THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A,
Volume 14, Special Issue 2013, pp. 269–285

A FULLY DYNAMIC UNIVERSAL ACCUMULATOR

Atefeh MASHATAN and Serge VAUDENAY

EPFL, Lausanne, Switzerland
http://lasec.epfl.ch

A dynamic universal accumulator is an accumulator that allows one to efficiently compute both membership and nonmembership witnesses in a dynamic way. It was first defined and instantiated by Li *et al.*, based on the Strong RSA problem, building on the dynamic accumulator of Camenisch and Lysyanskaya. We revisit their construction and show that it does not provide efficient witness computation in certain cases and, thus, is only achieving the status of a *partially dynamic* universal accumulator. In particular, their scheme is not equipped with an efficient mechanism to produce non-membership witnesses for a new element, whether a newly deleted element or an element which occurs for the first time.
We construct the first *fully dynamic* universal accumulator based on the Strong RSA assumption, building upon the construction of Li *et al.*, by providing a new proof structure for the non-membership witnesses. In a fully dynamic universal accumulator, we require that not only one can always create a membership witness without having to use the accumulated set for a newly added element, but also one can always create non-membership witnesses for a new element, whether a newly deleted element or an element which occurs for the first time, *i.e.*, a newcomer who is not a member, without using the accumulated set.

Key words: Dynamic Accumulators, Universal Accumulators.

1. INTRODUCTION

Cryptographic accumulators allow us to encapsulate a large number of elements in a single short *accumulator* along with short *witnesses* that can be used for proving whether or not an element has been accumulated. The notion of cryptographic accumulators was first introduced by Benaloh and de Mare [1] and further pursued by many researchers as they come very practical in many scenarios such as anonymous credential systems and group signatures, see for example [9, 8, 4], and that they can be instantiated based on a variety of techniques and hardness assumptions, for instance, the strong RSA assumption, bilinear maps, the Decisional Diffie-Hellman assumption, and one-way hash functions.

We are now going to focus on a number of schemes which are based on the Strong RSA assumption and they were built one after the other in an evolutionary process. Barić and Pfitzmann [2] followed the work of Benaloh and de Mare [1] and introduced collision-free accumulators. This scheme provided membership proofs. Later, Camenisch and Lysyanskaya [5] augmented the latter work and proposed a dynamic accumulator, in which elements can be efficiently added to and removed from the set of accumulated values. Finally, Li *et al.* [7] built their scheme based on the proposal of Camenisch and Lysyanskaya [5] and introduced universal accumulators in which there is a witness, whether a member or not, for every elements in the input domain. (See [6] for a survey.) Although Li *et al.* promise to provide efficient non-membership proofs, we will see that the structure of the witness fails to offer efficient dynamic proof computation for certain elements and, hence, achieves the desired dynamism only *partially*. In a *fully dynamic* universal accumulator, we require that not only one can always create a membership witness without having to use the accumulated set for a newly added element, but also one can always create non-membership witnesses for a new element, whether a newly deleted element or an element which occurs for the first time, *i.e.*, a newcomer who is not a member, without using the accumulated set.

Although accumulators are not so new elements in cryptographic schemes, formal security definitions and classifications on different requirements have not been adequately dealt with. The literature often fails to provide exact correctness or security definitions for different classes of accumulators and there have been several security notion proposed. We focus on the strongest security notion, considering a powerful adversary who can invoke the authority with polynomially many accumulator initiations. This notion is referred to as the *Chosen Element Attack* model, in the literature [11]. Informally, a polynomially bounded adversary interacts with the authority who maintains the accumulators. The adversary invokes the authority to initiate a polynomial number of accumulators and make changes to them according to the adversary's instructions on what element to add or delete. Finally, the adversary chooses an element and an accumulator and produces a witness. The adversary wins if the witness proves that the chosen element is not a member when in fact it is, or it proves that the chosen element is a member when in fact it is not.

Accumulators have proven to be a very strong mathematical tool with applications in a variety of privacy preserving technologies where there is a desire to represent a set of elements in a compact form, for example, certificate revocation schemes, anonymous credential systems, and group signatures. In particular, fully dynamic universal accumulators can come handy in a variety of real-life scenarios. For example, consider the set of people who have a medical condition that allows them to benefit from some discount medication, but denies them the access to certain areas, such as swimming pools. These people should be able to efficiently prove their membership at a pharmacy and everyone else should be able to show their nonmembership when entering a swimming pool, for example.

It is known that *batch updates* cannot be done [3]. This means that updating a (non)membership proof without the secret key requires to go through all the accumulator updates.

Our contributions.

In this paper, we first point out the lack of efficiency in the dynamic updating process of the dynamic universal accumulator of Li *et al.* [7], where the authority has to go through the already accumulated set to create non-membership witnesses for certain members, namely newly considered values which are not members and newly deleted members, defying the claim that the scheme provides efficient non-membership proofs in all cases.

Moreover, we introduce the notion of *weak* dynamic accumulators, a special case of dynamic accumulators where the only operation is addition and the elements can dynamically be added to the accumulator. Hence, a dynamic accumulator is trivially a weak dynamic accumulator. Further, we present a generic transformation from a weak dynamic accumulator with a domain having a certain structure, *e.g.*, the domain being a set of odd primes, to a weak dynamic accumulator with a domain of arbitrary form, *e.g.*, a subset of $\{0,1\}^*$.

Furthermore, we formally define what we require from a *fully* dynamic universal accumulator and instantiate the first such scheme based on the Strong RSA assumption and a weak dynamic accumulator with an arbitrary domain. This instantiation builds on the previous schemes based on the same hardness assumption by keeping the structure of the membership proofs, due to Camenisch and Lysyanskaya [5], but providing a new structure for the non-membership proofs. This property, *i.e.*, being *fully* dynamic, comes at a price. Our accumulators are a bit larger. However, as it is more efficient when introducing new elements compared to previously introduced partially dynamic accumulators, it achieves the same level of efficiency as the set of accumulated values is growing. Moreover, the efficacy of the new structure of non-membership proofs allows the authority to perform batch updates, a desired property that had not been achieved successfully so far.

Structure of the paper.

The rest of the paper is organized as follows. Section 2 is dedicated to briefly describing, notations, definitions, different classes of already existing accumulators, and the particular variant of the dynamic accumulator of Camenisch and Lysyanskaya [5] due to Li *et al.* [7]. In Section 3, we define the notion of Weak Dynamic Accumulators and present a generic transformation to obtain a Weak Dynamic Accumulators with arbitrary domain. Finally, Section 4 is devoted to defining Fully Dynamic Universal Accumulators followed by an instantiation whose security is based on the strong RSA assumption. Last but not least, we wrap up with some concluding remarks in Section 5.

