Strong Accumulators from Collision-Resistant Hashing

Philippe Camacho[1]⋆, Alejandro Hevia[1] ⋆⋆, Marcos Kiwi[2] ⋆⋆⋆, and Roberto Opazo[3]

[1] Dept. of Computer Science, University of Chile,
Blanco Encalada 2120, 3er piso, Santiago, Chile.
{pcamacho,ahevia}@dcc.uchile.cl
[2] Dept. Ing. Matemática & Ctr. de Modelamiento Matemático,
UMI 2807 U. Chile–CNRS
mkiwi@dim.uchile.cl
[3] CEO Acepta.com
roberto.opazo@acepta.com

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 [11], building on the work of Camenisch and Lysyanskaya [5], 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 practical concern, the so called e-Invoice Factoring Problem.

Key words: Accumulators, Collision-resistant Hashing, e-Invoice.

1 Introduction

Accumulator schemes were introduced by Benaloh and De Mare [3]. 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.

⋆ Gratefully acknowledges the support of CONICYT via FONDAP en Matemáticas Aplicadas.
⋆⋆ Gratefully acknowledges the support of CONICYT via FONDECYT No. 1070332.
⋆⋆⋆ Supported by CONICYT via FONDECYT No. 1010689 and FONDAP en Matemáticas Aplicadas.

Barić and Pfitzmann [1] refined the security definition of accumulator schemes, and introduced the concept of collision-free accumulators. This notion was further extended by Camenisch and Lysyanskaya [5] 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 [9].

Li, Li and Xue [11] recently introduced the notion of *universal accumulators*, which not only allow efficient generation of membership, but also of non-membership proofs. Building on [5], 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. However, their construction inherits an undesirable property from Camenisch and Lysyanskaya's scheme; updates of the set (addition and 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.

We propose a new accumulator scheme based on hash trees similar to those used in the design of digital timestamping systems [3, 2]. 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 Kocher [10] trick; 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 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 like the ones of [5, 3, 1, 11]. 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 to Li et al.'s [11] are: (1) the accumulator manager need not to 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 [7] while Strong-RSA is likely to be a stronger assumption than factoring [4].)

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 [5, 11], 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 [11], 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.

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. Section 5 briefly motivates the e-Invoice Factoring Problem. In Section 6, we conclude with some comments. Due to space restrictions the e-Invoice Factoring Problem is described in the full version of this paper where it is also shown how it can be solved using strong universal accumulator schemes.

2 Definitions and notations

Let $neg : \mathbb{N} \to \mathbb{N}$ 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 $a \overset{R}{\leftarrow} R()$ to denote 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. 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's 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. [11] 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 the input set $X \subseteq M$ is a tuple* $\mathfrak{A} = (\mathsf{Setup}, \mathsf{Witness}, \mathsf{Belongs}, \mathsf{Update}, \mathsf{CheckUpdate})$ *where*

- Setup *is a randomized algorithm which on input a security parameter $k \in \mathbb{N}$, outputs a public data structure $\mathfrak{m}_0$ (also called the memory), and returns an initial accumulator value $\mathfrak{Acc}_0$ in the set $Y = \{0,1\}^k$. Both the accumulator value $\mathfrak{Acc}$ and the memory $\mathfrak{m}$ will be typically held and updated by the accumulator manager.*
- $\mathsf{Witness}$ *is a randomized algorithm which 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 $x \notin X$.*
- $\mathsf{Belongs}$ *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.*
- $\mathsf{Update}_{\mathtt{op}}$ *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 $w_{\mathtt{op}}$.*
- $\mathsf{CheckUpdate}$ *is a randomized algorithm that takes as input a value $x \in M$, a pair of accumulator values $(\mathfrak{Acc}_{before}, \mathfrak{Acc}_{after})$, and an update witness w, 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 $b = 1$, w is deemed a valid witness that an 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 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 the memory $\mathfrak{m}$ is a public data structure which is computed from the set. Although public, this structure only needs to be maintained (stored) by the accumulator manager who updates the accumulator and generates membership and non-membership witnesses. In particular, the memory is *not* used to verify correct accumulator updates nor to check the validity of (non)membership witnesses.

