Strong Accumulators from Collision-Resistant Hashing *

Philippe Camacho[†] Alejandro Hevia[‡] Marcos Kiwi[§] Roberto Opazo[¶]

September 3, 2012

Abstract

Accumulator schemes were introduced in order to represent a large set of values as one short value called the *accumulator*. These schemes allow one to generate membership proofs, i.e. short witnesses that a certain value belongs to the set. In universal accumulator schemes, efficient proofs of non-membership can also be created. Li, Li and Xue [15], building on the work of Camenisch and Lysyanskaya [7], proposed an efficient accumulator scheme which relies on a trusted accumulator manager. Specifically, a manager that correctly performs accumulator updates.

In this work we introduce the notion of *strong universal accumulator schemes* which are similar in functionality to universal accumulator schemes, but do not assume the accumulator manager is trusted. We also formalize the security requirements for such schemes. We then give a simple construction of a strong universal accumulator scheme which is provably secure under the assumption that collision-resistant hash functions exist. The weaker requirement on the accumulator manager comes at a price; our scheme is less efficient than known universal accumulator schemes — the size of (non)membership witnesses is logarithmic in the size of the accumulated set in contrast to constant in the scheme of Camenisch and Lysyanskaya.

Finally, we show how to use strong universal accumulators to solve a problem of practical relevance, the so called e-Invoice Factoring Problem.

keywords: Strong Accumulators, Collision-resistant Hashing, e-Invoice.

1 Introduction

Accumulator schemes were introduced by Benaloh and De Mare [4]. These primitives allow to represent a potentially very large set by a short value called *accumulator*. Moreover, the accumulator together with a so called *witness* provides an efficiently verifiable proof that a given element belongs to the accumulated set.

Barić and Pfitzmann [2] refined the security definition of accumulator schemes, and introduced the concept of collision-free accumulators. This notion was further extended by Camenisch and Lysyanskaya [7] to a dynamic setting where updates (additions and deletions) to the accumulator are possible. They proposed a new construction and showed how to use it to efficiently implement membership revocation in group signatures, and anonymous credential systems. In particular, they show how to keep track of valid identities using an accumulator, so proving membership is done by arguing in zero-knowledge that a certain secret value was accumulated. For a thorough discussion of accumulators we refer the interested reader to the survey of Fazio and Nicolosi [10].

Li, Li and Xue [15] recently introduced the notion of *universal accumulators*, which not only allow efficient generation of membership, but also of non-membership proofs. Building on [7], Li et al. construct universal accumulator schemes and point out useful applications, e.g. proving that a certificate has not been revoked, or that a patient does not have a disease. Unfortunately, their construction inherits an undesirable property from Camenisch and Lysyanskaya's scheme; updates of the set (in partic-

*A preliminary version of this work appeared in proceedings of the 11th Information Security Conference, ISC'08, Lecture Notes in Computer Science 5222, pages 471-486, Springer-Verlag, 2008. Mr. Camacho gratefully acknowledges the support of CONICYT via FONDAP en Matemáticas Aplicadas. Mr. Hevia gratefully acknowledges the support of CONICYT via FONDECYT No. 1070332. Mr. Kiwi is supported by CONICYT via FONDECYT No. 1010689 and FONDAP-Basal in Applied Mathematics, and Millenium Nucleus Information and Coordination in Networks ICM/FIC P10-024F.

†Dept. of Computer Science, University of Chile, Blanco Encalada 2120, 3er piso, Santiago, Chile. pcamacho@dcc.uchile.cl.

‡Dept. of Computer Science, University of Chile, Blanco Encalada 2120, 3er piso, Santiago, Chile. ahevia@dcc.uchile.cl.

§Depto. Ing. Matemática & Ctr. Modelamiento Matemático UMI 2807, U. Chile. Web: www.dim.uchile.cl/~mkiwi.

¶CEO khipu.com, roberto@opazo.cl

ular, deletion of elements) require the accumulator manager to be trusted. This falls short of Benaloh and De Mare's initial goal: to provide membership proofs even if the accumulator manager is corrupted. In both [7] and [15], a malicious manager can compute witnesses for any element regardless of whether it was accumulated or not.

We propose a new accumulator scheme based on hash trees similar to those used in the design of digital time-stamping systems [4, 3]. 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 [14]; instead of associating values to the tree's leaves, we associated 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 $x_\alpha \prec x \prec x_\beta$, belongs to the tree but the pairs (x_α, x) and (x, x_β) do not.

The drawbacks of using a hash tree based scheme are twofold. First, the size of witnesses and the update time are logarithmic in the number of values accumulated. In contrast, witnesses and updates can be computed in constant time in RSA modular exponentiation based schemes like the ones of [7, 4, 2, 15]. We believe, nonetheless, that this problem may in fact not exist for reasonable set sizes — a claim that we will later support. The second drawback is the accumulator's manager storage space requirements which is linear in the number of elements. However, this is not an issue for the accumulator's users, since they only need logarithmic in the accumulated set size storage space.

Overall, the main advantages of our scheme in comparison with Li et al. [15] are: (1) the accumulator manager need not be trusted, and (2) since we only assume the existence of cryptographic hash functions 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 [9] while Strong-RSA is likely to be a stronger assumption than factoring [5].)

1.1 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 construct strong universal accumulators using only collision-resistant hash functions. Our construction has interesting properties of its own. As in [7, 15], we use auxiliary information to compute the (non)membership witness, but this information (called *memory*) need not 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 [15], 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.

1.2 Organization of the paper

In Section 2, we give some background definitions and formally introduce the notion of strong universal accumulator schemes. In Section 3, we describe our basic strong universal accumulator scheme and rigorously establish its security. In Section 4, we discuss the efficiency of the scheme in practice. The e-Invoice Factoring Problem is described in Section 5 where it is also shown how to solved it using strong universal accumulator schemes. In Section 6, we conclude with some comments.

2 Definitions and notations

Let $neg : \mathbb{N} \to \mathbb{R}$ denote a negligible function, that is, for every polynomial $p(\cdot)$ and any large enough integer n, $neg(n) < 1/p(n)$. Let also $||$ denote the operation of

concatenation between binary strings. If $R()$ is a randomized algorithm, we write $a \stackrel{R}{\leftarrow} R()$ to denote the process of choosing a according to the probability distribution induced by R. Let $\Pr\left[x_1 \stackrel{R}{\leftarrow} R_1(), \ldots, x_\ell \stackrel{R}{\leftarrow} R_\ell() : E\right]$ denote the probability of event E after the processes $x_1 \stackrel{R}{\leftarrow} R_1(), \ldots, x_\ell \stackrel{R}{\leftarrow} R_\ell()$ are performed in order. We also denote by $\langle R() \rangle$ the set of all possible values a returned by randomized algorithm R with positive probability. We distinguish between an *accumulator scheme* (the protocol, see below), its short representation or *accumulator value*, and its corresponding *accumulated set* X. For simplicity, however, we may use these terms indistinguishably when it is clear from the context.

