, x, w, Acc)$ *is a randomized algorithm which on input a value* $x \in U$*, a witness* w *and the accumulator value* $Acc \in Y$ *outputs a bit* 1 *if* w *is deemed a valid witness that* $x \in X$*, outputs* 0 *if* w *is deemed a valid witness that* $x \notin X$*, or outputs the special symbol* $\perp$ *if* w *is not a valid witness of either statement.*

All the above algorithms are supposed to have complexity polynomial in the security parameter κ*.*

In the above definition, memory $\mathfrak{m}$ is a public data structure which is computed from set X. Although public, this structure only needs to be maintained (stored) by the accumulator manager who requires it to update the accumulator, and to generate membership and non-membership witnesses. In particular, this memory is *not* used to verify correct accumulator updates nor to check the validity of (non)membership witnesses.

Strong universal accumulators with memory as defined above are intended for use in a multi-party protocol setting where procedures KeyGen, WitGen, and $\mathsf{Update}_{\mathsf{op}}$ are executed by a manager and Verify and $\mathsf{CheckUpdate}$ by the other participants of the multi-party protocol.

Security. Universal accumulators, as defined in [LLX07], satisfy a basic consistency property: It must be unfeasible to find both a valid membership witness and a valid non-membership witness for the same value $x \in U$. As mentioned there, this is equivalent to saying that given $X \subseteq U$ it is computationally hard to find $x \in X$ that has a valid non-membership witness or to find $x \in U \backslash X$ that has a valid membership witness.

In order to be able to cope with malicious accumulator managers, we first need to guarantee that the accumulator value is consistent with the elements supposedly added and removed by the manager. We need to introduce our main security notion for accumulators.

We adapt the security definition in [LLX07] as follows. First, we let the adversary control the computations of the accumulated values and the witnesses. However, we restrict it so that it must choose a pair (Acc, X) for which there exists a sequence of valid addition and delete operations yielding set X which accumulated value is Acc. This last restriction can be justified by noticing that, in the scenario we consider, parties other than the accumulator manager can externally verify the correctness of each update operation by using the $\mathsf{CheckUpdate}$ algorithm. Finally, to capture most setup assumptions, we parametrize the security definition with the following notion.

Definition 27 *An oracle Ω is an initialization procedure that, if given a security parameter $\kappa \in \mathbb{N}$ in unary generates a parameter $\zeta = (\zeta_0, 1^\kappa)$ where ζ_0 is of length polynomial in κ, chosen at random from the uniform distribution and is publicly available. Invoking Ω will be assumed to take a single time step.*

The initialization procedure will be used to model a setup process that is not under adversarial control. Clearly, the case of no setup assumptions corresponds to the special case when $\Omega(1^\kappa) = 1^\kappa$.

In the following definition, we consider a *consistent sequence of updates*, which means that an element can be inserted (resp. deleted) if it does not belong (resp. belongs) the set that is updated.

Definition 28 (Security of Strong Universal Accumulators with Memory) *Let* $\mathtt{Acc}$ *be a strong universal accumulator scheme (with memory) for universe U, $\kappa \in \mathbb{N}$ be the security parameter, and Ω be an initialization procedure that returns ζ. Let $\mathcal{A}$ be a PPT algorithm, $q \in \mathbb{N}$ a polynomial number in κ, we define the advantage for the adversary $\mathcal{A}$ as follows:*

$$Adv(\mathcal{A}) = \Pr\left[\begin{array}{c} (Acc_i, x_i, w_i, \mathsf{op}_i)_{i\in[q]}, x, w \leftarrow \mathcal{A}(\zeta) : \\ \forall i \in [q] : \mathsf{CheckUpdate}(\zeta, x_i, Acc_{i-1}, Acc_i, (w_i, \mathsf{op}_i)) = 1 \wedge \\ ((x \in X \wedge \mathsf{Verify}(\zeta, x, w, Acc_q) = 0) \vee \\ (x \notin X \wedge \mathsf{Verify}(\zeta, x, w, Acc_q) = 1)) \end{array}\right]$$

where $(x_i, \mathsf{op}_i)_{i\in[q]}$ is a consistent sequence of updates, $(w_i)_{i\in[q]}$ are the update proofs, X is the set resulting from the successives updates, that is $X = (\cup_{\{j:op[j]=\mathtt{add}\}} x_j) \setminus$

$(\cup_{\{k:op[k]=\texttt{del}\}} x_k)$, *and the probability is taken on the random coins of* $\mathcal{A}$. *The scheme is said secure if the probability that any PPT algorithm* $\mathcal{A}$ *wins is negligible in* κ.

The type of accumulators we consider in this work is not necessarily *quasi-commutative* [CL02, LLX07] as they may not hide the order in which the elements were added to the set. More precisely, our definition tolerates that the value of the accumulator may depend on a particular sequence of $\mathsf{Update}_{\mathtt{add}}$ and $\mathsf{Update}_{\mathtt{del}}$ operations that produced a particular accumulator value *Acc*.

Our security definition (Definition 28) for the dynamic scenario (where addition and deletion of elements are allowed) differs from the one in [CL02] where the adversary is only able to add and delete elements by querying the accumulator manager, who is incorruptible. In contrast, in our definition the adversary is allowed to control the accumulator. However, we require that during each update at least an uncorrupted participant verifies the update with $\mathsf{CheckUpdate}$ to guarantee the consistency between the accumulated value and the history of additions and deletions.

Other Differences with Previous Definitions. The standard definition of dynamic accumulators (see for example the one in [CL02]) adds two requirements which so far we have not considered. First, it requires the existence of an additional algorithm that allows to publicly and efficiently update membership witnesses after a change in the accumulator value so witnesses can be proven valid under the new accumulator value. And secondly, it requires that both the accumulator updating algorithm as well as the witness updating algorithm to run in time independent from the size N of the accumulated set. In our construction, we only achieve logarithmic dependency on N. In practice, such dependency may be appropriate for many applications.

4.3 Unbounded Merkle Trees

In this section, we introduce Merkle trees and show how to extend them so that they can be used to hash unbounded sequences. We call these new constructs *unbounded Merkle trees (UMT).*

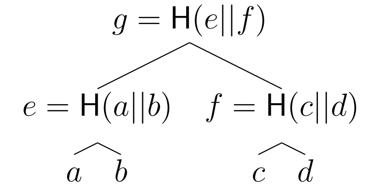


Figure 4.1: **Merkle tree for sequence** (a, b, c, d)**.** The hash function induced by the tree and a CRHF H takes the sequence (a, b, c, d) as input and returns the root hash value g as output. A proof that a belongs to the set is composed by the nodes containing values (b, f) which are the siblings of the nodes on the path from a to the root g. Checking the proof consists in computing $e' = \mathsf{H}(a||b)$, then computing $g' = \mathsf{H}(e'||f)$ and finally checking that $g = g'$.

Merkle Trees. Let $S = (v_1, v_2, \cdots, v_N)$ be a sequence of values in $Y = \{0, 1\}^\kappa$. Let $\mathcal{H} : K \times \{0, 1\}^* \to Y$ be a CRHF. A (binary) Merkle Tree [Mer89] is defined as follows:

- It has N leaves, where we assume for the sake of exposition that N is a power of 2.
- The value of the i^{th} leaf is set to $v_i \in Y$.
- The value $y \in Y$ for all internal node are computed recursively as follows: $y = \mathsf{H}(y_l||y_r)$ where $y_l, y_r \in Y$ are the values for the left and right node respectively.

If we consider the value obtained at the root as the final output, it is not hard to see that this data structure defines a CRHF $\mathcal{HS} : K \times Y^N \to Y$ where the input is the sequence and the output is the hash value obtained at the root node.

A useful property of Merkle trees is that, given the final hash value y, one can authenticate some element of the sequence by giving the siblings of the nodes between the element and the root (see Figure 4.1). Indeed, these nodes alone allow to recompute the hash value at the root and forging such nodes implies breaking the collision-resistance of $\mathcal{H}$.

The problem of such data structure is that the size of the input is fixed. That is, it is not possible to hash a sequence of more that N elements. In the following, we propose an unbounded version of the Merkle tree that will allow to hash sequences of arbitrary size and to update efficiently the hash value when the sequence changes.

Unbounded Merkle Trees. The idea to extend the previous construction so that we can handle dynamic and unbounded sequences consists of the following (see example of Figure 4.3): Instead of placing the values of the sequence at the leaves of the tree, we also use the internal nodes for hosting these elements. If a new value in the sequence is appended at the end, then we will add a new leaf so that the tree remains balanced and the leaves at the last level are contiguous starting from the left. In order to compute the hash value for the sequence, we introduce a proof value for each node which is defined as follows.

- We consider a CRHF $\mathcal{H} : K \times \{0,1\}^* \rightarrow Y$ and denote $\mathsf{H}(\cdot) = \mathcal{H}(k, \cdot)$ for some randomly chosen $k \in K$.
- The proof value for a Nil node is set to 0^κ.
- The proof value for an internal node N or a leaf is set to $\pi_\mathsf{N} = \mathsf{H}(v||\pi_l||\pi_r)$ where $v \in Y$ is the value of the node, $\pi_l, \pi_r \in Y$ are the proof values for the left and right child respectively.

If N is the root node of tree T, we abuse the notation and write π_T instead of π_N. The label of node N is set to $\ell(\mathsf{N}) = (V_\mathsf{N}, \pi_\mathsf{N})$. The collection of node values of T will be denoted by $\mathcal{V}(T)$.

At first sight, it seems natural to define the hash value for the sequence as the proof value corresponding to the root node. However, we need to be more careful, as in this case the depth of the tree we consider is variable and thus there is no guarantee that it is computationally hard to find a collision despite $\mathcal{H}$ is a CRHF. To illustrate this point let us consider a counter example that exhibits this issue (see Figure 4.2). Let $\pi_{Nil} = 0^\kappa$ be the proof value for a Nil node, and let H be a $(\kappa - 1)$-bit CRHF. Then, consider the following κ-bit hash function H':

$$\mathsf{H}'(x) = \begin{cases} 01^{\kappa-1} \text{ for } x = 0^\kappa||1^\kappa \\ 0^\kappa \text{ for } x = 01^{\kappa-1}||0^\kappa||0^\kappa \\ 1||\mathsf{H}(x) \text{ for all other } x \end{cases}$$

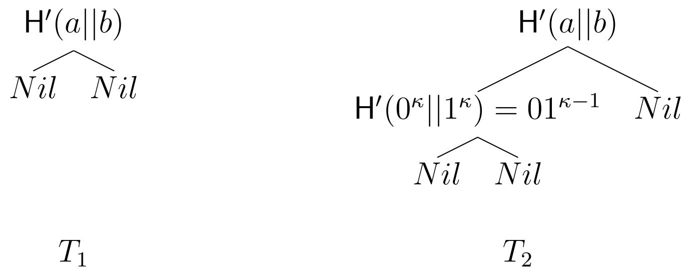


Figure 4.2: **Two different trees that yield the same proof value.** The tree on the left, T_1, and the one on the right, T_2, are different but yield the same proof value $\mathsf{H}'(\mathsf{H}'(a||b)||0^\kappa||0^\kappa)$ at their root.

Clearly, if H is collision-resistant, then so is H' . Then the proof π for a tree T_1 that contains a single node with value $\mathsf{H}'(a||b)$ is $\pi = \mathsf{H}'(\mathsf{H}'(a||b)||\pi_{Nil}||\pi_{Nil}) = \mathsf{H}'(\mathsf{H}'(a||b)||0^\kappa||0^\kappa)$ while the proof π' for a tree T_2 containing a root with value $\mathsf{H}'(a||b)$ and a left child with value $0^\kappa||1^\kappa$ is: $\pi' = \mathsf{H}'(\mathsf{H}'(a||b)||\pi''||\pi_{Nil})$ where $\pi'' = \mathsf{H}'(\mathsf{H}'(0^\kappa||1^\kappa)||\pi_{Nil}||\pi_{Nil}) = \mathsf{H}'(01^{\kappa-1}||0^\kappa||0^\kappa) = 0^\kappa$, thus $\pi' = \mathsf{H}'(\mathsf{H}'(a||b)||0^\kappa||0^\kappa) = \pi$. So we can observe that the proof values π and π' of each tree are equal, while no collision on H' has occurred.

This kind of issue is usually fixed using structure (e.g. length) padding techniques [ANPS07, BR97] so that the hash function allows to compare inputs that have the same format. In our case, we need to take in account the depth and the number of leaves at the last level (remember our tree is balanced) that are contiguous from left to right. These both values (depth and quantity of leaves) indeed define the unique structure of the tree so that no collision can occur.

Definition 29 *A balanced binary tree T is said to be* compact *if, and only if, for every i^{th} leaf L_i, each $j > i$, and considering d the distance of L_i to the root, we have that L_j is at distance at most d to the root.*

Definition 30 (Unbounded Merkle Trees (UMT)) *Let $S = (v_1, v_2, \ldots, v_N)$ be a sequence of values in $Y = \{0,1\}^\kappa$. Let $\mathcal{H}$ a collision-resistant hash function family where $\mathcal{H} : K \times \{0,1\}^* \to Y$. As before, we write $\mathsf{H}(\cdot)$ for the function $\mathcal{H}(k,\cdot)$ where k is selected at random. We say that T is an unbounded Merkle tree if, and only if:*

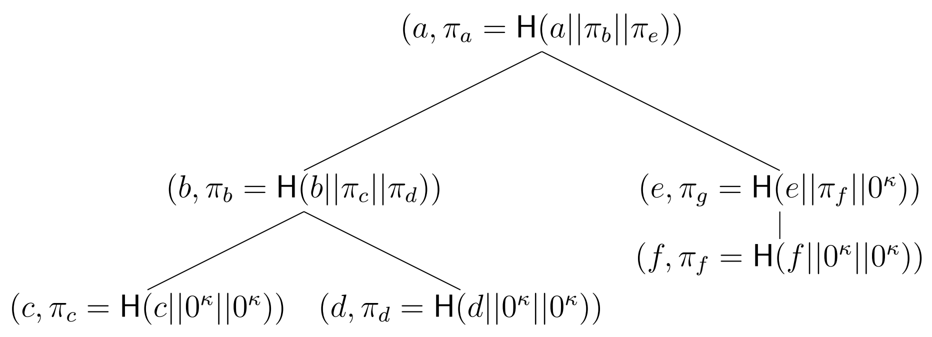

$(a, \pi_a = \mathsf{H}(a||\pi_b||\pi_e))$
$(b, \pi_b = \mathsf{H}(b||\pi_c||\pi_d))$
$(e, \pi_g = \mathsf{H}(e||\pi_f||0^\kappa))$
$(f, \pi_f = \mathsf{H}(f||0^\kappa||0^\kappa))$
$(c, \pi_c = \mathsf{H}(c||0^\kappa||0^\kappa))$
$(d, \pi_d = \mathsf{H}(d||0^\kappa||0^\kappa))$

Figure 4.3: **Unbounded Merkle Tree and associated CRHF for sequences.** Each node hosts a value and a proof value that is computed by hashing the node value along with the proof values of the children. The leaves of the last level are grouped from left to right in a contiguous block. For example, the leaf with value d could not be placed as a child of the node with value e, as it would leave a "hole" just after the first leaf at the last level. The hash value of the sequence $S = (a, b, e, c, d, f)$ is computed from the proof value of the root node π_a, the depth and the number of leaves *at the last level*, that is $\mathtt{HS}(S) = \mathsf{H}(\pi_a||\langle 2\rangle_\kappa||\langle 3\rangle_\kappa)$.

- *T is a binary tree with exactly N nodes.*
- *Each node N contains a pair $\ell(\mathsf{N}) = (v, \pi)$ where $v \in Y$ is the node value and $\pi \in Y$ is a proof value that links the node to its children. For a node N we write $V_\mathsf{N} = v$ and $\pi_\mathsf{N} = \pi$.*
- *T is balanced and all its leaves at the last level are grouped from left to right contiguously, that is T is also* compact.
- *The values of S are assigned to the nodes of T (first component of label) in breadth traversal order.*
- *The proof values for all the empty nodes are arbitrarily set to 0^κ.*
- *For each internal node N, we have that $\pi_\mathsf{N} = \mathsf{H}(v||\pi_l||\pi_r)$ where v is the value of the node and π_l, π_r are the proofs for the left and right child respectively. If N is the root node of tree T, we abuse the notation and write π_T instead of π_N.*

If T is an unbounded Merkle tree, then we denote by $\mathcal{V}(T)$ the set of node values for each node of T, that is the set $\{v_1, v_2, \ldots, v_N\}$.

It is straightforward to see that given a sequence there is only an UMT possible. Now, given a sequence S, we define a hash function family for that sequence (see Figure 4.3). Then we prove that this hash function is collision-resistant.

Construction 9 (Hash Function based on UMT) *Let $S = (v_1, \ldots, v_N)$ a sequence of elements in $Y = \{0,1\}^*$. Let $\mathcal{S} = Y^*$ be the space of sequences of elements in Y and let $T_{\mathcal{H}}$ be the associated UMT using the CRHF $\mathcal{H} : K \times \{0,1\}^* \to Y$. We define the hash function family $\mathcal{HS} : K \times \mathcal{S} \to Y$ as follows. Given $\mathsf{H} = \mathcal{H}(k, \cdot)$ for some $k \in K$, compute $T_{\mathcal{H}}$ and return the hash value $\mathtt{HS}(S) = \mathsf{H}(\pi_{T_{\mathcal{H}}} || \langle d \rangle_\kappa || \langle l \rangle_\kappa)$, where $\pi_{T_{\mathcal{H}}}$ is the proof value for the root node of $T_{\mathcal{H}}$, d is the depth of $T_{\mathcal{H}}$ and l is the number of leaves at the last level.*

Note that we do not use the following proposition to prove the security of our accumulator scheme (Section 4.4). However, this proposition is relevant from a conceptual point of view as it shows that the accumulated value (see Section 4.4.2) is computed through a CRHF.

Proposition 6 *The hash function family $\mathcal{HS}$ is collision-resistant.*

Proof. Let $\mathcal{A}$ be the adversary that breaks the collision-resistance of $\mathcal{HS}$. We build the following adversary $\mathcal{B}$ that breaks the collision-resistance of $\mathcal{H}$.

Given $k \in K$, $\mathcal{B}$ runs $\mathcal{A}$ on $\mathtt{HS} = \mathcal{HS}(k, \cdot)$. $\mathcal{A}$ returns two sequences S, S' such that $\mathtt{HS}(S) = \mathtt{HS}(S')$ but S and S' are distinct. In particular this means that the values of S are different from the values of S' or that they are in different order.

Let T, T' be the UMT of S, S' respectively. If $d_T \neq d_{T'}$ or $l_T \neq l_{T'}$, then $\mathcal{B}$ outputs a collision on message $M || \langle d_T \rangle_\kappa || \langle l_T \rangle_\kappa = M || V$ and $M' || \langle d_{T'} \rangle_\kappa || \langle l_{T'} \rangle_\kappa = M' || V'$ where M and M' may be equal but V, V' are different. Now we assume that both trees have the same depth and same number of leaves at the last level. For each node if the pre-images relative to the roots of T and T' are different then $\mathcal{B}$ outputs the collision. Otherwise $\mathcal{B}$ continues to compare the node and their hash values from top to bottom. Eventually, $\mathcal{B}$ will find a collision, as otherwise this would mean that all the nodes are equal and thus that the inputs (sequences S and S') are equal as well. ■

Merkle trees allow to authenticate a path from leaves to the root. Here we get the same property, except that the path can start from an internal node. Given T, T' trees, if T is a subtree of T', we write $T \subseteq T'$, otherwise we write $T \nsubseteq T'$.

Lemma 3 *Let $S = (v_1, \ldots, v_N)$ be a sequence of values in Y, $\mathcal{H} : K \times \{0,1\}^* \to Y$ a hash function family and T the corresponding UMT. If $\mathcal{H}$ is collision-resistant, then for every PPT algorithm $\mathcal{A}$ we have*

$$\Pr\left[\, T^* \leftarrow \mathcal{A}(k, S) : \pi_{T^*} = \pi_T \wedge T^* \nsubseteq T \,\right] = \mathsf{neg}(\kappa)$$

where T^* is a binary hash tree of depth at most d_T, with at most l_T leaves at the last level (grouped from left to right) and the probability is taken over the random coins of $\mathcal{A}$.

Proof. Let T be the UMT for the sequence S and $\mathcal{A}$ an adversary that finds a tree T^* such that $\pi_{T^*} = \pi_T$, $d(T^*) \leq d(T)$, $l_{T^*} \leq l_T$, but $T^* \nsubseteq T$. We define the adversary $\mathcal{B}$ as follows. It runs $\mathcal{A}$ and obtains T^*. Let T' be the subtree of T which nodes are in the same position of the nodes of T^*. By comparing the proof values of T^* and T' from top to bottom, $\mathcal{B}$ will eventually find a collision for $\mathcal{H}$ like described in the proof of Proposition 6. ∎

4.4 A Strong Universal Accumulator using UMT

In this section we describe our scheme. We assume that there exists a public broadcast channel with memory. Depending on the required security level, this can be a simple trusted web server, or a bulletin board that guarantees that every participant can see the published information and that nobody can delete a posted message. For a discussion on bulletin boards and an example of their use in another cryptographic protocol, the interested reader is referred to [CGS97]. We rely on broadcast channels in order to ensure that the publication of the successive accumulator values that correspond to updates of the set cannot be forged. In particular, an adversary who controls the manager of the accumulator cannot publish different accumulator values to different groups of participants.

4.4.1 Preliminaries

Our construction relies on UMT previously introduced. In order to handle proof of non-membership, we use the trick of Kocher [Koc98]: Instead of storing elements of the set in each node of the tree, we store pairs of consecutive elements of the set. Then, proving that an element x is *not* in the accumulated set X, amounts to simply proving that there exists elements x_α and x_β such that $x_\alpha < x < x_\beta$ and a pair (x_α, x_β) is stored in the tree.

Notation for Sets. We assume the set X we want to accumulate is ordered and denote by x_i the i^{th} element of $X = \{x_1, x_2, \ldots, x_N\}$, $N \in \mathbb{N}$. Let $x_0 = -\infty$ and $x_{N+1} = +\infty$ two special elements such that $-\infty < x_j < +\infty$ for all $x_j \in X$, where $\leq$ is the order relation on X (for example, the lexicographic order on bit strings) and $a < b$ if, and only if, $a \leq b$ and $a \neq b$.

Observe that showing $x \in X$ is equivalent to proving that:

$$(x_\alpha, x_\beta) \in \{(x_i, x_{i+1}) : 0 \leq i \leq N\} \ \wedge \ (x = x_\alpha \vee x = x_\beta).$$

On the other hand, showing that $x \notin X$ corresponds to proving:

$$(x_\alpha, x_\beta) \in \{(x_i, x_{i+1}) : 0 \leq i \leq N\} \wedge (x_\alpha < x < x_\beta).$$

Model for $X = \{x_1, \ldots, x_N\}$ under H. Given a set $X = \{x_1, \ldots, x_N\}$ and a CRHF H, we define a model for X under H to be the UMT for some sequence taking values in the set $\{\mathsf{H}(x_i || x_{i+1}) : 0 \leq i \leq N\}$. Figure 4.4 depicts a toy example of a model for a set.

Minimal subtrees generated by a set. Let T be a UMT. We say that $\mathcal{V} \subseteq \mathcal{V}(T)$ *generates a minimal subtree* R *of* T, if R is a subtree of T obtained by: (1) taking all nodes in T that belong to all paths from T's root to a node whose value is in $\mathcal{V}$ (the paths include both the root of T and the nodes of value in $\mathcal{V}$), and (2) all the (direct) children of the nodes taken in the previous step. Figure 4.5 illustrates the concept of minimal subtree. If R is generated by a singleton $\{v\}$, then we say that R

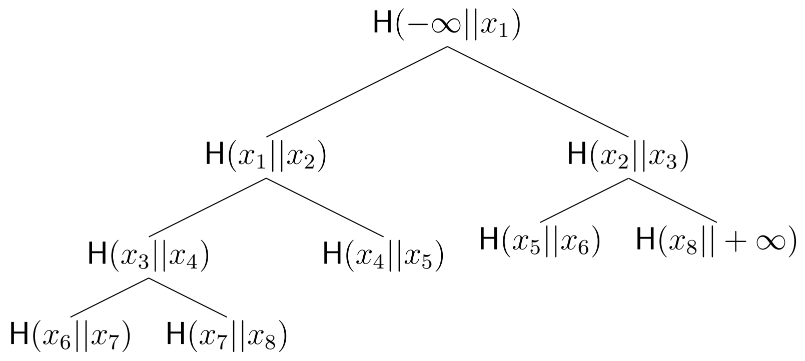


Figure 4.4: A model $\mathcal{T}$ for the set $X = \{x_1, \ldots, x_8\}$. The sequence for this model is $(\mathsf{H}(-\infty||x_1), \mathsf{H}(x_1||x_2), \mathsf{H}(x_2||x_3), \mathsf{H}(x_3||x_4), \mathsf{H}(x_4||x_5), \mathsf{H}(x_5||x_6), \mathsf{H}(x_8||+\infty), \mathsf{H}(x_6||x_7), \mathsf{H}(x_7||x_8))$.

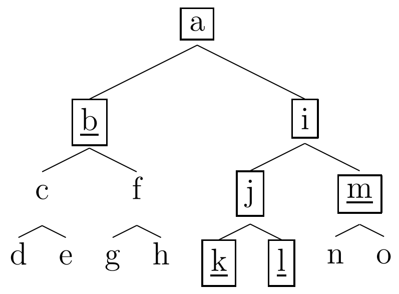


Figure 4.5: A tree and its minimal subtree (nodes with values in a box) generated by the node of value j. Children of the nodes that are on the path from j to a are underlined.

is generated by v.

In this section we use hash trees to build our proposed universal accumulator with memory. At a high level, our accumulator scheme relies on an accumulator manager that creates and updates a tree T which is a UMT of a sequence S_X corresponding to the set $X = \{x_1, x_2, ..., x_N\}$ under H. The UMT T_{H} of S_X will satisfy two conditions: (1) the accumulator manager can guarantee that $x \in X$ by proving that there is a node N of T_{H} such that $V_{\mathsf{N}} = (x_\alpha, x_\beta)$ where $x = x_\alpha$ or $x = x_\beta$, and (2) to demonstrate that $x \notin X$, the accumulator manager proves that there is a node N of T_{H} such that $V_{\mathsf{N}} = (x_\alpha, x_\beta)$ where $x_\alpha < x < x_\beta$. When adding or deleting elements from X, the accumulator manager needs to update T_{H} and guarantee that both of the stated conditions are satisfied.

In terms of setup assumptions, our scheme can be instantiated with any trusted initialization algorithm $\Omega(1^\kappa)$ which includes picking a hash function H uniformly at random from the family $\mathcal{H}$ (say by computing a random index $k \in K$ where $|K| = \kappa$ and then setting $\mathsf{H} = \mathcal{H}(k, \cdot)$). Of course, such assumption can also be instantiated with an ephemeral trusted third party running Ω, or alternatively using standard multi-party computation techniques among all participants, including the accumulator manager. Moreover, a common heuristic to avoid interaction is to simply pick H =SHA-256 [Nat06], for example. A detailed description of the proposed scheme follows.

4.4.2 Construction

Let $\kappa \in \mathbb{N}$ be the security parameter, $Y = \{0,1\}^\kappa$ and let $X = \{x_1, x_2, \ldots, x_N\} \subset U$ where $U = Y$. We define the accumulator scheme $\mathsf{HashAcc}$ below.

- $\mathsf{KeyGen}(\zeta)$: The algorithm starts by setting X equal to the empty set. Then, it extracts the security parameter κ and the description (index k) of the hash function $\mathsf{H} \in \mathcal{H}$ from ζ. We use $\mathcal{HS}$ as defined previously. Let $\mathtt{HS} = \mathcal{HS}(k, \cdot)$ for some $k \in K$. The initial sequence is $S = (\mathsf{H}(-\infty || +\infty))$. The memory $\mathfrak{m}$ is the UMT corresponding to the sequence S. The accumulated value is set to $\mathtt{HS}(S) = \mathsf{H}(\mathsf{H}(\mathsf{H}(-\infty|| + \infty)||0^\kappa||0^\kappa))||\langle 0\rangle_\kappa||\langle 1\rangle_\kappa)$. Finally, the algorithm sets the creation witness $w_{\mathtt{crt}}$ to $(\mathfrak{m}, \mathtt{crt})$, where $\mathtt{crt}$ is a fixed label.

- $\mathsf{Update}_{\mathtt{op}}(\zeta, x, Acc_{before}, \mathfrak{m}_{before})$: On input element $x \in U$, accumulator value Acc_{before}, and memory $\mathfrak{m}_{before}$, it proceeds as follows. Consider two cases depending on whether the update is an addition ($\mathtt{op} = \mathtt{add}$) or a deletion ($\mathtt{op} = \mathtt{del}$).

  If $\mathtt{op} = \mathtt{add}$ and $x \notin X$, the algorithm adds x into X by modifying $\mathfrak{m}_{before}$ through the two following steps:

  1. It replaces the value $\mathsf{H}(x_\alpha||x_\beta)$ from the appropriate node in $\mathfrak{m}_{before}$ (where $x_\alpha < x < x_\beta$) by the value $\mathsf{H}(x_\alpha||x)$.
  2. It augments the tree $\mathfrak{m}_{before}$ adding a new leaf N of value $\mathsf{H}(x||x_\beta)$ so the resulting tree $\mathfrak{m}_{after}$ is a balanced tree and such that the leaves at the last

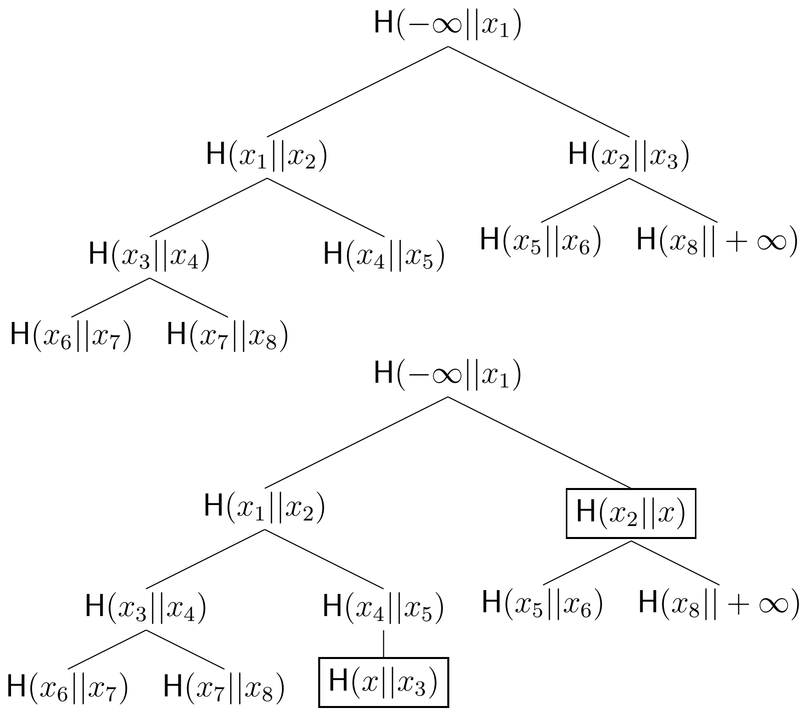


Figure 4.6: Inserting x into the tree of Figure 4.4 (at the top) where $x_2 < x < x_3$. The resulting tree after insertion (at the bottom, with new nodes in a box) must be kept balanced and such that the leaves at the last level are grouped from left to right. This means in particular that the new leaf could not have been placed under the node with value $\mathsf{H}(x_5||x_6)$ or the one with value $\mathsf{H}(x_8|| + \infty)$.

level are grouped from left to right (see Figure 4.6). Let $Par(\mathsf{N})$ be the (parent) node where N is attached as a leaf.