2. PRELIMINARIES

In this section, we list definitions, notations, and the building blocks which will be used to construct and analyze our scheme in the following section.

Throughout this paper, we use the expression $y \leftarrow A(x)$ to mean that y is the output of algorithm A running on input *x*. An algorithm is said to have polynomial running time, if its running time can be expressed as a polynomial in the size of its inputs. If X denotes a set, $|X|$ denotes its cardinality and $x \in_R X$ expresses that x is chosen from X according to the uniform distribution. If X and Y are sets, then $X \setminus Y$ denotes the set of elements in *X*, but not in *Y*. For convenience, we also use $X + \{x\}$ and $X - \{x\}$ when an element x is being added in or deleted from a set *X*. We also use the classical notion of a negligible function: $f : N \to R$ is said to be negligible in k if for any positive polynomial function $p(.)$ there exists a k_0 such that if $k \geq k_0$, then $f(k) < 1/p(k)$.

The strong RSA assumption states that given an RSA modulus n and a random x drawn from Z_n^*, it is infeasible to find $e > 1$ and $y \in \mathsf{Z}_n^*$ such that $y^e = x \bmod n$.

2.1. Evolution of Cryptographic Accumulators

In this section, we illustrate the evolution of the cryptographic accumulators in the literature. There are different notions of security used in the literature and, due to lack of space, we only focus on the strongest notion, sometimes referred to as the *Chosen Element Attack* model. As for the notion of correctness of an accumulator, one requires that correctly accumulated values have verifying witnesses, regardless of the type of accumulator. The literature has often stopped here and has failed to provide a more precise definition of correctness. We will provide the first formal definition of correctness that can be applied to several categories of accumulators with different functionalities.

There is an authority who initiates and maintains the accumulator and interacts with other participants. The authority generates the secret and public keys and keeps a state including the keys, the accumulated value, the set of elements which are accumulated. The authority delivers the proofs to the participants. In the security definition, the adversary asks the authority to provide certain proofs in the form of an oracle.

The definitions below are mostly gathered by Wang et al. [12], but contain some twists to make them consistent with the following sections.

Definition 1 (Accumulators). *An* accumulator *scheme* Acc*, with a domain P, a set* $X \subseteq P$*, and values* $x \in X$ *to be accumulated, consists of the following algorithms.*

– *A setup probabilistic algorithm* $\mathrm{KeyGen}(1^k) \to (K_s, K_p)$*, where* K_s *is only used by the authority and* K_p *is public.*

– *An algorithm* $\mathrm{AccVal}(K_s, K_p, X) \to c$*, which computes an accumulator value* $\mathbf{c}$ *, for the set X, from the keys.*[1]

– *An algorithm* $\mathrm{WitGen}(K_s, c, X, x) \to W$ *to generate a proof of membership for* $x \in X$ *in accumulator* c*.*

– *A predicate* $\mathrm{Verify}(K_p, c, x, W)$ *to check a proof.*

One problem with this primitive is the lack of any possibility to dynamically update a set *X*. Ideally, we would like to insert or delete members of X and update the accumulator c accordingly. To make it efficient we want all values to have a length which only depends on k and not on the cardinality of *X*. So, we amend the definition by introducing the UpdEle and UpdWit algorithms. This defines the notion of *Dynamic Accumulator* (see Def. 11 in Appendix).

[1]For some constructions, K_s is not used by AccVal.

The accumulator does not consider proofs of non-membership. This is the next desired functionality to have. This is done with the notion of *Universal Accumulator* (see Def. 12 in Appendix).

A proof W for x is said to be *valid* with respect to (X,c) if and only if the predicate $\text{Verify}(K_p,c,x,W)$ holds. Moreover, a proof W for x is said to be *coherent* with respect to (X,c) if and only if it is valid and $\text{IsMem}(W)$ is equivalent to $x \in X$. One problem with this construction is that the authority must figure out whether or not x is a member of X to generate a coherent proof. That is, the information on X cannot be compressed. The lack of dynamism is also a problem.

Finally, we formalize the dynamic universal accumulator notion due to Li *et al.* [7] as follows. The reason why we call it a *partially* dynamic universal accumulator is discussed below. Note that our formalism is a bit more detailed that theirs to be consistent with the rest of our paper.

Definition 2 (Partially Dynamic Universal (PDU) Accumulators). A partially dynamic universal accumulator PDUAcc*, with a domain P, a set* $X \subseteq P$*, and values* $x \in X$ *to be accumulated, consists of the following algorithms.*

– A setup probabilistic algorithm $\text{KeyGen}(1^k) \to (K_s,K_p)$*, where* K_s *is only used by the authority and* K_p *is public.*

– An algorithm $\text{AccVal}(K_s,K_p,X) \to c$*, which computes an accumulator value* c *of the set X, from the keys.*

– An algorithm $\text{UpdEle}(K_s,K_p,c,\text{op},x) \to (c',\text{extra})$*, where* $\text{op} = +$ *or* $\text{op} = -$*, which computes the accumulator* c' *for* $X\text{op}\{x\}$ *from the accumulator* c *for X. When* $\text{op} = +$*, we must have* $x \notin X$ *and we say that* x *is inserted into X. When* $\text{op} = -$*, we must have* $x \in X$ *and we say that* x *is deleted from X. The algorithm also returns some extra information* extra*, which might be needed for dynamic witness update.*

– An algorithm $\text{WitGen}(K_s,c,X,x) \to W$ *to generate a proof of membership or non-membership for the value* x *with respect to accumulator* c *of X.*

– An algorithm $\text{UpdWit}(K_p,c,c',\text{extra},\text{op},x,W,y) \to W'$ *to generate a proof* W' *for* y *in accumulator* c' *from a proof* W *for* y *in accumulator* c*, where* $\text{UpdEle}(K_s,K_p,c,\text{op},x) \to (c',\text{extra})$*. It must be the case that* $x \neq y$*.*

– A predicate $\text{IsMem}(W)$ *telling whether* W *is a proof of membership (true case) or a proof of non-membership (false case).*

– A predicate $\text{Verify}(K_p,c,x,W)$ *to check a proof.*

We let X be a subset of P which is initially empty. Every time we run $\text{UpdEle}(.,.,.,\text{op},x)$, we replace X by $X\text{op}\{x\}$. Clearly, for WitGen to generate coherent proofs, $\text{IsMem} \circ \text{WitGen}$ must be a predicate to decide whether an arbitrary x belongs to X or not. Since we want c to have a length which only depends on k (and not on the cardinality of X), it is not possible to require WitGen to generate coherent proofs without X as an input while X has been filled. It is still possible to invoke WitGen to create from X a new proof. Then, we count on UpdWit to update all coherent proofs.

With this notion, UpdWit cannot create a new witness for $x = y$, that is for a value y which has just been added or deleted. This is why we call it *Partially* Dynamic Universal.