SYNTAX. We formally define the syntax of a strong universal accumulator scheme (with memory). Our definition differs from that of Li et al. [15] as we consider an algorithm to verify if the accumulator value has been updated correctly (by adding or deleting a certain value), and we are not interested in hiding the order in which the elements are inserted into the accumulated set.

Definition 1 (Strong Universal Accumulators with Memory) *Let M be a set of values. A strong universal accumulator scheme (with memory) for M is a tuple $\mathfrak{A} =$ (Setup, Witness, CheckWitness, Update, CheckUpdate) where*

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

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

- CheckWitness($\kappa, x, w, \mathfrak{Acc}$) *is a randomized algorithm which on input a value $x \in M$, a witness w and the accumulator value $\mathfrak{Acc} \in Y$ 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.*

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

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

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

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 Setup, Witness, and $\mathsf{Update}_{\mathtt{op}}$ are executed by a manager and CheckWitness and CheckUpdate by the other participants of the multi-party protocol.

SECURITY. Universal accumulators as defined in [15] 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 $x \in M$. As mentioned there, this is equivalent to saying that given $X \subseteq M$ it is impossible to find $x \in X$ that has a valid non-membership witness or to find $x \in M \backslash X$ 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 therefore define what it means for an accumulator to represent a set, and then the security conditions that guarantee that such representation is sound even under the presence of malicious (but computationally bounded) managers. Our presentation uses a formalism based on initialization procedures to accommodate protocols whose security may require some setup assumptions (eg. an initial round of generation of trusted system-wide parameters[1]).

Definition 2 *An accumulator value $\mathfrak{Acc}$ represents the set $X \subseteq M$ under initialization parameters κ, denoted by $\mathfrak{Acc} \stackrel{\kappa}{\Rightarrow} X$, if and only if there exists a sequence $\{(\mathfrak{Acc}_i, x_i, \mathfrak{m}_i, \mathtt{op}_i)\}_{1 \leq i \leq n}$, where $n = |X|$, $x_i \in M$ for $1 \leq i \leq n$, $\mathtt{op}_i \in \{\mathtt{add}, \mathtt{del}\}$, and $\mathfrak{Acc}_0 \in Y$ and $\mathfrak{m}_0$ are values such that*

- $X = \{x_i\}_{1 \leq i \leq n}$,
- $(\mathfrak{Acc}_0, \mathfrak{m}_0) \in \langle \mathsf{Setup}(\kappa) \rangle$,
- $(\mathfrak{Acc}_i, \mathfrak{m}_i, w_i) \in \langle \mathsf{Update}_{\mathtt{op}_i}(\kappa, x_i, \mathfrak{Acc}_{i-1}, \mathfrak{m}_{i-1}) \rangle$ *for all* $1 \leq i \leq n$.

If no such sequence exists $\mathfrak{Acc}$ does not represents set X, denoted by $\mathfrak{Acc} \stackrel{\kappa}{\not\Rightarrow} X$.

Note that, regardless of the choice of κ, there could be two different sequences that make $\mathfrak{Acc} \stackrel{\kappa}{\Rightarrow} X$. More importantly, even for fixed κ, the above definition does not imply that the represented set X is unique for a given accumulated value $\mathfrak{Acc}$. Our definition of security, however, will ensure that as long as each accumulator update operation is verified by a honest observer (running CheckUpdate), such *collisions* will not happen except with negligible probability. Before proving this, we need to introduce our main security notion for accumulators.

We adapt the security definition in [15] as follows. First, we let the adversary select not only the value x and the witness w but also the accumulated set $X \subseteq M$, the accumulator value $\mathfrak{Acc} \in Y$ and whether x belongs or not to X. We restrict the adversary so he must choose a pair $(\mathfrak{Acc}, X)$ for which there exists a sequence of valid addition operations (namely, $\mathsf{Update}_{\mathtt{add}}$ with values in X) that can produce an accumulated value $\mathfrak{Acc}$. 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 CheckUpdate algorithm. Finally, to capture most setup assumptions, we parameterize the security notion with the following notion:

Definition 3 *An oracle Ω is an initialization procedure if given a security parameter $k \in \mathbb{N}$ in unary it generates a parameter $\kappa = (\kappa_0, 1^k)$ where κ_0 is of length polynomial in k. 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 $\Omega(1^k) = 1^k$.

Definition 4 (Security of Strong Universal Accumulators with Memory) *Let $\mathfrak{A}$ be a strong universal accumulator scheme (with memory) for universe M, $k \in \mathbb{N}$ be a security parameter, and Ω be an initialization procedure. Then $\mathfrak{A}$ is secure under Ω if for every probabilistic polynomial-time adversary $\mathcal{A}(\mathtt{atk}, \cdot)$ with $\mathtt{atk} \in \{\mathtt{cons}, \mathtt{crt}, \mathtt{add}, \mathtt{del}\}$, the following conditions hold:*

- *(Consistency)*

$$Pr\left[\begin{array}{c} \kappa = (\kappa_0, 1^k) \stackrel{R}{\leftarrow} \Omega(1^k), \\ (x, w_1, w_2, X, \mathfrak{Acc}) \leftarrow \mathcal{A}(\mathtt{cons}, \kappa) : \mathfrak{Acc} \stackrel{\kappa}{\Rightarrow} X, \\ \mathsf{CheckWitness}(\kappa, x, w_1, \mathfrak{Acc}) = 1, \\ \mathsf{CheckWitness}(\kappa, x, w_2, \mathfrak{Acc}) = 0 \end{array}\right]$$

is $neg(k)$.

- *(Secure creation)*

$$Pr\left[\begin{array}{c} \kappa = (\kappa_0, 1^k) \stackrel{R}{\leftarrow} \Omega(1^k), \\ (w, \mathfrak{Acc}_{after}) \leftarrow \mathcal{A}(\mathtt{crt}, \kappa) : \mathfrak{Acc}_{after} \stackrel{\kappa}{\not\Rightarrow} \emptyset, \\ \mathsf{CheckUpdate}(\kappa, \bot, \bot, \mathfrak{Acc}_{after}, (w, \mathtt{crt})) = 1 \end{array}\right]$$

is $neg(k)$.

- *(Secure addition)*

$$Pr\left[\begin{array}{c} \kappa = (\kappa_0, 1^k) \stackrel{R}{\leftarrow} \Omega(1^k), \\ (\mathfrak{Acc}_{before}, X, \mathfrak{Acc}_{after}, x, w) \leftarrow \mathcal{A}(\mathtt{add}, \kappa) : \\ \mathfrak{Acc}_{before} \stackrel{\kappa}{\Rightarrow} X,\ \mathfrak{Acc}_{after} \stackrel{\kappa}{\not\Rightarrow} X \cup \{x\}, \\ \mathsf{CheckUpdate}(\kappa, x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, (w, \mathtt{add})) = 1 \end{array}\right]$$

is $neg(k)$.