Definition 2. *An accumulator value $\mathfrak{Acc}$ represents the set $X \subseteq M$, denoted by $\mathfrak{Acc} \Rightarrow X$, if and only if there exists a sequence $\{(\mathfrak{Acc}_i, x_i, \mathfrak{m}_i)\}_{1 \le i \le n}$, where $n = |X|$, and values $\mathfrak{Acc}_0, \mathfrak{m}_0$ where $x_i \in M$ for $1 \le i \le n$ and*

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

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

Note that this definition also considers sets that have been formed by successive addition and deletions of elements as there is always a sequence of only addition operations that leads to the same set.

SECURITY. Universal accumulators as defined in [11] satisfy a basic consistency property: it must be unfeasible to find both a valid membership witness and a valid nonmembership 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 nonmembership 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 adapt the security notion in [11] as follows. First, we let the adversary select not only the value x and the witness w but also the accumulated set $X \subset Y$, 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, $\mathsf{Update}_{\mathsf{add}}$ 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 $\mathsf{CheckUpdate}$ algorithm. Thus, security holds as long as it is unfeasible for the adversary to fool the $\mathsf{CheckUpdate}$ verification, namely given an accumulator value $\mathfrak{Acc}_{before}$, the adversary is unable to efficiently generate an accumulator value $\mathfrak{Acc}_{after}$, a set X, an input value x, and a valid update witness w for which $\mathfrak{Acc}_{before}$ actually represents set X and $\mathsf{CheckUpdate}(x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w) = 1$, but $\mathfrak{Acc}_{after} \not\Rightarrow X \cup \{x\}$ if w is an addition witness or $\mathfrak{Acc}_{after} \not\Rightarrow X \setminus \{x\}$ if w is an deletion witness.

Definition 3 (Security of Strong Universal Accumulators with Memory). *A strong universal accumulator with memory is secure if for every probabilistic polynomial-time adversary* $\mathcal{A}$ *the following conditions hold:*

- *(Consistency)*

$$Pr\left[\begin{array}{c}(x, w_1, w_2, X, \mathfrak{Acc}) \leftarrow \mathcal{A}(k);\\ \mathfrak{Acc} \Rightarrow X,\ \mathsf{Belongs}(x, w_1, \mathfrak{Acc}) = 1,\ \mathsf{Belongs}(x, w_2, \mathfrak{Acc}) = 0\end{array}\right] = neg(k)\,.$$

- *(Secure addition)*

$$Pr\left[\begin{array}{c}(\mathfrak{Acc}_{before}, X, \mathfrak{Acc}_{after}, x, w) \leftarrow \mathcal{A}(k):\\ \mathfrak{Acc}_{before} \Rightarrow X,\ \mathfrak{Acc}_{after} \not\Rightarrow X \cup \{x\},\\ \mathsf{CheckUpdate}(x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w) = 1\end{array}\right] = neg(k)\,.$$

- *(Secure deletion)*

$$Pr\left[\begin{array}{c}(\mathfrak{Acc}_{before}, X, \mathfrak{Acc}_{after}, x, w) \leftarrow \mathcal{A}(k):\\ \mathfrak{Acc}_{before} \Rightarrow X,\ \mathfrak{Acc}_{after} \not\Rightarrow X \backslash \{x\},\\ \mathsf{CheckUpdate}(x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w) = 1\end{array}\right] = neg(k)\,.$$

The type of accumulators we consider in this work is not necessarily *quasi-commutative* [5, 11] 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 $\mathfrak{Acc}$. Our only requirement is that the accumulated set X represented by any accumulator value is well defined. The following proposition shows this is so if we use a secure strong universal accumulator scheme. The proof is omitted due to space constraints.