The resulting tree is denoted $\mathfrak{m}_{after}$. Figure 4.6 illustrates the process of inserting an element into $\mathfrak{m}_{before}$.

Once tree $\mathfrak{m}_{after}$ is built, the new accumulator is simply computed as the hash of the value of the root of the tree concatenated with the new depth d' of the tree as well as the updated number of leaves l' at the last level. Namely we have $Acc_{after} = \mathsf{H}(\pi_{\mathfrak{m}_{after}}||\langle d'\rangle_\kappa||\langle l'\rangle_\kappa)$. The witness $w_{add} = ((U_{add,1}, U_{add,2}, U_{add,3}), \texttt{add})$ that the update (addition) has been done correctly is computed as follows:

- $U_{add,1}$ corresponds to the minimal subtree of $\mathfrak{m}_{before}$ generated by the set

$\{\mathsf{H}(x_\alpha || x_\beta), V_{Par(\mathsf{N})}\}$,

- $U_{add,2}$ corresponds to the minimal subtree of $\mathfrak{m}_{after}$ generated by $\{\mathsf{H}(x_\alpha || x), \mathsf{H}(x || x_\beta)\}$, and
- $U_{add,3}$ is the tuple (d, d', l, l') where d, l are respectively the depth and the number of leaves at the last level of the tree $\mathfrak{m}_{before}$ and similarly d', l' are the new depth and number of leaves at the last level for the new tree $\mathfrak{m}_{after}$.

If $\mathtt{op} = \mathtt{del}$, deleting x from X is done in a similar way as follows. First, the update algorithm locates the two nodes of $\mathfrak{m}_{before}$ that contain x. Let α and β be those nodes, and let $\mathsf{H}(x_\alpha || x)$ and $\mathsf{H}(x || x_\beta)$ be their respective values, for some $x_\alpha < x < x_\beta$. The goal is to remove these nodes and replace them with a new node with value $\mathsf{H}(x_\alpha || x_\beta)$ in a way that the derived tree is still balanced. This is done by first replacing α with the single node with value $\mathsf{H}(x_\alpha || x_\beta)$, and then replacing β with the rightmost leaf R on the last level of the tree. These replacements yield a new tree $\mathfrak{m}_{after}$ whose root label is set to the value of the accumulator $Acc_{after} = \mathsf{H}(\pi_{\mathfrak{m}_{after}} || \langle d' \rangle_\kappa || \langle l' \rangle_\kappa)$ where d' and l' are respectively the new depth and updated number of leaves at the last level. The witness $w_{del} = ((U_{del,1}, U_{del,2}, U_{del,3}, U_{del,4}), \mathtt{del})$ is then computed as follows:

- $U_{del,1}$ corresponds to the minimal subtree of $\mathfrak{m}_{before}$ generated by the set $\{\mathsf{H}(x_\alpha || x), \mathsf{H}(x || x_\beta), V_R\}$,
- $U_{del,2}$ is the pair (x_α, x_β) such that $x_\alpha < x < x_\beta$,
- $U_{del,3}$ is the minimal subtree of $\mathfrak{m}_{after}$ generated by $\{\mathsf{H}(x_\alpha || x_\beta), V_R, V_{Par(R)}\}$, and
- $U_{del,4}$ is the tuple d, d', l, l' where d, l are respectively the depth and the number of leaves at the last level of the tree $\mathfrak{m}_{before}$ and similarly d', l' are the new depth and number of leaves at the last level for the new tree $\mathfrak{m}_{after}$.

The algorithm $\mathsf{Update}_{\mathsf{op}}$ outputs the new accumulator value Acc_{after}, the modified memory $\mathfrak{m}_{after}$, and the update witness w_{op}.

- $\mathsf{CheckUpdate}(\zeta, x, Acc_{before}, Acc_{after}, w_{\mathtt{op}})$: On input an element $x \in U$, two accumulator values $Acc_{before}, Acc_{after}$, and an update witness $w_{\mathtt{op}} = (w, \mathtt{op})$ for $\mathtt{op} \in \{\mathtt{add}, \mathtt{del}, \mathtt{crt}\}$, it proceeds as follows. If $\mathtt{op} = \mathtt{crt}$, then the algorithm outputs 1 if, and only if $Acc_{after} = \mathsf{H}(\mathsf{H}(\mathsf{H}(-\infty|| +\infty)||0^\kappa||0^\kappa)||\langle 0\rangle_\kappa||\langle 1\rangle_\kappa)$ and w is the model of the empty set under H.

  If $w = ((U_1, U_2, U_3), \mathtt{add})$, then U_1, U_2 are parsed as trees and U_3 as $(\langle d\rangle_\kappa, \langle d'\rangle_\kappa, \langle l\rangle_\kappa, \langle l'\rangle_\kappa)$. The algorithm returns 1 provided that:

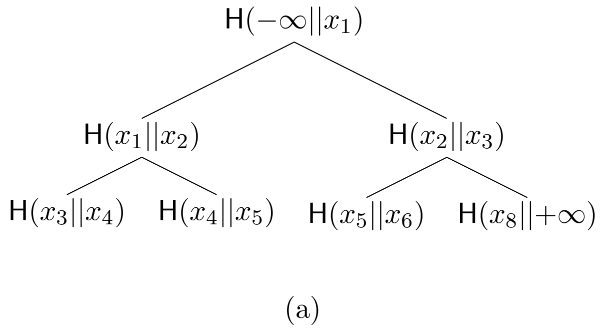
  - $d' = d$ or $d' = d + 1$,
  - $l' = l + 1 \mod 2^{d-1}$,
  - U_2 is a tree obtained by adding a leaf to U_1 so that (1) this leaf is placed at depth d' and (2) at the last position l' (note that not all the leaves at the last level will necessarily appear in U_2),
  - except for the node of value $\mathsf{H}(x_\alpha||x_\beta)$ (for $x_\alpha < x < x_\beta$) all nodes which are common to U_1 and U_2 have the same value in either one of the trees,
  - $\mathsf{H}(\pi_{U_1}||\langle d\rangle_\kappa||\langle l\rangle_\kappa) = Acc_{before}$ and $\mathsf{H}(\pi_{U_2}||\langle d'\rangle_\kappa||\langle l'\rangle_\kappa) = Acc_{after}$, and
  - $\mathsf{H}(x_\alpha||x), \mathsf{H}(x||x_\beta) \in \mathcal{V}(U_2)$.

  Otherwise, it outputs 0. We omit the case $w = ((U_1, U_2, U_3, U_4), \mathtt{del})$ which is similar.

- $\mathsf{WitGen}(\zeta, x, \mathfrak{m})$: On input $x \in U$ and memory $\mathfrak{m}$, it computes the witness $w = (w_1, w_2, w_3)$ as follows. First, the algorithm sets $w_1 = (x_\alpha, x_\beta)$ where $x = x_\alpha$ or $x = x_\beta$ if $x \in X$. Otherwise, if $x \notin X$ the algorithm sets $w_1 = (x_\alpha, x_\beta)$ where $x_\alpha < x < x_\beta$. Finally, it sets w_2 as the minimal subtree of $\mathfrak{m}$ generated by the value $\mathsf{H}(x_\alpha||x_\beta)$ and $w_3 = (\langle d\rangle_\kappa, \langle l\rangle_\kappa)$ where d, l are respectively is the depth and the number of leaves at the last level of the UMT $\mathfrak{m}$ [2].

- $\mathsf{Verify}(\zeta, x, w, Acc)$: On input $x \in U$ and witness $w = ((x', x''), T, (\langle d\rangle_\kappa, \langle l\rangle_\kappa))$ where T is purportedly a minimal subtree of the memory value $\mathfrak{m}$ associated

[2]This information is not related to the minimal subtree itself, however it is necessary in order to recompute the accumulated value in the next algorithm Verify.



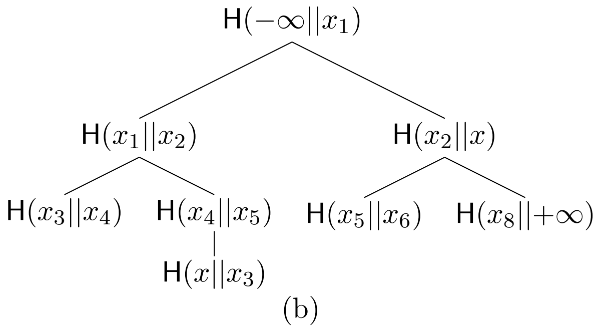


Figure 4.7: (a) The minimal subtree of the tree shown in Figure 4.4 and generated by $\{\mathsf{H}(x_2||x_3), \mathsf{H}(x_4, x_5)\}$. (b) The minimal subtree of the bottom tree shown in Figure 4.6 and generated by $\{\mathsf{H}(x_2||x), \mathsf{H}(x||x_3)\}$.

to the accumulator value Acc and $\langle d\rangle_\kappa, \langle l\rangle_\kappa$ are respectively the depth and the number of leaves at the last level of the memory $\mathfrak{m}$. We compute the depth d_T of the tree T and the position l_T (starting from the left) of the last leaf at the last level (note that T is not necessarily balanced).

Then we need to check if the following conditions hold:

- (1) $\mathsf{H}(\pi_T||\langle d\rangle_\kappa||\langle l\rangle_\kappa) = Acc$,
- (2) $\mathsf{H}(x'||x'') \in \mathcal{V}(T)$,
- (3) ($x = x'$ or $x = x''$), and (3') ($x' < x < x''$), and
- (4) $d_T \leq d$ and $l_T \leq l$.

The algorithm outputs 1 if conditions (1), (2), (3) and (4) hold; it outputs 0 if (1), (2), (3') and (4) hold. Otherwise, it outputs $\perp$.

Security. We now prove that the scheme HashAcc of the previous section is secure under Definition 28. First, note that if memory $\mathfrak{m}$ is a model of X, then the memory obtained after executing Update in order to add a new element $x \notin X$, is a model of $X \cup x$. Indeed, suppose $x_\alpha < x < x_\beta$ and let $\mathsf{H}(x_\alpha||x_\beta)$ be the value of a node N in $\mathfrak{m}$. By replacing node N with the node of value $\mathsf{H}(x_\alpha||x)$ and adding the node of value $\mathsf{H}(x||x_\beta)$, we clearly obtain a set of values $\{\mathsf{H}(x_i||x_{i+1}),\ 0 \leq i \leq N+1\}$ that corresponds to the successive intervals of the set $X \cup \{x\}$ (where $N = |X|$).

Intuitively, $\mathsf{CheckUpdate}$ must guarantee that the updated memory (tree) used to compute the new accumulated value still has the property of having all the successive intervals of the accumulated set as node values, that each interval appears once and only once in the tree, and that no other node value can belong to the tree.

Theorem 6 *Let $Y = \{0,1\}^\kappa$ and $\mathcal{H} : K \times \{0,1\}^* \to Y$ be a CRHF. Let $\Omega_\mathcal{H}$ be the initialization procedure that on input κ in unary returns $\zeta = (\mathsf{H}, 1^\kappa)$ where H is chosen uniformly at random from the family $\mathcal{H}$. If $\mathcal{H}$ is a CRHF, then the accumulator scheme $\mathsf{HashAcc}$ is a secure strong universal accumulator scheme (with memory) under $\Omega_\mathcal{H}$.*

Proof. Let $\mathcal{A}$ be the adversary that breaks the security of our scheme $\mathsf{HashAcc}$. We build the following adversary $\mathcal{B}$. $\mathcal{B}$ is given the description of the CRHF $\mathcal{H}$. $\mathcal{B}$ forwards this description to $\mathcal{A}$ which outputs the sequence introduced in Definition 28: $(Acc_1^*, w_1^*, x_1, \mathsf{op}_1), \ldots, (Acc_q^*, w_q^*, x_q, \mathsf{op}_q)$ and (x^*, w^*), an element from U along with its proof of membership/(non-membership). Adversary $\mathcal{B}$, using the same hash key, computes the correct sequence (remember that the computation of the accumulated value and the proofs are deterministic) related to updates $(x_1, \mathsf{op}_1), \ldots, (x_q, \mathsf{op}_q)$, and obtains the following sequence of accumulated values and witnesses (Acc_1, w_1, x_1, op_1), $\ldots$, $(Acc_q, w_q, x_q, \mathsf{op}_q)$. Let X be the set resulting from the successive updates and whose correct accumulated value is Acc_q, and T be the model of X computed by adversary $\mathcal{B}$.

Let us first consider the case where for all $i \in [q]$ we have that $Acc_i^* = Acc_i$. This means that in order to win, $\mathcal{A}$ must be able to compute a fake witness of membership / (non-membership).

- First, consider that $w^* = (w_1, w_2, w_3)$ is such that $w_1 = (x_\alpha^*, x_\beta^*)$ where (x_α^*, x_β^*) is not an interval for successive values in X, but $\mathsf{H}(x_\alpha^* || x_\beta^*) \in \mathcal{V}(T)$. In this case adversary $\mathcal{B}$ can output a collision for $\mathcal{H}$: $\mathsf{H}(x_\alpha^* || x_\beta^*) = \mathsf{H}(x_\alpha || x_\beta)$ where (x_α, x_β) is a pair of successive values in X.
- Secondly, if w_1 is such that $\mathsf{H}(x_\alpha^* || x_\beta^*) \notin \mathcal{V}(T)$ then by applying lemma 3 the adversary $\mathcal{B}$ is able to output a collision for $\mathcal{H}$.

Now, let us consider the first $i \in [1..q]$ such that $Acc_i^* \neq Acc_i$. As the procedure CheckUpdate is successful, this means that $\mathcal{A}$ has been able to compute a witness that is not a subtree of the correctly computed memory T_{i-1} (otherwise we would have $Acc_i^* = Acc_i$). Once again, applying lemma 3, $\mathcal{B}$ obtains a collision for $\mathcal{H}$. ■

Efficiency. We analyze the computational efficiency of the proposed scheme.

Theorem 7 *Let N be the size of X. The witnesses of (non)membership and of updates have size $O(\log(N))$. The update process* Update*, the verification processes* Belongs *and* CheckUpdate *can be done in time $O(\log(N))$.*

Proof. Assuming the accumulator manager uses a pointer based data structure representation for labeled binary trees, it is enough to show that a minimal subtree U of T generated by a constant number of node values has size $O(\log(N))$. Indeed, first note that a minimal subtree of a tree generated by a constant number of node values is the union of the minimal subtrees generated by each of the values. It is easy to see that the size of a minimal subtree generated by a node value is proportional to the depth of the node. This, and the fact that T is balanced, implies the desired conclusion. ■

4.5 Comparison with Previous Proposals

4.5.1 Efficiency of Our Scheme

Our solution is theoretically less efficient than the scheme proposed in [LLX07]. Nonetheless, if one considers practical instances of these schemes the difference effectively vanishes, as in most implementations hash function evaluation is significantly faster than RSA exponentiation – which is the core operation used by the schemes in [LLX07, CL02].

Table 4.1 shows the time taken by one single RSA exponentiation versus the time taken by our scheme for update operations as a function of the number of the accumulated elements. For the time measurements, we used the *openssl* benchmarking command (see [ope]) on a personal computer. Notice that RSA timings were obtained

N	**RSA-512**	**RSA-1024**	**RSA-2048**	**SHA-256**	**SHA-512**
2^3	0.85	4.23	25.00	0.55	2.13
2^{10}	0.85	4.23	25.00	1.83	7.125
2^{20}	0.85	4.23	25.00	3.66	14.25
2^{30}	0.85	4.23	25.00	5.49	21.36

Table 4.1: Comparison of performance between simple RSA exponentiation and logarithmic number of computations of SHA where N is the number of accumulated elements. Time is represented in milliseconds.

using signing operations, as in the scheme proposed in [LLX07] where exponents may not be small. Timings for SHA operations were measured using an input block of 1024 bits. The comparison is based on the fact that our scheme requires at most $6 \times 2\log(N)$ hash computations, where N is the number of accumulated elements, given that at most six branches of the Merkle tree used in our construction (three for $U_{del,1}$ and three for $U_{del,3}$, see Section 4.4.2) will have to be recomputed in the case of deletions. We now explain the relevance of Table 4.1. For clarity's sake, we focus on the efficiency of witness generation. In [CL02, LLX07, BdM93, BP97] schemes, the time to generate a witness is at least a single RSA signing operation, independently of the number of accumulated values. Hence, for both the aforementioned schemes, the time required to generate a witness is at least the one given by the columns of Table 4.1 with headers RSA-512, RSA-1024, and RSA-2048, depending on the size of the modulus used. In contrast, if we instantiate the hash function of our proposed scheme by one of the SHA family of hash functions, the time required to generate the witness for a set of size N is given by the columns of Table 4.1 with headers SHA-256 and SHA-512, depending on which version of SHA is used. A similar situation holds for operations such as addition or deletion of accumulated values. In conclusion, using our hash-based scheme is still very efficient, even for large values of N, in comparison with previous proposals.

Table 4.2 compares the functionality provided and sizes of parameters appearing in aforementioned schemes and our solution.

Scheme	Strong	Dynamic	Witness size
Benaloh et al. [BdM93]	Yes	No	$O(\kappa)$
Barić et al. [BP97]	Yes	No	$O(\kappa)$
Camenisch et al. [CL02]	No	Yes	$O(\kappa)$
Li et al. [LLX07]	No	Yes	$O(\kappa)$
This work	Yes	Partially†	$O(\kappa \log N)$

Table 4.2: Comparison of properties of previous and our scheme, where N is again the number of accumulated elements and κ is the security parameter. In all schemes the accumulator size is $O(\kappa)$. † Our solution allows dynamic addition and deletion of elements but no witness update.

4.5.2 Modeling Strong Accumulators

On the Setup Assumptions. We prove the security of our scheme under the assumption that there is a *trusted* procedure that chooses hash functions uniformly at random from a given family. We model such assumption using an initialization procedure Ω, which cannot be corrupted. Notice that once Ω finishes execution, no other trusted process or entity is required. In contrast, both previous solutions for dynamic accumulators [CL02, LLX07] not only require a trusted party – the accumulator manager itself – but also that such trusted entity be available for as long as the accumulator is active.

In practice, trusted initialization can efficiently be implemented using standard secure multi-party computation techniques. For our protocol, we only need to generate a hash function index, that is, a κ-bit uniformly distributed random string. This can be done using standard coin tossing algorithms [RBO89] (or more practical variants [KG09]) if a majority of participants during the initialization is honest. Alternatively, we could cast our results in the *human ignorance* setting proposed by Rogaway [Rog06]. In that case, it would suffice to take Ω as the identity function and make the reductions behind the proof of Theorem 6 more explicit (that is, detailing how new collision-finding adversaries are built from the given protocol adversaries). Note that all reductions in this chapter are in fact constructive.

Nontriviality of the strong property. Both in the construction of Camenisch et al. [CL02] as well as the one by Li et al. [LLX07], a corrupted manager can compute witnesses for arbitrary elements (regardless of whether the elements belong to the accumulated set or not). For example, in both schemes, the manager is able to generate a valid membership witness w for an arbitrary element $x \in \mathbb{Z}_n$ by simply computing $w = u^{x^{-1} \bmod (p-1)(q-1)}$, where $n = pq$ is the RSA modulus and $u \in \mathbb{Z}_n$ is the current accumulator value.

In contrast, our solution achieves the strong property as long as, at any given time, the party verifying the correctness of an accumulator update is able to remember the current and previous accumulated values.

4.6 The e-Invoice Factoring Problem

In this section, we describe an application of strong universal accumulators that yields an electronic analog of a mechanism called *factoring* through which a company, henceforth referred to as the Provider (P), sells a right to collect future payment from a company Client (C). The ensuing discussion is particularly concerned with the transfer of payment rights associated to the turn over of invoices, that is, *invoice factoring*. The way invoice factoring is usually performed in a country like Chile is that P turns over a purchase order from C to a third party, henceforth referred to as Factor Entity (FE). The latter gives P a cash advance equal to the amount of C's purchase order minus a fee. Later, FE collects payment from C.

There are several benefits to all the parties involved in a factoring operation. The provider obtains liquidity and avoids paying interests on credits that he/she would otherwise need (it is a common practice for some clients as well as several trading sectors in Chile to pay up to 6 months after purchase). The client gets a credit at no cost and is able to perform a purchase for which he might not have found a willing provider.

According to the Chilean Association of Factoring (*Asociación Chilena de Factoring - ACHEF*) during 2010, its 19 members accumulated almost 2 million documents worth more than 18 billion dollars [ACH]. Factoring's origins lie in the financing of trade, particularly international trade. Factoring as a fact of business life was underway in England prior to 1400 [Hur39]. The reader is referred to the website of the

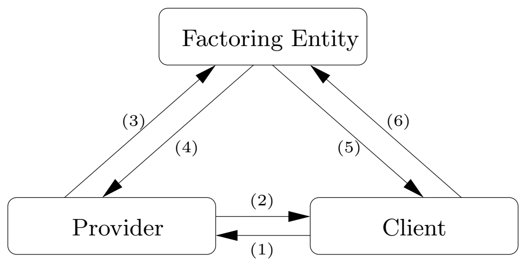


Figure 4.8: Steps of a factoring operation.

International Factors Group [ifg] for information on current trends and practices concerning factoring worldwide. Although factoring is performed in many contexts, as the reader will see, our proposed solution exploits peculiarities of the way it is locally implemented in Chile — thus, its applicability in other scenarios, if at all possible, would require adaptations.

The main phases of an invoice factoring operation are summarized below and illustrated in Figure 4.8:

1. C requests from P either goods or services,
2. P delivers the goods/services to C,
3. P makes a factoring request to FE,
4. FE either rejects or accepts P's request — in the latter case FE gives P a cash advance on C's purchase,
5. later, FE asks C to settle the outstanding payment, and finally,
6. C pays FE.

A risk for FE is that P can generate fake invoices and obtain cash advances over them. This danger is somewhat diminished by the fact that such dishonest behavior has serious legal consequences. More worrisome for FE is that P may duplicate real invoices and request cash advances from several FEs simultaneously. But, Chile's local practice makes this behavior hard to carry forth. Indeed, invoices are printed in blocks, serially numbered and pressure sealed by the local IRS agency (known as *Servicio de Impuestos Internos (SII)*). A FE will request the physical original copy of an invoice when advancing cash to P. It is illegal, and severely punished, to make fake copies or issue unsealed invoices.

Less than a decade ago, an electronic invoicing system began operating in Chile. Background and technical information concerning this initiative can be downloaded from the website of the SII, specifically from [efa]. The deployed electronic invoicing system has been widely successful. It has been hailed as a major step in the government modernization. Furthermore, it has created strong incentives for medium to small size companies to enter the so called "information age". Nevertheless, the system somewhat disrupts the local practice concerning factoring. Specifically, a FE will not be able to request the original copy of an invoice, since in a digital world there is no difference between an original and a copy. This creates the possibility of short term large scale fraud being committed by unscrupulous providers. Indeed, a provider can "sell" the same invoice to many distinct FEs. We refer to the aforementioned situation created by the introduction of electronic invoicing as the *e-Invoice Factoring Problem.*

We show below how to address this problem using strong universal accumulator schemes, but first it is important to note that there are other issues of concern for participants of an e-invoice factoring system, among the most relevant are:

- Privacy of the commercially sensitive information contained by invoices (e.g. private customer's information like for example tax identification numbers, volume of transactions, etc.)
- Robustness of the e-invoice factoring system — no small size colluding party should be able to disrupt the system's operation and/or break its security.
- Confidentiality of the FE's commercially sensitive information (customer pool, number of transactions, volume of transactions, etc.)

The latter of these issues arises because the FEs are in competition among themselves. They have an incentive to collaborate in order to avoid fraud, but they do not wish to disclose information about their customer base and transaction volumes to competitors. Moreover, a widespread sharing of invoices would not be welcomed by the providers who issue them, given that they probably want to keep the profile of their clients confidential.

In order to describe our proposed strong accumulator based solution for the e-Invoice Factoring Problem it is convenient to introduce additional terminology. A factoring protocol $\mathcal{F} = (FE_1, \ldots, FE_l; FA)$ involves $l \geq 2$ participants $FE_1, \ldots, FE_l$ called factor entities, and a special participant FA called the factoring authority. The factoring protocol $\mathcal{F}$ is defined by the concurrent execution of several instances of a decision protocol consisting of:

1. The transmission from FE_i to FA of a digest x of an invoice Inv that FE_i wants to buy.

2. The computation by FA, based on x and the identity of FE_i, of a value V and its publication through a public broadcast channel.

3. The decision on whether or not to buy the invoice associated to x made by FE_i based on V and other possible information previously collected. If FE_i concludes that Inv has not been previously sold, then it decides to buy and outputs $Inv||0||i$, otherwise it outputs $Inv||1||i$.

The previous description of a factoring protocol captures the fact that the FE_i's interact concurrently with FA in order to decide whether or not to buy invoices.

In order to formalize the security requirements involved in a factoring protocol we proceed as follows. Let $\kappa \in \mathbb{N}$ be a security parameter, Ω an initialization procedure, and let $\mathcal{F} = (FE_1, \ldots, FE_l; FA)$ be a factoring protocol for $l = P(\kappa) \geq 2$ where P is a polynomial. Consider an experiment, denoted $Exp^{\text{fac}}_{\mathcal{F},\Omega,\mathcal{A}}(\kappa)$, where $\mathcal{A}$ is a polynomial time bounded adversary that can corrupt FA and choose the elements for which the FE_i's want to make a purchase decision. The adversary can run a polynomial number of decision protocol instances. Also, FA can invoke the initialization procedure Ω which is not under the control of the adversary. We say that the experiment outputs 1 if the adversary wins, i.e. if either one of the following situations occur; (1) for

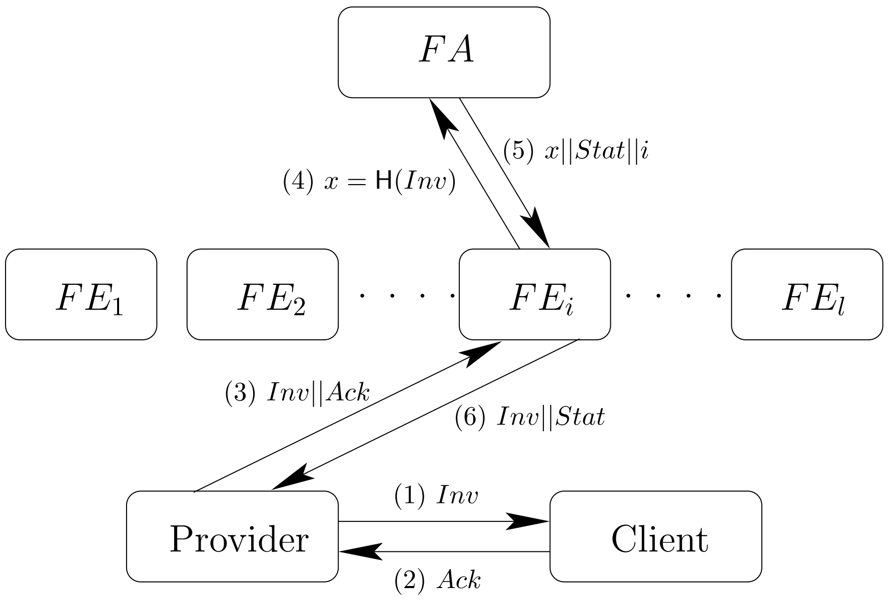


Figure 4.9: **Base Protocol** (trustworthy FA).

$i \neq j$, FE_i, FE_j publish $Inv||0||i$ and $Inv||0||j$ respectively, meaning they are willing to purchase the same invoice Inv, or (2) FE_i publishes $Inv||1||i$ meaning it rejects the invoice Inv, despite no other honest factor entity took the decision to buy it (there is no published message $Inv||0||j$ where j is the index of an honest factor entity FE_j).

Definition 31 (Security of a factoring protocol) *Let $\kappa \in \mathbb{N}$ be a security parameter and $l = P(\kappa) \geq 2$ for some polynomial P. We say that a factoring protocol $\mathcal{F}$ is secure under initialization procedure Ω if $\Pr\left[Exp^{fac}_{\mathcal{F},\Omega,\mathcal{A}}(\kappa) \right] = neg(\kappa)$ for every probabilistic polynomial time adversary $\mathcal{A}$.*

A FACTORING PROTOCOL BASED ON A SECURE STRONG UNIVERSAL ACCUMULATOR SCHEME. We now describe how any secure strong universal accumulator scheme can serve as the basis on which a secure factoring protocol can be built.

For the sake of clarity of exposition we first describe a general protocol, called **Base Protocol**, that involves all the participants of a factoring transaction: the client C, the provider P, the factor entities $FE_1, \ldots, FE_l$ and the factoring authority FA. In this **Base Protocol**, we assume moreover that FA is trustworthy. Afterwards, we shall show how to remove this assumption. Also assume that FA has access to a hash function H uniformly chosen from a collision-resistant hash function family. In our trustworthy factoring authority based solution, FA stores the hash values of all acquired invoices and replies to queries from the factor entities concerning the status (either acquired or available) of an invoice with a given digest value. Henceforth, we assume that the origin and authencity of all messages are guaranteed through a bulletin board (as implemented in other cryptographic protocols, e.g. [CGS97]).

The **Base Protocol** is illustrated in Figure 4.9 and its phases are described next:

Base Protocol

1. P sends an e-invoice Inv to C.
2. C sends an acknowledgement of receipt of the e-invoice $Ack = Sign_C(Inv)$.
3. P sends the message $Inv||Ack$ to the FE_i of his choice.
4. FE_i sends the message $x = \mathsf{H}(Inv)$ to FA.
5. FA checks whether x is in its database. If not, FA sets $Stat$ to 0 and adds x to its database. Otherwise, $Stat$ is set to 1. Then, FA broadcasts through the public channel the message $x||Stat||i$. Upon receiving $x||Stat||i$, the factor entity FE_i agrees to purchase Inv if $Stat = 0$, and declines if $Stat = 1$.
6. FE_i sends the message $Inv||Stat$ to P.