[1]In fact, our main construction does require trusted selection of a random hash function.

- *(Secure deletion)*

$$Pr\left[\begin{array}{c}\kappa = (\kappa_0, 1^k) \stackrel{R}{\leftarrow} \Omega(1^k),\\ (\mathfrak{Acc}_{before}, X, \mathfrak{Acc}_{after}, x, w) \leftarrow \mathcal{A}(\mathtt{del}, \kappa):\\ \mathfrak{Acc}_{before} \stackrel{\kappa}{\Rightarrow} X,\ \mathfrak{Acc}_{after} \stackrel{\kappa}{\not\Rightarrow} X\backslash\{x\},\\ \mathsf{CheckUpdate}(\kappa, x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, (w, \mathtt{del})) = 1\end{array}\right]$$

is $neg(k)$.

The type of accumulators we consider in this work is not necessarily *quasi-commutative* [7, 15] 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 $\mathfrak{Acc}$.

At this point, we need to justify our claim that the accumulated set X represented by an accumulator value is unique with overwhelming probability as long as honest parties verify all accumulator updates.

Definition 5 *Let $\mathfrak{A}$ be a strong universal accumulator scheme for some universe M and initialization Ω, and $k \in \mathbb{N}$ a security parameter. Given an adversary $\mathcal{A}$ with oracle access, consider the following two phase experiment $Exp^{VUAcc}_{\mathfrak{A},\Omega,\mathcal{A}}$: First, on input the security parameter k, the experiment generates a system-wide parameter κ by invoking $\Omega(1^k)$. Second, on input κ, the adversary outputs a tuple $(w, \mathfrak{Acc})$ which is taken to represent the creation witness, and the accumulated value (respectively) corresponding to the accumulated set $X = \emptyset$. If* $\mathsf{CheckUpdate}(\kappa, \bot, \bot, \mathfrak{Acc}, w) \neq 1$ *then the experiment aborts. In the second phase, the adversary is allowed to submit as many queries to an oracle $O()$ as it wants and then stops. The oracle $O()$ is stateful with initial state $(\kappa, \mathfrak{Acc}, \mathfrak{m}, w)$. For each query of the form $(\mathtt{op}, x, \mathfrak{Acc}', w)$ where $\mathtt{op} \in \{\mathtt{add}, \mathtt{del}\}$, the oracle proceeds as follows: it first computes a bit $b \leftarrow \mathsf{CheckUpdate}(\kappa, x, \mathfrak{Acc}, \mathfrak{Acc}', w)$. Then, if $b = 1$, it sets $\mathfrak{Acc} \leftarrow \mathfrak{Acc}'$, and $X \leftarrow X \cup \{x\}$ if $\mathtt{op} = \mathtt{add}$ (similarly if $\mathtt{op} = \mathtt{del}$ it sets $X \leftarrow X \setminus \{x\}$). In the case $b = 0$ the oracle does not modify $\mathfrak{Acc}$ or X. In both cases, the oracle returns bit b as the answer to the adversary. We say that an $\mathfrak{A}$ is* verifiably updatable *under Ω if for each probabilistic polynomial-time adversary $\mathcal{A}$, after $\mathcal{A}$ stops (without the experiment ever aborting), it holds that $\mathfrak{Acc} \stackrel{\kappa}{\Rightarrow} X$ except with negligible probability in k. If $\mathfrak{Acc} \stackrel{\kappa}{\not\Rightarrow} X$ we say adversary $\mathcal{A}$* wins *the experiment.*

The following proposition shows that the accumulated set represented by any accumulator value is well defined (with overwhelming probability) if we use a secure strong universal accumulator scheme.

Proposition 6 *If a strong universal accumulator scheme $\mathfrak{A}$ is secure under initialization procedure Ω, then $\mathfrak{A}$ is verifiably updatable under Ω.*

Proof: Let $\mathcal{A}^*$ be a probabilistic polynomial-time adversary that wins $Exp^{VUAcc}_{\mathfrak{A},\Omega,\mathcal{A}}$ with non-negligible probability in k making at most $\mu = \mu(k)$ queries for some polynomial μ. First of all, the initial accumulator value $\mathfrak{Acc}$ must represent the empty set, since otherwise adversary $\mathcal{A}^*$ would contradict the secure creation property. Then, for some index $1 \leq i \leq \mu$ there must exist queries $(\mathtt{op}_{i-1}, x_{i-1}, \mathfrak{Acc}_{i-1}, w_{i-1})$ and $(\mathtt{op}_i, x_i, \mathfrak{Acc}_i, w_i)$ such that $\mathfrak{Acc}_{i-1}$ does represent some set X^* while $\mathfrak{Acc}_i$ represents neither $X^* \cup \{x_i\}$ nor $X^* \setminus \{x_i\}$. Clearly, the polynomial-time adversary that runs $\mathcal{A}^*$ - while simulating the oracle - up to the i-th query and then outputs $\mathfrak{Acc}_{i-1}$, X^*, $\mathfrak{Acc}_i$, x_i, w_i breaks the secure addition or secure deletion property of $\mathfrak{A}$. □

Our security definition (Definition 4) for the dynamic scenario (where addition and deletion of elements are allowed) differs from the one in [7] 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.

DYNAMIC ACCUMULATORS. The standard definition of dynamic accumulators (see for example the one in [7]) adds two requirements which so far we have not considered. First, it requires the existence of an additional efficient 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 for the accumulator updates. In practice, such dependency may be appropriate for many applications.

3 Our scheme

We assume that there exist 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 [8]. 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.

3.1 Preliminaries

Our scheme is inspired by time stamping systems like those described in [4, 3]. In these systems a document needs to be associated to a certain moment in time. The solution proposed there is to divide the time in periods (e.g. hours, days), and place each document as a leaf at the bottom of a binary tree (say, T) with other documents that belong to the same period of time, say t. Then the values associated to each pair of leaves with the same parent node are hashed in order to derive the value of the parent node. This process is repeated until the value v of the root node of the tree is computed. This value v is then published as a representative of the tree T for period t. Later, a given document m can be proven to belong to a certain period of time t by presenting a valid subtree of tree T corresponding to time period t that includes the document m.

We use the above approach to build an accumulator scheme that works for dynamic sets and also allows proofs of nonmembership. In this case, building a proof of nonmembership is somehow similar to the trick of Kocher (in [14]) — instead of storing elements of the set, 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, $x_\alpha < x < x_\beta$, such that a pair (x_α, x_β) is stored in the tree.

BASIC TOOLS. Our solution uses collision-resistant hash functions, which we formalize as families of functions. In practice we can use a well-known hash function like SHA-256, for example. We start recalling the standard notion of collision-resistant hash functions.