Proposition 1. *Let $\mathfrak{A}$ a secure strong universal accumulator scheme, and $k \in \mathbb{N}$ a security parameter. Given any adversary $\mathcal{A}$, consider the experiment $Exp^{SUAcc}_{\mathfrak{A},\mathcal{A}}$ in which the adversary is allowed to submit as many queries to oracle $\mathcal{O}()$ as it wants and then stops. Oracle $\mathcal{O}()$ is stateful and operates as follows: on any first query, the oracle creates an empty set X', runs $\mathsf{Setup}(k)$ to obtain $(\mathfrak{Acc}', \mathfrak{m}')$ which it returns as the query answer. Then, for each subsequent query of the form $(x, \mathfrak{Acc}, w)$ the oracle computes $b \leftarrow \mathsf{CheckUpdate}(x, \mathfrak{Acc}', \mathfrak{Acc}, w)$, and if $b = 1$, it sets $\mathfrak{Acc}' \leftarrow \mathfrak{Acc}$, $X' \leftarrow X' \cup \{x\}$, and returns bit b as the answer to the oracle query. If $b = 0$ it does not modify $\mathfrak{Acc}$ or X' and it simply returns $\perp$. We say adversary $\mathcal{A}$ wins $Exp^{SUAcc}_{\mathfrak{A},\mathcal{A}}$ if after $\mathcal{A}$ stops, it holds that $\mathfrak{Acc} \not\Rightarrow X'$. Then, for every probabilistic polynomial time adversary $\mathcal{A}$, $\Pr\left[\mathcal{A} \text{ wins in } Exp^{SUAcc}_{\mathfrak{A},\mathcal{A}}\right]$ is negligible in k.*

Our security definition (Definition 3) for the dynamic scenario (where addition and deletion of elements are allowed) differs from the one in [5] where the adversary is only able to add and delete elements by querying the accumulator manager, who is uncorruptible. 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 [5]) 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 level of security required, 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 posted message. For a discussion on bulletin boards and an example of their use in another cryptographic protocol, the interested reader is referred to [6].

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 [3, 2]. 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 [10]) — 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.

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 4. *A hash-function family is a function $\mathcal{H} : K \times M \rightarrow Y$ where K and Y are non-empty sets and M and Y are sets of strings.*

Definition 5. *(Collision-Resistance) Let $\mathcal{H} : K \times M \rightarrow Y$ be a hash-function family. Let k be a security parameter, where $k = |K| = |Y|$. Then $\mathcal{H}$ is collision-resistant if and only if for every polynomial time probabilistic algorithm A we have:*

$$Pr[\kappa \overset{R}{\leftarrow} K; (m, m') \leftarrow A(k) : m \neq m',\ \mathcal{H}_\kappa(m) = \mathcal{H}_\kappa(m')] = neg(k)$$

where $\kappa \overset{R}{\leftarrow} K$ means that κ is selected uniformly at random in the set of keys K.

In the following H will denote a randomly selected function of a collision-resistant hash-family function $\mathcal{H} : K \times M \rightarrow Y$, where M is the set of all binary strings and Y is the set $\{0, 1\}^k$, for a large enough security parameter $k \in \mathbb{N}$.

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 $-\infty \prec x_j \prec +\infty$ for all $x_j \in X$, where $\prec$ is the order relation on X (for example, the lexicographic order on bit strings).

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

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

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

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

Consider now the following recursive definition of *labeled binary tree* T:

- T equals the empty tree Nil, or
- $T = (S; \mathit{left}, \mathit{right})$ where S is a label (string) and $\mathit{Left}(T) = left$ and $\mathit{Right}(T) = \mathit{right}$ are trees.