2.2. Formalizing the Notions of Correctness and Security for Accumulators

We now describe our notion of correctness than can be applied to several types of accumulators. This definition is our attempt to formalize the notion of correctness which was missing in the literature. One difficulty is that the accumulator interface introduces many options, and that we want to formalize that whatever sequence of option is selected, the accumulator always keep consistent properties. Here, the sequence of options is arbitrary. We formalize this by introducing an adversary who can select it maliciously.

Intuitively, the notation $c \prec X$ means that c is a correct accumulator value computed for the set *X*, while $W \vdash (x, X, c, bool)$ means that W is a valid computed proof for x being/not being (depending on the Boolean $bool$) in $c \prec X$.

Definition 3 (Correctness). *Consider a game in which we first run* $\mathrm{KeyGen}(1^k) \to (K_s, K_p)$ *and allow the adversary to take* K_p *and* K_s *and play with the rest of the algorithms available for the respective accumulator, e.g.,* $\mathrm{AccVal}(K_s, K_p,.)$, $\mathrm{MemWitGen}(K_s,.,.,.)$, $\mathrm{NonMemWitGen}(K_s,.,.,.)$, $\mathrm{UpdEle}(K_s, K_p,., \mathrm{op},.)$, *or* $\mathrm{UpdWit}(K_p,.,.,., \mathrm{op},.,.,.)$. *We recursively define the relations* $c \prec X$ *and* $W \vdash (x, X, c, bool)$ *by the following conditions:*

– If the adversary calls $\mathrm{AccVal}(K_s, K_p, X) \to c$*, then* $c \prec X$.

– If $c \prec X$ *and the adversary queries* $\mathrm{UpdEle}(K_s, K_p, c, \mathrm{op}, x) \to (c', \mathrm{extra})$*, then* $c' \prec X \mathrm{op}\, x$.

– If $c \prec X$ *and the adversary queries* $\mathrm{WitGen}(K_s, c, X, x) \to W$ *or* $\mathrm{MemWitGen}(K_s, c, X, x) \to W$*, then* $W \vdash (x, X, c, true)$*. If* $c \prec X$ *and the adversary calls* $\mathrm{NonMemWitGen}(K_s, c, X, x) \to W$*, then* $W \vdash (x, X, c, false)$.

– If $c \prec X$ *and the adversary called* $\mathrm{UpdEle}(K_s, K_p, c, \mathrm{op}, x) \to (c', \mathrm{extra})$ then $\mathrm{UpdWit}(K_p, c, c', \mathrm{extra}, \mathrm{op}, x, W, y) \to W'$ with $x \neq y$ and $W \vdash (y, X, c, b)$, then $W' \vdash (y, X\mathrm{op}, c', b)$.

The accumulator scheme is said to be correct if for all probabilistic polynomial time adversaries, and for all possible choices of W, x, X and c, we have that $W \vdash (x, X, c, b)$ *implies* $\mathrm{Verify}(K_p, c, x, W)$ *and, in the case of universal accumulators,* $\mathrm{IsMem}(W) \Leftrightarrow x \in X$.

Note that it is unusual to have an adversary in a definition of correctness. This is necessary here as we want to have correctness whatever the history of interactions with the interface.

Next, we describe the Chosen Element Attack scenario [12] when defining security for a PDU accumulator.

Definition 4 (Chosen Element Attack (CEA) Model). *The security of a PDU accumulator is defined in terms of a game, based on a security parameter* k*, played by a polynomially bounded adversary. Firstly,* KeyGen *is run and* K_p *is given to the adversary. Secondly, the adversary selects a polynomially bounded number* ℓ*. There are registers* K_s, X_i*, and* c_i, $i = 1, \ldots, \ell$*, for a secret key and to keep track of* ℓ *sets* X_i *and their accumulator values* c_i*. Initially, all* X_i*'s are empty and* c_i *is set to* $\mathrm{AccVal}(K_s, K_p, X_i)$*. The adversary can then call an* $\mathrm{UpdEle}(K_s, K_p, c_i,.,.)$ *oracle for a selected* i *which updates* X_i *and* c_i *accordingly. It is not allowed to add an* x *to* X_i *when* x *is already in* X_i*, nor is it allowed to delete* x *from* X_i *when* x *is not in* X_i*. The adversary can also call a* $\mathrm{WitGen}(K_s, c_i, X_i,.)$ *oracle for a selected* i *and an* $\mathrm{AccVal}(K_s, K_p,.)$ *oracle which do not update an* X_i *of* c_i*. After making many oracle queries, the adversary ends by producing some* (i, x, W)*. The adversary wins if* W *is an incoherent proof for* x *with respect to* X_i *and accumulator* c_i.

Note that when the algorithms are all deterministic, we can always reduce to $\ell = 1$ and remove access to the $\mathrm{AccVal}(K_s, K_p,.)$ oracle. This follows because calling AccVal on the same inputs is going to produce the same outputs. Moreover, the information that the adversary obtains from calling AccVal for the sets X_i can all be simulated by a single set *X*.

2.3. An Instantiation of Partially Dynamic Universal Accumulators due to Li *et al.*

We now describe a PDU accumulator based on the scheme presented by Li *et al.* [7].

In what follows, we have a domain P of possible values for x with some specific form. A subset $X \subseteq P$ has an accumulator c and public parameters n and g. A proof of non-membership for $x \in P \setminus X$ for c is a tuple (a,d) such that

$$c^a \equiv d^x g (\bmod n).$$

By writing a Euclidean division $a = a' + qx$, we can easily transform such a proof (a,d) into a new proof (a',d'), with $d' = dc^{-q} \bmod n$, such that $0 \le a' < x$. We refer to

$$[a,d]_x = (a \bmod x, dc^{-\frac{a-(a \bmod x)}{x}} \bmod n)$$

as the "reduced" proof, where the public parameters are implicit. Note that $[a,d]_x$ can be computed without the secret key for any integer a. When a is a rational number, we need the secret key to compute it.

Domain: P is a set of odd prime numbers, e.g., all odd prime numbers up to a given bound B.

KeyGen(1^k): pick two different prime numbers p and q such that $\frac{p-1}{2}$ and $\frac{q-1}{2}$ are both prime and not in P, take $n = pq$, $r = \frac{1}{2}\lambda(n)$, and g an element of order r in Z_n^*. Then, $K_p = (n,g)$ and $K_s = r$.

AccVal (K$_p$,X): the value of c is defined by

$$c = g^{\prod_{x \in X} x} \bmod n. \tag{1}$$

Note that K_s is not required to compute c.

UpdEle(K_s,K_p,c,op,x): if $\mathrm{op} = +$, we have $c' = c^x \bmod n$ (K_s is not required). If $\mathrm{op} = -$, we have $c' = c^{\frac{1}{x}} \bmod n$ (K_s is required). It returns $\mathrm{extra} = (c, \mathrm{op}, x, c')$. Since it is not allowed to delete a non-member of X, c' can be computed without K_s but using X by applying (1).