Discussion. Note that during Step 2 a receipt is produced by C and then transmitted to FE_i during Step 3. This is to prevent client C from being framed by P as having made a purchase whose payment FE_i could try to collect later on. Also, note that Inv is not transmitted to FA during Step 4. This is done to allow protocol extensions to support confidentiality of P's and FE_i's commercially sensitive information[3]. The reason for including the identifier i in FA's reply in Step 5 is to guarantee that nobody besides FE_i can exhibit a valid proof, purportedly sent by FA, claiming that x was not in FA's database at a given instant (otherwise anyone capturing FA's replies could obtain a certificate that purchase of an invoice with digest x is warranted).

In the protocol, FA's message $x||Stat||i$ certifies that an invoice with digest x either is or is not present in FA's database. Note also that FE_i's message $Inv||0$ is a proof of commitment that FE_i has agreed to acquire Inv from P.

Attacks outside of the model. Observe that collusion between C and P is possible. Indeed, it is easy to see that together they can produce e–invoices not tied to a real commercial transaction. To avoid this risk, a factor entity should check the validity of every e–invoice it is offered, before even contacting FA. The current e–factoring system deployed by the Chilean internal revenue service provides on–line functionality to check the validity of e–invoices (every issuer of e–invoices must submit to the tax collecting agency an electronic copy of every e–invoice it creates within 12 hours of having issued it).

Refined protocol. Now let us consider a more realistic scenario where the factoring authority is not trustworthy. We describe a solution for the e-Invoice Factoring Problem that is based on five algorithms that rely on a secure strong universal accumulator under initialization procedure Ω, denoted ACC. To avoid confusion, each algorithm related to the accumulator will be referred to by $\mathsf{ACC}.\langle\mathsf{algorithm\ name}\rangle$.

- $\mathsf{Setup}(1^\kappa)$: The factoring authority acts as the accumulator manager. First it invokes Ω on input 1^κ and obtains ζ. Then, it runs accumulator setup algorithm $\mathsf{ACC.Setup}$ on ζ, and stores the accumulator value Acc and the memory $\mathfrak{m}$.

- $\mathsf{Add}(\zeta, x, Acc_{before}, \mathfrak{m}_{before})$: The factoring authority runs procedure $\mathsf{ACC.Update_{add}}(\zeta, x, Acc_{before}, \mathfrak{m}_{before})$, returns $(Acc_{after}, \mathfrak{m}_{after})$ and w_{add}. Then, it publishes[4] in the bulletin board (Acc_{before}, Acc_{after}, w_{add}).

[3]This is straightforward by using perfectly one-way hash functions [CMR98] for H in Step 4.
[4]Note that *conceptually* it would be enough to publish only the elements that are inserted into /

- $\mathsf{Check}_{add}(\zeta, x, Acc_{before}, Acc_{after}, w_{add})$: The factor entity that wants to check whether the element x was correctly added to the accumulated set executes $\mathsf{ACC.CheckUpdate}(\zeta, x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w_{add})$.

- $\mathsf{Belongs}(\zeta, x, \mathfrak{m})$: the factoring authority sets $w = \mathsf{ACC.WitGen}(\zeta, x, \mathfrak{m})$. Then, it sets $Stat = 1$ if x has been accumulated, and $Stat = 0$ otherwise. Finally, it returns $(Stat, w)$.

- $\mathsf{Check}_{belongs}(\zeta, x, Stat, w, Acc)$: the factor entity that wants to check whether or not the element x belongs to the accumulated set verifies that $\mathsf{ACC.Verify}(\zeta, x, w, Acc)$ equals $Stat$.

Below we describe the refinement of Step 5 of the **Base Protocol** which corresponds to a factoring protocol. As in Section 4.4 we assume the availability of a public broadcast channel. We also assume that when **Base Protocol** starts the procedure $\mathsf{Setup}(1^\kappa)$ is invoked, where κ is the security parameter, thus generating the system-wide parameter ζ.

deleted from the set after each update. However, this would force every participant to store the whole history of changes in order to recompute the successive accumulated values. Thus, by publishing the accumulated values and the update proofs the accumulator manager allows an external party to check the consistency of the updates without having to parse all the intermediate values: It only needs to remember the last accumulated value that was checked successfully whether by itself or some other (trusted) verifier.

Refinement of Step 5

5.0 Assume the current accumulator value is Acc_{before} and the memory state is $\mathfrak{m}_{before}$.

5.1 Upon receiving x from factoring entity FE_i, the factoring authority FA determines $(Stat, w) = \mathsf{Belongs}(\zeta, x, \mathfrak{m}_{before})$ and then broadcasts $x||(Stat, w)||i$. If $Stat = 0$, then FA executes $\mathsf{Add}(\zeta, x, Acc_{before}, \mathfrak{m}_{before})$, obtains $(Acc_{after}, \mathfrak{m}_{after})$ and w_{add}, and broadcasts $x||(Stat, Acc_{after}, \mathfrak{m}_{after}, w_{add})||i$.

5.2 The following verifications are performed:

(a) FE_i runs $\mathsf{Check}_{\mathsf{belongs}}$ with input $(\zeta, x, Stat, w, Acc_{before})$.

(b) If $Stat = 0$, then every factor entity executes $\mathsf{Check}_{\mathsf{add}}(\zeta, x, Acc_{before}, Acc_{after}, w_{add})$.

5.3 If no factor entity objects by exhibiting a valid proof that it has previously purchased an invoice with digest value x, $Stat = 0$ in Step 1, and no message $x||(Stat, Acc_{after}, \mathfrak{m}_{after}, w_{add})||j$ with $j \neq i$ is published before FA updates the accumulator value to Acc_{after} (and the memory state to $\mathfrak{m}_{after}$),[5] then FE_i agrees to the purchase of an e-invoice with digest x. Otherwise ($Stat = 1$ or one of the verification fails), FE_i rejects the invoice with digest x.

It is important to point out that every FE_i has to check each change on the memory $\mathfrak{m}$ using the values published in the broadcast channel. Theses checks guarantee *continuity* in the evolution of the history of the FE_i's decisions (buy or reject an invoice).

[5]In practice this can be implemented by a round during which each factor entity that disagrees with FA's broadcast values has to publish its complaint.

For clarity of exposition, in our proposed solution to the e-Invoice Factoring Protocol, we have omitted explanations of how to deal with e-invoice digest removals from FA's database. However, it should be obvious how to implement this feature relying on the secure deletion functionality provided by secure strong universal accumulators with memory. Implementation of this functionality is essential for maintaining efficiency. Specifically, to upper bound the size of the intermediate outputs and per operation processing time by a logarithm in the number of accumulated invoices.

We henceforth denote by $\mathcal{F}_{\mathsf{ACC}}$ the protocol described above (the **Base Protocol** together with its refinement) when instantiated with a universal accumulator scheme ACC.

Proposition 7 *Let* ACC *be a secure strong universal accumulator scheme (with memory) under initialization procedure* Ω*. Then,* $\mathcal{F}_{\mathsf{ACC}}$ *is a secure factoring protocol under* Ω*.*

Proof. Assume an adversary can make either one of the following situations to occur: (1) for $i \neq j$ the adversary manages to get two messages $Inv||0||i$, $Inv||0||j$ broadcasted by two honest factor entities FE_i and respectively FE_j, or (2) the adversary is able to convince an honest factor entity FE_i to publish the message $Inv||1||i$ while no message $Inv||0||j$, where j is the index of an honest participant, has been broadcasted previously . If situation (1) occurs, the verification in Step 5.2 of the factoring protocol guarantees that only one message of the type $\mathsf{H}(Inv)||(Stat, Acc_{after}, \mathfrak{m}_{after}, w_{add})||z$ can be published between two consecutive changes of the accumulated value, thus the attacker needs to be able to find a witness of non-membership for $\mathsf{H}(Inv)$, although $\mathsf{H}(Inv)$ has already been accumulated. Since $\mathcal{F}_{\mathsf{ACC}}$ is secure, this can only happen with a negligible probability. If situation (2) occurs, then in order to succeed in Step 5.2, the attacker needs to be able to find a witness of membership for $\mathsf{H}(Inv)$, although $\mathsf{H}(Inv)$ has not been previously accumulated. Since $\mathcal{F}_{\mathsf{ACC}}$ is secure, this can not happen except with negligible probability. $\blacksquare$

Theorem 6 and Proposition 7 immediately yield,

Corollary 1 *The factoring protocol* $\mathcal{F}_{\mathsf{HashAcc}}$ *is a secure factoring protocol under* $\Omega_{\mathcal{H}}$*.*

4.7 Conclusion

We introduced the notion of strong universal accumulator scheme which provide almost the same functionality as do the universal accumulator schemes defined in [LLX07], namely (1) a set is represented by a short value called accumulator, (2) it is possible to add and remove elements dynamically from the (accumulated) set, and (3) proofs of membership and non-membership can be generated using a witness and the accumulated value. In this notion, however, the accumulator manager does not need to be trustworthy and might be compromised by an adversary.

We also give a construction of a strong universal accumulator scheme based on cryptographic hash functions which relies on a public data-structure to compute accumulated values and witnesses (of membership and non-membership in the accumulated set). We argue that the proposed scheme is practical and efficient for most applications. In particular, we discuss an application to a concrete and relevant problem, the e-invoice factoring problem.

Chapter 5

Batch Update for Cryptographic Accumulators is Impossible

Contents

5.1 Introduction

In their survey on cryptographic accumulators [FN02], Fazio and Nicolosi pointed out that, in Camenisch and Lysyanskaya's construction, the time to recompute the witnesses once the accumulated set has been modified was proportional to m, the number of changes of the accumulated set. This raised a natural question: *"Is it possible to construct dynamic accumulators in which the update of several witnesses can be performed in constant (independently of m) time?"*

Wang et al. [WWP07, WWP08] answered this question in the affirmative by proposing a construction that allows *batch update*. Unfortunately, we show that this construction is not secure. Moreover, we prove that there is no way to fix Wang et al.'s accumulator by giving a lower bound of $\Omega(m)$ in the time required to update witnesses after m updates.

Since the publication of [WWP07, WWP08], the existence of accumulators with *batch update* seems to have been taken for granted, and in fact assumed to exist in subsequent work. Damgård and Triandopoulos [DT08] cite their availability as an example of an accumulator construction based on the Paillier cryptosystem. Camenisch et al. [CKS09] also mentioned Wang et al.'s accumulator and claim to support *batch update*. If we consider strictly Fazio and Nicolosi's definition however, this is not the case, as the witness update algorithm in [CKS09] performs a number of operations proportional to the combined size of the set of added elements and removed elements.

We remark that our impossibility result also applies to any batch update variant of the accumulator schemes proposed in [DT08, LLX07], which allow both membership and non-membership proofs.

Batch Update. As originally proposed [FN02], the batch update property for an accumulator scheme states that each user should be able to update each of his witnesses using the algorithm UpdWit in time independent of the number of changes (additions and deletions) to the accumulated set. The formal definition is as follows:

Definition 32 (Batch update for accumulator schemes). *Let $\kappa \in \mathbb{N}$ be a security parameter and let $\mathtt{Acc}$ be an accumulator scheme. Also, let X_i be a set of accumulated values at some time i, and $U_i \subset X_i$ be a set of elements for which some user U holds valid witnesses. Suppose that after $m > 0$ updates (insertions or deletions) to set X_i, the new accumulated set is X_{i+m} and the associated accumulated value is $Acc_{X_{i+m}}$. We say that $\mathtt{Acc}$ has the* batch update property *if given some information $Upd_{X_i,X_{i+m}}$, user U running UpdWit can recompute a valid witness for any element in $U_i \bigcap X_{i+m}$ in constant time (with respect to m). More precisely, if t is the time of a witness update for a single added/removed element, then the update for a set after m changes must take time $O(t)$.*

5.2 Wang et al.'s Construction

In this section, we briefly recall Wang et al.'s accumulator with *batch update* [WWP07, WWP08]. The first version of their work [WWP07] suffered from two correctness problems which were later fixed [WWP08]. Below we review the improved version [WWP08].

The algorithms of Wang et al.'s scheme slightly deviate from the general syntax of definition 18 as they allow adding and deleting sets of more than one element to the accumulator. Moreover, the algorithms AddEle and AccVal are randomized which means that the accumulated value does not only depend on the elements of the set, contrary to the definition of Fazio and Nicolosi [FN02]. The general idea however remains the same.

Definition 33 *(Syntax for dynamic accumulators with batch update [WWP08]) Let κ be the security parameter. A dynamic accumulator with* batch update $\mathtt{Acc_{BU}}$ *consists of the following algorithms:*

- $\mathsf{KeyGen}(1^\kappa)$*: is a probabilistic algorithm that takes as input the security parameter κ in unary and returns a parameter $\mathcal{P} = (PK, SK)$ where PK is the public key and SK is the private key*[1]*.*
- $\mathsf{AccVal}(X, \mathcal{P})$*: is a probabilistic algorithm that computes an accumulated value. It takes as input the set $X = \{c_1, ..., c_N\}$ and the parameter $\mathcal{P}$ and returns an accumulated value Acc_X along with some auxiliary information a_c that will be used by other algorithms.*
- $\mathsf{WitGen}(a_c, X, \mathcal{P})$*: this probabilistic algorithm returns a list of witnesses corresponding to each element in the set X. It takes as input an auxiliary information a_c, the set X and the parameter $\mathcal{P}$.*
- $\mathsf{AddEle}(X^\oplus, a_c, Acc_X, \mathcal{P})$*: this probabilistic algorithm adds some new elements to the accumulated value. The input values are the set of new elements to add*

[1]In the original paper, the authors mention another parameter M which is an upper bound to the number of elements that can be accumulated. In order to simplify the notations, we omit it and recall that this upper bound must be a polynomial in κ.

$X^{\oplus} = \{c_1^{\oplus}, ..., c_l^{\oplus}\}$, *the auxiliary information* a_c, *the accumulated value* Acc_X *and the parameter* $\mathcal{P}$. *The returned values are* $Acc_{X \cup X^{\oplus}}$ *the accumulated value corresponding to the set* $X \cup X^{\oplus}$, *a list of witnesses* $(\pi_1^{\oplus}, ..., \pi_l^{\oplus})$ *associated with the inserted elements* $(c_1^{\oplus}, ..., c_l^{\oplus})$, *and the auxiliary information* a_c, a_u, *that will be used for future update operations.*

- DelEle$(X^{\ominus}, a_c, Acc_X, \mathcal{P})$: *this probabilistic algorithm is analogous to* AddEle *and allows a set of elements* $X^{\ominus}$ *to be deleted. The input values are the set of elements to delete* $X^{\ominus} = \{c_1^{\ominus}, ..., c_l^{\ominus}\}$, *the auxiliary information* a_c, *the accumulated value* Acc_X *and the parameter* $\mathcal{P}$. *The returned values are* $Acc_{X \setminus X^{\ominus}}$ *the accumulated value corresponding to the set* $X \setminus X^{\ominus}$, *and the auxiliary information* a_c, a_u, *that will be used for future update operations.*

- Verify(x, π, Acc_X, PK): *this deterministic algorithm checks whether an element* x *belongs to the set* X *represented by the accumulated value* Acc_X *using the witness* π *and the public key* PK. *It returns* valid *if the witness* π *for* x *is valid or* $\perp$ *otherwise.*

- UpdWit(π_i, a_u, PK) : *this deterministic algorithm updates witnesses for the elements accumulated in* Acc_X *and that are still accumulated in* $Acc_{X'}$ *(the new set after update). The inputs are* π_i, *the witness to update, the auxiliary information* a_u *and the public key* PK. *It returns an updated witness* π_i' *that allows to prove that* c_i *is still accumulated in the new accumulated value* $Acc_{X'}$.

Note that in this definition UpdWitGen does not appear. The reason is that in Wang et al.'s construction, the update information required to recompute the witnesses is generated by algorithms AddEle and DelEle.

Wang et al.'s accumulator relies on the Paillier cryptosystem [Pai99], which we recall here. The Paillier cryptosystem [Pai99] consists of the following three algorithms.

- KeyGen: let $n = pq$ be a RSA modulus, with p, q large prime integers. Let g be an integer multiple of n modulo n^2. The public key is defined by $PK = (n, g)$ and the private key by $SK = \lambda = \lambda(n) = lcm(p-1, q-1)$.

- $\mathsf{Encrypt}$: let $M \in \mathbb{Z}_n$ be a plaintext message and r a random element in $\mathbb{Z}_n^*$, the encrypted message is $c = g^M r^n \bmod n^2$.
- $\mathsf{Decrypt}$: to recover M from ciphertext c, compute $M = \frac{L(c^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n$ w here $L : u \to \frac{u-1}{n}$ takes as argument elements from the set $\mathcal{S}_n = \{u < n^2 | u = 1 \bmod n\}$.

The homomorphic property of the Paillier cryptosystem follows from the fact that $\forall x, y \in \mathcal{S}_n$ and $\sigma \in \mathbb{Z}^+$:

$$\begin{aligned} L((x.y)^\lambda \bmod n^2) \bmod n &= L(x^\lambda \bmod n^2) + L(y^\lambda \bmod n^2) \bmod n \\ L(x^{\sigma\lambda} \bmod n^2) \bmod n &= \sigma L(x^\lambda \bmod n^2) \bmod n \end{aligned}$$

We can now describe Wang et al.'s concrete scheme.

Construction 10 *(Wang et al.'s accumulator, [WWP08])*

- $\mathsf{KeyGen}(1^\kappa)$*:*
  Given the security parameter κ in unary, compute a safe-prime product $n = pq$ that is κ-bits long and create an empty set V. Let $\mathcal{C} = \mathbb{Z}_{n^2}^* \setminus \{1\}$ and $T' = \{3, ..., n^2\}$. Let $\beta \xleftarrow{R} \mathbb{Z}_{\phi(n^2)}^*$ and $\sigma \xleftarrow{R} \mathbb{Z}_{n^2}$ be two random numbers. The public key PK is set to (n, β) and the private key SK to (σ, λ). The output is the parameter $\mathcal{P} = (PK, SK)$.
- $\mathsf{AccVal}(X, \mathcal{P})$*:*
  Given a set $X = \{c_1, ..., c_N\}$ with $X \subset \mathcal{C}$, and the parameter $\mathcal{P}$, take $c_{N+1} \xleftarrow{R} \mathcal{C}$ and compute

$$\begin{aligned} x_i &= L(c_i^\lambda \bmod n^2) \bmod n \ \ (\mathit{for}\ i = 1, ..., N+1) \\ Acc_X &= \sigma \textstyle\sum_{i=1}^{N+1} x_i \bmod n \\ y_i &= c_i^{\lambda\sigma\beta^{-1}} \bmod n^2 \ \ (\mathit{for}\ i = 1, ..., N+1) \\ a_c &= \Pi_{i=1}^{N+1} y_i \bmod n^2 \end{aligned}$$

Output the accumulated value Acc_X and the auxiliary information a_c.

- $\mathsf{WitGen}(a_c, X, \mathcal{P})$:
  Given the auxiliary information a_c, a set $X = \{c_1, ..., c_N\}$, and the parameter $\mathcal{P}$, choose uniformly at random a set of m numbers $T = \{t_1, ..., t_N\} \subset T' \setminus \{\beta\}$ and compute
  $$w_i = a_c c_i^{-t_i \beta^{-1}} \bmod n^2 \ \ (\textit{for } i = 1, ..., N)$$
  Output the witness $\pi_i = (w_i, t_i)$ for c_i (for $i = 1, ..., N$).

- $\mathsf{AddEle}(X^{\oplus}, a_c, Acc_X, \mathcal{P})$:
  Given a set $X^{\oplus} = \{c_1^{\oplus}, ..., c_l^{\oplus}\} (X^{\oplus} \subseteq \mathcal{C} \setminus X)$ to be inserted, the auxiliary information a_c, the accumulated value Acc_X, and the parameter $\mathcal{P}$, choose $c_{l+1}^{\oplus} \overset{R}{\leftarrow} \mathcal{C}$ and a set of l numbers $T^{\oplus} = \{t_1^{\oplus}, ..., t_l^{\oplus}\} \overset{R}{\leftarrow} T' \setminus (T \cup \{\beta\})$, and compute
  $$\begin{aligned} x_i^{\oplus} &= L((c_i^{\oplus})^{\lambda} \bmod n^2) \bmod n \ \ (\textit{for } i = 1, ..., l+1) \\ Acc_{X \cup X^{\oplus}} &= Acc_X + \sigma \textstyle\sum_{i=1}^{l+1} x_i^{\oplus} \bmod n \\ y_i^{\oplus} &= (c_i^{\oplus})^{\lambda \sigma \beta^{-1}} \bmod n^2 \ \ (\textit{for } i = 1, ..., l+1) \\ a_u &= \Pi_{i=1}^{l+1} y_i^{\oplus} \bmod n^2 \\ w_i^{\oplus} &= a_c a_u (c_i^{\oplus})^{-t_i^{\oplus} \beta^{-1}} \bmod n^2 \ \ (\textit{for } i = 1, ..., l) \end{aligned}$$
  Set $a_c = a_c a_u \bmod n^2, T = T \cup T^{\oplus}$, and $V = V \cup \{a_u\}$. Then output the new accumulated value $Acc_{X \cup X^{\oplus}}$ corresponding to the set $X \cup X^{\oplus}$, the witness $\pi_i^{\oplus} = (w_i^{\oplus}, t_i^{\oplus})$ for the new added elements $c_i^{\oplus}$ (for $i = 1, ..., l$), and the auxiliary information a_u and a_c.

- $\mathsf{DelEle}(X^{\ominus}, a_c, Acc_X, \mathcal{P})$:
  Given a set $X^{\ominus} = \{c_1^{\ominus}, ..., c_l^{\ominus}\}$ $(X^{\ominus} \subseteq X)$ to be deleted, the auxiliary information a_c, the accumulated value Acc_X, and the parameter $\mathcal{P}$, choose $c_{l+1}^{\ominus} \overset{R}{\leftarrow} \mathcal{C}$ and compute
  $$\begin{aligned} x_i^{\ominus} &= L((c_i^{\ominus})^{\lambda} \bmod n^2) \bmod n \ \ (\textit{for } i = 1, ..., l+1) \\ Acc_{X \setminus X^{\ominus}} &= Acc_X - \sigma \textstyle\sum_{i=1}^{l} x_i^{\ominus} + \sigma x_{l+1}^{\ominus} \bmod n \\ y_i^{\ominus} &= (c_i^{\ominus})^{\lambda \sigma \beta^{-1}} \bmod n^2 \ \ (\textit{for } i = 1, ..., l+1) \\ a_u &= y_{l+1}^{\ominus} \Pi_{j=1}^{l} (y_j^{\ominus})^{-1} \bmod n^2 \end{aligned}$$

Set $a_c = a_c a_u \bmod n^2$ *and* $V = V \cup \{a_u\}$*. Then output the new accumulated value* $Acc_{X \setminus X^{\ominus}}$ *corresponding to the set* $X \setminus X^{\ominus}$ *and the auxiliary information* a_u *and* a_c*.*

- $\mathsf{Verify}(c, \pi, Acc_X, PK)$*:*
  Given an element c*, its witness* $\pi = (w, t)$*, the accumulated value* Acc_X*, and the public key* PK*, test whether* $\{c, w\} \subset \mathcal{C}, t \in T'$ *and* $L(w^{\beta} c^{t} \bmod n^2) \equiv Acc_X (\bmod n)$*. If so, output* valid*, otherwise output* $\perp$*.*

- $\mathsf{UpdWit}(\pi_i, a_u, PK)$:
  Given the witness π_i*, the auxiliary information* a_u *and the public key* PK*, compute* $w'_i = w_i a_u \bmod n^2$ *then output the new witness* $\pi'_i = (w'_i, t_i)$ *for the element* c_i*.*

In the following section we show that Wang et al.'s construction is not secure.

5.3 An Attack on the Accumulator with Batch Update of Wang et al.

5.3.1 Problems with the proof

A security proof for the scheme was presented in the original paper [WWP07][2]. In their work, Wang et al. described a security reduction assuming the *Extended Strong RSA assumption* (or *es-RSA*), also proposed in [WWP08] as an analogous to the *Strong RSA assumption* [BP97] but relative to modulus n^2 instead of n. Unfortunately, there are two main problems in the proof. First, it states that adversary $\mathcal{B}$ must "run the KeyGen algorithm" which means it knows the factorization of the product $n = pq$, or at least knows $\phi(n^2)$ and $\lambda = lcm(p-1, q-1)$ since $\beta = \sigma\lambda \bmod n^2$ (see [WWP07]). Therefore, it is not clear how the reduction to break the *es-RSA* assumption can be achieved.

[2]As mentioned before, the subsequent paper [WWP08] fixes two correctness flaws in [WWP07] but does not give a new security proof. Our attack is against the improved version [WWP08].

The second problem is that, to break the *es-RSA* assumption, $\mathcal{B}$ needs to find non trivial values (y, s) such that $y^s = x \bmod n^2$ where x is given as input to $\mathcal{B}$. This value x does not seem to be mentioned in the proof.

5.3.2 Description of the attack

In order to show that the construction is not secure, i.e. the proof of security cannot be fixed, we present an attack. This attack considers the updated scheme [WWP08]. The idea is simply to delete an element from the set, and then update the witness of this element with the update information obtained by the execution of the algorithm DelEle. We then observe that this new witness is a valid one for the deleted element, which of course should not happen. We start with the set $X = \{c_1\}$ for some c_1, and let $x_1 = L(c_1^\lambda \bmod n^2) \bmod n$. Then, a random element c_* is chosen and $x_* = L(c_*^\lambda \bmod n^2) \bmod n$ is computed. The accumulated value is set to $v = \sigma(x_1 + x_*) \bmod n$. The witness value $\pi_1 = (w_1, t_1)$ for c_1 is defined by $\pi_1 = a_c c_1^{-t_1\beta^{-1}} \bmod n^2$ where $a_c = y_* y_1 \bmod n^2$, $y_1 = c_1^{\lambda\sigma\beta^{-1}} \bmod n^2$, $y_* = c_*^{\lambda\sigma\beta^{-1}} \bmod n^2$, and t_1 is random.

Then, the adversary asks the *manager* to delete element c_1. This means that the new accumulated value is $v' = v - \sigma x_1 + \sigma(x_{**}) \bmod n = \sigma(x_* + x_{**}) \bmod n$ where $x_{**} = L(c_{**}^\lambda \bmod n^2) \bmod n$ and c_{**} is random. The auxiliary value a_u used to update the witnesses is $a_u = y_{**} y_1^{-1} \bmod n^2$ where $y_{**} = c_{**}^{\lambda\sigma\beta^{-1}} \bmod n^2$. So, by updating the witness w_1 with a_u we obtain $w_1' = a_u w_1 \bmod n^2 = y_{**} y_1^{-1} y_* y_1 c_1^{-t_1\beta^{-1}} \bmod n^2 = y_{**} y_* c_1^{-t_1\beta^{-1}} \bmod n^2$. Then $w_1'^\beta c_1^{t_1} \equiv (y_{**} y_* c_1^{-t_1\beta^{-1}})^\beta c_1^{t_1} \bmod n^2 = (y_{**} y_*)^\beta \bmod n^2 = (c_{**} c_*)^{\lambda\sigma} \bmod n^2$. It follows that

$$\begin{aligned}
L(w_1'^\beta c_1^{t_1} \bmod n^2) &\equiv L((c_{**} c_*)^{\lambda\sigma} \bmod n^2) \bmod n \\
&\equiv \sigma(L(c_{**}^\lambda \bmod n^2) + L(c_*^\lambda \bmod n^2)) \bmod n \\
&\equiv \sigma(x_* + x_{**}) \bmod n \\
&\equiv v' \bmod n
\end{aligned}$$

This shows that (w_1', t_1) is a valid witness for the deleted element x_1. Therefore the scheme is not secure. Indeed the problem is simply that the information a_u allows *every* old witness, including w_1, to be updated. However such an update should not be possible.

5.4 A Lower Bound for Updating the Witnesses

The attack of the last section is an indication that the proposed construction may have some design flaws. In this section, we show that the problem indeed is more fundamental and the *batch update* property is essentially unrealizable. We argue this by presenting a lower bound on the size of $Upd_{X,X'}$, the information needed to update the witnesses after m changes (more precisely, deletions). Any deterministic update algorithm UpdWit must at least read $Upd_{X,X'}$, and so it also bounds the running time of any such algorithm in the case of sequential memory access.

Theorem 8 *Let* Acc *be a secure accumulator scheme with deterministic* UpdWit *and* Verify *algorithms. For an update involving* m *delete operations in a set of* N *elements, the size of the information* $Upd_{X,X'}$ *required by the algorithm* UpdWit *is* $\Omega(m \log \frac{N}{m})$*. In particular if* $m = \frac{N}{2}$ *we have* $|Upd_{X,X'}| = \Omega(m) = \Omega(N)$*.*

Proof. The idea of the proof is that the update information must encode a minimum amount of information in order for the accumulator scheme to be correct and secure. We prove this by considering a theoretical game between the accumulator *manager* and some user U. In the game, starting from an accumulated set X, the accumulator *manager* updates the accumulator in a way that is not known to user U (namely, the *manager* deletes some elements in an arbitrary set $X_d \subset X$) while still providing the update information $Upd_{X,X'}$ to U, where $X' = X \setminus X_d$. We prove that, as long as the scheme is correct and secure, there is a simple strategy that allows the user U to recover the exact changes made by the *manager*, that is, the set of deleted elements X_d. We conclude that the information provided by the manager to the user must at least encode a description of the set X_d. Details follow.

Consider the following game. The set accumulated at some point in time is
$X = \{x_1, x_2, \ldots, x_N\}$, and the corresponding accumulated value is Acc_X. We suppose the user possesses all the witnesses for each element in X and knows the accumulated value. Then m DelEle operations are performed, that is the new set obtained is $X' =$

$X \setminus X_d$ where $X_d = \{x_{i_1}, x_{i_2}, ..., x_{i_m}\}$. The *manager* computes the new accumulated value $Acc_{X'}$ and sends it to the user along with the update information $Upd_{X,X'}$ required to update all the witnesses that are still in X'.

Armed with this update information $Upd_{X,X'}$ and the new accumulated value $Acc_{X'}$, user U is able to reconstruct the set X_d of deleted elements as follows: For each element in X, the user checks if the corresponding witness can be successfully updated using algorithm UpdWit with input $Upd_{X,X'}$.

More specifically, the user computes $\pi'_x = \mathsf{UpdWit}(\pi_x, Upd_{X,X'}, PK)$ and checks whether or not π'_x is a valid witness. If not, that is, if $\mathsf{Verify}(x, \pi'_x, Acc_{X'}, PK) = \perp$, then the user U deduces that the element x must have been deleted from X.

Let us consider the set $Good = \{X_d \text{ can be reconstructed by user } U\}$. As the scheme must be correct and as algorithms $\mathsf{UpdWit}, \mathsf{Verify}$ are deterministic, we note that if x is still in X' the user U necessarily will be able to successfully update the corresponding witness π_x.