Here *left* and *right* are the *left* and *right* child of T respectively. Each tree T has associated a node $N = \mathit{node}(T)$ which is called the root of T as well as the *parent* of $\mathit{Left}(T)$ and $\mathit{Right}(T)$. Each node $N = \mathit{node}(S; \mathit{left}, \mathit{right})$ has associated a string $\mathit{Label}(N) = S$. Sometimes we identify the tree with its root and we write $\mathit{Label}(T)$ to denote $\mathit{Label}(\mathit{node}(T))$. We say that N' is a node of T if $N' = \mathit{node}(T)$ or N' is a node of $\mathit{Left}(T)$ or $\mathit{Right}(T)$. A *leaf* is a node of the form $(S; Nil, Nil)$. If $T = Nil$, then we say that T has depth 0 and denote it as $\mathit{depth}(T) = 0$. Otherwise, let $\mathit{depth}(T) = 1 + \max\{\mathit{depth}(\mathit{Left}(T)), \mathit{depth}(\mathit{Right}(T))\}$. A tree T is *balanced* if $|\mathit{depth}(\mathit{Left}(T)) - \mathit{depth}(\mathit{Right}(T))| \leq 1$. It is a well known fact that a balanced tree with n nodes has maximum depth $O(\log(n))$.

The set $\{H(x_i||x_{i+1}) : 0 \leq i \leq n\}$ will be called the *base* of X under H. Since H is a collision-resistant hash function and no two x_i are identical, $H(x_i||x_{i+1}) \neq H(x_j||x_{j+1})$ for $i \neq j$, except with negligible probability.

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 $\mathit{Label}(N) = (\mathit{Val}_N; \mathit{Proof}_N)$.
- The base of X is $\{\mathit{Val}_N : N \text{ is a node of } T\}$.
- T has $n+1$ nodes.
- $\mathit{Proof}_N = H(\mathit{Val}_N||\mathit{Proof}_{\mathit{Left}(N)}||\mathit{Proof}_{\mathit{Right}(N)})$ for every node N of T (where Proof_{Nil} corresponds to the empty string).

Figure 1 depicts a toy example of a model of a set.

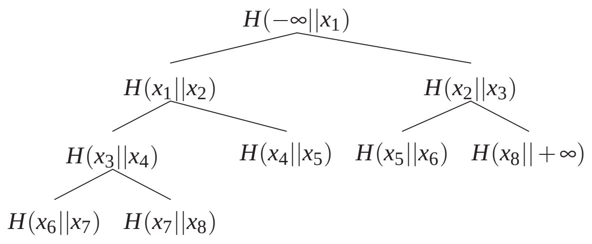

Fig. 1. A tree model T of the set $X = \{x_1, \ldots, x_8\}$. Only node values are shown. Note that the place of the values in the tree is irrelevant.

A subtree T' of a labeled binary tree T is a tree such that: (a) $\mathit{Label}(T') = \mathit{Label}(T)$, (b) $\mathit{Left}(T')$ is a subtree of $\mathit{Left}(T)$ or $\mathit{Left}(T') = \mathit{Nil}$, and (c) $\mathit{Right}(T')$ is a subtree of $\mathit{Right}(T)$ or $\mathit{Right}(T') = \mathit{Nil}$.

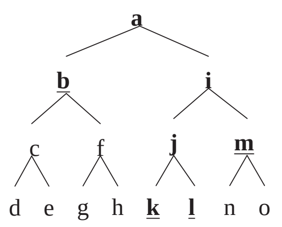

Fig. 2. A tree and its minimal subtree (nodes with values in boldface) generated by the node of value j. Children of the nodes that are on the path from j to a are underlined.

Let T be a labeled binary tree. We denote its collection of node values by $\mathcal{V}(T)$. 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 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 all the children of these nodes. 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.