WitGen(Ks,c,X,x): if $x \in X$, then $W = (\mathrm{mem}, w)$ with $w = c^{\frac{1}{x}} \bmod n$. This does not require X. It can also be computed without K_s, but using X by observing that

$$w = c^{\prod_{y \in X - \{x\}} y} \bmod n. \tag{2}$$

If $x \notin X$, then $W = (\mathrm{nonmem}, [a,d]_x)$ with $a = \left(\prod_{y \in X} y\right)^{-1}$ and $d = 1$.

Clearly, the final (a,d) is such that

$$a = \left(\prod_{y \in X} y\right)^{-1} \bmod x \quad and \quad d = \left(c^a g^{-1}\right)^{\frac{1}{x}} \bmod n. \tag{3}$$

The above computation requires K_s. Below, we give an algorithm to compute $[a,d]_x$ without K_s, but by going through all X members.

UpdWit(K_p,extra, W,y): with $\mathrm{extra} = (c, \mathrm{op}, x, c')$, there are four cases to update the proof for y in X after adding or deleting x:

- W of the form (mem, w) and x just added ($\text{op} = +$): set $W' = (\text{mem}, w')$ with $w' = w^x \bmod n$.
- W of the form (mem, w) and x just deleted ($\text{op} = -$): set $W' = (\text{mem}, w')$ with $w' = w^z c'^{\frac{1-xz}{y}} \bmod n$ and $z = \frac{1}{x} \bmod y$.
- W of the form (nonmem, a, d) and x just added ($\text{op} = +$): set $W' = (\text{nonmem}, [a', d']_y)$ with $a' = az$, $d' = dc^{-a\frac{1-xz}{y}} \bmod n$ and $z = \frac{1}{x} \bmod y$.
- W of the form (nonmem, a, d) and x just deleted ($\text{op} = -$): set $W' = (\text{nonmem}, [a', d]_y)$ with $a' = ax$.

IsMem(W): is true if and only if W is of the form $(\text{mem},.)$.

Verify(K_p, c,x,W): if W is of the form (mem, w), it is true if and only if $c = w^x \bmod n$. If W is of the form (nonmem, a, d), it is true if and only if $c^a \equiv d^x g (\bmod n)$.

We can compute the non-membership proof $[a,d]_y$ for $y \notin X$ by iteratively going through all $x \in X$. For this, we start with $(a,d) = (1,1)$, which is a proof of non-membership for y in the empty set with accumulator $c = 1$. Then, for each $x \in X$ we update $(a,d) \leftarrow \text{UpdWit}(K_p, (c,+,x,c^x \bmod n), (a,d), y)$ and $c \leftarrow c^x \bmod n$.

Note that all algorithms are deterministic here. It can be easily proven that all oracle calls in the security game preserve the relations (1), (2), and (3). Then, we easily prove that UpdWit preserves (2) and (3) as well. Since (1) can be computed without K_s and that the game does not allow the adversary to add a member or to delete a non-member, all oracle calls in the game can be simulated without knowing K_s. So, the security is equivalent to forging $X \subseteq P$ and $x \in P$ together with W which is an incoherent proof for x with respect to X with K_p as a sole input. We can easily see that this implies breaking the strong RSA assumption (c.f. [7] for more details).

Quite importantly, it is necessary that all proofs of non-membership satisfy (3). Indeed, assuming that $x \notin X$ and we have some (a',d') such that $c^{a'} \equiv d'^x g(\bmod n)$, the adversary can compute (a,d) satisfying (3) by going through all X members and without requiring K_s. Then, he would obtain a relation $c^{a'-a} \equiv (d'/d)^x (\bmod n)$. If the proof (a',d') does not satisfy (3), then x does not divide $a' - a$. So, the adversary can invert $a' - a$ modulo x and obtain a relation $w^x \bmod n = c$ which is a proof of membership although x has been deleted. Security collapses. So, it is important to provide only proofs (a,d) such that (3) holds. A similar weakness was spotted in [10]. It was based on a proposal to compute (a,d) from $\prod_{y \in X} y \bmod \varphi(n)$ instead of $\prod_{y \in X} y$, which is quite a bad idea! The above procedure is safe. (Indeed, its computation does not require the secret key, so it is zero-knowledge.)

One drawback of LLX is that after deleting x from X we cannot create a proof of non-membership for x except by recomputing a from scratch, since (3) must be satisfied. Although it seems dynamic, it fails to provide the promised efficiency since we need X to compute a and, hence, only offers a partially dynamic universal accumulator. A similar drawback exists in the WitGen algorithm. Indeed, the authority willing to issue a proof of non-membership for a non-member which never needed such a proof has to go through the entire X structure.

3. WEAK DYNAMIC ACCUMULATORS FOR ARBITRARY DOMAINS

In this section, we define the notion of *weak* dynamic accumulator where elements can be dynamically added to the accumulator. It's called a weak dynamic accumulator because it only considers adding and not deleting elements, and that is all we need for our construction in Section 4. When compared to Dynamic Accumulators in Def. 11, WD accumulators do not require an AccVal to compute c from X.

Definition 5 (Weak Dynamic (WD) Accumulators). A weak dynamic accumulator WDAcc*, with a domain P, a set* $X \subseteq P$*, and values* $x \in X$ *to be accumulated, consists of the following algorithms.*

– A setup probabilistic algorithm $\mathrm{KeyGen}(1^k) \to (K_s, K_p)$*, where* K_s *is only used by the authority and* K_p *is public.*

– An algorithm $\mathrm{InitAccVal}(K_s, K_p) \to c$*, which computes an initial accumulator value* c *of the empty set, from the keys.*

– An algorithm $\mathrm{AddEle}(K_s, K_p, c, x) \to (c', \mathrm{extra})$*, which computes the accumulator* c' *for* $X + \{x\}$ *from the accumulator* c *for* X*. We must have* $x \notin X$ *and we say that* x *is inserted into* X*. The algorithm also returns some extra information* extra*, which might be needed for dynamic witness update.*

– An algorithm $\mathrm{UpdWit}(K_p, c, c', \mathrm{extra}, x, W, y) \to W'$ *to generate a proof* W' *for* y *in accumulator* c' *from a proof* W *for* y *in accumulator* c*, where* $\mathrm{AddEle}(K_s, K_p, c, x) \to (c', \mathrm{extra})$.

– A predicate $\mathrm{Verify}(K_p, c, x, W)$ *to check a proof.*

Note that all WD accumulators are dynamic, by definition. Hence, the constructions due to Li *et al.* or Camenisch and Lysyanskaya are both WD accumulators. Moreover, Def. 3, the correctness definitions, can be applied to WD accumulators considering $+$ as the only possibility for op. Furthermore, the security notion for a WD accumulator is a special case of a Chosen Element Attack scenario since there are no deletions.