Moreover, as the scheme must be secure, we claim that

$$\Pr[X_d \notin Good] \leq \epsilon(\kappa)$$

where κ is the security parameter and ϵ is a negligible function. The reason is that this whole experiment can be transformed in an adversary that tries to find a fake witness choosing X_d at random. We deduce that the size $\#Good$ of the set $Good$ is such that

$$\#Good \geq \binom{N}{m}(1 - \epsilon(\kappa))$$

Hence, the user must be able to recompute the set of deleted elements X_d only from values $Upd_{X,X'}$ and $Acc_{X'}$, assuming that $X_d \in Good$.

We therefore conclude that $(Upd_{X,X'}, Acc_{X'})$ must contain at least the information required to encode any element of $Good$. There are at least $\binom{N}{m}(1-\epsilon(\kappa))$ elements in $Good$, so the minimum amount of information required is $\log(\binom{N}{m}(1-\epsilon(\kappa))$ bits. Since $\log(\binom{N}{m}) \geq m \log \frac{N}{m}$ we obtain that $|(Acc_{X'}, Upd_{X,X'})| = \Omega(m \log \frac{N}{m} + \log(1 - \epsilon(\kappa)))$. Recall that $|Acc_{X'}|$ must be sufficiently short (say, at most $O(\log(N))$, otherwise the accumulated value is not longer a "short representation" of the set, and the scheme is not really useful. Moreover ϵ vanishes to 0 when κ grows, so we can conclude that $|Upd_{X,X'}| = \Omega(m \log \frac{N}{m})$. The result then follows. ■

Given a the security parameter κ, the above theorem says that any update algorithm must take time at least $O(m) = O(N) = O(P(\kappa)) = \omega(t)$ for some *arbitrary* polynomial P, in order to *read* the input (in the case of sequential memory access). This goes beyond $O(t)$, the desired "constant time" in the number of changes.

Corollary 2 *Cryptographic accumulators with* batch update *(and deterministic update and verification) do not exist.*

5.5 Conclusion

This result shows that the *batch update* property as proposed in [FN02] essentially cannot be obtained, as the time to update all the witnesses cannot be linear in t, the time required to verify a single witness, but it must be at least $O(P(\kappa)) = \omega(t)$ for some polynomial P. Notice that our lower bound is not tight since Camenisch and Lysyanskaya's accumulator requires $O(P(\kappa) \cdot t)$ time to update the witnesses after $O(P(\kappa))$ changes. Nonetheless, in principle, it leaves some (potential) room to improve their construction by at most a factor of t.

Finally, one may consider getting around this impossibility result by not allowing deletions in the set. Unfortunately, such an accumulator can be trivially implemented by signing the elements of the set, as in this case there is no replay-attack. The witness for every element consists in its signature under the manager's private key, and clearly needs not to be updated.

Short Transitive Signatures for Directed Trees

Contents

6.1 Introduction

In this chapter, we describe a new construction for a `TSDT` scheme that enjoys better worst-case complexity than Neven's [Nev08]. We obtain a scheme where, for any $\lambda \geq 1$, (a) signing or verifying an edge signature requires $O(\lambda)$ cryptographic operations, and (b) computing (without the secret key) an edge signature in the transitive closure of the tree requires $O(\lambda(\frac{N}{\kappa})^{1/\lambda})$ cryptographic operations. The signature size is substantially improved: Our signatures are only $O(\kappa\lambda)$ bits, where κ is the security parameter. In particular, if $\lambda = \log(N)$, then signatures are $O(\kappa \log(N))$ bits, while allowing efficient signature computation ($O(\log(N)$ time). Alternatively, by setting for example $\lambda = 2$, we obtain optimal signature sizes of $O(2 \cdot \kappa) = O(\kappa)$ bits if we are willing to afford $O(\sqrt{\frac{N}{\kappa}})$ computation.

Our Approach. There are two main ideas in our construction. First, we use the following fact observed by Dietz [Die82]. Let `Pre` and `Post` be the two strings representing the sequences of nodes obtained by pre-order and post-order (respectively) traversal of a given tree $\mathcal{T}$. Dietz observed that 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`. This property captures the fact that the tree is directed (from top to bottom) and gives us a characterization of the existence of a path between two nodes. Armed with this result, we reduce the problem of deciding whether there is a path between vertices a and b, to comparing the position of a and b in a string sequence $\mathcal{S}$. Doing this efficiently is not trivial as the tree can grow, which means the string $\mathcal{S}$ dynamically changes. An *order data structure* – a concept also introduced by Dietz [Die82] – does exactly what we need: It supports element insertions into a sequence while still providing an efficient method to decide element order. Roughly speaking, we implement such data structure via a binary search tree $\mathcal{B}$, where each pair of elements x and y in $\mathcal{S}$ are associated to nodes $x, y \in \mathcal{B}$ (respectively), each with efficiently computable short labels $\ell(x)$ and $\ell(y)$. We then are able to define the relation "a appears before b in the sequence $\mathcal{S}$" as a total order relation $\prec$ which can be efficiently evaluated only from $\ell(a)$ and $\ell(b)$.

To achieve this, we use a labelling technique – based on *tries* [Fre60] – which allows efficient and *incremental* computations of labels for new elements. Any newly inserted element v in $\mathcal{T}$ is mapped to a node v in $\mathcal{B}$ whose label $\ell(v)$ will share all but the last bit with another already computed label (see details in Section 6.4). Thus, whether an element a comes before some other element b in $\mathcal{S}$, can then be efficiently tested by lexicographical comparison between the labels associated with the corresponding nodes in $\mathcal{B}$. With this at hand, we then use *two* of these data structures to keep track of `Pre` and `Post` lists and to test Dietz's condition on any pair of elements.

The problem, however, is that labels of $O(N)$ bits are now associated to vertices of the N-node tree $\mathcal{T}$, so at first sight, little has been gained: Signature length is now

$O(N)$ bits compared to $N \log N$ bits in Neven's construction.

That is when our second idea comes into play: We use a CRHF that preserves the predicate `CommonPrefix`, defined as follows: Given $A, B \in \Sigma^m$, for some $m \in \mathbb{N}$, $\texttt{CommonPrefix}(A, B, i) = 1$ if, and only if, A and B share a common prefix up to position i. This tool allows us to prove Dietz's condition using only *hashed* labels (and a constant size proof), effectively compressing the signature. Thus, we propose a generic construction for `TSDT` schemes based on the new primitive mentioned above. Morevover, we describe a concrete instantiation for a CRHF that preserves the `CommonPrefix` predicate and whose security relies on the *N-Bilinear Diffie Hellman Inversion* assumption.

We further improve our construction by showing 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 a novel tradeoff $O(\lambda(\frac{N}{\kappa})^{1/\lambda})$ vs. $O(\lambda)$ for $\lambda \geq 1$ between the time to compute a proof versus the time to verify a proof.

Our solution, like previous ones, is stateful. However, we note that our construction does not allow the *combiner* to compute edge signatures without using a state shared with the *signer*. This means that, strictly speaking, signature edge composition is impossible in our construction (contrary to Neven's [Nev08] where a state is also required but only for signing edges). Nonetheless, the functionality of computing signatures for any edge in the transitive closure of the tree (which is done by the *combiner*) – without relying on the *signer*'s secret – is maintained. To the best of our knowledge, if we require these edge signatures to have constant size in the number of nodes of the tree, avoiding the use of a state for edge signature composition is an open problem.

6.2 A CRHF that preserves the *Common Prefix* predicate

In this section we describe a CRHF that implements the `CommonPrefix` predicate mentioned above. We first observe that using `CommonPrefix` predicate we can implement other useful predicates over strings such as `Compare`, defined as $\texttt{Compare}(A, B) = 1$ if, and only if, $A \prec B$ (here $\prec$ is the extended lexicographical order): If $A \prec B$, it follows that there exists a (possibly empty) common prefix C for A and B and symbols σ, σ', such that (1) $D = C||\sigma$ 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 short proofs for `CommonPrefix`, using incremental hashing we can compare any two strings by only their hash values.

Definitions. Let $\texttt{PP} \in \{0,1\}^\kappa$ some public parameter, and $N \in \mathbb{N}$ a bound on the size of the input[1] which is a polynomial in κ. We denote by $\mathcal{H}$ a hash function family.

Definition 34 *(CRHF that preserves* `CommonPrefix` *- Syntax) A hash function family $\mathcal{H}$ that preserves the* `CommonPrefix` *predicate is a 4-tuple of algorithms* $(\mathsf{HGen}, \mathsf{HEval}, \mathsf{ProofGen}, \mathsf{ProofCheck})$ *where:*

- $\mathsf{HGen}(1^\kappa, N)$*: Given a bound N on the length of the strings to hash, this probabilistic algorithm returns a public parameter* $\texttt{PP}$*. Value* $\texttt{PP}$ *implicitly defines a hash function* $\mathsf{H} = \mathsf{H}_{\texttt{PP},N,\kappa} \in \mathcal{H}$ *where* $\mathsf{H} : \{0,1\}^N \rightarrow \{0,1\}^\kappa$.

- $\mathsf{HEval}(M, \texttt{PP})$*: Given $M \in \{0,1\}^N$, this* deterministic *algorithm efficiently computes and returns the string* $\mathsf{H}(M) \in \{0,1\}^\kappa$.

- $\mathsf{ProofGen}(A, B, i, \texttt{PP})$*: 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* $\mathsf{ProofCheck}$ *algorithm.*

- $\mathsf{ProofCheck}(H_A, H_B, i, \pi, \texttt{PP})$*: A deterministic algorithm that, given two hash values $H_A, H_B \in \{0,1\}^\kappa$ and a proof $\pi \in \{0,1\}^\kappa$, returns either* valid *or* $\perp$.

[1] Here we intentionally use the same variable name N for the size of the input of the hash function as well as the number of nodes of the tree. Indeed, our full construction for trees of N nodes presented in section 6.4.2 requires hashing N-bit strings.

The scheme is said to be correct if for any strings A, B and $i \in \mathbb{N}$ such that $\texttt{CommonPrefix}(A, B, i) = 1$, and $\pi = \mathsf{ProofGen}(A, B, i, \mathsf{PP})$, we have that $\mathsf{ProofCheck}$ on inputs $(\mathsf{H}(A), \mathsf{H}(B), i, \pi, \mathsf{PP})$ returns valid.

The security requirement is the following: For any PPT adversary $\mathcal{A}$, it has to 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 $\mathsf{ProofCheck}(\mathsf{H}(A), \mathsf{H}(B), \pi, i, \mathsf{PP})$ 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 guarantees that the hash values $\mathsf{H}(A)$ and $\mathsf{H}(B)$ have been correctly computed.

Definition 35 *(Secure preservation of the* `CommonPrefix` *predicate) Let $\mathcal{H}$ be a family of hash functions with common-prefix proofs and $\mathcal{A}$ a PPT adversary. The* `CommonPrefix` *advantage of $\mathcal{A}$ is*

$$Adv_{\mathcal{H}}^{\texttt{CommonPrefix}}(\mathcal{A}, \kappa, N) = \Pr\left[\begin{array}{c} \mathsf{PP} \leftarrow \mathsf{HGen}(1^\kappa, N); A, B, \pi, i \leftarrow \mathcal{A}(1^\kappa, N, \mathsf{PP}) : \\ A[1..i] \neq B[1..i] \wedge H_A = \mathsf{H}(A) \wedge H_B = \mathsf{H}(B) \wedge \\ \mathsf{ProofCheck}(H_A, H_B, i, \pi, \mathsf{PP}) = \mathsf{valid} \end{array}\right]$$

We say $\mathcal{H}$ preserves securely `CommonPrefix` *if, for every PPT $\mathcal{A}$, we have $Adv_{\mathcal{H}}^{\texttt{CommonPrefix}}(\mathcal{A}, \kappa, N) = \mathsf{neg}(\kappa)$.*

The following proposition states that a hash function family that preserves the `CommonPrefix` predicate must be collision-resistant.

Proposition 8 *Let $\mathcal{H}$ be a family of hash functions that preserves securely the* `CommonPrefix` *predicate. Then $\mathcal{H}$ is a collision-resistant hash function family.*

Proof. Assume an adversary $\mathcal{A}$ finds A, B two strings of size N with $A \neq B$ such that $\mathsf{H}(A) = \mathsf{H}(B)$. Due to the correctness of the scheme it is trivial to compute a proof π such that A shares a common prefix of length N with itself. As $A \neq B$ we can deduce that (A, B, π) is a forgery for $\mathcal{H}$ relative to the predicate `CommonPrefix`.
■

The Construction. The description of the hash function H is a tuple $(g, g^s, g^{s^2}, ..., g^{s^N})$ of the N-BDHI problem. We assume it has been computed securely by a trusted third party or using multi-party computations techniques. The basic idea is to represent a N-bit binary string M by $\mathsf{H}(M) \overset{\text{def}}{=} g^{M[1]s} \cdot g^{M[2]s^2} \cdots g^{M[N]s^N}$. Now if some message M' is equal to M up to position i then the value $\Delta = \frac{\mathsf{H}(M)}{\mathsf{H}(M')} = \prod_{j=i+1}^{N} g^{c_j s^j}$, where $c_j \in \{-1, 0, 1\}$, will be a product of powers of g^{s^j} for $1 \leq j \leq N$ where for all $j \leq i$ the exponents are equal to 0. Yet, the related value $\pi = \prod_{j=i+1}^{N} g^{c_j s^{j-(i+1)}}$ can easily be computed given M,M' and H. The intuition behind the proof is that as M and M' are equal up to position i, then we can represent the difference between M and M' using only $N - i$ positions. Thus, verifying proof π simply consists in testing to see if by using the bilinear map, we can "shift forward" the exponents in the proof by i positions to obtain Δ. More precisely, π will be a valid proof for $\mathsf{H}(M), \mathsf{H}(M')$ if, and only if, $e(\frac{\mathsf{H}(M)}{\mathsf{H}(M')}, g) = e(\pi, g^{s^{i+1}})$. Details follow.

Construction 11 *(A CRHF that preserves* `CommonPrefix`*) Let* `PH` *be the scheme defined by the following algorithms:*

- $\mathsf{HGen}(1^\kappa, N)$*: Run* $\mathsf{BMGen}(1^\kappa)$ *to obtain* $P = (p, \mathbb{G}, \mathbb{G}_T, e, g)$*. Let* $s \overset{R}{\leftarrow} \mathbb{Z}_p$*, and* $T = (g, g^s, g^{s^2}, ..., g^{s^N})$*. Return* $\mathtt{PP} = (P, T)$*.*
- $\mathsf{HEval}(M, \mathtt{PP})$*:* $M \in \{0,1\}^N$*. Compute* $\mathsf{H}(M) = \prod_{j=1}^{N} g^{M[j]s^j}$*. Return* $\mathsf{H}(M)$*.*
- $\mathsf{ProofGen}(A, B, i, \mathtt{PP})$*: Given* N*-bit strings* A, B*, let* C *be the array such that* $\forall j \in \{1, \ldots, N\} : C[j] = A[j] - B[j]$*. Return* $\pi = \prod_{j=i+1}^{N} g^{C[j]s^{j-(i+1)}}$*.*
- $\mathsf{ProofCheck}(H_A, H_B, \pi, i, \mathtt{PP})$*: Compute* $\Delta = \frac{H_A}{H_B}$*. If* $i = N$ *and* $\Delta = 1$ *return* valid*. If* $i < N$ *return* valid *if* $e(\Delta, g) = e(\pi, g^{s^{i+1}})$*, otherwise return* $\perp$*.*

Proposition 9 *Under the N-BDHI assumption the hash function family defined by the scheme* `PH` *securely preserves the predicate* `CommonPrefix`*.*

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 the public parameter $\mathtt{PP} = (P, T = (g, g^s, g^{s^2}, \ldots, g^{s^N}))$ as input, it forwards them to $\mathcal{A}$. Eventually $\mathcal{A}$ will output values A, B, π, i such that $\mathsf{ProofCheck}(\mathsf{H}(A), \mathsf{H}(B), i, \pi, \mathtt{PP}) = \mathsf{valid}$. Then, $\mathcal{B}$ computes the array C defined as $C[j] = A[j] - B[j] = c_j$ for

$j \in \{1, \ldots, N\}$. Let k be the smallest index such that $c_k \neq 0$. Clearly $i - k > 0$ since $A[1..i] \neq B[1..i]$. From the validity of π, we have that $e(\Delta, g) = e(\pi, g^{s^{i+1}})$, and thus $\pi = \Delta^{\frac{1}{s^{i+1}}}$. 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 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. The family $\mathcal{H}$ is incremental [BGG94]: Let $\mathsf{H} \in \mathcal{H}$ be a hash function, $M = M[1..N] \in \{0,1\}^N$ be any message, and M' another message such that $M'[j] = M[j]$ for all $j \in [N]$ except for $j = i$ where $i \in [N]$. We have that $\mathsf{H}(M) = \prod_{j=1}^{N} g^{M[j]s^j}$ and thus $\mathsf{H}(M') = \mathsf{H}(M) \cdot g^{(M'[i]-M[i])}s^i$. In terms of efficiency, both the computation of the hash function and the proof are easily parallelizable as they involve only group *multiplications*. In particular, with $O(N)$ processors, we can compute a proof using only $O(\log N)$ (sequential) group multiplications. Also, note that handling strings of length $m > N$ can be done dynamically, *without having to re-compute 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 $g^{s^{N+1}}, \ldots, g^{s^m}$.

6.3 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 compute $\mathsf{H}(S) = \prod_{i=1}^{N/\kappa} g^{\sigma_i s^i}$ 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 [2] required to compute a proof from $O(N)$ to $O(\frac{N}{\kappa})$ [3]. Although simple,

[2]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.

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

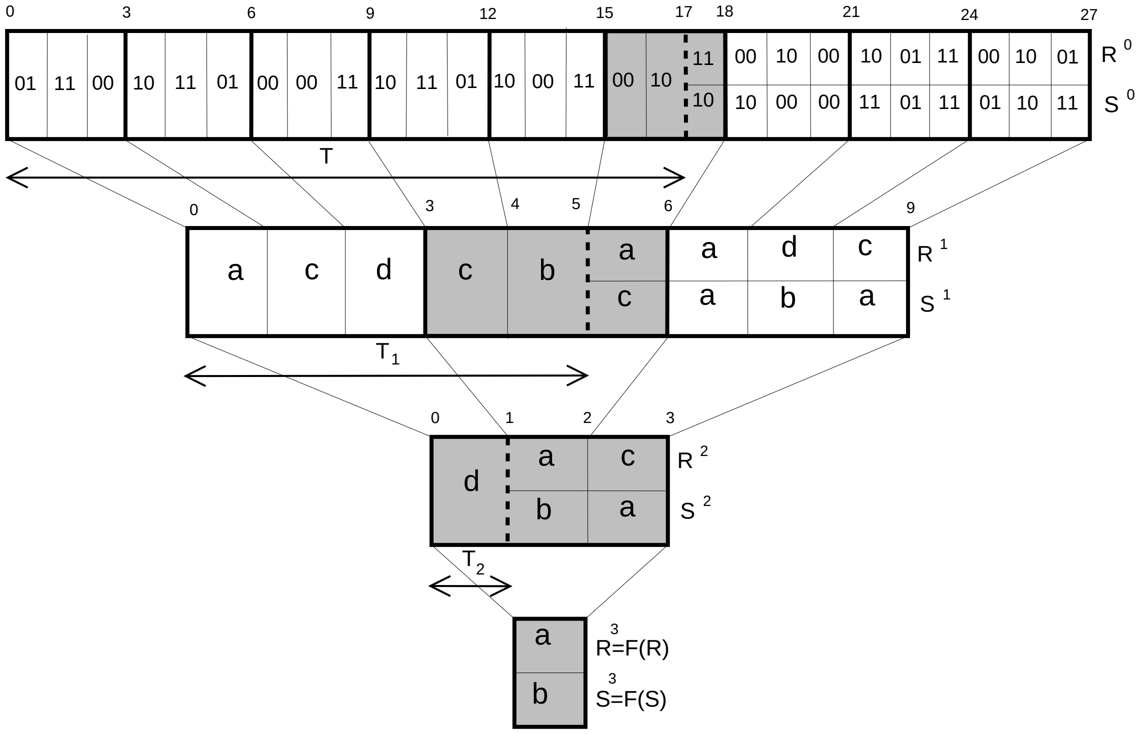


Figure 6.1: Toy example of tradeoff data structure.

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 $i = i_0 = 17 \cdot 2 = 34$. At level 3 we have the final hash values for $R = R^0$ and $S = S^0$ which are respectively R^3 and S^3. We move now to level 2. The *combiner* shows that R^2 and S^2 are equal up to the position $i_2 = 1$ using hash values $R^3 = \mathsf{F}(R)$ and $S^3 = \mathsf{F}(S)$. Then, it 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 $\mathsf{G}^{-1}(a) = cba$ and $\mathsf{G}^{-1}(b) = cbc$. So we move up to level 1. Now the *combiner*, using $\mathsf{G}^{-1}(a)$ and $\mathsf{G}^{-1}(b)$, 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 to 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 $\mathsf{G}^{-1}(a) = 001011$ and $\mathsf{G}^{-1}(c) = 001010$ 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 $i = i_0 = 34 = (5 \cdot 3 + 2) \cdot 2 = (i_1 \cdot t + j_0)\kappa$.

this observation is crucial for the following tradeoff.

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 scheme such that, for any $\lambda \geq 1$, the time it takes to compute a proof is reduced to $O(\lambda(\frac{N}{\kappa})^{1/\lambda})$ 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, that $\frac{N}{\kappa}$ is a power of t. In order to compute $\mathsf{H}(S)$, we first cut S in chunks of size $t\kappa$ bits.

Let $\mathsf{G} : \{0,1\}^{t\kappa} \rightarrow \{0,1\}^{\kappa}$ be a CRHF that preserves the common-prefix predicate (that is G operates as H but takes $t\kappa$ bits (i.e. t symbols) as input). For each chunk $S_0, S_1, ..., S_{N/(t\kappa)-1}$ we compute the hash value $\mathsf{G}(S_i)$ and obtain, by concatenating $(\mathsf{G}(S_0), \mathsf{G}(S_1), \ldots, \mathsf{G}(S_{N/(t\kappa)-1}))$, 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 $S^{\frac{\log(N/\kappa)}{\log t}}$ with only one symbols (i.e. κ bits). The final output corresponds to the output of the common-prefix preserving CRHF $\mathsf{F} : \{0,1\}^N \rightarrow \{0,1\}^{\kappa}$.

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 $R^0, R^1, \ldots, R^{\lambda-1}, R^{\lambda}$ and $S^0, S^1, \ldots, S^{\lambda-1}, S^{\lambda}$ 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 λ. Here we have the final hash value of κ bits. At the next level, $\lambda - 1$, there is only one chunk of size $t = (\frac{N}{\kappa})^{1/\lambda}$ (number of symbols). Using $\mathsf{ProofGen}$, 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 some position $i_{\lambda-1}$. 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 $R^{\lambda-1}$ (where $\sigma_{R,\lambda-1}$ is a symbol), and another one (say $\pi_{\lambda-1,S}$), showing that $T^{\lambda-1}||\sigma_{S,\lambda-1}$ (where $\sigma_{S,\lambda-1}$ is also a symbol) 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 G: the $t\kappa$-bit long strings $R^{\lambda-2}_{i_{\lambda-1}t} = \mathsf{G}^{-1}(\sigma_{R,\lambda-1})$ (that is the chunk number $i_{\lambda-1}t$ of $R^{\lambda-2}$) and similarly $S^{\lambda-2}_{i_{\lambda-1}t} = \mathsf{G}^{-1}(\sigma_{S,\lambda-1})$. The *combiner* then moves up one level and repeats the procedure at level $\lambda-2$ now working on the strings $\mathsf{G}^{-1}(\sigma_{R,\lambda-1}), \mathsf{G}^{-1}(\sigma_{S,\lambda-1})$ and generating a proof (say $\pi_{\lambda-2}$) that they have some $j_{\lambda-2}$-symbol common prefix (inside the blocks relative to $\mathsf{G}^{-1}(\sigma_{R,\lambda-1})$ and $\mathsf{G}^{-1}(\sigma_{S,\lambda-1})$). 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 6.1) where $i_0 = (i_1 t + j_0)\kappa$, with i_0 the absolute position at level 0, i_1 the absolute position at level 1 and j_0 the relative position at level 0. 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 $\mathsf{G}^{-1}(\sigma_{R,\ell-1})$ and $\mathsf{G}^{-1}(\sigma_{S,\ell-1})$. These considerations lead to the following result.

Theorem 9 *Let $\lambda \geq 1$. Under the N-BDHI assumption we can build a CRHF that securely preserves the* `CommonPrefix` *predicate where (a) the time to compute a hash value is $O(\lambda)$, (b) the time to compute a proof is $O(\lambda(\frac{N}{\kappa})^{1/\lambda})$, and (c) the time to verify a proof is $O(\lambda)$.*

Proof. First we observe that the mappings between each level are a collision-resistant hash function family. We denote by F_i the mapping between level 0 and level i. Also note that finding a collision for F_i enables to find a collision for G described previously.

Assume an adversary $\mathcal{A}$ manages to break the security of the tradeoff scheme. Then we build an adversary $\mathcal{B}$ that breaks the (simple) `PH` by computing a forged proof or finding a collision for G. Adversary $\mathcal{B}$ sends the public parameters of the scheme to $\mathcal{A}$ who answers with a forgery for the tradeoff 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 $R[k] \neq S[k]$. Consider level λ where R^λ and S^λ are the final hash values. Let $t = (\frac{N}{\kappa})^{1/\lambda}$. Using the pre-images of t bits present at level $\lambda-1$ and the proof that the pre-images share a common prefix until position $i_{\lambda-1}$ we consider three cases:

1. We are in presence of a forgery, then $\mathcal{B}$ has broken the security of the simple `PH`.

2. There is no proof forgery, but we have that $R[1..(i_{\lambda-1}t^{\lambda-1})] \neq S[1..(i_{\lambda-1}t^{\lambda-1})]$ then we found a collision for the mapping $\mathsf{F}_{\lambda-1}$.

3. There is no proof forgery nor collision.

If we are in case (3) we repeat the procedure at the next level $\lambda - 2$ and so forth until reaching level 0. Eventually we will find a collision for G or a forgery for `PH`, as otherwise the adversary $\mathcal{A}$ would not have won. ■

6.4 Short Transitive Signatures for Directed Trees

Our construction for `TSDT` is based on the following idea: Handling a growing tree can be reduced to maintaining two ordered sequences, one corresponding to the pre-order tree traversal and another to the post-order tree traversal in a depth-first search. This was first observed by Dietz [Die82].

Proposition 10 *([Die82]) Let $\mathcal{T}$ be a tree of N nodes 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, consider the tree depicted in Figure 6.2, last row, first column. For that tree, `Pre` $= acdbe$ and `Post` $= dcbea$. Since 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 from some node x to another node y 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 in using this result is that the ordered sequences are dynamic – new elements can be inserted between any two existent elements. More concretely, in order to use Dietz's property for our problem we require to:

- assign labels to the nodes of the tree so that these labels can be compared,
- create these labels dynamically, as the directed tree we want to sign grows, and
- produce fixed labels, as recomputing all or part of them after a node insertion would be unpractical.

It is the role of the *signer* to handle these labels so that the *combiner* can use them in order to convince a *verifier* that there is a path between two nodes.

This problem is addressed by the so called *order data structure* [Die82, MRB$^+$02]. Such a data structure allows us to compare any pair of labels and also insert a new label so that it may lie between two existing ones. A naive way - mentioned in [Die82] - to implement the proposed data structure would be to consider the interval $[0..2^N - 1]$ for the labels; to insert an element between X and Y one would use label $Z = \lfloor \frac{X+Y}{2} \rfloor$. This way, we can always find the label for an element between any two others and the comparison algorithm consists in comparing the labels. Unfortunately, this solution does not suffice for our application since the string representation of a new label cannot be easily obtained from already computed labels, and the *signer* must sign labels of length N for each new edge. So our first improvement is a new order data structure with the following property: If X and Y are two consecutive labels (that is $X \prec Y$ and there is no Z between X and Y) then every new computed label Z such that $X \prec Z \prec Y$ will share all bits except one with X or Y.

Before describing our construction we introduce the formal definition of order data structure [MRB$^+$02]. Jumping ahead, we use this data structure to efficiently create and update labels in $\mathcal{U} = \{0,1\}^*$, as well as to compare them using the extended lexicographical order as the relation $\prec_{\mathcal{U}}$. The particular mapping between elements in lists `Pre` and `Post` to labels will depend on our construction.