Proposition 2. *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 $X \subset M$ 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 $V \in \mathcal{V}(T') \setminus \mathcal{V}(T)$ and $\mathit{Proof}_{T'} = \mathit{Proof}_T$, 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 $X \subset M$ 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 $\mathit{Proof}_{T'} = \mathit{Proof}_T$ 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 $\mathit{Proof}_{N'} = H(\mathit{Val}_{N'}||\mathit{Proof}_{\mathit{Left}(N')}||\mathit{Proof}_{\mathit{Right}(N')})$ and $\mathit{Proof}_N = H(\mathit{Val}_N||\mathit{Proof}_{\mathit{Left}(N)}||\mathit{Proof}_{\mathit{Right}(N)})$ are equal but $\mathit{Val}_N||\mathit{Proof}_{\mathit{Left}(N)}||\mathit{Proof}_{\mathit{Right}(N)} \neq \mathit{Val}_{N'}||\mathit{Proof}_{\mathit{Left}(N')}||\mathit{Proof}_{\mathit{Right}(N')}$. Nodes N and N' can be found efficiently by simply traversing both trees in some fixed order.

Now, let B be an adversary that is given a uniformly selected at random collision-resistant hash function $H \in \mathcal{H}$. 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 $\mathit{Proof}_T = \mathit{Proof}_{T'}$ and $V \in \mathcal{V}(T') \setminus \mathcal{V}(T)$. Finally, following the procedure mentioned above, B will be able to find a collision for H.

3.2 A Strong Universal Accumulator with Memory using Hash Trees

In this section we use hash trees to build a universal accumulator with memory.

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

- The memory $\mathfrak{m}$ is a model of X.
- Setup: The algorithm first sets X equal to the empty set. Then, it picks a hash function H uniformly at random from the family $\mathcal{H}$ by first computing a random index $i \in K$ (say using standard multiparty computation techniques among all participants, including the accumulator manager) and then setting $H = H_i$.[1] The algorithm then initializes $\mathfrak{m}$ to a single root node $N_{\mathfrak{m}}$ with value $H(-\infty || + \infty)$. Finally, the accumulator manager publishes $\mathfrak{Acc}_{init} = Proof_{N_{\mathfrak{m}}}$.
- $\mathsf{Witness}$: 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 $H(x_\alpha || x_\beta)$.
- $\mathsf{Belongs}$: On input $x \in M$, witness $w = ((x', x''), u)$, and accumulator value $\mathfrak{Acc}$, it first checks if the following conditions hold: (a) $Proof_u = \mathfrak{Acc}$, (b) $H(x' || x'') \in \mathcal{V}_u$, (c) $(x = x'$ or $x = x'')$, and (c') $(x' \prec x \prec x'')$. The algorithm outputs 1 if conditions (a), (b), and (c) hold; it outputs 0 if (a), (b), and (c') hold. Otherwise, it outputs $\perp$.
- $\mathsf{Update}_{\mathtt{op}}$: On input an element $x \in M$, an accumulator value $\mathfrak{Acc}_{before}$, and a 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 $\mathfrak{m}_{before}$ 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 $\mathfrak{m}_{before}$ with 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 $w_{add} = (\mathtt{add}, w_{add,1}, w_{add,2})$ that the update (addition) has been done correctly is computed as follows:

  - $w_{add,1}$ corresponds to the minimal subtree of $\mathfrak{m}_{before}$ generated by the set $\{H(x_\alpha || x_\beta), Val_{V_{Par(N)}}\}$, and,
  - $w_{add,2}$ corresponds to the minimal subtree of $\mathfrak{m}_{after}$ generated by the set $\{H(x_\alpha || x), H(x || x_\beta)\}$.

[1] A common heuristic to avoid interaction is to simply pick H =SHA-256 [12], for example.

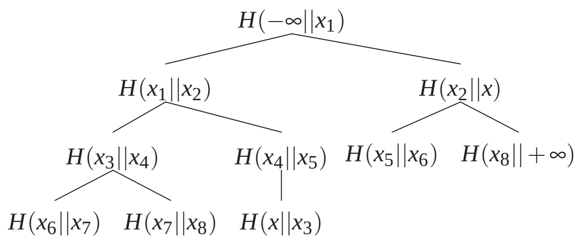

Fig. 3. Inserting x into the tree of Figure 1 where $x_2 \prec x \prec x_3$.