Definition 6 (Security of a WD accumulator). *The security of the WD accumulator is defined in terms of a game, based on a security parameter* k*, played by a polynomially bounded adversary. Firstly,* KeyGen *is run and* K_p *is given to the adversary. Secondly, the adversary selects a polynomially bounded number* ℓ*. There are registers* K_s*,* X_i*, and* c_i*,* $i = 1, \dots, \ell$*, for a secret key and to keep track of* ℓ *sets* X_i *and their accumulator values* c_i*. Initially, all* X_i*'s are empty and* c_i *is set to* $\mathrm{InitAccVal}(K_s, K_p)$*. The adversary can then call an* $\mathrm{AddEle}(K_s, K_p, c_i, ., .)$ *oracle for a selected* i *which updates* X_i *and* c_i *accordingly. It is not allowed to add an* x *to* X_i *when* x *is already in* X_i*. After making many oracle queries, the adversary ends by producing some* (i, x, W)*. The adversary wins if* W *is an incoherent proof for* x *with respect to* X_i *and accumulator* c_i*.*

We now describe a generic way of transforming a WD accumulator WDAcc_0 with domain P, set of elements x with some special form, to a WD accumulator with an arbitrary domain, *i.e.*, a finite subset of $\{0,1\}^*$. In this generic transformation, we will make use of a signature scheme. In the following description, the algorithms, values, and predicates with index of 0, *e.g.*, KeyGen_0, refer to the corresponding items in WDAcc_0 defined as above, whereas values indexed by sig refer to those of a signature scheme.

We assume that the elements in P can be enumerated starting from $h = h_{\mathrm{init}}$ and then iterating by means of an operation $h \leftarrow$next.element(h). The overall idea is that the value of the new accumulator consists of a pair $(c^0, \mathrm{last.h})$, where c^0 is the accumulator value for WDAcc_0 and the last added $\mathrm{last.h}$ value. To add a new string x, we get a new h using next.element and we bind h to x using a signature on (x, h). A witness for x is a triplet (h, σ, w), where h is the bound value, σ is a valid signature, and w is a witness for h in WDAcc_0. When a new x is added, the witness for it is computed and returned as the extra information.

A generic transformation to obtain a WD accumulator with arbitrary domain:
Domain: S is a large enough subset of $\{0,1\}^*$.

KeyGen(1^k): run $\mathrm{KeyGen}_0(1^k)$ and obtain K_p^0 and K_s^0. Further, run $\mathrm{KeyGen}_{\mathrm{sig}}(1^k)$ and obtain $(K_s^{\mathrm{sig}}, K_p^{\mathrm{sig}})$. Then $K_p = (K_p^0, K_p^{\mathrm{sig}})$ and $K_s = (K_s^0, K_s^{\mathrm{sig}})$.

InitAccVal(K_p,X) $\rightarrow$c=(c^0, h_{init}): where c^0 is the output of $\mathrm{InitAccVal}^0(K_s^0, K_p^0)$.

AddEle(K_s, K_p,(c^0, last.h),x): Let $h \leftarrow \mathrm{next.element(last.h)}$ denote the next element in the list and call $\mathrm{AddEle}^0(K_s^0, K_p^0, c^0, h) \rightarrow (c^0{}', \mathrm{extra}_0)$. Then let $\sigma \leftarrow \mathrm{sig}(K_s^{\mathrm{sig}}, x, h)$. Return $((c^0{}', h), (\sigma, \mathrm{extra}_0))$.

UpdWit(K$_p$,(c^0, last.h), ($c^0{}'$, next.h), (σ,extra), x, W, y): If $x = y$, then return $(\mathrm{next.h}, \sigma, \mathrm{extra})$. If $x \neq y$, extract h and w from $W = (h, \sigma, w)$ and then return $(h, \sigma, \mathrm{UpdWit}^0(K_p^0, c^0, c^0{}', \mathrm{extra}, \mathrm{next.h}, w, h))$.

Verify(K_p, (c^0, last.h), x, (h,σ,w)): is true if and only if both $\mathrm{Verify}^{\mathrm{sig}}(K_p^{\mathrm{sig}}, (x, h), \sigma)$ and $\mathrm{Verify}^0(K_p^0, c^0, h, w)$ are true.

Very similar generic transformations exist in the literature, see for example [4], where it is suggested that the issuer of the accumulator would need to publish a mapping from S to P used along with a signature scheme. The advantage of our scheme compared to those approaches is that we do not require any mapping to be published: we just assume that the elements of P can be enumerated and that it is easy to move to the next element, given the previous one.

The following theorem states that starting from a correct and secure WD accumulator with domain P and using a secure signature scheme, we obtain a correct and secure WD accumulator with the arbitrary domain S with the above generic transformation.

Theorem 7. *Consider a correct and secure WD accumulator* WDAcc_0 *and a secure digital signature scheme* sig *(i.e., signatures are unforgeable under chosen message attacks). The resulting WD accumulator* WDAcc *of the above generic transformation is correct and secure.*

Proof. For simplicity, we show the proof for the case of deterministic algorithm, hence assume $\ell = 1$. The general case follows similarly.

The correctness follows immediately as both the signature's and the original accumulator's verification predicate are being verified. More precisely, we need to show that WDAcc is correct according to Def. 3. Since it is not a universal accumulator, all we need to show is that $W \vdash (x, X, c, b)$ implies $\mathrm{Verify}(K_p, c, x, W)$, for all probabilistic polynomial time adversaries, and for all possible choices of $W, x, X,$ and c. Note that both AddEle and UpdWit call upon the respective algorithms in WDAcc_0. In particular, we have that $c = (c^0, h)$, for some h. Hence, $W \vdash (x, X, c, b)$ for WDAcc implies that

$W \vdash (h, X^0, c^0, b)$ for WDAcc_0. Now, since WDAcc_0 is a correct accumulator, $W \vdash (h, X^0, c^0, b)$ implies $\mathrm{Verify}(K_p, c^0, h, W)$, which in turn implies $\mathrm{Verify}(K_p, c, x, W)$.

Moreover, as the signature scheme is only binding elements of the two domains together, any incoherent witness with respect to the domain S of WDAcc produces an incoherent witness for a corresponding element in the domain P. More precisely, we can reduce an adversary $\mathcal{A}$ who can find an incoherent proof with respect to WDAcc to an adversary $\mathcal{B}$ who produces an incoherent proof with respect to WDAcc_0. We now outline this reduction.