Definition 36 *Let $(\mathcal{U}, \prec_{\mathcal{U}})$ be a totally ordered set of labels. An order data structure over $\mathcal{U}$ consists of three algorithms:*

- $\mathsf{ODSetup}()$: *Initialize the data structure and in particular return two bounds $-\infty$ and $+\infty$ such that for every element $X \in \mathcal{U}$ we have $-\infty \prec X \prec +\infty$.*

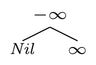
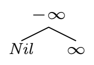
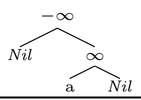
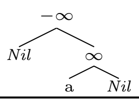
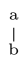
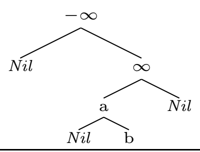
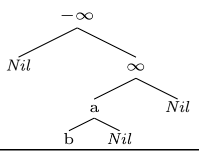
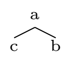
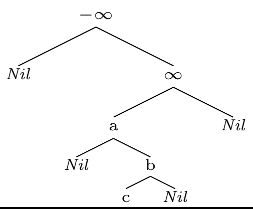
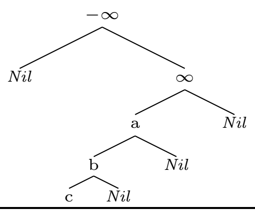
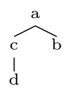
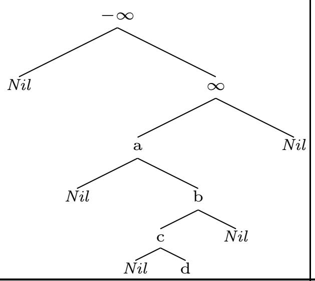
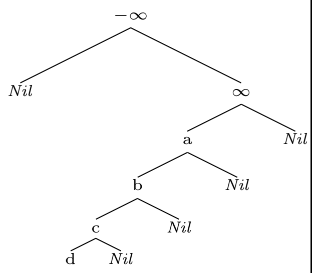
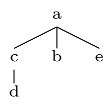
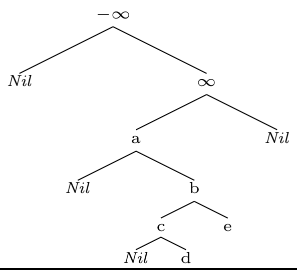
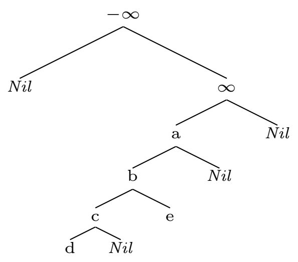
Step	**Tree $\mathcal{T}$**	**Pre/Post order**	`OrderDS`$_{\texttt{Pre}}$	`OrderDS`$_{\texttt{Post}}$	**Labels**
0	*Nil*	`Pre` = *Nil* `Post` = *Nil*			$\ell_{\texttt{Pre}}[-\infty] = \$$ $\ell_{\texttt{Pre}}[\infty] = 1\$$ $\ell_{\texttt{Post}}[-\infty] = \$$ $\ell_{\texttt{Post}}[\infty] = 1\$$
1	a	`Pre` = a `Post` = a			$\ell_{\texttt{Pre}}[a] = 10\$$ $\ell_{\texttt{Post}}[a] = 10\$$
2		`Pre` = ab `Post` = ba			$\ell_{\texttt{Pre}}[b] = 101\$$ $\ell_{\texttt{Post}}[b] = 100\$$
3		`Pre` = acb `Post` = cba			$\ell_{\texttt{Pre}}[c] = 1010\$$ $\ell_{\texttt{Post}}[c] = 1000\$$
4		`Pre` = $acdb$ `Post` = $dcba$			$\ell_{\texttt{Pre}}[d]$ = 10101\$ $\ell_{\texttt{Post}}[d]$ = 10000\$
5		`Pre` = $acdbe$ `Post` = $dcbea$			$\ell_{\texttt{Pre}}[e] = 1011\$$ $\ell_{\texttt{Post}}[e] = 1001\$$

Figure 6.2: 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 `OrderDS`$_{\texttt{Pre}}$ and `OrderDS`$_{\texttt{Post}}$ contain two nodes $-\infty$ and $+\infty$ 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 – through a call to `OrderDS`$_{\texttt{Pre}}$.ODInsert$(-\infty, \infty)$ and `OrderDS`$_{\texttt{Post}}$.ODInsert$(-\infty, \infty)$ – in such a way they reflect the order $-\infty \prec a \prec \infty$. In particular we have that labels $\ell_{\texttt{Pre}}[a] = \ell_{\texttt{Post}}[a] = 10\$$. The end marker \$ is appended to the label so it allows direct order label comparison through the standard lexicographical order assuming 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. After invoking `OrderDS`$_{\texttt{Pre}}$.ODInsert$(a, +\infty)$ and `OrderDS`$_{\texttt{Post}}$.ODInsert$(-\infty, a)$, the order data structure tries for `Pre` and `Post` are updated as follows: As b comes after a in `Pre` we have that b is the right child of a in `OrderDS`$_{\texttt{Pre}}$. In `OrderDS`$_{\texttt{Post}}$, b is the left child of a 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 $\ell_{\texttt{Pre}}[v]$ and $\ell_{\texttt{Post}}[v]$ 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 $\ell_{\texttt{Pre}}[a] = 10\$$ and $\ell_{\texttt{Pre}}[d] = 10101\$$ which means $\ell_{\texttt{Pre}}[a] \prec \ell_{\texttt{Pre}}[d]$. Also we can check that $\ell_{\texttt{Post}}[d] = 10000\$ \prec \ell_{\texttt{Post}}[a] = 10\$$.

- ODInsert(X, Y) : *Let X and Y be two consecutive labels. Compute a new element Z such that $X \prec_{\mathcal{U}} Z \prec_{\mathcal{U}} Y$.*

- ODCompare(X, Y)*: Return* valid *if, and only if, $X \prec_{\mathcal{U}} Y$.*

Our construction for order data structure uses binary tries (which are a special kind of trees) [Fre60] to handle labels. In order to avoid confusion, the trees used in the order data structure will be denoted by the letter $\mathcal{B}$ and we will use the term *tries* when referring to them. Moreover, we will keep using the symbol $\mathcal{T}$ for the directed tree we want to sign and also use the term *tree* in this case. As mentioned in Section 2.1, in a trie, the label for a node is obtained by concatenating the labels on the edges in a path from the root to the node. Comparing two labels then consists of comparing the labels as binary strings w.r.t. the *extended* lexicographical order.

Construction 12 . *Let* `OrderDS` *be the order data structure over universe $(\{0, 1\}^*, \prec)$ defined by the following operations:*

- ODSetup()*: Create a trie $\mathcal{B}$ with two nodes, a root for the bound $-\infty$ and its right child for the bound $+\infty$. The label of the root is ϵ (the empty string) and the label of the right child is* 1*. Intuitively $-\infty$ represents the lowest element of the universe and $+\infty$ 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).*

- ODCompare(X, Y) : *Return* 1 *if, and only if, $X \prec Y$, that is X is lower than Y w.r.t. to the* extended lexicographical order*.*

We observe that in the worst case the largest path of the trie 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 $\mathsf{H}(Z)$ from $\mathsf{H}(X)$ or $\mathsf{H}(Y)$.

6.4.1 Basic Construction

Our first construction is based only on standard digital signatures, as Neven's construction. The scheme is as follows: Each time an edge (and thus a vertex) is inserted into the tree $\mathcal{T}$, two lists $\mathtt{Pre}$ and $\mathtt{Post}$ corresponding the the pre-order and post-order traversals of $\mathcal{T}$ respectively are updated with the newly inserted element. We efficiently implement the latter by maintaining two order data structures (see Figure 6.2) $\mathtt{OrderDS}_{\mathtt{Pre}}$ and $\mathtt{OrderDS}_{\mathtt{Post}}$, one for each list. More precisely, each node $x \in \mathcal{T}$ is associated with a label $\ell_{\mathtt{Pre}}[x]$ (resp. $\ell_{\mathtt{Post}}[x]$) computed by the order data structure for pre-order (resp. post-order). Note that the labels $\ell_{\mathtt{Pre}}[x]$ and $\ell_{\mathtt{Post}}[x]$ will mention explicitly the end marker symbol \$. The reason to introduce this change is that in the full construction these labels will be hashed and, given that our hash function introduced in Section 6.2 considers fixed length inputs, we need to rely on this extra symbol to avoid trivial collisions. Thus, comparing two labels will be implemented through *standard* lexicographical order which is why we will use the operator $<$ instead of $\prec$ henceforth.

We use $\mathsf{ODCompare}$ to evaluate if x appears before some y in $\mathtt{Pre}$ (resp. $\mathtt{Post}$), which simply verifies that $\ell_{\mathtt{Pre}}[x] < \ell_{\mathtt{Pre}}[y]$ and $\ell_{\mathtt{Post}}[y] < \ell_{\mathtt{Post}}[x]$.

Construction 13 *(*$\mathtt{TSDT}$ *from Standard Digital Signatures) Let* $\mathtt{SSig} = (\mathsf{SKG}, \mathsf{SSig}, \mathsf{SVf})$ *be a standard digital signature scheme. The scheme* $\mathtt{BasicTSDT}$ *is as follows.*

- $\mathsf{TSKG}(1^\kappa)$: *Run* 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* $\mathtt{OrderDS}_{\mathtt{Post}}$ *to maintain the sequences for pre-order and post-order tree traversal respectively. Set* $\mathcal{T} = (V, E)$ *as the empty tree. Define two tables* $\ell_{\mathtt{Pre}}[\cdot]$ *and* $\ell_{\mathtt{Post}}[\cdot]$ *that map* *nodes in $\mathcal{T}$ to their respective labels in* $\texttt{OrderDS}_{\texttt{Pre}}$ *and* $\texttt{OrderDS}_{\texttt{Post}}$ *respectively. That is, if* $x \in V$, $\ell_{\texttt{Pre}}[x]$ *will be the label bound to* x *in* $\texttt{OrderDS}_{\texttt{Pre}}$ *and* $\ell_{\texttt{Post}}[x]$ *will be the label bound to* x *in* $\texttt{OrderDS}_{\texttt{Post}}$*. Set* $\ell_{\texttt{Pre}}[-\infty] = \ell_{\texttt{Post}}[-\infty] = \$$ *and* $\ell_{\texttt{Pre}}[+\infty] = \ell_{\texttt{Post}}[+\infty] = 1\$$*. Return* (tsk, tpk)*.*

- $\mathsf{TSign}(tsk, a, b)$: *If both* $a, b \in V$ *or if the insertion does not preserve the tree structure of* $\mathcal{T}$*, return* $\perp$*. If both* $a, b \notin V$ *then insert* a *in* $\texttt{OrderDS}_{\texttt{Pre}}, \texttt{OrderDS}_{\texttt{Post}}$*, that is such that* $-\infty < a < +\infty$ *in lists* $\texttt{Pre}$ *and* $\texttt{Post}$*. Then insert* b *in* $\texttt{OrderDS}_{\texttt{Pre}}, \texttt{OrderDS}_{\texttt{Post}}$ *so to reflect the pre-order list* $\texttt{Pre} = ab$ *and post-order list* $\texttt{Post} = ba$ *respectively.*

  Otherwise, let $z \in \{a, b\}$ *be the new vertex not yet in* V *and* $x \in \{a, b\} \setminus \{z\}$ *be the other. Insert edge* (a, b) *in* $\mathcal{T}$*, and update* $\texttt{OrderDS}_{\texttt{Pre}}, \texttt{OrderDS}_{\texttt{Post}}$ *data structures to reflect the new pre-order and post-order tree traversal of* $\mathcal{T}$ *as follows. Let* y *be the element in* $\texttt{Pre}$ *such that* z *lies (strictly) between* x *and* y*. Assume that* $x < z < y$ *(the other case is similar). Compute* $\texttt{OrderDS}_{\texttt{Pre}}.\mathsf{ODInsert}(\ell_{\texttt{Pre}}[x], \ell_{\texttt{Pre}}[y])$ *to obtain* $\ell_{\texttt{Pre}}[z]$ *the label associated to* z *in* $\texttt{OrderDS}_{\texttt{Pre}}$*. Similarly obtain* $\ell_{\texttt{Post}}[z]$ *the label associated to* z *in* $\texttt{OrderDS}_{\texttt{Post}}$*: That is, compute* $\ell_{\texttt{Post}}[z] = \texttt{OrderDS}_{\texttt{Post}}.\mathsf{ODInsert}(\ell_{\texttt{Post}}[y], \ell_{\texttt{Post}}[x])$*.*

  Then, compute $M_z = (z, \ell_{\texttt{Pre}}[z], \ell_{\texttt{Post}}[z])$ *and sign it to obtain* $\sigma_z = \mathsf{SSig}(tsk, M_z)$*. Now, using* $\ell_{\texttt{Post}}[x]$*, (re)compute signature* $\sigma_x = \mathsf{SSig}(tsk, M_x)$ *on* $M_x = (x, \ell_{\texttt{Pre}}[x], \ell_{\texttt{Post}}[x])$*. Return* $\tau_{(x,z)} = (M_x, \sigma_x, M_z, \sigma_z)$*.*

- $\mathsf{TSComp}((a, b), \tau_{(a,b)}, (b, c), \tau_{(b,c)}, tpk)$*: 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* $(M_a, \sigma_a, M_b, \sigma_b)$*. If* M_a *or* M_b *are not of the form* $(a, A_{\texttt{Pre}}, A_{\texttt{Post}})$ *or* $(b, A_{\texttt{Pre}}, A_{\texttt{Post}})$ *where* $A_{\texttt{Pre}}, A_{\texttt{Post}}, A_{\texttt{Pre}}, A_{\texttt{Post}}$ *are labels, or if any signature is invalid, then return* $\perp$*. Otherwise, verify that* a *appears before (resp. after)* b *in* $\texttt{Pre}$ *(resp.* $\texttt{Post}$*) by checking that* $A_{\texttt{Pre}} < B_{\texttt{Pre}}$ *and* $B_{\texttt{Post}} < A_{\texttt{Post}}$*. If verification succeeds return* valid *else return* $\perp$*.*

Correctness and Security. We require that correct signatures, those honestly computed by the *signer* as well as those combined by anyone from two existent signatures, verify correctly, meaning that the verification algorithm on them returns valid. To see that this holds, it suffices to observe first, that the signing operation preserves the tree structure of the graph, and second, that $\mathsf{ODCompare}(X, Y)$ is true, if, and only if, $x = \ell^{-1}_{\mathtt{Pre}}[X]$ (resp. $y = \ell^{-1}_{\mathtt{Post}}[Y]$) appears before $y = \ell^{-1}_{\mathtt{Pre}}[Y]$ (resp. $x = \ell^{-1}_{\mathtt{Post}}[X]$) in $\mathtt{Pre}$ (resp. $\mathtt{Post}$) which allows correct verification by Dietz's condition (namely, Proposition 10). These considerations yield the following theorem.

Theorem 10 *If* $\mathtt{SSig}$ *is a signature scheme existentially unforgeable under chosen message attack then* $\mathtt{BasicTSDT}$ *a is secure transitive signature scheme for directed trees where edge signatures are* $O(N + \kappa)$ *bits long.*

6.4.2 Full Construction

We extend our basic construction as follows. Instead of comparing strings directly, we compare them by proving that labels from certain nodes contain the labels of other nodes as prefixes. Such proofs are done using scheme $\mathtt{PH}$. As mentioned before, we use the alphabet $\Sigma = \{0, \$, 1\}$ where $0 < \$ < 1$. That is, in order to prove that two labels X, Y are such that $X \prec Y$ using their hashes $\mathsf{H}(X), \mathsf{H}(Y)$ instead of the strings, the *combiner* must compute: (1) the longest common prefix C for X and Y, (2) a proof that C is a prefix of X up to position $i = |C|$, (3) a proof that C is a prefix of Y up to position $i = |C|$, (4) a proof that $C||c_1$ is a prefix of $X||\$$ up to position $i + 1$ for some $c_1 \in \Sigma$, and (5) a proof that $C||c_2$ is a prefix of $Y||\$$ up to position $i + 1$, for some $c_2 \in \Sigma$. Then, verifying that $X \prec Y$ reduces to the checking of the proofs and verifying that $c_1 < c_2$.

Our full construction requires the *combiner* to be *stateful*, in particular to hold at any moment the tree $\mathcal{T}$, and the two tries $\mathcal{B}_{\mathtt{Pre}}$ and $\mathcal{B}_{\mathtt{Post}}$ for both order data structures. In particular our scheme does not allow for edge signature compositions as the data present in a pair of edge signatures is not enough to compute the combined signature. This is an important limitation of our scheme despite the fact that the main functionality (i.e. being able to compute a signature for any path in the tree $\mathcal{T}$ without the secret key of the *signer*) is maintained. Thus, the main change in our construction compared to the inital syntax for a transitive signature scheme (see

Definition 23), is the replacement of the algorithm TSComp used to combine two signatures by the algorithm $\mathsf{TSCompStat}$ which takes as input all the data produced by the *signer*: the order data structures, the hash values and the signatures that are computed after each update of the directed tree $\mathcal{T}$.

In the following scheme, we denote by $\mathfrak{m}$ the state that is shared by the *combiner* and the *signer* and contains:

- the order data structures (with the tries) for the pre-order and post-order lists,
- the standard signatures produced by the *signer* and their relative messages,
- the hash values of the labels computed through the order data structures, and
- a table $\mathfrak{t}$ to map the nodes of the tree $\mathcal{T}$ to their labels, the hash values of these labels and the corresponding (standard) signatures.

Construction 14 *(*`TSDT` *using CRHF that preserves* `CommonPrefix`*) Let* $\mathtt{PH} = (\mathsf{HGen}, \mathsf{HEval}, \mathsf{ProofGen}, \mathsf{ProofCheck})$ *be a family of hash functions that preserves* `CommonPrefix`*. The scheme* `PHTSDT` *consists of the following algorithms:*

- $\mathsf{TSKG}(1^\kappa)$: *Do as in* `BasicTSDT` *and also generate the public parameters (ie.* `PP`*) for the* `PH` *scheme. Return* $(tsk, tpk, \mathtt{PP})$*.*
- $\mathsf{TSign}(tsk, a, b, \mathfrak{m})$: *Do as in* `BasicTSDT` *except that the message* M_z *to sign is now* $M_z = z||H_{z_{\mathtt{Pre}}}||H_{z_{\mathtt{Post}}}$*, where* $H_{z_{\mathtt{Pre}}} = \mathsf{H}(\ell_{\mathtt{Pre}}[z])$ *and* $H_{z_{\mathtt{Post}}} = \mathsf{H}(\ell_{\mathtt{Post}}[z])$*. Store in a table* $\mathfrak{t}$ *(which is part of* $\mathfrak{m}$*) the association between the node* z *the values* $H_{z_{\mathtt{Pre}}}$ *and* $H_{z_{\mathtt{Post}}}$ *and the corresponding digital signature. Notice that* $H_{z_{\mathtt{Pre}}}, H_{z_{\mathtt{Post}}}$ *can be computed incrementally due to* H*'s homomorphism.*
- $\mathsf{TSCompStat}(a, b, \mathfrak{m}, tpk)$*: If there is no path between* a *and* b *then reject. Retrieve from* $\mathfrak{m}$ *the values* $(M_a, \sigma_a, M_b, \sigma_b)$*. If* σ_a *or* σ_b *is invalid, then reject. Parse* $M_a = (a, H_{A_{\mathtt{Pre}}}, H_{A_{\mathtt{Post}}})$ *and* $M_b = (b, H_{B_{\mathtt{Pre}}}, H_{B_{\mathtt{Post}}})$*. Compute proof* $\pi_{\mathtt{Pre}}$ *as follows. Let* $D_{\mathtt{Pre}}$ *be the longest common prefix between* $A_{\mathtt{Pre}}$ *and* $B_{\mathtt{Pre}}$*. Note that* $\mathsf{H}(D_{\mathtt{Pre}})$ *has already been computed by the* signer *and thus* $M_d = d$

$||\mathsf{H}(D_{\texttt{Pre}})||\mathsf{H}(D_{\texttt{Post}})$, *and the signature* σ_d *on* M_d *can be fetched from* $\mathfrak{m}$, *as well as* $A_{\texttt{Pre}}, B_{\texttt{Pre}}, D_{\texttt{Pre}}, A_{\texttt{Post}}, B_{\texttt{Post}}$ *and* $D_{\texttt{Post}}$. *Let* $t = |D_{\texttt{Pre}}|$. *Compute the following values:*

- $\pi_1 = \mathsf{ProofGen}(D_{\texttt{Pre}}, A_{\texttt{Pre}}, t, \mathsf{PP})$,
- $\pi_2 = \mathsf{ProofGen}(D_{\texttt{Pre}}, B_{\texttt{Pre}}, t, \mathsf{PP})$
- $\pi_3 = \mathsf{ProofGen}(D_{\texttt{Pre}}||d_1, A_{\texttt{Pre}}||\$, t+1, \mathsf{PP})$, *and*
- $\pi_4 = \mathsf{ProofGen}(D_{\texttt{Pre}}||d_2, B_{\texttt{Pre}}||\$, t+1, \mathsf{PP})$

where $(d_1, d_2) \in \{(0, \$), (0, 1), (\$, 1)\}$. *Set* $\Theta = (M_a, \sigma_a, M_b, \sigma_b, M_d, \sigma_d, t, d_1, d_2)$ *and* $\pi_{\texttt{Pre}} = (\pi_1, \pi_2, \pi_3, \pi_4)$. *Compute similarly* $\pi_{\texttt{Post}}$ *and return* $\tau_{(a,d)} = (\Theta, \pi_{\texttt{Pre}}, \pi_{\texttt{Post}})$.

- $\mathsf{TSVf}((a, b), \tau, tpk, \mathsf{PP})$: *Parse* τ *as* $\tau = (\Theta, \pi_{\texttt{Pre}}, \pi_{\texttt{Post}})$. *Parse* $\pi_{\texttt{Pre}}$ *as* $(\pi_1, \pi_2, \pi_3, \pi_4)$. *Parse* Θ *as* $(M_a, \sigma_a, M_b, \sigma_b, M_d, \sigma_d, t, d_1, d_2)$ *where* $M_x = x||H_{X_{\texttt{Pre}}}||H_{X_{\texttt{Post}}}$ *for* $x \in \{a, b, d\}$. *Check that all (standard) signatures are valid under public key* tpk. *Check that* $d_1, d_2 \in \Sigma$ *and* $d_1 < d_2$. *Verify proofs* $\pi_1, \pi_2, \pi_3, \pi_4$ *using* ProofCheck*:*

  - $\mathsf{ProofCheck}(H_{D_{\texttt{Pre}}}, H_{A_{\texttt{Pre}}}, \pi_1, t, \mathsf{PP})$,
  - $\mathsf{ProofCheck}(H_{D_{\texttt{Pre}}}, H_{B_{\texttt{Pre}}}, \pi_2, t, \mathsf{PP})$,
  - $\mathsf{ProofCheck}(H_{D_{\texttt{Pre}}} \cdot \mathsf{H}(0^t||d_1), H_{A_{\texttt{Pre}}} \cdot \mathsf{H}(0^t||\$), \pi_3, t+1, \mathsf{PP})$, *and*
  - $\mathsf{ProofCheck}(H_{D_{\texttt{Pre}}} \cdot \mathsf{H}(0^t||d_2), H_{B_{\texttt{Pre}}} \cdot \mathsf{H}(0^t||\$), \pi_4, t+1, \mathsf{PP})$

  Perform the similar verifications relative to $\texttt{OrderDS}_{\texttt{Post}}$. *If all these verifications pass return* valid *otherwise return* $\perp$.

This new construction combines the basic one with hashing with common-prefix proofs so we can shrink the size of an edge signature to $O(\kappa)$ bits.

Theorem 11 *If* `SSig` *is a signature scheme existentially unforgeable under chosen message attack and* $\mathcal{H}$ *securely preserves the predicate* `CommonPrefix`, *then* `PHTSDT` *is a secure* `TSDT` *scheme where the size of an edge signature is* $O(\kappa)$ *bits.*

Proof. Let $\mathcal{A}$ be a PPT adversary that breaks our scheme. We build an adversary $\mathcal{B}$ that breaks the security of $\texttt{SSig}$ or $\texttt{PH}$. $\mathcal{B}$ has access to a signing oracle $\mathcal{O}_{\texttt{SSig}}(\cdot)$ and is given the description of the hash function H as defined in construction 11. $\mathcal{B}$ forwards all the public parameters to $\mathcal{A}$. $\mathcal{A}$ asks for edges signing to $\mathcal{B}$ who replies using the signing oracle $\mathcal{O}_{\texttt{SSig}}(\cdot)$ and H. Finally, $\mathcal{A}$ outputs a compressed edge 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 inserted in the tree $\mathcal{T}$. This means that $\mathcal{B}$ has been able to compute some signature for a message M' not previously requested to the oracle $\mathcal{O}_{\texttt{SSig}}$. So now we assume that all signed messages reflect the history of edge insertions in the tree.

Let $A_{\texttt{Pre}}, B_{\texttt{Pre}}$ and $A_{\texttt{Post}}, B_{\texttt{Post}}$ be the order labels associated to vertices a, b in $\texttt{OrderDS}_{\texttt{Pre}}$ and $\texttt{OrderDS}_{\texttt{Post}}$ respectively. As there is no path from a to b, this means that (i) a appears before b in $\texttt{Pre}$ and also a appears before b in $\texttt{Post}$ or (ii) b appears before a in $\texttt{Pre}$ and also b appears before a in $\texttt{Post}$ or (iii) there is a path, but from b to a. Assume we are in case (i). If indeed a appears before b in $\texttt{Pre}$, then the adversary $\mathcal{A}$ managed to prove that b appears before a in $\texttt{Post}$, 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 $C||c_1$ is a prefix of $B_{\texttt{Post}}$ and $C||c_2$ is a prefix of $A_{\texttt{Post}}$ and $c_1 < c_2$. It is worth noting that, although the proofs $\pi_1, \pi_2, \pi_3, \pi_4$ do not explicitly mention the strings tied to the nodes (only hash values and lengths), these strings are present in the data structures $\texttt{OrderDS}_{\texttt{Pre}}$ and $\texttt{OrderDS}_{\texttt{Post}}$ which are maintained by the *signer* and hence are accessible to the simulator $\mathcal{B}$. 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 $A_{\texttt{Post}} \prec B_{\texttt{Post}}$, there exists no such string C, so this means that either $C||c_1$ is not a prefix of $A_{\texttt{Post}}$ or $C||c_2$ is not a prefix of $B_{\texttt{Post}}$, therefore $\mathcal{B}$ has been able to break the security of the predicate preserving hashing scheme $\texttt{PH}$. The case (ii) and (iii) can be treated similarly. ■

Using the tradeoff technique we can generalize the previous result as follows.

Theorem 12 *Let* $\lambda \geq 1$. *If* $\texttt{SSig}$ *is a signature scheme existentially unforgeable under chosen message attack and* $\mathcal{H}$ *securely preserves the* $\texttt{CommonPrefix}$ *predicate, then* $\texttt{PHTSDT}$ *with tradeoff is a secure* $\texttt{TSDT}$ *scheme. Moreover, (a) an edge signature is*

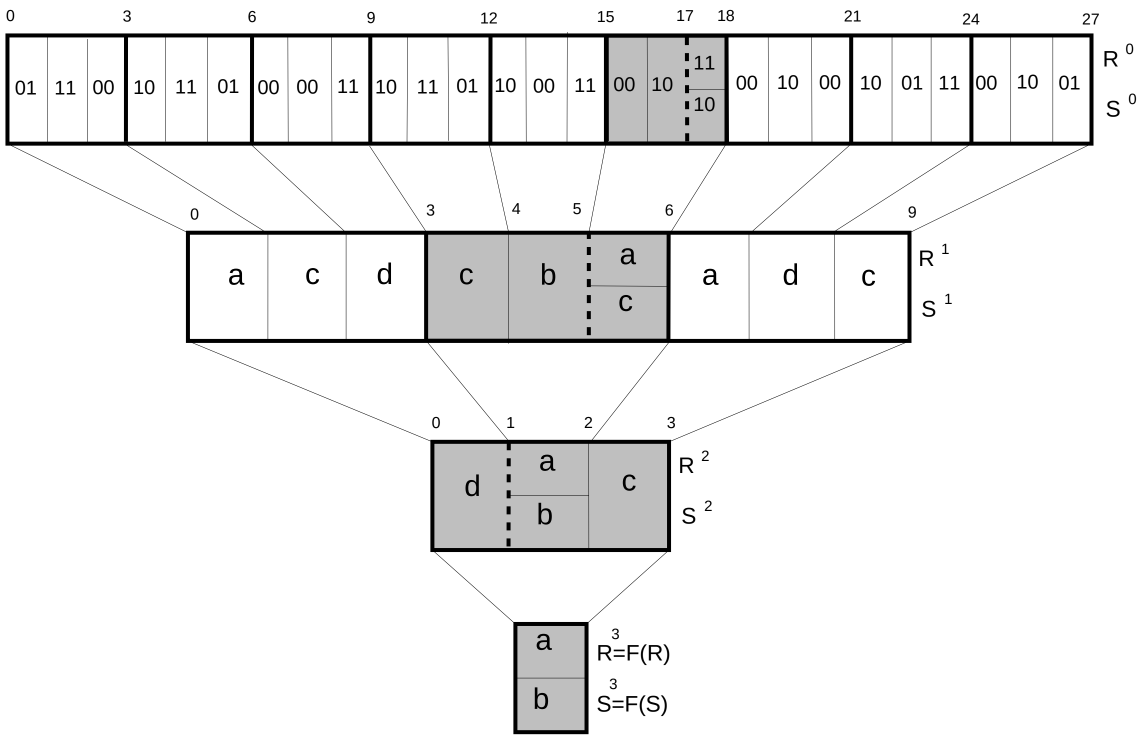


Figure 6.3: Example of hash value update for the tradeoff scheme.

In this example we show how to compute the hash value for the string $S^0 = S$ given the hash value (and all intermediate values) of the string $R^0 = R$ that share all bits except one (in position 36). We start at level 0. The first 5 chunks starting from the left are already in the table as they correspond to the chunks of string R for which the hash values have already been computed. The same applies for the chunks in position $7, 8$ and 9 respectively. The chunk in position 6 differs between string R and S by one bit (one symbol). Thus the *signer* fetches the hash value of this chunk for string R and updates it using the incremental property of the hash function G to obtain the hash value for the chunk in position 6 corresponding to the string S. This chunk now affects the string at level 1 within the chunk number 2. Again the *signer* using the hash value of this chunk relative to string R^1, can compute the hash value of chunk number 2 for the string S^1. This hash value will feed the level 2 and change the (single) chunk present at this level. The *signer* by computing the new hash value for the updated chunk will obtain the final hash value $\mathsf{F}(S)$.

$O(\lambda\kappa)$ bits long and can be verified in $O(\lambda)$ cryptographic operations, (b) the signer *has to perform $O(\lambda)$ cryptographic operations to sign an edge, (c) computing the signature for any edge (in the transitive closure of the tree) takes $O(\lambda(\frac{N}{\kappa})^{1/\lambda})$ cryptographic operations for the* combiner*, and (d) the initialization requires $O(N)$ cryptographic operations for the* signer*.*

Proof. The new construction consists conceptually of replacing the hash function H in the Construction 14 by the one with tradeoff described in Section 6.3. As before, we will denote this hash function by F.

The security of the new construction still holds as F preserves the `CommonPrefix` predicate (see Theorem 9). Moreover the size of an edge signature in the transitive closure of the tree is $O(\lambda\kappa)$ as it consists of a constant number of proofs and (standard) digital signatures. Some technical details need to be addressed though. The problem we need to solve is how to compute incrementally the hash values through all the levels of the tradeoff hash function. Recomputing the hash value from scratch every time a label is produced would be inefficient for the *signer*.

We describe here the full construction (see example Figure 6.3). Let `OrderDS` be an order data structure (both data structures for `Pre` and `Post` lists are treated in the same way). At the beginning the *signer* will compute the hash value for the first string corresponding to $-\infty$ which is equal to $S = \$0^N$. To do so, he will compute the intermediate hash values for each layer of the hash scheme with tradeoff as depicted in Figure 6.1. When computing each string at each layer the *signer* will store in a table the association between each chunk (that is the string of size $(\frac{N}{\kappa})^{1/\lambda}$ bits), the layer, the position within the layer, its hash value and also the relationship between the input string S and this chunk. This costs the *signer* $O(N)$ cryptographic operations but note it is required only once for the setup phase. Henceforth the *signer* will only need $O(\lambda)$ operations as explained below. In order to compute the hash of a new string (for example the string $1\$0^{N-1}$ that stands for $+\infty$), the *signer* will proceed from the first to the last layer.

1. Identify using the trie of the data structure the string R that share all bits except one with the string S for which we want to compute the hash value.

2. Start at the first layer. For each chunk of input string S, if it is present in the table, skip it.

3. When finding the first (and single) chunk which has not been stored previously in the table, compute the hash of this string and store it in the table. This hash value will replace at the next layer the corresponding chunk belonging to string R.

4. Jump to the next layer and repeat the same procedure.

5. When reaching the last level λ output to the final hash value $\mathsf{F}(S)$.

We claim that the number of cryptographic operations involved for computing $\mathsf{F}(S)$ for a new string S is $O(\lambda)$. As mentioned before, the string S shares all the bits except one with some label R which hash value has already been computed. Thus, this means that in our layered hash function scheme, only one chunk per level will be affected by this change and thus the *signer* will only have to recompute the hash value for this chunk. Moreover, due to the incremental incremental property of the hash function G (see Section 6.2, additional properties), this computation can be done using a constant number of cryptographic operations.