Definition 7 *A hash function family is a collection of functions* $\left\{\mathcal{H}_\tau : M \to Y\right\}_{\tau\in K}$ *where* M *and* Y *are sets of strings, and* K *and* Y *are non-empty sets.*

Definition 8 *Let* $\mathcal{H} = \left\{\mathcal{H}_\tau : M \to Y\right\}_{\tau\in K}$ *be a hash function family and* k *a security parameter, where* $|K| = |Y| = 2^k$*, Then,* $\mathcal{H}$ *is collision-resistant if and only if for every polynomial-time probabilistic algorithm* A *we have:*

$$Pr\left[\begin{array}{l}\tau \stackrel{R}{\leftarrow} K; (m,m') \leftarrow A(\tau,k) : \\ m \neq m',\ \mathcal{H}_\tau(m) = \mathcal{H}_\tau(m')\end{array}\right] = neg(k)$$

where $\tau \stackrel{R}{\leftarrow} K$ *means that* τ *is selected uniformly at random in the set of keys* K*.*

Often, we will view a mapping $\mathcal{H} : K \times M \to Y$ as the hash function family $\left\{\mathcal{H}_\tau : M \to Y\right\}_{\tau\in K}$ where $\mathcal{H}_\tau(\cdot) = \mathcal{H}(\tau,\cdot)$. Henceforth, M will be the set of all binary strings, and K and Y will be the set $\{0,1\}^k$, for a large enough security parameter $k \in \mathbb{N}$.

NOTATION FOR SETS. We assume the set X we want to accumulate is ordered and denote by x_i the i-th element of $X = \{x_1, x_2, ..., x_n\}$, $n \in \mathbb{N}$. Let $x_0 = -\infty$ and $x_{n+1} = +\infty$ two special elements such that $-\infty \prec x_j \prec +\infty$ for all $x_j \in X$, where $\preceq$ is the order relation on X (for example, the lexicographic order on bit strings) and $a \prec b$ if and only if $a \preceq b$ 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:

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

For sets A and B we denote their symmetric difference by $A \triangle B$.

LABELED TREES. Our proposal relies on labeled binary trees. The root node of a tree T will be denoted $root(T)$. The left subtree (respectively right subtree) rooted at the left (respectively right) child node of T will be denoted $Left(T)$ (respectively $Right(T)$). The node $root(T)$ is said to be the *parent* of $Left(T)$ and $Right(T)$. Each node N of T will be labeled by a string henceforth denoted $Label(N)$. Sometimes we identify the tree T with its root $N = root(T)$ and we write $Label(T)$ to denote $Label(root(T))$. As usual, a leaf of T corresponds to a node of T that has no children. If T consists of only one node, then we say that T has depth 0 and denote it as $depth(T) = 0$. Otherwise, let $depth(T) = 1 + \max\{depth(Left(T)), depth(Right(T))\}$. A tree T is *balanced* if $|depth(Left(T)) - depth(Right(T))| \leq 1$. It is a

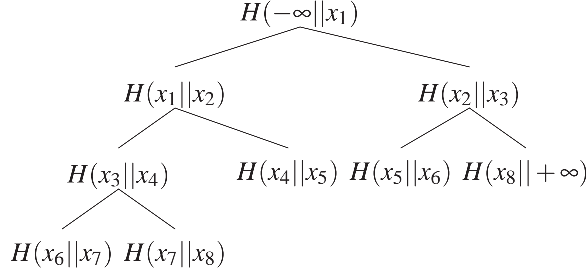


Figure 1: A tree model T of the set $X = \{x_1, \ldots, x_8\}$. Only node values are shown. Note that the place of the values in the tree is irrelevant.

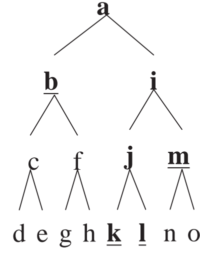


Figure 2: A tree and its minimal subtree (nodes with values in boldface) generated by the node of value j. Children of the nodes that are on the path from j to a are underlined.

well-known fact that a balanced tree with n nodes has maximum depth $O(\log(n))$.

MODEL OF $X = \{x_1, \ldots, x_n\}$ UNDER H. Informally, a model of X under H is a labeled balanced binary tree such that for every node N of T the label of N is of the form $H(x_i||x_{i+1})$ plus some additional information linking $H(x_i||x_{i+1})$ and the labels of the roots of the left and right tree whose parent is N. Formally, the set $\{H(x_i||x_{i+1}) : 0 \leq i \leq n\}$ will be called the *base* of X under H. A balanced binary tree T is called a *model* of X under H if:

- For every node N in T there are strings Val_N and $Proof_N$, called node value and node proof respectively, such that $Label(N) = (Val_N; Proof_N)$.
- The base of X is $\{Val_N : N \text{ is a node of } T\}$.
- T has $n+1$ nodes.
- $Proof_N = H(Val_N||Proof_{Left(N)}||Proof_{Right(N)})$ for every node N of T (where $Proof_{Nil}$ corresponds to the empty string).

If N is the root node of tree T, we abuse the notation and write $Proof_T$ instead of $Proof_N$. The collection of node values of T will be denoted by $\mathcal{V}(T)$. Figure 1 depicts a toy example of a model of a set.

MINIMAL SUBTREES GENERATED BY A SET. Let T be a labeled binary tree. We say that $\mathcal{V} \subseteq \mathcal{V}(T)$ *generates a minimal subtree U of T* if U 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 2 illustrates the concept of minimal subtree. If U is generated by a singleton $\{S\}$, then we say that U is generated by S.

FORGING BINARY TREES. We claim that for a set X and H a uniformly chosen hash function from a collision-resistance hash function family, given a model T of X under H it is computationally hard to distill a new labeled tree T' with the same root value but different values elsewhere. Formally, we claim the following result.

Proposition 9 *Let $\mathcal{H}: K \times M \to Y$ be a collision-resistant hash function family and H a uniformly chosen function in $\mathcal{H}$. Let $X \subseteq M$ be an adversarially-chosen polynomial size set (on the security parameter k), and T be a model of X under H. Then, given T, no adversary can efficiently compute a labeled binary tree T' and a value V such that $V \in \mathcal{V}(T') \triangle \mathcal{V}(T)$ and $Proof_{T'} = Proof_T$, except with negligible probability.*

Proof: Let A be a polynomial-time stateful adversary which works in two phases. First, on input the security parameter and a hash function $H \in \mathcal{H}$, A outputs a set $X \subseteq M$ of size polynomial on k. Then, given a model T for X under H, it outputs a labeled binary tree T' and a value V satisfying the conditions of the proposition. Since $Proof_{T'} = Proof_T$ and value V is in $\mathcal{V}(T')$ but not in $\mathcal{V}(T)$ there must exist a node N' in T' and a node N in T such that $Proof_{N'} = H(Val_{N'}||Proof_{Left(N')}||Proof_{Right(N')})$ and $Proof_N = H(Val_N||Proof_{Left(N)}||Proof_{Right(N)})$ are equal but $Val_N||Proof_{Left(N)}||Proof_{Right(N)} \neq Val_{N'}||Proof_{Left(N')}||Proof_{Right(N')}$. Nodes N and N' can be found efficiently by simple traversal of both trees in some fixed order.