According to the security game of Def. 6 for WDAcc, KeyGen is run and $K_p = (K_p^0, K_p^{\mathrm{sig}})$ is given to the adversary. Initially, X is empty and the accumulator is set to $\mathrm{InitAccVal}(K_s, K_p)$. In particular, the value of the accumulator c is (c^0, h_{init}), where c^0 is the output of $\mathrm{InitAccVal}^0(K_s^0, K_p^0)$. Then $\mathcal{A}$ can call the $\mathrm{AddEle}(K_s, K_p, c, ., .)$ oracle which updates X and c accordingly, but is not allowed to add an x to X when x is already in X. In particular, each AddEle query lets $h := \mathrm{next.element(last.h)}$, calls $\mathrm{AddEle}^0(K_s^0, K_p^0, c, h) \rightarrow (c^0, \mathrm{extra}_0)$, computes $\sigma := \mathrm{sig}(K_s^{\mathrm{sig}}, x, h)$, and returns $((c^0, h), (\sigma, \mathrm{extra}_0))$. After enough oracle queries, the adversary ends the game by producing some (x, W) which is an incoherent proof for x with respect to X and accumulator c, where $W = (x, \sigma_x, w)$.

Now let's look at the game played by $\mathcal{B}$. Firstly, KeyGen_0 is run and K_p^0 is given to the adversary. Initially, X^0 is empty and c^0 is set to $\mathrm{InitAccVal}^0(K_s^0, K_p^0)$. The adversary calls $\mathrm{AddEle}^0(K_s^0, K_p^0, c^0, ., .)$ oracles to update X^0 and c^0 accordingly, but is not allowed to add an x^0 to X^0 when x^0 is already in X^0. Once the adversary has made enough queries, she produces some (x^0, w). She wins if w is an incoherent proof for x^0 with respect to X^0 and accumulator c^0.

We are now going to use $\mathcal{A}$ to help $\mathcal{B}$ win his game. Upon receiving K_p^0, $\mathcal{B}$ runs $\mathrm{KeyGen}_{\mathrm{sig}}(1^k)$ and obtains $(K_s^{\mathrm{sig}}, K_p^{\mathrm{sig}})$. Then provides $\mathcal{A}$ with $K_p = (K_p^0, K_p^{\mathrm{sig}})$. For every $\mathrm{AddEle}(K_s, K_p, c, ., .)$ oracle query of $\mathcal{A}$, $\mathcal{B}$ does the following. He lets $h := \mathrm{next.element(last.h)}$, calls $\mathrm{AddEle}^0(K_s^0, K_p^0, c, h) \rightarrow (c^0, \mathrm{extra}_0)$, computes $\sigma := \mathrm{sig}(K_s^{\mathrm{sig}}, x, h)$, and returns $((c^0, h), \sigma, (\mathrm{extra}_0))$ to $\mathcal{A}$. Note that $\mathcal{B}$ posses K_s because he ran $\mathrm{KeyGen}_{\mathrm{sig}}(1^k)$ at the beginning of this reduction. Finally, $\mathcal{A}$ provides $\mathcal{B}$ with some (x, W) which is an incoherent proof for x with respect to X and accumulator c. Note that witnesses of WDAcc are of the form (h, σ, w), where $\sigma := \mathrm{sig}(K_s^{\mathrm{sig}}, x, h)$ and the accumulator c is of the form $((c^0, h), (\sigma, \mathrm{extra}_0))$. Hence, an incoherent witness (h, σ, w) of x in WDAcc implies an incoherent witness W of h in WDAcc_0. □

4. FULLY DYNAMIC UNIVERSAL ACCUMULATORS

In this section we formalize the notion of fully dynamic universal accumulator, then show how to construct some based on the [LLX] accumulator and a weak dynamic accumulator.

4.1. Definitions

We say that a dynamic universal accumulator is *fully dynamic* if the following conditions are satisfied:

1. We can always create a non-membership proof for a new x (*i.e.*, a value x occurring for the first time or a newly deleted x) without using X.

2. We can create a proof of membership without using X for a newly added x.

That is, we can create proofs for newly occurring values x (*e.g.*, non-members) with an algorithm which does not depend on the cardinality of *X*. To make this change possible, we introduce a new operation dec in UpdEle to "declare" new non-members, and we make UpdWit run with $x = y$ without any prior witness W for operations $+$ and dec. This UpdEle can be used to compute the initial proof of non-membership for x. However, it requires to update the accumulator value *c*. More formally, we define a fully dynamic universal accumulator as follows.

Definition 8 (Fully Dynamic Universal (FDU) Accumulator). A fully dynamic universal accumulator FDUAcc*, with a domain P, a set* $X \subseteq P$*, and values* $x \in X$ *to be accumulated, consists of the following algorithms.*

– A setup probabilistic algorithm $\mathrm{KeyGen}(1^k) \to (K_s, K_p)$*, where* K_s *is only used by the authority and* K_p *is public.*

– An algorithm $\mathrm{AccVal}(K_s, K_p, X) \to c$*, which computes an accumulator value* c *of the set X, from the keys.*

– An algorithm $\mathrm{UpdEle}(K_s, K_p, c, \mathrm{op}, x) \to (c', \mathrm{extra})$*, where* $\mathrm{op} = +$*,* $\mathrm{op} = -$*, or* $\mathrm{op} = \mathrm{dec}$*, which computes the accumulator* c' *from an accumulator* c*. When* $\mathrm{op} = +$*, we must have* $x \notin X$ *and we say that* x *is inserted into X. When* $\mathrm{op} = -$*, we must have* $x \in X$ *and we say that* x *is deleted from X. When* $\mathrm{op} = \mathrm{dec}$*, we must have* $x \notin X$ *and we say that* x *is declared as a non-member of X. The algorithm also returns some extra information* extra*. For* $\mathrm{op} = \mathrm{dec}$*, the algorithm also returns some new proof of non-membership for* x *(if not already in* extra*).*

– An algorithm $\mathrm{WitGen}(K_s, c, X, x) \to W$ *to generate a proof of membership or non-membership for the value* x *with respect to accumulator* c *of X.*

– An algorithm $\mathrm{UpdWit}(K_p, \mathrm{extra}, W, y) \to W'$ *to generate a proof* W' *for* y *in accumulator after* UpdEle *returned* extra *from a previous proof W.*

– A predicate $\mathrm{IsMem}(W)$ *telling whether* W *is a proof of membership (true case) or a proof of non-membership (false case).*

– A predicate $\mathrm{Verify}(K_p, c, x, W)$ *to check a proof.*

Note that UpdWit no longer requires that y is different from the element involved in the last UpdEle call.

The correctness notion is similar to that of WD accumulators with the obvious change that the witnesses are not only produced by UpdWit, but also by WitGen and that witnesses are either for membership or non-membership proofs. This change was foreseen in our general correctness notion in Def. 3.