The other complexities follow directly from the complexities of Theorem 9. ■

6.5 Conclusion

In this chapter we show how to use a CRHF that preserves the `CommonPrefix` predicate in order to build an efficient transitive signature scheme for directed trees. Moreover we show that this CRHF also enables a practical tradeoff between the time to combine a signature, $O(\lambda(\frac{N}{\kappa})^{1/\lambda})$, and the time to verify it, $O(\lambda)$. We recall, however, that the problem of building *short* and *stateless* transitive signatures, even for directed trees, remains open. In particular, enabling signature composition seems to imply the need for global information on the tree that is signed. Finally, we believe that the introduced CRHF may lead to useful applications, especially in the area of authenticated data structures.

Fair Exchange of Short Signatures without Trusted Third Party

Contents

7.1 Introduction

Let p be a prime number of κ bits, $\theta \in \mathbb{Z}_p$ and $\vec{b}$ a bit vector of κ components. In this chapter we design a pair of CRHFs that preserve the predicate $\texttt{Equiv}(\theta, \vec{b}) = 1 \Leftrightarrow \theta = \sum_{i=1}^{\kappa} \vec{b}[i]2^{i-1}$. These hash functions combined with some zero-knowledge techniques are used to build a protocol for gradually releasing a secret. A protocol is said to gradually release a secret if this secret (but verifiable) value can be opened bit by bit. The correctness of each bit released is *checkable* and moreover the process *does not leak* any additional information. We consider, as introduced in section 2.3, a bilinear map

$e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ and a public parameter $\texttt{PP} = \langle (p, \mathbb{G}, \mathbb{G}_T, e, g), (g_0, g_1, g_2, \cdots, g_\kappa) \rangle$. More concretely, we use the following pair of CRHF that preserve the predicate `Equiv`:

- $\mathsf{H}_1 : \{0,1\}^\kappa \rightarrow \mathbb{G}$, where $\mathsf{H}_1(\theta) = \prod_{i \in [\kappa]} g_i^{\theta[i]}$.
- $\mathsf{H}_2 : \mathbb{Z}_p \rightarrow \mathbb{G}$ where $\mathsf{H}_2(\theta) = g^\theta$.

We apply this idea for gradually releasing a secret to the problem of fair exchange of digital signatures which we motivate next.

MOTIVATION. Nowadays, it is more and more common to trade digital goods on the web: E-books, software licenses, avatar-games currencies like *Ultima Online*[1], to cite a few. Whether these goods are exchanged on *E-bay* through *Paypal* or bought directly to a provider like *Amazon* or *Microsoft*, securing this transaction requires a trusted third party (TTP). Though it works quite well in practice, enabling a totally distributed and at the same time secure virtual market place is of clear interest: It would avoid the presence of single points of failure, and also allow smoother electronic commercial transactions that would not need to rely on some intermediary. A lot of these transactions may be captured by the exchange of digital signatures. Suppose for example you want to buy a software license from some independent developer: Indeed exchanging the software license as well as the money transfer (digital check) can be modeled by signed messages. However we face a non-trivial problem. Given that the transaction is made on-line, a malicious participant may fool his counterpart by not sending his signature or simply sending garbage. A protocol that prevents such a behavior from a corrupted party is called *fair*: At the end of the execution of protocol either both parties obtain the signature they expected or neither do.

GRADUAL RELEASE OF A SECRET AND FAIR EXCHANGE. There are two main approaches to solving this problem. On the one hand, one can assume that both players interact through a TTP. Though this solution does not fit our goal, it is important to note that an important line of research has focused on designing protocols where the TTP is only required when "something goes wrong". These protocols are said to be *optimistically fair*: (see [ASW97, Mic03] and [HWS12] for some recent related work).

[1]http://en.wikipedia.org/wiki/Ultima_Online

On the other hand, if no TTP exists and we assume that both participants have exactly the same computational resources, then it is impossible, *in general*, to achieve *complete fairness* [Cle86]. In [Blu83, EGL85] was proposed a way to relax the notion of fairness in order to overcome Cleve's impossibility result [Cle86]. The idea is to assume that both players have roughly the same amount of time, so we can achieve *partial fairness*. Several secure multi-party computations and specific protocols, like [BCvD87, Cle90, Dam95, BN00, GMPY06], were built on top of this security notion. The recurrent idea behind these constructions consists in enabling each player to release their secret, bit by bit, in alternation. Thus, if a player aborts, the other participant will have "only one bit of disadvantage". Formalizing this idea is not an easy task though, in particular because it is hard to reason on the specific amount of time for the players. This issue was noticed in [GK10] where authors point out that (1) assigning more time to the honest party in order to allow him to recover his value is somehow artificial as it does not depend on the participant himself, and (2) implementing such definitions seems to imply the use of strong assumptions related to the *exact* time required to solve some computational problem.

In this work we propose a new security definition that still captures the intuition of partial fairness for the exchange of digital signatures, but without forcing the participants to have access to almost equal computational resources as proposed in [GMPY06]. The idea of our definition is to compare the probabilities of computing valid signatures on the agreed messages at the end of the protocol. More precisely, if the adversary aborts the protocol, the honest participant[2] will compute the expected signature by choosing randomly a value from the space of signature candidates, which is defined by the remaining bits to be obtained. The adversary will keep running its own algorithm and also output a signature candidate. We say the protocol is secure if the probabilities that each participant output a valid signature only differ by a polynomial factor. Note that this definition, like previous ones that circumvent Cleve's impossibility result [Cle86], allows the adversary to get some advantage, but it guarantees that this advantage is polynomially bounded. With that definition in

[2]Note that we need to consider that at least one participant is honest, as otherwise we cannot really avoid that one of the two adversaries, which are arbitrary polynomial time algorithms, wins.

hand we can prove the security of our protocol without having to rely on the strong assumptions mentioned above. Our protocol is designed to exchange short signatures [BB08] without the presence of a TTP. We use bilinear maps as the underlying signature scheme, and also the idea of gradually releasing each bit of some secret θ that will enable the signature to recovered. The security of our construction relies on two complexity assumptions for bilinear maps, namely the κ-Strong Diffie-Hellman [BB08] and the κ-Bilinear Diffie-Hellman assumptions [BB04a], and the security proof holds in the common reference string model. As we use non-interactive zero-knowledge proofs of knowledge (ZKPoK) to make the protocol simpler and more efficient, we require the use of random oracle [FS86] or some non-black-box assumptions [Gro10]. Of course, we can use interactive ZKPoK at a minor expense in round efficiency.

Our Contributions.

1. We propose a practical protocol for exchanging short signatures [BB08] without relying on a TTP. To the best of our knowledge this is the first construction that meets such a goal. The number of rounds of our protocol is $\kappa + 5$, where κ is the security parameter. The communication complexity is $16\kappa^2 + 12\kappa$ bits. The protocol requires a linear number of group exponentiations, group multiplications, bilinear map applications, hash computations and also a constant number of group divisions (see Fig. 7.5 for more details).

2. We introduce a new non-interactive ZKPoK to prove that the prover is able to open a commitment to a bit vector. This protocol may be of independent interest.

3. We introduce another non-interactive ZKPoK to prove that the prover is able to open a commitment to a bit vector that corresponds to the binary decomposition of some value θ which is hidden as the discrete logarithm of some group element. We think this technique may lead to other interesting applications.

4. As stated earlier, we propose a new security definition for partial fairness in the context of the exchange of digital signatures. This definition is simple and avoids the issue of involving the *exact* running time of the participants.

Our Approach. Let $\kappa \in \mathbb{N}$ be the security parameter. Let $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BMGen}(1^\kappa)$ be the public parameter where $p = |\mathbb{G}| = |\mathbb{G}_T|$ is prime, $\mathbb{G}, \mathbb{G}_T$ are cyclic groups, $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is the bilinear map and g is a random generator. Let s be a random element in $\mathbb{Z}_p$, we consider the following common reference string: $(g, g^s, g^{s^2}, ..., g^{s^\kappa}) = (g_0, g_1, g_2, ..., g_\kappa)$. In practice this common reference string can be computed using generic multi-party computation techniques (see [CHK$^+$12] for an efficient implementation) so that the secret s is randomly generated and remains unknown to all the participants. Another alternative is to rely on a TTP that would "securely delete" the secret after the generation of the common reference string. Obviously the intention of this work is to avoid the use of a TTP, but note however that even in this case, the TTP would be required only once.

Our construction can be summarized as follows: The prover chooses a secret $\theta \xleftarrow{R} \mathbb{Z}_p$, then commits each bit of this secret into a Pedersen [Ped91] commitment, where the bit b_i in position i with randomness $r_i \in \mathbb{Z}_p$ will be committed with respect to the base (g, g_i): That is $\mathsf{Commit}((b_i, i), r_i) = g^{r_i} g_i^{b_i}$. Then, we use a ZKPoK protocol in order to verify that the prover is able to open these commitments to bits. The next step is to publish $D = g^\theta$ and show, using another ZKPoK protocol, that θ, the discrete logarithm of D, is "equivalent" to the bit vector committed in $\vec{C} = (\mathsf{Commit}((b_i, i), r_i))_{i \in [\kappa]}$. More precisely, the prover shows that $\theta = \sum_{i \in [\kappa]} b_i 2^{i-1}$. To release a signature σ, the prover will blind it using θ to obtain $\tilde{\sigma} = \sigma^\theta$. Using bilinear maps it is straightforward to verify that $\tilde{\sigma}$ contains a valid signature σ which is blinded in the exponent by θ, the discrete logarithm of D. The other verifications will simply check the proofs. By releasing each bit in turn, both players will reconstruct their own blinding factor θ and obtain the signature.

Related Work. Among the abundant literature on the topic of gradual release and fair exchange for digital signatures, the work in [Dam95] is probably what is the most similar to ours: It describes a practical fair exchange protocol for digital signatures based on gradual release of a secret. The protocol described in [Dam95] works for Rabin, RSA and El Gamal signatures. The number of rounds of the protocol described in [Dam95] is roughly 2κ for RSA and Rabin signatures and κ for El Gamal signatures.

Due to Cleve's impossibility result [Cle86], the question of building complete fair protocols with dishonest majority seemed to be closed. However, Gordon et al. showed that non-trivial functions can be computed fairly in the two-party model [GHKL11], and left the question of finding a tight characterization of these functions open. In particular it is not known whether functions with a non-polynomial size domain and that return multiple bits as output (like computing a signature) can be computed fairly in Cleve's setting.

In [GK10] a definition is proposed for partial fairness that may exhibit some similarities with ours (both definitions involve a $Q(\kappa)$ factor where Q is a polynomial). However, our definition and approach differs quite a bit from [GK10]. First, the setting in [GK10] is more general than our specific construction to exchange digital signatures. Secondly, in their protocol, the number of rounds is variable and defines the level of fairness, whereas in our construction fairness only depends on the computational power of the participants.

Our ZKPoK protocol to prove that a commitment can be opened to a bit vector is inspired by [GOS06, Gro10]. We remark that, though [GMPY06] uses the idea of gradual release, the construction proposed is not practical in our setting as it requires coding the functionality (signing in our case) as an arithmetic circuit.

7.2 A new argument to prove that a commitment encodes a bit

In this section we describe a commitment scheme to encrypt a vector of values in $\mathbb{Z}_p$ and then provide a ZKPoK that the prover knows how to open each commitment to a bit. Our construction, as [GOS06], uses the idea that if the value b encrypted is a bit then $b(b-1)$ must be equal to 0, and also borrows from [Gro10] by instantiating a basic form of the restriction argument.

Our commitment scheme requires the generation of a common reference string $\mathsf{CRS} = (g, g^s, g^{s^2}, ..., g^{s^N}) = (g_0, g_1, ..., g_N)$ where $s \overset{R}{\leftarrow} \mathbb{Z}_p$ is the trapdoor. To commit a bit b_i in position i using randomness $r_i \in \mathbb{Z}_p$, we compute the Pedersen commitment

Common reference string: Input $(1^\kappa, N)$

1. $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BMGen}(1^\kappa)$
2. $s \xleftarrow{R} \mathbb{Z}_p$
3. Return $\mathsf{CRS} = \langle (p, \mathbb{G}, \mathbb{G}_T, g), (g_0, g_1, g_2, ..., g_N) \rangle$ where for all $i \in [0..N] : g_i = g^{s^i}$.

Statement: The statement is formed by a vector of elements of $\mathbb{G}$: $(C_1, C_2, ..., C_N)$. The claim is that for each $i \in [N]$, the prover knows (r_i, b_i) such that $C_i = g^{r_i} g_i^{b_i}$ where $b_i \in \{0, 1\}$.

Proof: Input $(\mathsf{CRS}, \vec{b}, \vec{r})$

1. Check that $\vec{b} = (b_1, ..., b_N) \in \{0,1\}^N$. Return $\perp$ if this is not the case.
2. Check that $\vec{r} = (r_1, ..., r_N) \in \mathbb{Z}_p^N$. Return $\perp$ if this is not the case.
3. For each $i \in [N]$ compute an argument π_i that C_i is the commitment to a bit in base g_i: $\pi_i = (A_i, B_i)$ where $A_i = g_{N-i}^{r_i} g_N^{b_i}$ and B_i is such that $e(A_i, C_i g_i^{-1}) = e(B_i, g)$. More concretely compute $B_i = g_{N-i}^{r_i^2} g_N^{r_i(2b_i-1)}$.
4. Compute the ZKPoK $\vec{PK} = (PK\{(r_i, b_i) : C_i = g^{r_i} g_i^{b_i}\})_{i \in [\kappa]}$ as described in section 2.6.2.
5. Return $\pi = (\pi_{bit}, \vec{PK})$ where $\pi_{bit} = (\pi_i)_{i \in [N]}$.

Verification: Input $(\mathsf{CRS}, \vec{C}, \pi)$

1. Parse $\vec{C}$ as $(C_i)_{i \in [N]}$. Check that $C \in \mathbb{G}^N$.
2. Parse π as $(\pi_{bit}, \vec{PK})$. Check that $\pi_{bit} \in (\mathbb{G} \times \mathbb{G})^N$.
3. Verify the proof of knowledge for each commitment using the vector $\vec{PK}$.
4. For each $i \in [N]$ check that:
   (a) $e(C_i, g_{N-i}) = e(A_i, g)$.
   (b) $e(A_i, C_i g_i^{-1}) = e(B_i, g)$.
5. Return valid if and only if all check pass, otherwise return $\perp$.

Figure 7.1: ZKPoK that the prover can open a commitment to a bit vector.

$\mathsf{Commit}((b_i, i), r_i) = C_i = g^{r_i} g_i^{b_i}$. The commitment to the vector $\vec{b} = (b_1, b_2, ..., b_N)$ using the randomness $\vec{r} = (r_i)_{i \in [N]}$ will simply be the vector formed by the commitments for each bit in position i: $\vec{C} = (C_i)_{i \in [N]}$. Abusing a little our notation we will write $\vec{C} = \mathsf{Commit}(\vec{b}, \vec{r})$.

We still need a ZKPoK that each commitment C_i can be opened to a bit by the prover. The prover proceeds as follows (see Figure 7.1): First it computes a (non-interactive) ZKPoK for the representation of each commitment C_i in base (g, g_i). Then, it computes the "translation" of the commitment $C_i = g^{r_i} g_i^{b_i}$ by $N - i$ positions to the right, by providing the value $A_i = g_{N-i}^{r_i} g_N^{b_i}$. If we compute $e(A_i, C_i g_i^{-1})$ and express this quantity in terms of $e(B_i, g)$ we realize by simple inspection (see completeness paragraph in the proof of Theorem 13) that $B_i = g_{N-i}^{r_i^2} g_N^{r_i(2b_i-1)} g_{N+i}^{b_i(b_i-1)}$.

As the prover does not know g_{N+i}, if $b_i \notin \{0,1\}$, it will not be able to provide the second part of the proof B_i. In case b_i is indeed a bit, then the prover will compute the proof $\pi_i = (A_i, B_i)$ in order to convince the verifier that C_i is the encryption of a bit relative to position i.

Proposition 11 *The vector commitment scheme described above is perfectly hiding and computationally binding under the N-BDHI assumption.*

Proof. Here we have that $\tau = s$. If we define $\mathsf{TCommit}(\tau) = (C = g^r g_i^{m'}, ek = (m', r))$ for $m', r \in \mathbb{Z}_p$, we have that $\mathsf{TOpen}(C, ek, m)$ will return $r' = r + s^i(m' - m)$. The scheme is perfectly trapdoor.

Assume an adversary $\mathcal{A}$ computes $\vec{b}, \vec{b'} \in \mathbb{Z}_p^N$ two vectors of messages and $\vec{r} = (r_1, ..., r_N), \vec{r'} = (r'_1, ..., r'_N) \in \mathbb{Z}_p^N$ two randomness vectors such that $\mathsf{Commit}(\vec{b}, \vec{r}) = \mathsf{Commit}(\vec{b'}, \vec{r'})$ and $B[j] \neq B'[j]$ for (at least) one $j \in [N]$: We obtain the equation $g^{r_j - r'_j} g^{(B[j]-B'[j])s^j} = 1_{\mathbb{G}}$. If we set $X = s^j$, we can deduce that $(r_j - r'_j) + (B[j] - B'[j])X = 0$ and then $X = s^j = \frac{r'_j - r_j}{B[j]-B'[j]} \bmod p$. Once s^j is recovered we can compute $g_N^X = g_{N+j}$ and by proposition 1, the N-BDHI assumption is broken. ■

Theorem 13 *The protocol in Figure 7.1 is a ZKPoK that for each $i \in [N]$, the prover knows (r_i, b_i) such that $C_i = g^{r_i} g_i^{b_i}$ and $b_i \in \{0,1\}$. The protocol is perfectly complete and perfect zero-knowledge. Moreover, if the N-BDHI assumption holds, for any $i \in [N]$, the prover has negligible probability to output (r_i, b_i) and a valid proof π_i such that the verification passes and $b_i \notin \{0,1\}$.*

Proof. **Completeness.** Let $i \in [N]$. It's clear that the prover can compute $A_i = g_{N-i}^{r_i} g_N^{b_i}$. Then if $b_i := \vec{b}[i] \in \{0,1\}$ we have $b_i(b_i - 1) = 0$ and

$$\begin{aligned}
e(A_i, C_i g_i^{-1}) &= e(g^{(r_i + b_i s^i)s^{N-i}}, g^{r_i} g^{(b_i-1)s^i}) \\
&= e(g^{(r_i s^{N-i} + b_i s^N)(r_i + (b_i-1)s^i)}, g) \\
&= e(g^{r_i^2 s^{N-i} + r_i b_i s^N + r_i(b_i-1)s^N + b_i(b_i-1)s^{N+i}}, g) \\
&= e(g^{r_i^2 s^{N-i} + s^N r_i(2b_i-1)}, g) \\
&= e(B_i, g)
\end{aligned}$$

We can see that the prover can compute B_i as he knows r_i and b_i and the group elements g_{N-i}, g_N are public.

Hardness to open the commitment to a non-bit vector. Assume there exists some adversary $\mathcal{A}$ that is able to open the commitment to some vector that does not contain bits, for at least one i. We build the following adversary $\mathcal{B}$ that breaks the N-BDHI assumption. $\mathcal{B}$ receives the challenge tuple $(g_0, g_1, g_2, ..., g_N)$. Then, $\mathcal{B}$ runs $\mathcal{A}$ using the tuple as the CRS and obtains C_i, r_i, b_i such that $C_i = g^{r_i} g_i^{b_i}$ (using the proof of knowledge $\vec{PK}$) and $\pi_i = (A_i, B_i)$ where

$$\begin{aligned} e(C_i, g_{N-i}) &= e(A_i, g) \quad (1) \\ e(A_i, C_i g_i^{-1}) &= e(B_i, g) \quad (2) \end{aligned}$$

From (1) we can deduce that $A_i = C_i^{s^{N-i}}$. From (2) (as seen in the correctness proof) we have that $B_i = g^{r_i^2 s^{N-i} + r_i(2b_i-1)s^N + b_i(b_i-1)s^{N+i}} = g_{N-i}^{r_i^2} g_N^{r_i(2b_i-1)} g_{N+i}^{b_i(b_i-1)}$

$\mathcal{B}$ can compute $X = g_{N-i}^{r_i^2} g_N^{r_i(2b_i-1)}$, so it can obtain $g_{N+i} = (B_i \cdot X^{-1})^{\frac{1}{b_i(b_i-1)}}$ where $b_i(b_i-1) \neq 0$ as $b_i \notin \{0,1\}$, and thus the N+i-DHE assumption is broken. Using Proposition 1 we have that the N-BDHI assumption is broken as well.

Perfect Zero-Knowledge. We justify why the proof is perfectly witness-indistinguishable. Consider a pair of witnesses $(r_i, b_i), (r_i', b_i')$ for some commitment $C_i = g^{r_i} g_i^{b_i} = g^{r_i'} g_i^{b_i'}$. First observe that there is only one possible value that satisfies equations $A_i = A_i' = C_i^{s^{N-i}}$. Secondly we have that $B_i = g^{r_i^2 s^{N-i} + r_i(2b_i-1)s^N}$ and $B_i' = g^{r_i'^2 s^{N-i} + r_i'(2b_i'-1)s^N}$ are uniformly random and thus are perfectly indistinguishable.

We describe now the zero-knowledge simulator. It generates the common reference string correctly, and also learns the trapdoor so it can create commitments that can be opened to any value. As the commitment can be opened, it is straightforward to compute a proof A_i, B_i.

Let us justify why the simulator perfectly simulates the real proof. Consider the *hybrid* stateful algorithm where the simulator generates the trapdoor and the common reference string but opens the commitment to the real bit b_i. Then as the randomness is known, the hybrid algorithm can compute the proof as well. As the commitment is perfectly trapdoor the real argument is perfectly indistinguishable from the hybrid algorithm. Finally, as the argument is perfect witness-indistinguishable, the hybrid is perfectly indistinguishable from the simulated proof. ■

7.3 Base equivalence argument

Let $\theta \xleftarrow{R} \mathbb{Z}_p$. Consider the commitment to the bit vector $\vec{C} = (C_i)_{i\in[\kappa]} = (g^{r_i} g_i^{\theta[i]})_{i\in[\kappa]}$ where $r_i \in \mathbb{Z}_p$ for each $i \in [\kappa]$ and also $D = g^{\theta}$. In this section we introduce a ZKPoK that the prover can open a commitment to a vector of bits such that each bit in position i, corresponds to the i^{th} bit of θ, which is hidden as the discrete logarithm of D. This protocol will allow us to blind the signature with some factor θ (in the exponent) and then reveal each bit of this exponent gradually without leaking any additional information. The idea is the following. Given $\theta \in \mathbb{Z}_p$ and $\vec{C} = (g^{r_i} g_i^{\theta[i]})_{i\in[\kappa]}$, the prover proceeds in two steps. First it computes a proof of knowledge of the representation for C_i in base (g, g_i) (for each $i \in [N]$), and of the discrete logarithm of D in base g. It also runs the protocol described in Figure 7.1 to prove that the commitment can be opened to a bit vector. Then it computes $D' = \frac{\prod_{i\in[\kappa]} g^{r_i} g_i^{\theta[i]}}{g^r}$ where $r = \sum_{i\in[\kappa]} r_i$. Here the prover computes some compressed representation of the bit vector commitment and removes the randomness. Intuitevely we can observe that, though the randomness is removed, D' does not reveal any further information about θ, as the bits are still hidden in the discrete logarithm and D' is uniformly random. The prover will need to convince the verifier that r is indeed the accumulated randomness of the bit vector commitment. To do so, it computes $U = D'^{\frac{1}{s}} = (\prod_{i\in[\kappa]} g_i^{\theta[i]})^{\frac{1}{s}} = \prod_{i\in[\kappa]} g_{i-1}^{\theta[i]}$ where we recall that $g_0 = g$. Observe that this value can be computed without knowing s. In order to verify this proof, the verifier will check that $e(\frac{\prod_{i\in[\kappa]} C_i}{g^r}, g) = e(U, g_1)$. Once the randomness of the bit vector is removed one can "move the vector to the left by one position". If r would not be equal to $\sum_{i\in[\kappa]} r_i$ this would not be possible without breaking some of the assumptions. The second step consists in checking that the condensed bit vector commitment $U = \prod_{i\in[\kappa]} g_{i-1}^{\theta[i]}$ is "equivalent" to the simple commitment g^{θ}. This is done by noting that $U = \prod_{i\in[\kappa]} g_{i-1}^{\theta[i]} = g^{P(s)}$ where $P(\cdot)$ is the polynomial $P(X) = \sum_{i\in[\kappa]} \theta[i] X^{i-1}$. This means in particular that $P(2) = \sum_{i\in[\kappa]} \theta[i]2^{i-1} = \theta$. Thus, we need to prove that $P(s) - P(2) = P(s) - \theta$ is divisible by $s - 2$. The prover can compute the coefficients of the formal polynomial[3] $W(\cdot)$ such that $P(X) - P(2) = W(X)(X - 2)$. Then, using the common reference string CRS the prover obtains $V = g^{W(s)}$. Verifying the "base equivalence" statement consists in checking that $e(\frac{U}{D}, g) = e(V, g_1 g^{-2}) = e(V, g^{s-2})$. This means that indeed $\theta = P(2)$ and thus, the coefficients of $P(\cdot)$ correspond to the binary decomposition of θ. The full protocol is detailed in figure 7.2.

Theorem 14 *The protocol in Figure 7.2 is a ZKPoK that the prover knows the discrete logarithm θ of D, and $(r_i, b_i)_{i\in[\kappa]}$ for each $i \in [\kappa]$, such that $C_i = g^{r_i} g_i^{b_i}$ with $b_i \in \{0, 1\}$, and $\theta = \sum_{i\in[\kappa]} b_i 2^{i-1}$. The protocol is perfectly complete and perfect zero-knowledge. Moreover if the κ-SDH and the κ-BDHI assumptions hold, the prover has negligible probability to output $(r_i, b_i)_{i\in[\kappa]}$, θ and valid proofs π_{equiv}, π_{bit} such that $\vec{b} = (b_i)_{i\in[\kappa]} \notin \{0, 1\}^\kappa$, or $\sum_{i\in[\kappa]} b_i 2^{i-1} \neq \theta$.*

Proof. Perfect Completeness. The prover can compute $U = (\frac{\prod_{i\in[\kappa]} C_i}{g^r})^{\frac{1}{s}}$ without knowing s because $U = \prod_{i\in[\kappa]} g_{i-1}^{\theta[i]}$. Indeed U corresponds to the vector $(0, \theta[1], ..., \theta[\kappa])$ that is moved by one position to the left. Similarly V can be computed because the prover knows the coefficients $W[i]$ of the polynomial $W(\cdot)$ of degree $\kappa - 2$, so we have $V = \prod_{i\in[\kappa-1]} g_{i-1}^{W[i]}$. The rest follows by inspection.

Hardness to open the commitment to a vector which is not the binary expansion of θ. Assume there exists an adversary $\mathcal{A}$ that is able to open the commitment $\vec{C}$ to a vector which is not the binary expansion of θ, the discrete logarithm of D. We build the following adversary $\mathcal{B}$ that breaks the κ-BDHI assumption or the κ-SDH assumption. $\mathcal{B}$ receives the challenge tuple $(g_0, g_1, g_2, \cdots, g_\kappa)$. This challenge tuple stands for the CRS and is sent to $\mathcal{A}$. $\mathcal{B}$, using $\vec{PK}$ and π_{bit} can break the κ-BDHI assumption in case $\vec{C}$ can be opened to a non-bit vector (see Theorem 13). Henceforth we assume that $\vec{C}$ can be opened by the prover to a bit vector. As adversary $\mathcal{A}$ wins, $\mathcal{B}$ obtains

- θ such that $D = g^\theta$ (using PK_θ).

[3]That is a polynomial which variable X stands for s which is unknown.

Common reference string: Input $(1^\kappa, \kappa)$

1. $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BMGen}(1^\kappa)$.
2. $s \overset{R}{\leftarrow} \mathbb{Z}_p$.
3. Return $\mathsf{CRS} = \langle (p, \mathbb{G}, \mathbb{G}_T, e), (g_0, g_1, g_2, ..., g_\kappa) \rangle$ where for all $i \in [0..\kappa] : g_i = g^{s^i}$.

Statement: The statement is formed by a vector of elements of $\mathbb{G}$: $(D, C_1, C_2, ..., C_\kappa)$ where $\vec{C} = (C_i)_{i \in [\kappa]}$ is a commitment to a bit vector as defined in section 7.2. The claim is that the prover can open the commitment to a bit vector which is the binary expansion of the discrete logarithm of D, also known by the prover.

Proof: Input $(\mathsf{CRS}, \theta, r_1, ..., r_\kappa)$

1. Compute $D = g^\theta$.
2. Compute for every $i \in [\kappa]$: $C_i = g^{r_i} g_i^{\theta[i]}$.
3. Compute the π_{bit} as described in Figure 7.1, with $b_i = \theta[i]$ for each $i \in [N]$.
4. Compute $r = \sum_{i \in [\kappa]} r_i$.
5. Compute $U = (\frac{\prod_{i \in [\kappa]} C_i}{g^r})^{\frac{1}{s}}$ using the common reference string CRS and the bit vector $\theta[\cdot]$.
6. Compute the formal polynomial $W(\cdot)$ such that $P(X) - P(2) = W(X)(X-2)$ where $P(X) = \sum_{i \in [\kappa]} \theta[i] X^{i-1}$, and $P(2) = \sum_{i \in [\kappa]} \theta[i] 2^{i-1} = \theta$. Compute $V = g^{W(s)}$ using the coefficients of the formal polynomial $W(\cdot)$ and the common reference string CRS.
7. Compute the ZKPoK $PK_\theta = PK\{\theta : g^\theta\}$ and $\vec{PK} = (PK\{(r_i, b_i) : C_i = g^{r_i} g_i^{b_i}\})_{i \in [\kappa]}$ as described in section 2.6.2.
8. Return $\pi = (\pi_{equiv}, \pi_{bit}, \vec{PK}, PK_\theta)$ where $\pi_{equiv} = (r, U, V)$.

Verification: Input (CRS, C, π)

1. Parse C as $(D, (C_i)_{i \in [\kappa]})$.
2. Parse π as $(\pi_{equiv}, \pi_{bit}, \vec{PK}, PK_\theta)$.
3. Check π_{bit} as described in Figure 7.1.
4. Check that $r \in \mathbb{Z}_p$.
5. Check that $(U, V, D, C_1, ..., C_\kappa) \in \mathbb{G}^{\kappa+3}$.
6. Compute $D' = \frac{\prod_{i \in [\kappa]} C_i}{g^r}$.
7. Check that $e(D', g) = e(U, g_1)$.
8. Check that $e(\frac{U}{D}, g) = e(V, g_1 g^{-2})$.
9. Verify the proofs of knowledge $\vec{PK}$ and PK_θ.
10. Accept if all tests pass in which case return valid otherwise return $\perp$.