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 $w_{del} = (\mathtt{del}, w_{del,1}, w_{del,2}, w_{del,3})$ is then computed as follows:

- $w_{del,1}$ 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
- $w_{del,3}$ 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 $w_{\mathtt{op}}$.

- $\mathsf{CheckUpdate}$: On input an element $x \in M$, two accumulator values $\mathfrak{Acc}_{before}$, $\mathfrak{Acc}_{after}$, and an update witness w, it proceeds as follows. If $w = (\mathtt{add}, w_1, w_2)$ then, the algorithm outputs 1 provided that:
  - w_1 is a tree obtained by adding a leaf to w_2,
  - 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 w_1 and w_2 have the same value in either one of the trees,
  - $Proof_{w_1} = \mathfrak{Acc}_{before}$ and $Proof_{w_2} = \mathfrak{Acc}_{after}$, and
  - $H(x_\alpha||x), H(x||x_\beta) \in \mathcal{V}(w_2)$.

  Otherwise, it outputs 0. We omit the case $w = (\mathtt{del}, w_1, w_2, w_3)$ which is similar.

SECURITY. We now prove that the scheme $\mathsf{HashAcc}$ of the previous section is secure under Definition 3.

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

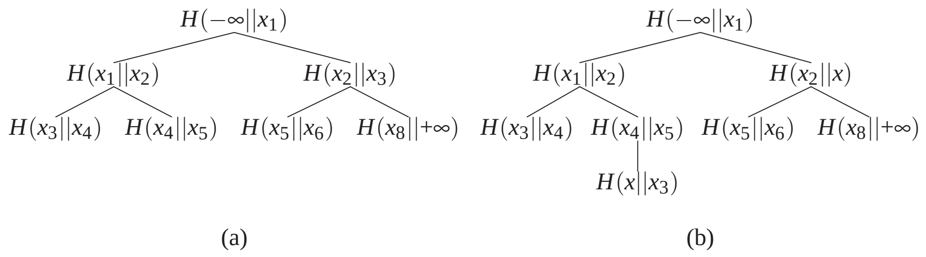

Fig. 4. (a) The minimal subtree of the tree shown in figure 1 and generated by $\{H(x_2||x_3), H(x_4, x_5)\}$. (b) The minimal subtree of the tree shown in Figure 3 and generated by $\{H(x_2||x), H(x||x_3)\}$.

$x_\alpha \prec x \prec x_\beta$ and let $H(x_\alpha||x_\beta)$ be the value of a node V in $\mathfrak{m}$. By replacing node V 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.

Theorem 1. *Let $\mathcal{H} : K \times M \to Y$ be a collision-resistant hash function family. Then, the accumulator scheme* HashAcc *is a secure strong universal accumulator scheme (with memory).*

Proof. We need to prove the properties *Consistency*, *Addition*, and *Deletion*.

- *(Consistency)* First, we note that $\mathfrak{Acc} \Rightarrow X$ implies that there exists a memory $\mathfrak{m}$ which is a model of X. 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 $\mathsf{Belongs}(x, w_1, \mathfrak{Acc}) = 1$ and $\mathsf{Belongs}(x, w_2, \mathfrak{Acc}) = 0$. 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 2 this adversary will only succeed with negligible probability. The argument for $x \notin X$ is analogous.
- *(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 $\mathfrak{Acc}_{before} \Rightarrow X$. Assume that $\mathsf{CheckUpdate}(x, \mathfrak{Acc}_{before}, \mathfrak{Acc}_{after}, w) = 1$ where both x and $w = (\mathtt{add}, U_{before}, U_{after})$ are arbitrarily chosen by the adversary, and $\mathfrak{Acc}_{after} \not\Rightarrow X \cup \{x\}$. Then, for some two elements $u, v \in M$ the adversary is effectively able to 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 2.