Next, we detail a variant of the Chosen Element Attack scenario corresponding to FDU accumulators, in which the adversary is allowed to declare new elements as well as add or delete.

Definition 9 (Extended Chosen Element Attack (ECEA) Model). *The security of an FDU accumulator is defined in terms of a game, based on a security parameter k, played by a polynomially bounded adversary. Firstly,* KeyGen *is run and* K_p *is given to the adversary. Secondly, the adversary selects a polynomially bounded number* ℓ*. There are registers* K_s*,* X_i*, and* c_i*,* $i = 1, \ldots, \ell$*, for a secret key and to keep track of* ℓ *sets* X_i *and their accumulator values* c_i*. Initially, all* X_i*'s are empty and* c_i *is set to* $\mathrm{AccVal}(K_s, K_p, X_i)$*. The adversary can then call an* $\mathrm{UpdEle}(K_s, K_p, c_i, ., .)$ *oracle for a selected* i

which updates X_i and c_i accordingly. It is not allowed to add an x to X_i when x is already in X_i, nor is it allowed to delete x from X_i when x is not in X_i. Moreover, the adversary is not allowed to declare an element which has already been declared, i.e., is either a member or a non-member. The adversary can also call a $\mathrm{WitGen}(K_s, c_i, X_i, .)$ *oracle for a selected i and an* $\mathrm{AccVal}(K_s, K_p, .)$ *oracle which do not update an X_i of c_i. After making many oracle queries, the adversary ends by producing some (i, x, W). The adversary wins if W_X is an incoherent proof for x with respect to X_i and accumulator c_i.*

4.2. Instantiating an FDU Accumulator based on the Strong RSA Assumption

Below we construct a fully dynamic universal accumulator based on a variant of the [LLX] accumulator. The main idea relies on providing a two-layer accumulator. The lower layer c_2 is a WD accumulator where we accumulate all declared elements. *I.e.*, all values which have ever be used, whether they are member or not. There is no withdrawal in this layer. The upper layer c_1 is a fully dynamic accumulator which can only treat elements of the lower layer. Essentially, (a, d, h, w) is a proof of non-membership for x in accumulator (c_1, c_2), if $c_1^a \equiv d^x h (\bmod n_1)$ and w is a proof of membership for h in accumulator c_2, *i.e.*, $w^h \bmod n_2 = c_2$. To create a proof of non-membership for a newly deleted member or a new comer who is not a member, we only have to pick a random $a \in \mathrm{Z}_x$ and $d \in \mathrm{Z}_n^*$ and compute the corresponding h to add in the second accumulator. That is, the WD accumulator is only used to validate new h values which are needed to create new proofs. Since proofs also have a part in the PDU accumulator, it is not necessary to delete h from the WD accumulator.

Equation (3) is now replaced by

$$a = a_0 \frac{\prod_{y \in X_0} y}{\prod_{y \in X} y} \bmod x \quad \mathit{and} \quad d = \left(c_1^a h^{-1}\right)^{\frac{1}{x}} \bmod n_1, \tag{4}$$

where a_0, respectively X_0, is the initial value for a, respectively X, and a_0 is chosen at random.

Our accumulator is defined as follows. The value of c is defined by $c = (c_1, c_2)$ corresponding to two accumulators c_1 and c_2. It will be convenient in the security game to define two sets X_1, defined as before, and X_2, a finite set of elements, corresponding to this value c.

The set X_1 is updated by UpdEle and is accumulated in the value c_1, whereas the set X_2 is accumulated in c_2 by means of a WD accumulator with arbitrary domain. Hence, the value of c is deterministically defined by c_1 and c_2, where

$$c_1 = g^{\prod_{x \in X_1} x} \bmod n_1.$$

The value of c will corresponds to $X = X_1$. So, there are many c's corresponding to the same X depending on X_2.

In the following description, the algorithms, values, and predicates with index of 1, *e.g.*, KeyGen_1, refer to the corresponding items in the PDU accumulator of Li *et al.* [7], whereas the algorithms, values, and predicates with index of 2, *e.g.*, Verify_2, refer to those of a WD accumulator with arbitrary domain as defined in Section 3.

A Concrete FDU accumulator:
Domain: P is a large enough set of odd prime numbers, *e.g.*, all odd numbers up to a given bound B.
KeyGen(1^k): run $\text{KeyGen}_1(1^k)$ and obtain $K_p^1 = (n_1, g_1)$ and $K_s^1 = r_1$. Further, run $\text{KeyGen}_2(1^k)$ and obtain (K_s^2, K_p^2). Then $K_p = (K_p^1, K_p^2)$ and $K_s = (K_s^1, K_s^2)$.
AccVal(K_p,X): compute

$$c_1 = g^{\prod_{x\in X} x} \bmod n_1$$

and return $c = (c_1, \text{InitAccVal}_2(K_s^2))$. Note that K_s is not used.
UpdEle(K_s, Kp, c, op, x): if $\text{op} = +$, we have $c_1' = c_1^x \bmod n_1$ and $c_2' = c_2$ (K_s^1 is not required). Set $\text{extra} = (c, +, x, c')$. If $\text{op} = -$, we set $c_1' = c_1^{\frac{1}{x}} \bmod n_1$ (K_s^1 is required) and proceed like for $\text{op} = \text{dec}$. If $\text{op} = \text{dec}$, we pick $a \in \mathrm{Z}_x$ and $d \in \mathrm{Z}_{n_1}^*$ at random, and compute $h = c_1^a d^{-x} \bmod n_1$, then we set $W = (\text{nonmem}, (a, d, h, c_2))$. We set $(c_2', \text{extra}_2) = \text{AddEle}_2(K_s^2, K_p^2, c_2, h)$. The extra information is then set to $\text{extra} = (c, \text{op}, x, c', W, \text{extra}_2)$.
WitGen(K_s, c, X, x): if $x \in X$, then $W = (\text{mem}, w)$ with $w = c_1^{\frac{1}{x}} \bmod n_1$. This requires K_s, but not X. It can also be computed without K_s, but using X by observing that

$$w = c_1^{\prod_{y\in X-\{x\}} y} \bmod n_1.$$

If $x \notin X$, then $W = (\text{nonmem}, (a, d))$ as in the [LLX] accumulator by using X.
UpdWit(K_p, extra, W,y): with $\text{extra} = (c, \text{op}, x, c', e, \text{extra}_2)$, there a several cases to update the proof for y in X after adding or deleting x for $x \neq y$:
– W of the form (mem, w) and x just added ($\text{op} = +$): set $W' = (\text{mem}, w')$ with $w' = w^x \bmod n_1$.
– W of the form (mem, w) and x just deleted ($\text{op} = -$): set $W' = (\text{mem}, w')$ with $w' = w^z c'^{\frac{1-xz}{y}} \bmod n_1$ and $z = \frac{1}{x} \bmod y$.
– W of the form (mem, w) and x just declared ($\text{op} = \text{dec}$): $W' = W$.
– W of the form (nonmem, a, d) and x just added ($\text{op} = +$): set $W' = (\text{nonmem}, [a', d']_y)$ with $a' = az$, $d' = dc^{-a\frac{1-xz}{y}} \bmod n_1$ and $z = \frac{1}{x} \bmod y$.
– W of the form (nonmem, a, d) and x just deleted ($\text{op} = -$): set $W' = (\text{nonmem}, [a', d]_y)$ with $a' = ax$.
– W of the form (nonmem, a, d) and x just declared ($\text{op} = \text{dec}$): $W' = W$.
– W of the form $(\text{nonmem}, a, d, h, w)$ and x just added ($\text{op} = +$): set $W' = (\text{nonmem}, [a', d']_y, h, w)$ with $a' = az$, $d' = dc^{-a\frac{1-xz}{y}} \bmod n_1$ and $z = \frac{1}{x} \bmod y$.