Now, let B be an adversary that is given a uniformly selected at random collision-resistant hash function $H \in \mathcal{H}$.

Adversary B first queries A to obtain a set X which it uses to build a model T for X under H. Then, B runs A as a subroutine to obtain another labeled binary tree T' and a value V such that $Proof_T = Proof_{T'}$ and $V \in \mathcal{V}(T') \triangle \mathcal{V}(T)$. Finally, using the procedure mentioned above, B finds a collision for H. □

3.2 A Strong Universal Accumulator with Memory using Hash Trees

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 model of the accumulated set $X = \{x_1, x_2, ..., x_n\}$ under H. The model T of 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 such that $V_N = (x_\alpha, x_\beta)$ where $x = x_\alpha$ or $x = x_\beta$, and (2) to demonstrate that $x \notin X$, the accumulator proves that there is a node N of T such that $V_N = (x_\alpha, x_\beta)$ where $x_\alpha \prec x \prec x_\beta$. When adding or deleting elements from X, the accumulator manager needs to update T 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 $\Omega(1^k)$ which includes picking a hash function H uniformly at random from the family $\mathcal{H}$ (say by computing a random index $i \in K$ where $|K| = k$ and then setting $H = H_i$). Of course, such assumption can also be instantiated with an ephemeral trusted third party running Ω, or alternative using standard multiparty computation techniques among all participants, including the accumulator manager. Moreover, a common heuristic to avoid interaction is to simply pick $H =$SHA-256 [16], for example.

A detailed description of the proposed scheme follows.

THE CONSTRUCTION. Let $k = 2^t \in \mathbb{N}$ be the security parameter and let $X = \{x_1, x_2, ..., x_n\}$ be a subset of $M = \{0,1\}^t$. We define the accumulator scheme $\mathsf{HashAcc}$ below.

- $\mathsf{Setup}(\kappa)$: The algorithm starts by setting X equal to the empty set. Then, it extracts the security parameter k and the description (index) of the hash function $H \in \mathcal{H}$ from κ. The algorithm then sets $\mathfrak{m}$ to be the following model of X: a tree with a single root node N with value $Val_N = H(-\infty || +\infty)$ and the accumulator is initialized to $Proof_N = H(Val_N||\varepsilon||\varepsilon)$ where ε is the empty string. Finally, the algorithm sets the creation witness $w_{\texttt{crt}}$ to $(\mathfrak{m}, \texttt{crt})$, where $\texttt{crt}$ is a fixed label.

- $\mathsf{Witness}(\kappa, x, \mathfrak{m})$: On input $x \in M$ and memory $\mathfrak{m}$, it computes the witness $w = (w_1, w_2)$ 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 $x_\alpha \prec x \prec x_\beta$. Finally, it sets w_2 as the minimal subtree of $\mathfrak{m}$ generated by the value $H(x_\alpha||x_\beta)$.

- $\mathsf{CheckWitness}(\kappa, x, w, \mathfrak{Acc})$: On input $x \in M$ and witness $w = ((x', x''), U)$ where U is purportedly a minimal subtree of the memory value $\mathfrak{m}$ associated to the accumulator value $\mathfrak{Acc}$, it first checks if the following conditions hold: (1) $Proof_U = \mathfrak{Acc}$, (2) $H(x'||x'') \in \mathcal{V}(U)$, (3) ($x = x'$ or $x = x''$), and (3') ($x' \prec x \prec x''$). The algorithm outputs 1 if conditions (1), (2), and (3) hold; it outputs 0 if (1), (2), and (3') hold. Otherwise, it outputs $\perp$.

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

  If $\texttt{op} = \texttt{add}$ and $x \notin X$, the algorithm adds x into X by modifying $\mathfrak{m}_{before}$ as follows:

  1. It replaces the value $H(x_\alpha||x_\beta)$ from the appropriate node in $\mathfrak{m}_{before}$ (where $x_\alpha \prec x \prec x_\beta$) by the value $H(x_\alpha||x)$.
  2. It augments the tree $\mathfrak{m}_{before}$ adding a new leaf N of value $H(x||x_\beta)$ so the resulting tree $\mathfrak{m}_{after}$ is a balanced tree. Let $V_{Par(N)}$ be the (parent) node where N is attached as a leaf.

  The resulting tree is denoted $\mathfrak{m}_{after}$. Figure 3 illustrates the process of inserting an element into $\mathfrak{m}_{before}$.

  Once tree $\mathfrak{m}_{after}$ is built, the new accumulator is simply the value of the root of the tree, namely $\mathfrak{Acc}_{after} = Proof_{\mathfrak{m}_{after}}$. The witness $w_{add} = (\texttt{add}, U_{add,1}, U_{add,2})$ 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 $\{H(x_\alpha||x_\beta), Val_{V_{Par(N)}}\}$, and,

  – $U_{add,2}$ corresponds to the minimal subtree of $\mathfrak{m}_{after}$ generated by $\{H(x_\alpha||x), H(x||x_\beta)\}$.

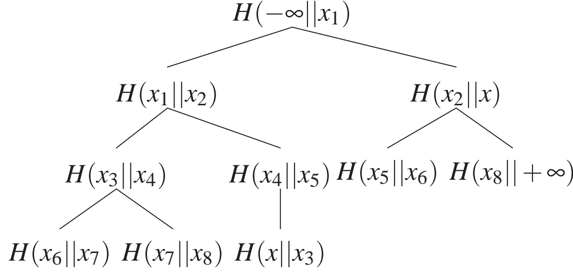
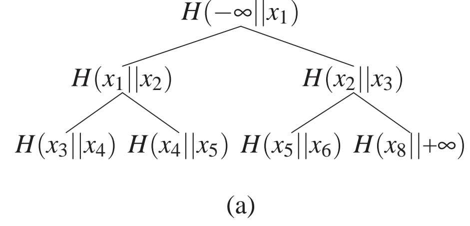
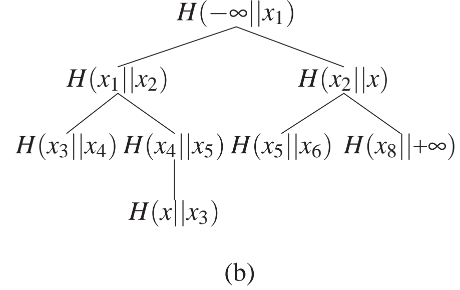


Figure 3: Inserting x into the tree of Figure 1 where $x_2 \prec x \prec x_3$.