Figure 7.2: ZKPoK that the prover can open a commitment to a bit vector which is the binary expansion of the discrete logarithm of some group element D.

- $(r_i, \theta^*[i]) \in (\mathbb{Z}_p \times \{0,1\})^\kappa$ for $i \in [\kappa]$ such that $\vec{C} = (C_i)_{i\in[\kappa]}$ where $C_i = g^{r_i} g_i^{\theta^*[i]}$ (using $\vec{PK}$), and $\theta^* \neq \theta$.

- $\pi_{equiv} = (r, U, V) \in \mathbb{Z}_p \times \mathbb{G} \times \mathbb{G}$.

Assume first that $r \neq \sum_{i\in[\kappa]} r_i \bmod p$, then we can deduce that $U = (g^{\sum_{i\in[\kappa]} r_i - r} g^{\theta_i^* s^i})^{1/s}$. Since $g^{\frac{\theta_i^* s^i}{s}} = g_{i-1}^{\theta_i^*}$ is easily computable from the known $\theta^*[i]$, $\mathcal{B}$ can deduce $g^{\frac{\sum_{i\in[\kappa]} r_i - r}{s}}$ and as $\delta = \sum_{i\in[\kappa]} r_i - r \neq 0 \bmod p$ is known, $\mathcal{B}$ can compute $g^{\frac{1}{s}} = (\frac{U}{g_{i-1}^{\theta_i^*}})^{\frac{1}{\delta}}$ and thus the κ-DHI assumption is broken. From now on we assume that $\sum_{i\in[\kappa]} r_i = r$. As the adversary $\mathcal{A}$ wins, this means that there exists some $j \in [\kappa]$ such that $\theta[j] \neq \theta^*[j]$. Moreover, as the verification involving V passes, we have that $V = g^{\frac{\sum_{i\in[\kappa]} \theta^*[i] s^{i-1} - \theta[i] 2^{i-1}}{s-2}}$. As the decomposition in binary is unique we have that $\Delta = \sum_{i\in[\kappa]} 2^{i-1}(\theta^*[i] - \theta[i]) \neq 0 \bmod p$. We can rewrite V as $V = g^{\frac{\Delta}{s-2} + \frac{\sum_{i\in[\kappa]} \theta[i]^* s^{i-1} - \theta[i] 2^{i-1}}{s-2}} = g^{\frac{\Delta}{s-2} + \frac{\sum_{i\in[\kappa]} \theta[i]^*(s^{i-1} - 2^{i-1})}{s-2}} = g^{\frac{\Delta}{s-2} + Z(s)}$ where the coefficients of $Z(\cdot)$ are efficiently computable by $\mathcal{B}$ because $\forall i \in [\kappa] : (s-2) | (s^{i-1} - 2^{i-1})$. As $\Delta \in \mathbb{Z}_p$ is also known this means $\mathcal{B}$ can compute $g^{\frac{1}{s-2}} = (\frac{V}{g^{Z(s)}})^{\frac{1}{\Delta}}$ and thus the κ-SDH assumption is broken.

Perfect Zero-Knowledge. The simulator works as follows. It generates the common reference string CRS correctly and saves the trapdoor s. Given the statements D and $\vec{C} = (C_i)_{i\in[\kappa]}$ such that $\vec{C}$ is formed by Pedersen commitments to bits in positions $1, ..., \kappa$ and such that D and $\vec{C}$ are equivalent with respect to bases $(2, s)$, the simulator chooses a random $r' \in \mathbb{Z}_p$ and reveals it as the randomness of $\prod_{i\in[\kappa]} C_i$. Then, the simulator sets $U = (\frac{\prod_{i\in[\kappa]} C_i}{g^{r'}})^{\frac{1}{s}}$ and $V = (\frac{U}{D})^{\frac{1}{s-2}}$. To see that r', U, V produced by the simulator are indistinguishable from values of a real experiment, we observe that:

- r' is uniformly distributed as well as r.

- D' is equal to $f_1(r')$ in the simulated experiment and $f_1(r)$ in the real experiment, where $f_1 : \mathbb{G} \to \mathbb{G}$ is defined as $f_1(x) = \frac{\prod_{i\in[\kappa]} C_i}{x}$

- U is equal to $f_2(r')$ in the simulated experiment and $f_2(r)$ in the real experiment, where $f_2 : \mathbb{G} \to \mathbb{G}$ is defined as $f_2(x) = f_1(x)^{\frac{1}{s}}$.

- V is equal to $f_3(r')$ in the simulated experiment and $f_3(r)$ in the real experiment, where $f_3 : \mathbb{G} \to \mathbb{G}$ is defined as $f_3(x) = (\frac{f_2(x)}{D})^{\frac{1}{s-2}}$.

■

7.4 Fair Exchange of Short Signatures without TTP

Our fair exchange protocol for digital signatures works as follows. At the beginning a common reference string CRS is generated. Then each participant runs $\mathsf{FEKeyGen}(1^\kappa)$ to obtain a pair of (public/private) keys (pk, sk) for the signing algorithm. At this point each participant executing $\mathsf{EncSigGen}(\mathsf{CRS}, sk, m)$ will compute an encrypted signature γ for the message m, using the signature σ_m blinded with some factor θ. This value γ will also contain the proofs that relate the signature σ_m with some bit vector commitment to θ.

The rest is straightforward: Each participant sends the encrypted signature. If all the verifications pass, the first participant $\mathcal{P}_A$ will ask to $\mathcal{P}_B$ to open the commitment of the first bit of θ_A. If the opening is successful, $\mathcal{P}_B$ will do the same for its own blinding factor θ_B. The process is repeated for each bit until all the bits of the blinding factors are recovered. Finally, each player can compute the signature by "canceling out" the blinding factor θ. The abstract syntax of the protocol is described in Fig. 7.3.

We now describe more in detail how the encrypted signature is constructed, which is the core of our construction. The encrypted signature contains:

1. A commitment $\vec{C}$ to the bit string formed by the bits of θ as described in Section 7.2.

2. $\tilde{\sigma}$, the signature of the message m blinded by θ.

3. Proofs to guarantee that the bit vector commitment encrypts the binary decomposition of the blinding factor θ.

4. A proof in order to convince the verifier that γ is the encryption of σ_m under some blinding factor θ which is hidden in the basic commitment g^θ.

5. A proof of knowledge of the discrete logarithm of D and a proof of knowledge of the representation of each bit commitment of the vector $\vec{C}$. These proofs of knowledge will allow us to keep simulating the adversary despite the fact that it aborts.

A detailed description of the concrete protocol is given in Fig. 7.4.

	$\mathcal{P}_A(\mathsf{CRS}, m_A, m_B)$		$\mathcal{P}_B(\mathsf{CRS}, m_A, m_B)$
1	$(sk_A, pk_A) \leftarrow \mathsf{FEKeyGen}(1^\kappa)$		
2	pk_A	$\longrightarrow$	
3			$(sk_B, pk_B) \leftarrow \mathsf{FEKeyGen}(1^\kappa)$
4		$\longleftarrow$	pk_B
5	$(\theta_A, \vec{r}_A, \gamma_A) \leftarrow \mathsf{EncSigGen}(\mathsf{CRS}, sk_A, m_A)$		
6	γ_A	$\longrightarrow$	
7			$(\theta_B, \vec{r}_B, \gamma_B) \leftarrow \mathsf{EncSigGen}(\mathsf{CRS}, sk_B, m_B)$
8		$\longleftarrow$	γ_B
10	$v \leftarrow \mathsf{EncSigCheck}(\mathsf{CRS}, pk_B, m_B, \gamma_B)$		
11	**if** $v = \perp$ **then ABORT**		
12			$v \leftarrow \mathsf{EncSigCheck}(\mathsf{CRS}, pk_A, m_A, \gamma_A)$
13			**if** $v = \perp$ **then ABORT**
	for $i = 1$ **to** κ:		
14	$\mathsf{open}_{A,i} \leftarrow \mathsf{KeyBitProofGen}(\mathsf{CRS}, \vec{r}_A, \theta_A, i)$		
15	$\mathsf{open}_{A,i}$	$\longrightarrow$	
16			$\mathsf{open}_{B,i} \leftarrow \mathsf{KeyBitProofGen}(\mathsf{CRS}, \vec{r}_B, \theta_B, i)$
17		$\longleftarrow$	$\mathsf{open}_{B,i}$
19	$v_i \leftarrow \mathsf{KeyBitCheck}(\mathsf{CRS}, \mathsf{open}_{B,i}, i)$		
20	**if** $v_i = \perp$ **then ABORT**		
21			$v_i \leftarrow \mathsf{KeyBitCheck}(\mathsf{CRS}, \mathsf{open}_{A,i}, i)$
22			**if** $v_i = \perp$ **then ABORT**
	end for		
23	$\sigma_{m_B} \leftarrow \mathsf{EncSigDecrypt}(\gamma_B, \theta_B)$		
24			$\sigma_{m_A} \leftarrow \mathsf{EncSigDecrypt}(\gamma_A, \theta_A)$

Figure 7.3: Abstract fair exchange protocol.

We say that the protocol is *perfectly complete*[4] if, and only if, both players $\mathcal{P}_A$ and $\mathcal{P}_B$ that follow the protocol obtain respectively $\sigma_A = \mathtt{SSig}(sk_B, m_B)$, the signature of message m_B and $\sigma_B = \mathtt{SSig}(sk_A, m_A)$, the signature of message m_A, with probability 1.

We say that the protocol is *(partially) fair* if, at the end of the execution of the protocol (be it normal or anticipated by the abortion of the adversary), the

[4]Here *complete* does not refer to fairness.

probability of both players to recover their corresponding signature differs at most by a polynomial factor in the security parameter κ. As mentioned in the introduction, the advantage of this approach is that it avoids trying to compare the *exact* running time of the participants and thus allows to capture in a simple, but precise manner, the intuition of *partial fairness.*

Definition 37 *(Partial fairness) We define the* partial fairness *of the protocol through the following experiment: The adversary $\mathcal{A}$ plays the role of the corrupted player say w.l.o.g. $\mathcal{P}_A$. Thus, $\mathcal{P}_B$ is honest and follows the protocol. $\mathcal{O}_{\mathsf{SSig}}(\cdot)$ is the signing oracle for the signature scheme* `SSig` *relative to the public key pk_B of $\mathcal{P}_B$.*

1. *$\mathcal{A}$ asks for signature computations for arbitrary messages to $\mathcal{O}_{\mathsf{SSig}}(\cdot)$.*
2. *$\mathcal{A}$ chooses the messages m_A and m_B on which the fair exchange protocol will be run, with the restriction that m_B must not have been requested before to $\mathcal{O}_{\mathsf{SSig}}(\cdot)$.*

   $\mathcal{A}$ computes also its public key pk_A and sends it to $\mathcal{P}_B$.
3. *$\mathcal{A}$ then interacts in arbitrary way with $\mathcal{P}_B$.*
4. *If $\mathcal{A}$ has aborted before ending the protocol, then let $\theta^*_A[1..i]$ $(0 \leq i \leq \kappa)$ be the partial blinding obtained by $\mathcal{P}_B$. At this point we assume that $\mathcal{P}_B$ will try to compute $\mathsf{SSig}(sk_A, m_A)$ by choosing at random some element in the remaining space of size $2^{\kappa-i}$. We call this tentative signature σ_B.*
5. *$\mathcal{A}$ asks for signature computations for arbitrary messages to $\mathcal{O}_{\mathsf{SSig}}(\cdot)$ with the restriction that the message must be different from m_B.*
6. *$\mathcal{A}$ keeps running its own algorithm and finally outputs a tentative signature σ_A on m_B relative to public key pk_B, and also another tentative signature σ^* on a message m^*, different from m_B and that has not been queried before to the signing oracle $\mathcal{O}_{\mathsf{SSig}}(\cdot)$.*

```
$\mathsf{FESetup}(1^\kappa)$
    1. $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BMGen}(1^\kappa)$
    2. $s \xleftarrow{R} \mathbb{Z}_p$
    3. Return $\mathsf{CRS} = \langle (p, \mathbb{G}, \mathbb{G}_T, e, g), (g_0, g_1, g_2, ..., g_\kappa) \rangle$ where for all $i \in [0..\kappa] : g_i = g^{s^i}$.
$\mathsf{FEKeyGen}(1^\kappa)$
    1. $(sk, pk) \leftarrow \mathsf{SKG}(1^\kappa)$ where $sk = (g, x, y)$ and $pk = (g, u, v)$ with $u = g^x$ and $v = g^y$, like described in Section 2.4.2.
    2. Return $(sk, pk)$.
$\mathsf{EncSigGen}(\mathsf{CRS}, sk, m)$
    1. Compute $\theta \xleftarrow{R} \mathbb{Z}_p$.
    2. Compute $D = g^\theta$. [RELEASE].
    3. Compute $\vec{C} = (C_i)_{i \in [\kappa]} = (g^{r_i} g_i^{\theta[i]})_{i \in [\kappa]}$. [RELEASE].
    4. Compute $\pi_{bit}$ that shows that $\vec{C}$ is the encryption of a binary vector as described in Figure 7.1.
    5. Compute $\pi_{equiv}$ that shows that $\vec{C}$ is the encryption of the bits of the binary decomposition of the blinding factor $\theta$
       as described in Figure 7.2.
    6. Compute $PK_\theta = PK\{\theta : g^\theta\} = (c = \mathsf{H}(g^r), z = r - c\theta)$ , where $r \xleftarrow{R} \mathbb{Z}_p$. (see Section 2.6.2). [RELEASE].
    7. Compute $\vec{PK}$, a vector where each component at position $i$ is ZKPoK for the representation of $C_i$ in base $(g, g_i)$: $\vec{PK} =$
       $(PK\{(r_i, \theta[i]) : g^{r_i} g_i^{\theta[i]}\})_{i \in [\kappa]} = ((c_i, z_{1,i}, z_{2,i}))_{i \in [\kappa]}$ where $c_i = \mathsf{H}(g^{r_{1,i}} g_i^{r_{2,i}})$,$z_{1,i} = r_{1,i} - c_i r_i$, $z_{2,i} = r_{2,i} - c_i \theta[i]$
       and $r_{1,i}, r_{2,i} \xleftarrow{R} \mathbb{Z}_p$ for each $i \in [\kappa]$ (see Section 2.6.2). [RELEASE].
    8. Parse $sk$ as $(g, x, y)$.
    9. Set $r_\sigma \xleftarrow{R} \mathbb{Z}_p$.
   10. Compute $\sigma = (\sigma', r_\sigma) \leftarrow \mathsf{SSig}(sk, m)$ where $\sigma' = g^{\frac{1}{x+m+yr_\sigma}}$.
   11. Set $\tilde{\sigma} \leftarrow (\sigma'^\theta = g^{\frac{\theta}{x+m+yr_\sigma}}, r_\sigma) = (\tilde{\sigma}', r_\sigma)$.
   12. Set $\gamma \leftarrow (D, \vec{C}, \pi_{bit}, \pi_{equiv}, PK_\theta, \vec{PK}, \tilde{\sigma})$.
   13. Return $(\theta, \vec{r}, \gamma)$, where $\vec{r} = (r_i)_{i \in [\kappa]}$ is the randomness vector of the commitment $\vec{C}$.
$\mathsf{EncSigCheck}(\mathsf{CRS}, pk, m, \gamma)$
    1. Parse $\gamma$ as $\gamma = (D, \vec{C}, \pi_{bit}, \pi_{equiv}, PK_\theta, \vec{PK}, \tilde{\sigma})$.
    2. Check $\pi_{bit}$ as described in Figure 7.1.
    3. Check $\pi_{equiv}$ as described in Figure 7.2.
    4. Check $PK_\theta$ by verifying that $c = \mathsf{H}(D^c g^z)$ (see Section 2.6.2).
    5. Check $PK_\theta$ by verifying that for each $i \in [\kappa]$ we have that $c_i = \mathsf{H}(C_i^{c_i} g^{z_{1,i}} g_i^{z_{2,i}})$ (see Section 2.6.2).
    6. Parse $pk$ as $pk = (g, u, v)$.
    7. Check that $e(\tilde{\sigma}, ug^m v^{r_\sigma}) = e(D, g)$.
    8. Return valid if all tests pass, $\perp$ otherwise.
$\mathsf{KeyBitProofGen}(\mathsf{CRS}, \vec{r}, \theta, i)$
    1. Opens the $i^{\text{th}}$ commitment of $\vec{C}$, that is $(\theta[i], r_i)$ such that $C_i = g^{r_i} g_i^{\theta[i]}$.
    2. Return open $\leftarrow (\theta[i], r_i)$.
$\mathsf{KeyBitCheck}(\mathsf{CRS}, \mathsf{open}, i)$
    1. Parse open as $(b, r)$.
    2. Check that $C_i = g^r g_i^b$ and $b \in \{0, 1\}$.
$\mathsf{EncSigDecrypt}(\gamma, \theta)$
    1. Parse $\gamma$ as $\gamma = (D, \vec{C}, \pi, PK_\theta, \vec{PK}, \tilde{\sigma})$.
    2. Parse $\tilde{\sigma}$ as $\tilde{\sigma} = (\tilde{\sigma}', r_\sigma)$.
    3. Compute $\sigma' = \tilde{\sigma}'^{\frac{1}{\theta}}$.
    4. Return $\sigma = (\sigma', r_\sigma)$.
```

Figure 7.4: **Implementation of the fair exchange protocol.** For the sake of simplicity we introduce the notation [**RELEASE**] which means that the first player(the one who sends the first value) must release the currently computed value and wait for the second player to send his before releasing the next one.

Operation	**# Exp**	**# Mult**	**# BM**	**# Div**	**# Hash**
Step 2 $\mathsf{EncSigGen}$	1				
Bit Vector commitment	κ	κ			
BV-NIZK	4κ	2κ			
BE-NIZK	2κ	$2\kappa - 3$		1	
ZKPoK	$2\kappa + 1$	κ			$\kappa + 1$
Check BV-NIZK		κ	4κ		
Check BE-NIZK	1	$\kappa - 1$	4	2	
Check ZKPoK	$3\kappa + 2$	$2\kappa + 1$			$\kappa + 1$
Step 7 $\mathsf{EncSigCheck}$	2	2	2		
$\mathsf{KeyBitCheck}$	κ	κ			
$\mathsf{EncSigDecrypt}$	1				
Sum	$13\kappa + 8$	$11\kappa - 1$	$4\kappa + 6$	3	$2\kappa + 2$

Figure 7.5: **Time complexities of the fair exchange protocol.** This figure shows the number of cryptographic operations performed by each participant during the whole protocol. The first block corresponds to the algorithm $\mathsf{EncSigGen}$, the second block to the algorithm $\mathsf{EncSigCheck}$. BV-NIZK stands for the NIZK argument to prove a commitment is the encryption of a bit vector as depicted in Fig. 7.1. BE-NIZK stands for the NIZK argument to prove the equivalence between a commitment to a bit vector and the discrete logarithm of $D = g^{\theta}$, as depicted in Fig. 7.2. #Exp, #Mult, #BM, and #Div correspond respectively to the number of group exponentiations, group multiplications, bilinear map applications and group inversions. #Hash is the number of hash evaluations.

The protocol is said to be partially fair *if, and only if,*

$$\frac{\Pr[\,\mathsf{SVf}(pk_B, m_B, \sigma_A) = \mathsf{valid}\,]}{\Pr[\,\mathsf{SVf}(pk_A, m_A, \sigma_B) = \mathsf{valid}\,]} \leq Q(\kappa) \wedge \Pr[\,\mathsf{SVf}(pk_B, m^*, \sigma^*) = \mathsf{valid}\,] = \mathsf{neg}(\kappa)$$

where $Q(\cdot)$ is some polynomial and the probability is taken over the random choices of $\mathcal{A}$ and $\mathcal{P}_B$.

Note that, in this definition, it is fundamental to consider the case where the adversary reveals its own signature to the honest player by following the protocol, but is also able to compute another signature on a message different from m_B. To understand why, it is sufficient to think of a protocol where the honest player releases at the end its private key for signing. In this case the honest player may have obtained the signature of the adversary, and we could say that somehow the exchange is fair, but obviously this solution should not be considered as secure.

Simultaneous hardness of bits for discrete logarithm. Our construction relies on the idea of releasing gradually the bits of $\theta \in \mathbb{Z}_p$ the discrete logarithm in base g of $D = g^\theta$. A problem that could arise in this situation would be that some θ values are somehow easier to find than others especially when some of the bits are released. This might help an adversary to retrieve θ much faster (by a factor greater than a polynomial) and thus break the security of our protocol. To overcome this issue we need to introduce the *Simultaneous hardness of bits of the discrete logarithm* assumption which states that a polynomial time adversary cannot distinguish[5] between a random sequence of $j = \kappa - \omega(\log \kappa)$ bits and the first j bits of θ when given $D = g^\theta$.

Definition 38 *(Simultaneous hardness of bits for discrete logarithm) Let $\mathbb{G}$ be a cyclic group of prime order p. We say that the* Simultaneous hardness of bits for discrete logarithm *(SHDL) assumption holds if for every PPT adversary $\mathcal{A}$ and for any $l = \omega(\log \kappa)$ we have that the following quantity is negligible in κ.*

$$Adv^{SHDL}(\mathcal{A}, \kappa) = |\Pr \left[\begin{array}{c} \theta \overset{R}{\leftarrow} \mathbb{Z}_p : \\ 1 \leftarrow \mathcal{A}(g^\theta, \theta[1..\kappa - l]) \end{array} \right] - \Pr \left[\begin{array}{c} \theta, \alpha \overset{R}{\leftarrow} \mathbb{Z}_p : \\ 1 \leftarrow \mathcal{A}(g^\theta, \alpha[1..\kappa - l]) \end{array} \right] |$$

where the probability is taken over the random choices of $\mathcal{A}$.

Schnorr [Sch98] showed that the $SHDL$ holds in the generic group model by computing the following upper bound on the advantage of the adversary

$$Adv^{SHDL}(\mathcal{A}, \kappa) = O(\kappa(\kappa - l)\sqrt{t}(\frac{2^{\kappa - l}}{2^\kappa})^{1/4})$$

where t is the number of generic group operations of the adversary. Thus, if we set $l = \omega(\log \kappa)$ we obtain that $Adv^{SHDL}(\mathcal{A}, \kappa) = O(\kappa(\kappa - \omega(\log \kappa))\sqrt{t}(2^{-\omega(\log \kappa)})^{1/4})$ which is negligible.

[5]Note that a PPT adversary can easily distinguish both bit strings if $j = \kappa - O(\log \kappa)$ by performing a brute force attack on the remaining bits as $2^{O(\log \kappa)}$ is a polynomial in κ.

The recent work [DJ12] by Duc and Jetchev suggests that results applying to groups of integers modulo a safe prime [PS98, MP05] can be extended to elliptic curves so to reduce the SHDL assumption to more standard ones.

Theorem 15 *The protocol described in Fig. 7.4 is complete. Moreover if the κ-SDH assumption, the κ-BDHI assumption and the SHDL assumption hold, and a securely precomputed common reference string is available, then it is secure in the random-oracle model according to definition 37.*

Proof. As the underlying ZKPoK protocols are perfectly complete, we can see by inspection that so is the fair exchange protocol.

Let us outline the intuition of the security proof. The main challenge for our proof is related to the following facts:

- The adversary may abort during the protocol. In order to cope with this issue, we will use the proofs of knowledge that conceptually will allow us to force the adversary to open all the commitments and thus break some complexity assumption in case the adversary lies, that is does not open the commitments to the bits of the blinding factor θ.

- The adversary might detect that it is simulated, especially when few bits remain to be released, exposing the fact that the simulator does not know the signature and is trying to get it from the adversary.

  In this case, we will rely on the idea that if the adversary opens the commitment correctly (follows the protocol) then it will not get a significative advantage and thus will not win. Moreover, we will need to use the SHDL assumption that will guarantee us that when the simulator releases bits that do no correspond to the blinding factor θ, then, if sufficient bits remain ($\omega(\log \kappa)$), the adversary will not be able to detect it.

Let $\mathcal{A}$ be the adversary that breaks the fairness of our protocol. We consider the following sequence of games.

Game 0. This game corresponds to the security definition 37. In this game, the simulator $\mathcal{B}$ invokes a signing oracle $\mathcal{O}_{\mathtt{SSig}}(\cdot)$ to obtain the signatures, and does not know the trapdoor s of common reference string CRS. We define S_0 to be the event that adversary $\mathcal{A}$ wins.

Game 1. This game is identical to the previous one except that the adversary $\mathcal{A}$ is not allowed to open the commitments C_i to values that are not equal to the i^{th} bit of θ_A, the discrete logarithm of D_A (computed by $\mathcal{A}$).

More precisely, the simulator $\mathcal{B}$ does the following: It asks $\mathcal{A}$ to compute γ_A which is parsed as $(D_A, \vec{C}_A, \pi_{A,bit}, \pi_{A,equiv}, PK_{\theta A}, \vec{PK}_A, \tilde{\sigma}_A)$. For the proofs of knowledge, $\mathcal{A}$ will use $\mathsf{H}(\cdot)$ a random oracle which is controlled by $\mathcal{B}$. So $\mathcal{B}$ will rewind $\mathcal{A}$ and obtain $PK_{\theta}{}'_A$ from which it will obtain θ_A. Similarly, $\mathcal{B}$ will use $\vec{PK}_A[i]$ and $\vec{PK}'_A[i]$ (after rewinding) to extract $(r_{Ai}, \theta_A[i])$ for each $i \in [\kappa]$.

Let S_1 be the event that adversary $\mathcal{A}$ wins. Applying the difference lemma [Sho04] and as we rely on the random-oracle methodology, we have that

$$|\Pr[\,S_0\,] - \Pr[\,S_1\,]\,| \leq 2Adv^{\kappa-BDHI}(\mathcal{B}_1, \kappa, \kappa) + Adv^{\kappa-SDH}(\mathcal{B}_2, \kappa, \kappa)$$

considering that $\mathcal{B}_1$ is the best adversary that breaks $\kappa - BDHI$ assumption and $\mathcal{B}_2$ is the best adversary that breaks the $\kappa - SDH$ assumption. This follows from Proposition 11, Theorem 13 and Theorem 14. Note that the rewindings are *not* nested because each ZKPoK is released in alternation and the committed values are fixed at the beginning of the interaction of $\mathsf{EncSigCheck}$ at steps 2 and 3 (see Figure 7.4).

Game 2. This game is identical to the previous one except that the adversary $\mathcal{A}$ is not allowed to win by outputting a valid signature σ^* on a message different from m_B and other messages the adversary $\mathcal{A}$ may have requested to the simulator $\mathcal{B}$ for signing.

More precisely, the simulator $\mathcal{B}$ will simply follow the protocol by invoking the signing oracle $\mathcal{O}_{\texttt{SSig}}(\cdot)$ when necessary. If $\mathcal{A}$ wins by outputting a valid signature σ^* on a message m^* not queried previously, then $\mathcal{B}$ has been able to forge a signature for scheme $\texttt{SSig}$.

Applying the difference lemma, we have that
$|\Pr[\,S_1\,] - \Pr[\,S_2\,]\,| \leq Adv_{\texttt{SSig}}^{\mathcal{UF-CMA}}(\mathcal{B}_3, \kappa)$, where $\mathcal{B}_3$ is the best adversary for breaking the security of $\texttt{SSig}$.

Game 3. This game is identical to the previous one except that now the common reference string CRS is computed by the simulator $\mathcal{B}$ which now knows the trapdoor s. Let S_3 be the event that $\mathcal{A}$ wins, we have that $\Pr[\,S_2\,] = \Pr[\,S_3\,]$.

Game 4. In this game we perform the following change: The simulator $\mathcal{B}$, instead of committing to the bits of θ_B, will commit (and release gradually) the bits of some other random value $\alpha \xleftarrow{R} \mathbb{Z}_p$.

Note that, as $\mathcal{B}$ knows the trapdoor s, it will be able to simulate the proofs and make all the tests pass. We detail next how $\mathcal{B}$ computes the values of the protocol.

1. Run $\mathsf{BMGen}(1^\kappa)$ to obtain $(p, \mathbb{G}, \mathbb{G}_T, e, g)$.
2. Get the public key for the signature scheme $pk = (g, u, v)$ from the oracle $\mathcal{O}_{\mathtt{SSig}}(\cdot)$.
3. Set $s \xleftarrow{R} \mathbb{Z}_p$.
4. Compute $\mathsf{CRS} \leftarrow \langle (p, \mathbb{G}, \mathbb{G}_T, e, g), (g_0, g_1, ..., g_\kappa) \rangle$ where $\forall i \in [0..\kappa]$: $g_i = g^{s^i}$.
5. Forward the public key of the signature scheme $\mathtt{SSig}$ to adversary $\mathcal{A}$.
6. Set $\alpha \xleftarrow{R} \mathbb{Z}_p$.
7. Set $\theta_B \leftarrow \mathbb{Z}_p$
8. Set $r_\sigma \xleftarrow{R} \mathbb{Z}_p$.
9. Set $D = g^{\theta_B}$.
10. Set $\tilde{\sigma} = (\sigma_{m_B}^{\theta_B}, r_\sigma)$ (σ_{m_B} is queried to the signing oracle $\mathcal{O}_{\mathtt{SSig}}(\cdot)$).
11. Set $(r_1, r_2, ..., r_\kappa) \xleftarrow{R} \mathbb{Z}_p^\kappa$.
12. $\vec{C} = (C_i)_{i\in[\kappa]} = (g^{r_i} g_i^{\alpha[i]})_{i\in[\kappa]}$.

As $\mathcal{A}$ cannot win by outputting a valid signature σ^* on a message different from m_B, this means that if $\mathcal{A}$ does not abort before less than $O(\log\kappa)$ bits remain to be released, it cannot win either. The reason is that, as stated in Game 1, $\mathcal{A}$ is not allowed to lie (that is to open the commitments to bits that do not correspond to the binary expansion of θ_A).

More precisely, we have that $\Pr[\,\mathsf{SVf}(pk_A, m_A, \sigma_B) = \mathsf{valid}\,] \geq \frac{1}{2^{O(\log\kappa)+1}}$ (because

$\mathcal{A}$ has got one bit of advantage, and only $O(\log \kappa)$ bits remain). We can deduce that

$$\frac{\Pr\left[\,\mathsf{SVf}(pk_B, m_B, \sigma_A) = \mathsf{valid}\,\right]}{\Pr\left[\,\mathsf{SVf}(pk_A, m_A, \sigma_B) = \mathsf{valid}\,\right]} \leq \frac{1}{2^{-(\log(O(\kappa)+1)}} = 2^{O(\log \kappa)+1}$$

which is still a polynomial, thus $\mathcal{A}$ did not win.