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

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

Theorem 2. *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. It is enough to show that a minimal subtree U of T generated by a constant number of node values has a 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 in Practice

Our solution is theorically less efficient than the scheme proposed in [11]. Nonetheless, if one considers practical instances of these schemes the difference effectively vanishes as in most implementations hash functions operations are significantly faster than RSA exponentiations – which is the core operation used by the schemes in [11, 5]. Table 2 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 [13]) on a personal computer. Notice that RSA timings were obtained using signing operations, as in the scheme proposed in [11] 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 $4 \times 2\log(N)$ 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.

Our results show that even for large values of N using a hash-based scheme is still very efficient. Moreover, our scheme is faster than using a single RSA operation with a 2048-bit key.

Table 1. Running time for RSA and SHA operations.

Algorithm	Note	Operations per second
SHA-256	input block of 1024 bits	65507
SHA-512	input block of 1024 bits	16856
RSA-512	signing operation	1179
RSA-1024	signing operation	236
RSA-2048	signing operation	40

Table 2. Comparison of performance between simple RSA exponentiation and logarithmic number of computations of SHA. N is the number of elements that are accumulated. Time is represented in milliseconds.

N	**RSA-512**	**RSA-1024**	**RSA-2048**	**SHA-256**	**SHA-512**
2^3	0,845	4,23	25	0,37	1,42
2^{10}	0,845	4,23	25	1,22	4,75
2^{20}	0,845	4,23	25	2,44	9,5
2^{30}	0,845	4,23	25	3,66	14,24

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 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 a purchase order from C to a third party, henceforth referred to as Factoring 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.

The main phases of a factoring operation are summarized below: (a) C requests from P either goods or services, (b) P delivers the goods/services to C, (c) P makes a factoring request to FE, (d) FE either rejects or accepts P's request — in the latter case FE gives P a cash advance on C's purchase, (e) later, FE asks C to settle the outstanding payment, and finally (f) 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.

Approximately half 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 [8].

The newly 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*. In the full version of this paper we show how to address this problem using strong universal accumulator schemes.

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 [11], namely (a) a set is represented by a short value called accumulator, (b) it is possible to add and remove elements dynamically from the (accumulated) set, and (c) proofs of membership and nonmembership 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 nonmembership 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. 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.
2. D. Bayer, S. Haber, and W. S. Stornetta. Improving the efficiency and reliability of digital time-stamping. In R.M. Capocelli, A. DeSantis, and U. Vaccaro, editors, *Sequences II: Methods in Communication, Security, and Computer Science*, pages 329–334. Springer–Verlag, 1993.
3. J. Benaloh and M. De Mare. One-way accumulators: A decentralised alternative to digital signatures. In *Advances in Cryptology - Proceedings of Eurocrypt '93*, volume 765 of *LNCS*, pages 274–285. Springer–Verlag, 1993.
4. 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.
5. J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *Advances in Cryptology - Proceedings of Crypto '02*, volume 2442 of *LNCS*, pages 61–76, Berlin, 2002. Springer–Verlag.
6. 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.

7. 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.
8. Servicio de Impuestos Internos. Información sobre factura electrónica. (`https://palena.sii.cl/dte/mn_info.html` [June 19, 2008]).
9. N. Fazio and A. Nicolisi. Cryptographic accumulators: Definitions, constructions and applications, 2003. (`http://www.cs.nyu.edu/~nicolosi/papers/accumulators.ps` [June 19, 2008]).
10. P. C. Kocher. On certificate revocation and validation. In R. Hirschfeld, editor, *Financial Cryptography*, volume 1465 of *LNCS*, pages 172–177. Springer–Verlag, 1998.
11. 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.
12. National Institute of Standards and Technology (NIST). *FIPS Publication 180: Secure Hash Standard (SHS)*, May 1993.
13. OpenSSL Project. OpenSSL Package, June 2008. (`http://www.openssl.org` [June 19, 2008]).