(a)



(b)

Figure 4: (a) The minimal subtree of the tree shown in Figure 1 and generated by $\{H(x_2||x_3), H(x_4, x_5)\}$. (b) The minimal subtree of the tree shown in Figure 3 and generated by $\{H(x_2||x), H(x||x_3)\}$.

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 V_α and V_β be those nodes, and let $H(x_\alpha||x)$ and $H(x||x_\beta)$ be their respective values, for some $x_\alpha \prec x \prec x_\beta$. The goal is to remove these nodes and replace them with a new node with value $H(x_\alpha||x_\beta)$ in a way that the derived tree is still balanced. This is done by first replacing V_α with the single node with value $H(x_\alpha||x_\beta)$, and then replacing V_β with a leaf node L (for example, the rightmost leaf 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 $\mathfrak{Acc}_{after} = Proof_{\mathfrak{m}_{after}}$. The witness $w_{del} = (\mathtt{del}, U_{del,1}, w_{del,2}, U_{del,3})$ is then computed as follows:

- $U_{del,1}$ corresponds to the minimal subtree of $\mathfrak{m}_{before}$ generated by the set $\{H(x_\alpha||x), H(x||x_\beta), Val_L\}$,
- $w_{del,2}$ is the pair $(x_\alpha||x_\beta)$ such that $x_\alpha \prec x \prec x_\beta$, and
- $U_{del,3}$ is the minimal subtree of $\mathfrak{m}_{after}$ generated by $H(x_\alpha||x_\beta)$.

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

- $\mathsf{CheckUpdate}(\kappa, x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w_{\mathtt{op}})$: On input an element $x \in M$, two accumulator values $\mathfrak{Acc}_{before}$, $\mathfrak{Acc}_{after}$, 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 $\mathfrak{Acc}_{after} = H(H(-\infty|| +\infty), \varepsilon, \varepsilon)$ and w is a model of the empty set under H. If $w = (\mathtt{add}, U_1, U_2)$, then the algorithm returns 1 provided that:
  - U_2 is a tree obtained by adding a leaf to U_1,
  - Except for the node of value $H(x_\alpha||x_\beta)$ (for $x_\alpha \prec x \prec x_\beta$) all nodes which are common to U_1 and U_2 have the same value in either one of the trees,
  - $Proof_{U_1} = \mathfrak{Acc}_{before}$ and $Proof_{U_2} = \mathfrak{Acc}_{after}$, and
  - $H(x_\alpha||x), H(x||x_\beta) \in \mathcal{V}(U_2)$.

  Otherwise, it outputs 0. We omit the case $w = (\mathtt{del}, U_1, w_2, U_3)$ which is similar.

SECURITY. We now prove that the scheme $\mathsf{HashAcc}$ of the previous section is secure under Definition 4.

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 $x_\alpha \prec x \prec x_\beta$ and let $H(x_\alpha||x_\beta)$ be the value of a node N in $\mathfrak{m}$. By replacing node N with the node of value $H(x_\alpha||x)$ and adding the node of value $H(x||x_\beta)$, we clearly obtain a set of values $\{H(x_i||x_{i+1}), 0 \leq i \leq n+1\}$ that corresponds to the successive intervals of the set $X \cup \{x\}$ (where $n = |X|$).

Intuitively, 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.

Definition 10 *Let $\mathcal{H}: K \times M \to Y$ be a collision-resistant hash function family. Let $\Omega_{\mathcal{H}}$ be the initialization procedure that on input k in unary returns $\kappa = (H, 1^k)$ where H is chosen uniformly at random from the family $\mathcal{H}$.*

Theorem 11 *The accumulator scheme* HashAcc *is a secure strong universal accumulator scheme (with memory) under $\Omega_{\mathcal{H}}$.*

Proof: We need to prove the properties *Consistency*, *Secure creation*, *Secure addition*, and *Secure deletion*.

- *(Consistency)* First, we note that $\mathfrak{Acc} \overset{\kappa}{\Rightarrow} X$ implies that there exists a memory $\mathfrak{m}$ which is a model of X under H. Let us now suppose that there is an adversary A that can compute a value x and two witnesses w_1, w_2 such that $\mathsf{CheckWitness}(\kappa, x, w_1, \mathfrak{Acc}) = 1$ and $\mathsf{CheckWitness}(\kappa, x, w_2, \mathfrak{Acc}) = 0$. We assume without lost of generality that $x \in X$. Any such adversary A is in fact able to find x_α and x_β, $x_\alpha \prec x \prec x_\beta$, such that $H(x_\alpha || x_\beta)$ belongs to $\mathcal{V}(\mathfrak{m})$. Since $\mathfrak{m}$ is a model for X, by Proposition 9, this adversary will only succeed with negligible probability. The argument for $x \notin X$ is analogous.

- *(Secure Creation)* Let $\mathcal{A}$ be a probabilistic polynomial-time adversary (playing the role of an accumulator manager) that outputs $(w_0, \mathfrak{Acc}_0)$ where $\mathfrak{Acc}_0 \overset{\kappa}{\not\Rightarrow} \emptyset$ but $\mathsf{CheckUpdate}(\kappa, \perp, \perp, \mathfrak{Acc}_0, (w_0, \mathtt{crt})) = 1$. By definition of Setup, one has that w_0 has the structure of a model (i.e. must be a labeled balanced binary tree plus some additional information). Following the same line of argument as that of Proposition 9 one reaches a contradiction to the collision-resistant assumption on $\mathcal{H}$.

- *(Secure Addition)* Consider the case where the update is the addition of a value x such that $x_\alpha \prec x \prec x_\beta$ and $H(x_\alpha, x_\beta)$ belongs to the base of X, where $\mathfrak{Acc}_{before} \overset{\kappa}{\Rightarrow} X$. Assume that $\mathsf{CheckUpdate}(\kappa, x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w) = 1$ where both $w = (\mathtt{add}, U_{before}, U_{after})$ and x are arbitrarily chosen by the adversary, and $\mathfrak{Acc}_{after} \overset{\kappa}{\not\Rightarrow} X \cup \{x\}$. Then, for two elements $u, v \in M$ the adversary can effectively build a tree $S^* = U_{after}$ containing a value $H(u||v)$ that does not belong to $(\mathcal{V}(\mathfrak{m}_{before}) \cup \{H(x_\alpha||x), H(x||x_\beta)\}) \setminus \{H(x_\alpha||x_\beta)\} = \mathcal{V}(\mathfrak{m}_{after})$ and such that in addition $Proof_{S^*} = Proof_{U_{after}} = \mathfrak{Acc}_{after} = Proof_{\mathfrak{m}_{after}}$. This contradicts Proposition 9.

- *(Secure Deletion)* This case is similar to the addition of an element.