Let S_4 be the event that $\mathcal{A}$ wins. We have that $|\Pr[\,S_3\,] - \Pr[\,S_4\,]| = Adv^{SHDL}(\mathcal{B}_4, \kappa)$, where $\mathcal{B}_4$ is the best adversary that breaks the SHDL assumption. To see this, we build the following distinguisher $\mathcal{D}$ for the SHDL assumption. $\mathcal{D}$ receives a group element E and a sequence α of $\kappa - \omega(\log \kappa)$ bits. $\mathcal{D}$ will use the simulator $\mathcal{B}$ and replace D by the value E. Then, $\mathcal{D}$ will complete the sequence α with random bits so that the new sequence is κ-bit long. We denote this new sequence $\alpha' = \alpha || \gamma$, where each bit of γ is random and $|\gamma| = \kappa - |\alpha|$. Then, $\mathcal{D}$ will compute the commitments for each bit of α' and also release each of these bits gradually as $\mathcal{B}$ (whether in Game 3 or in Game 4). If $\mathcal{A}$ aborts the simulation (not only the protocol) before, or precisely when, all the bits of α are released, then we can deduce that the discrete logarithm of E does not start with the sequence α (Game 4). Otherwise, this means that indeed α is a prefix of the discrete logarithm of E (Game 3).

Game 5. In this game the simulator will not ask to the oracle to produce the signature on m_B, instead it will compute the values as follows.

1. Run $\mathsf{BMGen}(1^\kappa)$ to obtain $(p, \mathbb{G}, \mathbb{G}_T, e, g)$.
2. Get the public key for the signature scheme $pk = (g, u, v)$ from the oracle $\mathcal{O}_{\texttt{SSig}}(\cdot)$.
3. Set $s \xleftarrow{R} \mathbb{Z}_p$.
4. Compute $\mathsf{CRS} \leftarrow \langle (p, \mathbb{G}, \mathbb{G}_T, e, g), (g_0, g_1, ..., g_\kappa) \rangle$ where $\forall i \in [0..\kappa]$: $g_i = g^{s^i}$.
5. Forward the public key of the signature scheme `SSig` to adversary $\mathcal{A}$.
6. Set $\theta_B \xleftarrow{R} \mathbb{Z}_p$.
7. Set $r_\sigma \xleftarrow{R} \mathbb{Z}_p$.
8. Set $D = (g^{m_B} u v^{r_\sigma})^{\theta_B}$.
9. Set $\tilde{\sigma} = (g^{\theta_B}, r_\sigma)$.

10. Set $(r_1, r_2, ..., r_\kappa) \xleftarrow{R} \mathbb{Z}_p^\kappa$.
11. $\vec{C} = (C_i)_{i \in [\kappa]} = (g^{r_i} g_i^{\theta_B[i]})_{i \in [\kappa]}$.

Note again that all the proofs can be simulated because $\mathcal{B}$ knows s. Moreover, $\mathcal{A}$ is "fooled" because we have $e(\tilde{\sigma}, g^{m_B} u v^{r_\sigma}) = e((g^{m_B} u v^{r_\sigma})^{\theta_B}, g) = e(D, g)$ and all values computed by the simulator $\mathcal{B}$ are perfectly indistinguishable from those of a real experiment. Let S_5 be the event that $\mathcal{A}$ wins. Both transcripts (from this game and the previous one) are indistinguishable as in both cases the bits released do not correspond to the binary expansion of the discrete logarithm of D. Thus we can deduce that $\Pr[S_4] = \Pr[S_5]$.

Moreover if $\mathcal{A}$ wins then $\mathcal{B}$ has found a forgery for the signature scheme `SSig`. Thus we have $\Pr[S_5] \leq Adv_{\mathtt{SSig}}^{\mathcal{UF-CMA}}(\mathcal{B}_3, \kappa)$, where, as mentioned before, $\mathcal{B}_3$ is the best adversary that breaks the security of the signature scheme `SSig`.

To summarize, we have that:

$$\begin{aligned}
\Pr[S_0] &\leq |\Pr[S_0] - \Pr[S_5]| + \Pr[S_5] \\
&\leq |\Pr[S_0] - \Pr[S_1]| + |\Pr[S_1] - \Pr[S_2]| \\
&\quad + |\Pr[S_2] - \Pr[S_3]| + |\Pr[S_3] - \Pr[S_4]| \\
&\quad + |\Pr[S_4] - \Pr[S_5]| + \Pr[S_5] \\
&\leq |\Pr[S_0] - \Pr[S_1]| + |\Pr[S_1] - \Pr[S_2]| + |\Pr[S_3] - \Pr[S_4]| \\
&\quad + \Pr[S_5] \\
&\leq 2Adv^{\kappa-BDHI}(\mathcal{B}_1, \kappa, \kappa) + Adv^{\kappa-SDH}(\mathcal{B}_2, \kappa, \kappa) \\
&\quad + Adv_{\mathtt{SSig}}^{\mathcal{UF-CMA}}(\mathcal{B}_3, \kappa) + Adv^{SHDL}(\mathcal{B}_4, \kappa) \\
&\quad + Adv_{\mathtt{SSig}}^{\mathcal{UF-CMA}}(\mathcal{B}_3, \kappa) \\
&= 2Adv^{\kappa-BDHI}(\mathcal{B}_1, \kappa, \kappa) + Adv^{\kappa-SDH}(\mathcal{B}_2, \kappa, \kappa) \\
&\quad + 2Adv_{\mathtt{SSig}}^{\mathcal{UF-CMA}}(\mathcal{B}_3, \kappa) + Adv^{SHDL}(\mathcal{B}_4, \kappa)
\end{aligned}$$

■

7.5 Conclusion

In this work we introduced a practical protocol to exchange short signatures [BB04b, BB08] fairly without relying on a TTP using between others a pair of CRHF that preserve the predicate $\texttt{Equiv}(\theta, \vec{b}) = 1 \Leftrightarrow \theta = \sum_{i=1}^{\kappa} \vec{b}[i]2^{i-1}$. It seems our approach can be applicable to other signature schemes, those where the signature is verified by a single application of the bilinear map. For example, in the BLS scheme [BLS04] a signature on message m has the form $\sigma_m = \mathsf{H}(m)^s$ where s is the secret key and $\mathsf{H}(\cdot)$ is a CRHF. The idea would be to blind σ_m with $\theta \overset{R}{\leftarrow} \mathbb{Z}_p$ and get the encrypted signature $\tilde{\sigma} = \sigma_m^{\theta}$. One could verify that the encrypted signature is well-formed by checking that $e(\tilde{\sigma}, g) = e(\mathsf{H}(m), A)$ and $e(A, g) = e(g^s, g^{\theta})$.

Thus, our techniques might be extended in order to obtain a general framework for building practical fair protocols involving bilinear maps.

Chapter 8

Conclusion

In this thesis, we started first by studying cryptographic accumulators and in particular the dynamic properties of such primitive. This research lead to the design of an authenticated data structure supporting (non)membership queries and the security of which does not rely on a trusted third party [CHKO08]. Then we solved an open question raised by Nicolisi and Fazio [FN02] asking whether accumulators allowing efficient update of all the witnesses at once could be built, answering in the negative [CH10]. Trying to design an optimal authenticated dictionary (where the time to compute a proof is $O(\log N)$ and the time to verify it is $O(1)$), we started to study transitive signature schemes. This lead us to design a new transitive signature scheme for directed trees [CH12] that is more efficient than previous proposals, and achieves the best known worst-case complexity. Moreover our construction enables a practical trade off between the time to combine a signature, $O(\lambda(\frac{N}{\kappa})^{1/\lambda})$, and the time to verify it, $O(\lambda)$. This construction is based on a hash function that allows efficient comparison of two strings (w.r.t. the lexicographical order) only using hashes and a short proof.

The concept of *predicate-preserving collision-resistant hashing* emerged: Indeed we can build CRHFs that somehow *preserve* some of the structure of the values being hashed, enabling the efficient computation of proofs for predicates involving the preimages. Then we applied this concept to build a commitment scheme for a bit vector with a proof that this bit vector is the binary decomposition of some value (hidden in another commitment). This commitment scheme and the related arguments served to build a fair exchange protocol for Boneh-Boyen short signatures [BB04b] that does not rely on a trusted third party, which is the first of the sort to the best of our knowledge.

Finally, we highlight some open problems related to this thesis.

- We studied CRHFs that preserve simple predicates like set membership, existence of a common prefix, and equivalence of two values expressed in different bases. What kind of predicates can be efficiently preserved? What is the relation between the logical complexity of the predicate (number of variables, quantifiers) and the size of the proof or the time to compute such a proof?

- The problem of building an optimal authenticated dictionary is still open. One way to solve it is by using a transitive signature scheme for directed graphs, but this primitive is not known to exist either.

- Can we generalize the concept of predicate-preserving hashing enough, providing concrete constructions, in order to build a general framework to compute on authenticated data like [ABC$^+$12]?

- It seems natural to apply the hash functions of Chapter 6 to build efficient authenticated string pattern schemes. This may lead to constructions where proofs are shorter than those of [Ver11, HL08, HT10, GHS10] which are linear in the size of the string.

Bibliography

[ABC$^+$12] Jae Hyun Ahn, Dan Boneh, Jan Camenisch, Susan Hohenberger, Abhi Shelat, and Brent Waters. Computing on Authenticated Data. In Ronald Cramer, editor, *TCC*, volume 7194 of *LNCS*, pages 1–20. Springer Berlin / Heidelberg, March 2012.

[ACH] Asociación Chilena de Factoring (ACHEF). `http://www.achef.cl/` (last access 5/12/2012).

[ANPS07] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-property-preserving iterated hashing: ROX. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *LNCS*, pages 130–146. Springer Berlin / Heidelberg, 2007.

[ASM08] Man Ho Au, Willy Susilo, and Yi Mu. Practical Anonymous Divisible E-Cash from Bounded Accumulators. In Gene Tsudik, editor, *Financial Cryptography and Data Security*, volume 5143 of *LNCS*, pages 287–301. Springer Berlin / Heidelberg, May 2008.

[ASW97] N. Asokan, Matthias Schunter, and Michael Waidner. Optimistic protocols for fair exchange. In *CCS*, pages 7–17. ACM Press, April 1997.

[ATSM09] Man Au, Patrick Tsang, Willy Susilo, and Yi Mu. Dynamic Universal Accumulators for DDH Groups and Their Application to Attribute-Based Anonymous Credential Systems. In Marc Fischlin, editor, *CT-RSA*, volume 5473 of *LNCS*, pages 295–308. Springer Berlin / Heidelberg, 2009.

[AWSM07] Man Au, Qianhong Wu, Willy Susilo, and Yi Mu. Compact E-Cash from Bounded Accumulator. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *LNCS*, pages 178–195. Springer Berlin / Heidelberg, 2007.

[BB04a] Dan Boneh and Xavier Boyen. Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *LNCS*, pages 223–238. Springer Berlin / Heidelberg, 2004.

[BB04b] Dan Boneh and Xavier Boyen. Short Signatures Without Random Oracles. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *LNCS*, pages 56–73. Springer Berlin / Heidelberg, 2004.

[BB08] Dan Boneh and Xavier Boyen. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *Journal of Cryptology*, 21(2):149–177, 2008.

[BBG05] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical Identity Based Encryption with Constant Size Ciphertext. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *LNCS*, pages 440–456. Springer Berlin / Heidelberg, 2005.

[BCvD87] Ernest Brickell, David Chaum, Jeroen van De Graaf, and Ivan Bjerre Damgård. Gradual and verifiable release of a secret. In Carl Pomerance, editor, *CRYPTO*, volume 293 of *LNCS*, pages 156–166. Springer Berlin / Heidelberg, 1987.

[BdM91] J Benaloh and M de Mare. Efficient Broadcast Time-Stamping. Technical report, Clarkson University Department of Mathematics and Computer Science, 1991.

[BdM93] Josh Benaloh and Michael de Mare. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In Tor Helleseth, editor, *EUROCRYPT*, volume 765 of *LNCS*, pages 274–285. Springer-Verlag, 1993.

[BEG$^+$91] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *Symposium on Foundations of computer science (SFCS)*, pages 90–99. IEEE Computer Society, 1991.

[BF11] Dan Boneh and David Freeman. Homomorphic Signatures for Polynomial Functions. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *LNCS*, pages 149–168. Springer Berlin / Heidelberg, 2011.

[BFKW09] 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*, volume 5443 of *LNCS*, pages 68–87. Springer Berlin / Heidelberg, 2009.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental Cryptography: The Case of Hashing and Signing. In Yvo G. Desmedt, editor, *CRYPTO*, volume 839 of *LNCS*, pages 216–233. Springer Berlin / Heidelberg, July 1994.

[BH92] Dave Bayer and Stuart Haber. Improving the Efficiency and Reliability of Digital Time-Stamping. In *Sequences II: Methods in Communication, Security and Computer Science*, pages 329–334. Springer-Verlag, 1992.

[BK12] Alexandra Boldyreva and Virendra Kumar. A New Pseudorandom Generator from Collision-Resistant Hash Functions. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *LNCS*, pages 187–202. Springer Berlin / Heidelberg, February 2012.

[BLL00] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In *CCS*, pages 9–17. ACM Press, November 2000.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[Blu83] Manuel Blum. How to exchange (secret) keys. *ACM Transactions on Computer Systems*, 1(2):175–193, May 1983.

[BN00] Dan Boneh and Moni Naor. Timed Commitments. In Mihir Bellare, editor, *CRYPTO*, volume 1880 of *LNCS*, pages 236–254. Springer Berlin / Heidelberg, August 2000.

[BN05] Mihir Bellare and Gregory Neven. Transitive Signatures: New Schemes and Proofs. *IEEE Transactions on Information Theory*, 51(6):2133–2151, June 2005.

[BP97] Niko Barić and Birgit Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In Walter Fumy, editor, *EUROCRYPT*, volume 1233 of *LNCS*, pages 480–494. Springer-Verlag, 1997.

[BR97] Mihir Bellare and Phillip Rogaway. Collision-Resistant hashing: Towards making UOWHFs practical. In *CRYPTO*, volume 1294 of *LNCS*, pages 470–484. Springer Berlin / Heidelberg, 1997.

[BS02] Dan Boneh and Alice Silverberg. Applications of Multilinear Forms to Cryptography. Technical report, 2002.

[BV98] Dan Boneh and Ramarathnam Venkatesan. Breaking RSA May Be Easier Than Factoring. In *EUROCRYPT*, volume 1233, pages 1–12. Springer-Verlag, 1998.

[Cam11] Philippe Camacho. Optimal Data Authentication from Directed Transitive Signatures. `http://eprint.iacr.org/2011/439` (last access 5/12/2012), 2011.

[Cam13] Philippe Camacho. Fair Exchange of Short Signatures without Trusted Third Party. In Ed Dawson, editor, *CT-RSA*, volume 7779 of *LNCS*, pages 34–49. Springer Berlin / Heidelberg, 2013.

[CDvdG87] David Chaum, Ivan Bjerre Damgård, and Jeroen van de Graaf. Multiparty Computations Ensuring Privacy of Each Party's Input and Correctness of the Result. In David L Chaum, editor, *CRYPTO*, pages 87–119. Springer-Verlag, August 1987.

[CFM08] Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-Knowledge Sets with Short Proofs. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *LNCS*, pages 433–450. Springer Berlin / Heidelberg, 2008.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. The Random Oracle Methodology, Revisited. *Journal of the ACM*, 51(4):557–594, July 2004.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A Secure and Optimally Efficient Multi-Authority Election Scheme. In Walter Fumy, editor, *EUROCRYPT*, volume 1233 of *LNCS*, pages 103–118. Springer Berlin / Heidelberg, July 1997.

[CH10] Philippe Camacho and Alejandro Hevia. On the Impossibility of Batch Update for Cryptographic Accumulators. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT*, volume 6212 of *LNCS*, pages 178–188. Springer Berlin / Heidelberg, 2010.

[CH12] Philippe Camacho and Alejandro Hevia. Short Transitive Signatures for Directed Trees. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *LNCS*, pages 35–50. Springer Berlin / Heidelberg, 2012.

[CHK$^+$12] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 416–432. Springer Berlin / Heidelberg, 2012.

[CHKO08] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong Accumulators from Collision-Resistant Hashing. In Tzong-Chen

Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security Conference*, volume 5222 of *LNCS*, pages 471–486. Springer Berlin / Heidelberg, September 2008.

[CKS09] 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*, volume 5443 of *LNCS*, pages 481–500. Springer Berlin / Heidelberg, 2009.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In Moti Yung, editor, *CRYPTO*, volume 2442 of *LNCS*, pages 61–76. Springer Berlin / Heidelberg, 2002.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC*, pages 364–369. ACM Press, November 1986.

[Cle90] Richard Cleve. Controlled Gradual Disclosure Schemes for Random Bits and Their Applications. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *LNCS*, pages 573–588. Springer Berlin / Heidelberg, July 1990.

[CMR98] Ran Canetti, Daniele Micciancio, and Omer Reingold. Perfectly one-way probabilistic hash functions. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 131–140. ACM Press, 1998.

[CS97] Jan Camenisch and Markus Stadler. Proof Systems for General Statements about Discrete Logarithms. `ftp://ftp.inf.ethz.ch/pub/crypto/publications/CamSta97b.ps` (last access 5/12/2012), 1997.

[Dam88] Ivan Bjerre Damgård. Collision Free Hash Functions and Public Key Signature Schemes. In David Chaum and Wyn L. Price, editors, *EUROCRYPT*, volume 304 of *Lecture Notes in Computer Science*, pages 203–216, Berlin, Heidelberg, December 1988. Springer Berlin Heidelberg.

[Dam95] Ivan Damgård. Practical and Provably Secure Release of a Secret and Exchange of Signatures. *Journal of Cryptology*, 8(4):201–222, September 1995.

[Die82] Paul F. Dietz. Maintaining Order in a Linked List. In *STOC*, pages 122–127. ACM Press, May 1982.

[DJ12] Alexandre Duc and Dimitar Jetchev. Hardness of Computing Individual Bits for One-Way Functions on Elliptic Curves. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *LNCS*, pages 832–849. Springer Berlin / Heidelberg, 2012.

[DNRV09] Cynthia Dwork, Moni Naor, Guy N Rothblum, and Vinod Vaikuntanathan. How Efficient Can Memory Checking Be? In Omer Reingold, editor, *TCC*, volume 5444 of *LNCS*, pages 503–520. Springer Berlin / Heidelberg, 2009.

[DT08] Ivan Damgård and Nikos Triandopoulos. Supporting Non-membership Proofs with Bilinear-map Accumulators. `http://eprint.iacr.org/2008/538` (last access 5/12/2012), 2008.

[efa] Servicio de Impuestos Internos. `https://palena.sii.cl/dte/mn_info.html` (last access 5/12/2012).

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, June 1985.

[FN02] Nelly Fazio and Antonio Nicolisi. Cryptographic Accumulators: Definitions, Constructions and Applications. Technical report, 2002.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, August 1960.

[FS86] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *LNCS*, pages 186–194. Springer-Verlag, January 1986.

[GHKL11] S. Dov Gordon, Carmit Hazay, Jonathan Katz, and Yehuda Lindell. Complete Fairness in Secure Two-Party Computation. *Journal of the ACM*, 58(6):1–37, December 2011.

[GHS10] Rosario Gennaro, Carmit Hazay, and Jeffrey S. Sorensen. Automata Evaluation and Text Search Protocols with Simulation Based Security. `http://eprint.iacr.org/2010/484` (last access 5/12/2012), 2010.

[GK10] S. Dov Gordon and Jonathan Katz. Partial Fairness in Secure Two-Party Computation. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *LNCS*, pages 157–176. Springer Berlin / Heidelberg, 2010.

[GMPY06] Juan A. Garay, Philip MacKenzie, Manoj Prabhakaran, and Ke Yang. Resource Fairness and Composability of Cryptographic Protocols. In Ran Canetti, editor, *TCC*, volume 3876 of *LNCS*, pages 404–428. Springer, 2006.

[GMR88] 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.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87*, pages 218–229, New York, New York, USA, January 1987. ACM Press.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect Non-interactive Zero Knowledge for NP. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *LNCS*, pages 339–358. Springer Berlin / Heidelberg, 2006.

[GR04] Craig Gentry and Zulfikar Ramzan. RSA Accumulator Based Broadcast Encryption. In Kan Zang and Yuliang Zheng, editors, *Information Security Conference*, volume 3225 of *LNCS*, pages 73–86. Springer Berlin / Heidelberg, 2004.

[Gro10] Jens Groth. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *LNCS*, pages 321–340. Springer Berlin / Heidelberg, 2010.

[GTH02] Michael T. Goodrich, Roberto Tamassia, and Jasminka Hasic. An Efficient Dynamic and Distributed Cryptographic Accumulator. In A. H. Chan and V. D. Gligor, editors, *Information Security Conference*, volume 2433 of *LNCS*, pages 372–388. Springer Berlin / Heidelberg, September 2002.

[GTT08] Michael Goodrich, Roberto Tamassia, and Nikos Triandopoulos. Super-Efficient Verification of Dynamic Outsourced Databases. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *LNCS*, pages 407–424. Springer Berlin / Heidelberg, 2008.

[HL08] Carmit Hazay and Yehuda Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. In Ran Canetti, editor, *TCC*, volume 4948 of *LNCS*, pages 155–175. Springer Berlin / Heidelberg, 2008.

[Hoh03] Susan Rae Hohenberger. *The Cryptographic Impact of Groups with Infeasible Inversion.* Master's thesis, Massachusetts Institute of Technology, 2003.

[HT10] Carmit Hazay and Tomas Toft. Computationally Secure Pattern Matching in the Presence of Malicious Adversaries. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *LNCS*, pages 195–212. Springer Berlin / Heidelberg, 2010.

[Hur39] W.H Hurd. Four Centuries of Factoring. *Quarterly Journal of Economics*, 53(2):305—-311, 1939.

[HW09] Susan Hohenberger and Brent Waters. Realizing Hash-and-Sign Signatures under Standard Assumptions. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *LNCS*, pages 333–350. Springer Berlin / Heidelberg, April 2009.

[HWS12] Qiong Huang, Duncan S. Wong, and Willy Susilo. The Construction of Ambiguous Optimistic Fair Exchange from Designated Confirmer Signature without Random Oracles. In Marc Fischlin, J. Buchmann, and Mark Manulis, editors, *PKC*, volume 7293 of *LNCS*, pages 120–137. Springer Berlin Heidelberg, 2012.

[ifg] International Factors Group (IFG). `http://www.ifgroup.com/` (last access 5/12/2012).

[JMSW02] Robert Johnson, David Molnar, Dawn Xiaodong Song, and David Wagner. Homomorphic Signature Schemes. In Bart Preneel, editor, *CT-RSA*, volume 2271 of *LNCS*, pages 244–262. Springer Berlin / Heidelberg, 2002.

[Jou00] Antoine Joux. A One Round Protocol for Tripartite Diffie-Hellman. In *International Symposium on Algorithmic Number Theory*, pages 385–394, July 2000.

[KG09] Aniket Kate and Ian Goldberg. Distributed Key Generation for the Internet. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 119–128. IEEE, June 2009.

[Koc98] Paul C Kocher. On Certificate Revocation and Validation. In Rafael Hirchfeld, editor, *Financial Cryptography*, volume 1465 of *LNCS*, pages 172–177. Springer Berlin / Heidelberg, 1998.

[KZG10] Aniket Kate, Gregory Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *LNCS*, pages 177–194–194. Springer Berlin / Heidelberg, 2010.

[LBLV98] Helger Lipmaa, Ahto Buldas, Peeter Laud, and Jan Villemson. Time-stamping with binary linking schemes. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *LNCS*, pages 486–501. Springer-Verlag, 1998.

[Lip12] Helger Lipmaa. Secure Accumulators from Euclidean Rings without Trusted Setup. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *ACNS*, volume 7341 of *LNCS*, pages 224–240. Springer Berlin / Heidelberg, 2012.

[LLX07] Jiangtao Li, Ninghui Li, and Rui Xue. Universal Accumulators with Efficient Nonmembership Proofs. In Jonathan Katz and Moti Yung, editors, *ACNS*, volume 4521 of *LNCS*, pages 253–269. Springer Berlin / Heidelberg, 2007.

[LY10] Benoît Libert and Moti Yung. Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs. In Daniele Micciancio, editor, *TCC*, volume 5978 of *LNCS*, pages 499–517, Berlin, Heidelberg, 2010. Springer Berlin / Heidelberg.

[Mer87] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In Carl Pomerance, editor, *CRYPTO*, volume 293 of *LNCS*, pages 369–378. Springer Berlin / Heidelberg, August 1987.

[Mer89] Ralph C. Merkle. A certified digital signature. pages 218–238, July 1989.

[Mic03] Silvio Micali. Simple and fast optimistic protocols for fair electronic exchange. In *PODC*, pages 12–19, New York, New York, USA, July 2003. ACM Press.

[MP05] Philip Mackenzie and Sarvar Patel. Hard Bits of the Discrete Log with Applications to Password Authentication. In Alfred Menezes, editor, *CT-RSA*, volume 3376 of *LNCS*. Springer Berlin Heidelberg, 2005.

[MR02] Silvio Micali and Ronald Rivest. Transitive Signature Schemes. In Bart Preneel, editor, *CT-RSA*, volume 2271 of *LNCS*, pages 236–243. Springer Berlin / Heidelberg, February 2002.

[MRB$^+$02] 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. *ESA*, 2461:219–223–223, August 2002.

[MRK03] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-Knowledge Sets. In *FOCS*, pages 80–91. ACM Press, October 2003.

[MSK02] S Mitsunari, R Sakai, and M Kasahara. A New Traitor Tracing. *EICE*, E85-A:481–484, 2002.

[Nao03] Moni Naor. On Cryptographic Assumptions and Challenges. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *LNCS*. Springer Berlin / Heidelberg, 2003.

[Nat06] National Institute Of Standards And Technology (NIST). *FIPS Publication 180-4: Secure Hash Standard (SHS)*, 2006.

[Nev08] Gregory Neven. A simple transitive signature scheme for directed trees. *Theoretical Computer Science*, 396(1-3):277–282, May 2008.

[Ngu05] Lan Nguyen. Accumulators from Bilinear Pairings and Applications. In Alfred Menezes, editor, *CT-RSA*, volume 3376 of *LNCS*, pages 275–292. Springer Berlin / Heidelberg, 2005.

[NR05] M Naor and G N Rothblum. The Complexity of Online Memory Checking. *IEEE Symposium on Foundations of Computer Science*, pages:573–584, 2005.

[Nyb96a] Kaisa Nyberg. Commutativity in cryptography. In *First International Workshop in Functional Analysis at at Trier University*, pages 331–342. Walter Gruyter, 1996.

[Nyb96b] Kaisa Nyberg. Fast Accumulated Hashing. In *Third International Workshop on Fast Software Encryption*, pages 83–87. Springer-Verlag, 1996.

[ope] OpenSSL Project. `www.openssl.org` (last access 5/12/2012).

[Pai99] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In Jacques Stern, editor, *EUROCRYPT*, volume 1592 of *LNCS*, pages 223–238. Springer Berlin / Heidelberg, 1999.

[Ped91] Torben Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In J. Feigenbaum, editor, *CRYPTO*, volume 576 of *LNCS*, pages 129–140. Springer Berlin / Heidelberg, 1991.

[Pol74] J. M. Pollard. Theorems on factorization and primality testing. *Mathematical Proceedings of the Cambridge Philosophical Society*, 76(03):521–528, November 1974.

[PS98] Sarvar Patel and Ganapathy S. Sundaram. An efficient discrete log pseudo random generator. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *LNCS*. Springer-Verlag, 1998.

[PTT08] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *CCS*, pages 437–448. ACM Press, 2008.

[PTT10] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal Authenticated Data Structures with Multilinear Forms. In Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors, *Pairing*, volume 6487 of *LNCS*, pages 246–264. Springer Berlin / Heidelberg, 2010.

[RBO89] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual*

ACM symposium on Theory of computing - STOC '89, pages 73–85. ACM Press, February 1989.

[Rog06] Phillip Rogaway. Formalizing human ignorance. In Phong Q. Nguyen, editor, *VIETCRYPT*, volume 4341 of *LNCS*, pages 211–228. Springer Berlin /Heidelberg, September 2006.

[San99] Tomas Sander. Efficient Accumulators without Trapdoor. In *ICICS '99*, volume 1726, pages 252–262, 1999.

[Sch98] C.P. Schnorr. Security of Almost ALL Discrete Log Bits. *Electronic Colloquium on Computational Complexity*, 1998.

[Sho04] Victor Shoup. *A Computational Introduction to Number Therory and Algebra*. 2004.

[SSM05] 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.

[TT02] Roberto Tamassia and Nikos Triandopoulos. On the Cost of Authenticated Data Structures. Technical report, Center for Geometric Computing, Brown University, 2002.

[TX03] Gene Tsudik and Shouhuai Xu. Accumulating Composites and Improved Group Signing. In Chi-Sung Laih, editor, *ASIACRYPT*, LNCS, pages 269–286. Springer Berlin / Heidelberg, 2003.

[Ver11] Damien Vergnaud. Efficient and Secure Generalized Pattern Matching via Fast Fourier Transform. In Abderrahmane Nitaj and David Pointcheval, editors, *AFRICACRYPT*, volume 6737 of *LNCS*, pages 41–58. Springer Berlin / Heidelberg, 2011.

[WWP07] Peishun Wang, Huaxiong Wang, and Josef Pieprzyk. A New Dynamic Accumulator for Batch Updates. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *Information and Communications Security*, volume 4861 of *LNCS*, pages 98–112. Springer Berlin / Heidelberg, 2007.

[WWP08] Peishun Wang, Huaxiong Wang, and Josef Pieprzyk. Improvement of a Dynamic Accumulator at ICICS 07 and Its Application in Multi-user Keyword-Based Retrieval on Encrypted Data. In *Asia-Pacific Conference on Services Computing. 2006 IEEE*, pages 1381–1386. IEEE Computer Society, 2008.

[XLL07] Rui Xue, Ninghui Li, and Jiangtao Li. A New Construction of Zero-Knowledge Sets Secure in Random Oracle Model. In *ISDPE*, pages 332–337. IEEE Computer Society, 2007.

[Xu09] Jia Xu. On Directed Transitive Signature. `http://eprint.iacr.org/2009/209` (last access 5/12/2012), 2009.

[Yi07] Xun Yi. Directed Transitive Signature Scheme. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *LNCS*, pages 129–144. Springer Berlin / Heidelberg, 2007.

[YSL08] Dae Hyun Yum, Jae Woo Seo, and Pil Joong Lee. Generalized Combinatoric Accumulator. *IEICE - Transactions on Information and Systems*, E91-D(5):1489–1491, 2008.

[ZSNS04] Fangguo Zhang, Reihaneh Safavi-Naini, and Willy Susilo. An Efficient Signature Scheme from Bilinear Pairings and Its Applications. In Feng Bao, Robert Deng, and Jianying Zhou, editors, *PKC*, volume 2947 of *LNCS*, pages 277–290. Springer Berlin / Heidelberg, 2004.