□

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

Theorem 12 *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 Efficiency of our scheme and comparison with previous proposals

Our solution is theoretically less efficient than the scheme proposed in [15]. 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 [15, 7]. Table 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 [17]) on a personal computer. Notice that RSA timings were obtained using signing operations, as in the scheme proposed in [15] where exponents

n	RSA-512	RSA-1024	RSA-2048	SHA-256	SHA-512
2^3	0.85	4.23	25.00	0.37	1.42
2^{10}	0.85	4.23	25.00	1.22	4.75
2^{20}	0.85	4.23	25.00	2.44	9.50
2^{30}	0.85	4.23	25.00	3.66	14.24

Table 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.

Scheme	Strong	Dynamic	Witness size
Benaloh et al. [4]	Yes	No	$O(k)$
Barić et al. [2]	Yes	No	$O(k)$
Camenish et al. [7]	No	Yes	$O(k)$
Li et al. [15]	No	Yes	$O(k)$
This work	Yes	Partially†	$O(k \log n)$

Table 2: Comparison of properties of previous and our scheme, where n is again the number of accumulated elements and k is the security parameter. (In all scheme's the accumulator size is $O(k)$). † Our solution allows dynamic addition and deletion of elements but no witness update.

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 $4 \times 2\log(n)$ hash computations, where n is the number of accumulated elements, given that at most four branches of the Merkle tree used in our construction (three for $w_{del,1}$ and one for $w_{del,3}$, see Section 3.2) will have to be recomputed in the case of deletions. We now explain the relevance of Table 1. For clarity's sake, we focus on the efficiency of witness generation. In [7, 15, 4, 2] 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 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 n-th witness is given by the columns of Table 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 2 compares the functionality provided and sizes of parameters appearing in aforementioned schemes and our solution.

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 [7, 15] not only require a trusted party – the accumulator manager herself – but also that such trust entity be available for as long as the accumulator is active. This seems unavoidable since managers require a trapdoor for the pseudo-collision free function in order to (efficiently) delete elements from the accumulator.

In practice, trusted initialization can efficiently be implemented using standard secure multiparty computation techniques. For our protocol, we only need to generate a hash function index, that is, a k-bit uniformly distributed random string. This can be done using standard coin tossing algorithms [18] (or more practical variants [13]) if a majority of participants during the initialization is honest. Alternatively, we could cast our results in the *human ignorance* setting proposed by Rogaway [19]. In that case, it would suffice to take Ω as the identity function and make the reductions behind the proof of Theorem 11 more explicit (that is, detailing how new collision-finding adversaries are built from the given protocol adversaries). Note that all reductions in this paper are in fact constructive.

NONTRIVIALITY OF THE STRONG PROPERTY: Both in the construction of Camenish et al. [7] as well as the one by Li et al. [15], 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.

Note that both these schemes [7, 15] are verifiably updatable. Moreover, witness computation is deterministic in the aforementioned schemes, depending only on the previous and current value of the accumulator at witness generation time. However, these conditions are not enough for these schemes to be strong. In Camenish et

al. scheme [7], a malicious manager can add an element $x = ab$ (for some a,b) and then prove either a or b belong to the accumulated set. Similarly, in the scheme by Li et al. [15], a malicious manager can easily compute non-membership witnesses for elements even if they are in the accumulated set. In this case though, a witness verification algorithm could detect the forgery if the list of all elements currently in the accumulated set is available. Yet, this is not a requirement of the protocol and would likely make such variant inefficient.

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

5 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 with 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 [1]. 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 [11]. The reader is referred to the website of the International Factors Group [12] 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 ex-

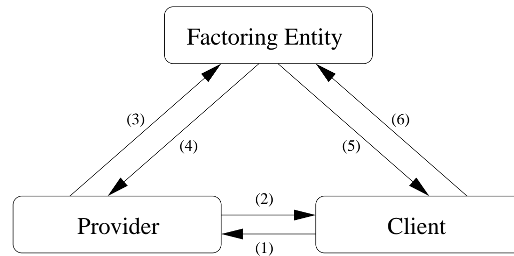


Figure 5: Steps of a factoring operation.

ploits 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 5:

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 [20]. 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 maintain confidential the profile of their clients.

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 $\mathcal{F} = (FE_1, \ldots, FE_n; FA)$ involves $n \geq 2$ participants $FE_1, \ldots, FE_n$ 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 $k \in \mathbb{N}$ be a security parameter, Ω an initialization procedure, and let $\mathcal{F} = (FE_1, \ldots, FE_n; FA)$ be a factoring protocol for $n = p(k) \geq 2$ where p is a polynomial. Consider an experiment, denoted $Exp^{\mathrm{fac}}_{\mathcal{F},\Omega,\mathcal{A}}(k)$, 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 $i \neq j$ the adversary obtains a signature from honest factor entities FE_i and FE_j for the same message $Inv||0$, or (2) the adversary obtains a signature from honest factor entity FE_i for a message $Inv||1$ such that no honest factor entity has previously generated a signature of $Inv||0$.

Definition 13 (Security of a factoring protocol) *Let $k \in \mathbb{N}$ be a security parameter and $n = p(k) \geq 2$ for some polynomial p. We say that a factoring protocol $\mathcal{F}$ is secure under initialization procedure Ω if $\Pr\left[Exp^{fac}_{\mathcal{F},\Omega,\mathcal{A}}(k)\right] = neg(k)$ for every probabilistic polynomial-time adversary $\mathcal{A}$.*

5.1 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 $FE_1, \ldots, FE_n$ and the factoring authority *FA*. In this **Base Protocol**, we assume moreover that *FA* is trustworthy. Afterward, 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 all messages are digitally signed by the entity that sends them. Moreover, we assume the factoring authority *FA* and factor entities interact through a bulletin board (as implemented in other cryptographic protocols, e.g. [8]).

The **Base Protocol** is illustrated in Figure 6 and its phases are described next:

Base Protocol

1. P sends an e-invoice *Inv* to C.
2. C sends a signed acknowledgment of receipt of the e-invoice $Ack = Sign_C(Inv)$.
3. P sends the signed message $Inv||Ack$ to the FE_i of his choice.
4. FE_i sends the signed message $x = H(Inv)$ 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 signed 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 signed message $Inv||Stat$ to P.

DISCUSSION. Note that during Step 2 a receipt is signed 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.[2] The reason for including the identifier

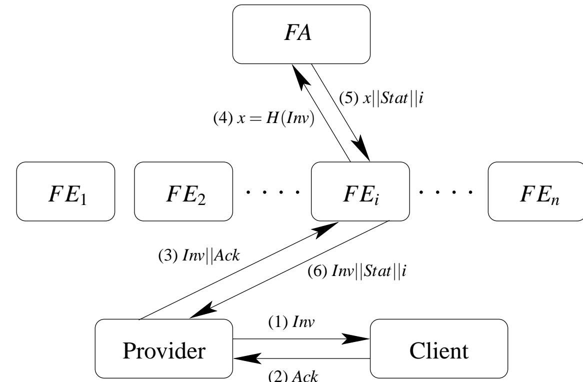


Figure 6: **Base Protocol** (trustworthy *FA*).

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 signature on $x||Stat||i$ certifies that an invoice with digest x either is or is not present in *FA*'s database. Note also that FE_i's signature on $Inv||0$ 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 now 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 ACC.⟨algorithm name⟩.

[2] This is straightforward by using perfectly one-way hash functions [6] for H in Step 4.

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

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

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

- $\mathsf{Add}(\kappa, x, \mathfrak{Acc}_{before}, \mathfrak{m}_{before})$: the factoring authority runs $\mathsf{ACC.Update}_{\mathsf{add}}(\kappa, x, \mathfrak{Acc}_{before}, \mathfrak{m}_{before})$, returns $(\mathfrak{Acc}_{after}, \mathfrak{m}_{after})$ and w_{add}. Then, it publishes in the bulletin board $(\mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w_{add})$.

- $\mathsf{Check}_{add}(\kappa, x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w_{add})$: the factor entity that wants to check whether the element x was correctly added to the accumulated set decides (non)membership of x based on the result returned by $\mathsf{ACC.CheckUpdate}(\kappa, x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w_{add})$.

Below we describe the refinement of Step 5 of the **Base Protocol** which corresponds to a factoring protocol. As in Section 3 we assume the availability of a public broadcast channel. We also assume that when **Base Protocol** starts the procedure $\mathsf{Setup}(1^k)$ is invoked, where k is the security parameter, thus generating the system-wide parameter κ.

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).

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.

Refinement of Step 5

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

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

5.2 The following verifications are performed:

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

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

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 5.1, and no message $x||(Stat, \mathfrak{Acc}_{after}, \mathfrak{m}_{after}, w_{add})||j$ with $j \neq i$ is published before FA updates the accumulator value to $\mathfrak{Acc}_{after}$ (and the memory state to $\mathfrak{m}_{after}$),[3] 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.

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 14 *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 obtains a signature from honest factor entities FE_i and FE_j for the same message $Inv||0$, or (2) the adversary obtains a signature from honest factor entity FE_i

[3] 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 a message $Inv||1$ such that no honest factor entity has previously generated a signature of $Inv||0$. If situation (1) occurs, the verification in Step 5.2 of the factoring protocol guarantees that only one message of the type $H(Inv)||(Stat, \mathfrak{Acc}_{after}, \mathfrak{m}_{after}, w_{add})||l$ can be published between two consecutive changes of the accumulated value, the attacker needs to be able to find a witness of non-membership for $H(Inv)$, although $H(Inv)$ has already been accumulated. Since $\mathcal{F}_{\mathsf{ACC}}$ is secure, this can only happen with a negligible probability. If situation (2) occurs, then for the verification in Step 5.2 to succeed with non-negligible probability the attacker needs to be able to find a witness of membership for $H(Inv)$, although $H(Inv)$ has not been previously accumulated. Since $\mathcal{F}_{\mathsf{ACC}}$ is secure, this can not happen. □

Theorem 11 and Proposition 14 immediately yield,

Corollary 15 *The factoring protocol $\mathcal{F}_{\mathsf{HashAcc}}$ is a secure factoring protocol under $\Omega_{\mathcal{H}}$.*

6 Conclusion

We introduced the notion of strong universal accumulator scheme which provide almost the same functionality as do the universal accumulator schemes defined in [15], 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

References

[1] Asociación Chilena de Factoring (ACHEF). Estadísticas. (`http://www.achef.cl/` [June 28, 2011]).

[2] N. Barić and B. Pfitzmann. Collision-free accumulators and fail-stop signed scheme without trees. In *Advances in Cryptology - Proceedings of Eurocrypt '97*, volume 1233 of *LNCS*, pages 480–494. Springer–Verlag, 1997.

[3] D. Bayer, S. Haber, and W. S. Stornetta. Improving the efficiency and reliability of digital time-stamping. In *Sequences II: Methods in Communication, Security, and Computer Science*, pages 329–334. Springer–Verlag, 1993.

[4] J. Benaloh and M. De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology - Proceedings of Eurocrypt '93*, volume 765 of *LNCS*, pages 274–285. Springer–Verlag, 1993.

[5] D. Boneh and R. Venkatesan. Breaking RSA may not be equivalent to factoring. In *Advances in Cryptology - Proceedings of Eurocrypt '98*, volume 1233 of *LNCS*, pages 59–71. Springer–Verlag, 1998.

[6] R. Canetti, D. Micciancio, and O. Reingold. Perfectly one-way probabilistic hash functions. In *30th Annual Symposium on the Theory of Computing*, pages 131–140, ACM Press, 1998.

[7] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Advances in Cryptology - Proceedings of Crypto '02*, volume 2442 of *LNCS*, pages 61–76, Springer–Verlag. 2002.

[8] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Advances in Cryptology - Proceedings of Eurocrypt '97*, volume 1233 of *LNCS*, pages 103–118. Springer–Verlag, 1997.

[9] I. Damgård. Collision free hash functions and public key signature schemes. In *Advances in Cryptology, Proceedings of Eurocrypt '87*, volume 308 of *LNCS*, pages 203–216. Springer–Verlag, 1988.

[10] N. Fazio and A. Nicolisi. Cryptographic accumulators: Definitions, constructions and applications, 2003. (`http://www.cs.nyu.edu/~nicolosi/papers/accumulators.ps` [June 19, 2008]).

[11] W.H. Hurd. Four Centuries of Factoring. Quarterly Journal of Economics 53(2):305–311, 1939

[12] International Factors Group (IFG). (`http://www.ifgroup.com/` [June 28, 2011]).

[13] A. Kate and I. Goldberg. Distributed Key Generation for the Internet. In *29th IEEE International Conference on Distributed Computing Systems*, June, pages 119–128. IEEE Press, 2009.

[14] P. C. Kocher. On certificate revocation and validation. In *Financial Cryptography*, volume 1465 of *LNCS*, pages 172–177. Springer–Verlag, 1998.

[15] J. Li, N. Li, and R. Xue. Universal accumulators with efficient nonmembership proofs. In *Proceedings of Applied Cryptography and Network Security - ACNS '07*, volume 4521 of *LNCS*, 2007.

[16] National Institute of Standards and Technology (NIST). *FIPS Publication 180: Secure Hash Standard (SHS)*, May 1993.

[17] OpenSSL Project. OpenSSL Package, June 2008. (`http://www.openssl.org` [June 19, 2008]).

[18] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In *21st Annual Symposium on the Theory of Computing*, pages 73–85, ACM Press, 1989.

[19] P. Rogaway. Formalizing Human Ignorance. In *Progress in Cryptology - Proceedings of Vietcrypt '06*, volume 4341 of *LNCS*, pages 211–228. Springer–Verlag, 2006.

[20] Servicio de Impuestos Internos. Información sobre factura electrónica. (`https://palena.sii.cl/dte/menu.html` [June 24, 2011]).