– W of the form $(\text{nonmem}, a, d, h, w)$ and x just deleted ($\text{op} = -$): set $W' = (\text{nonmem}, [a', d]_y, h, w')$ with $a' = ax$ and $w' = \text{UpdWit}_2(K_p^2, c_2, c_2', \text{extra}_2, w, h)$.

– W of the form $(\text{nonmem}, a, d, h, w)$ and x just declared ($\text{op} = \text{dec}$): $w' = \text{UpdWit}_2(K_p^2, c_2, c_2', \text{extra}_2, w, h)$.

For $x = y$, there are two cases:

– x just added ($\text{op} = +$): set $W' = (\text{mem}, w)$ with $w = c_1$ from $c = (c_1, c_2)$.

– x just deleted ($\text{op} = -$) or just declared ($\text{op} = \text{dec}$): set $W' = e$.

IsMem(W): is true if and only if W is of form $(\text{mem}, .)$.

Verify(*Kp, c,x,W*): if W is of the form (mem, w), it is true if and only if $c_1 = w^x \bmod n_1$. If W is of form (nonmem, a, d), it is true if and only if $c^a \equiv d^x g (\bmod n_1)$. If W is of the form $(\text{nonmem}, a, d, h, w)$, it is true if and only if $c_1^a \equiv d^x h (\bmod n_1)$ and $\text{Verify}_2(K_p^2, c_2, w, h)$ holds.

We now prove the correctness and security of our FDU accumulator scheme based on the Strong RSA assumption.

Theorem 10. *If the Strong RSA Assumption holds, the aforementioned FDU accumulator is correct and secure under the ECEA model.*

Proof. The correctness follows immediately since both PDU and WDU considered as building blocks of our FDU are correct accumulators according to Def. 3.

All oracle calls in the security game preserve the relations described in the AccVal and WitGen algorithms. Furthermore, since the game does not allow the adversary to add a member or to delete a non-member, all oracle calls in the game can be simulated without knowing K_s^1. Hence, the security is equivalent to forging $X \subseteq P$ and $x \in P$ together with W which is an incoherent proof for x with respect to X and K_p as the only input. Hence, an incoherent witness implies breaking the strong RSA assumption. In other words, computing an incoherent witness for $x \in X$ implies an incoherent witness for the PDU accumulator of Li *et al.* [7] and computing an incoherent witness for $x \notin X$ implies that, given n_1 and a random c_1 drawn from $\mathbf{Z}_{n_1}^*$, the adversary has found $w \in \mathbf{Z}_{n_1}^*$ and $x > 1$ such that $c_1 = w^x \bmod n_1$.

More precisely, we can reduce an adversary $\mathcal{A}$ who produces incoherent proofs in the aforementioned FDU accumulator to an adversary $\mathcal{B}$ who produces incoherent proofs in either the WD accumulator with arbitrary domain as defined in Section 3 or the PDU accumulator of Li *et al.* [7], which, in turn, implies an adversary who breaks the Strong RSA Assumption. The reduction is detailed below.

According to Def. 9, the security game starts with running KeyGen and giving K_p to the adversary. That is, both KeyGen_1 and KeyGen_2 are run to obtain $K_p^1 = (n_1, g_1), K_s^1 = r_1$, and (K_s^2, K_p^2). Then, $K_p = (K_p^1, K_p^2)$ is given to the adversary. Initially, X is empty and c is set to $\text{AccVal}(K_p, X)$. That is, $c_1 = g^{\prod_{x \in X} x} \bmod n_1$ is computed and $c = (c_1, \text{InitAccVal}_2(K_s^2))$ is returned. Note that K_s is not used in this computation.

Then, the adversary calls $\text{UpdEle}(K_s, K_p, c, ., ., .)$ oracle queries to update X and c accordingly. She is not allowed to add an x to X when x is already in X, nor is she allowed to delete x from X when x is not in *X*. Moreover, the adversary is not allowed to declare an element which has already been declared, *i.e.*, is either a member or a non-member. When an element x is being added, we have $c_1' = c_1^x \bmod n_1$, *i.e.*, using the PDU accumulator of Li *et al.* [7], and $c_2' = c_2$, *i.e.*, in the WDU accumulator. If x is being declared, random $a \in \mathrm{Z}_x$ and $d \in \mathrm{Z}_{n_1}^*$ are picked to compute $h = c_1^a d^{-x} \bmod n_1$, again as in the PDU accumulator of Li *et al.* [7], to obtain $W = (\mathrm{nonmem}, (a,d,h,c_2))$. If the element x is being deleted, we have $c_1' = c_1^{\frac{1}{x}} \bmod n_1$ and proceed like the declaration process. Note that K_s^1 is not required in any of these steps.

The adversary can also call a $\mathrm{WitGen}(K_s, c, X, .)$ oracle query. If $x \in X$, then $W = (\mathrm{mem}, w)$ with $w = c_1^{\prod_{y \in X - \{x\}} y} \bmod n_1$, which can be computed without K_s, but using *X*. If $x \notin X$, then $W = (\mathrm{nonmem}, (a,d))$ as in the PDU accumulator of Li *et al.* [7]. Note that, again, K_s is not required in any of these steps.

Once the adversary has made enough oracle queries, she ends the game by producing some (x, W) that is an incoherent proof for x with respect to X and accumulator $c = (c_1, c_2)$. The witness W is of the form (mem, w), (nonmem, a, d), or $(\mathrm{nonmem}, a, d, h, w)$. We are going to consider each case separately.

– If W, the incoherent proof, is of the form (mem, w), then, by definition of the verification algorithm Verify, we must have that $c_1 = w^x \bmod n_1$, which directly breaks the Strong RSA assumption.

– In order for an incoherent witness W of form (nonmem, a, d) to pass the verification step, we must have that $c^a \equiv d^x g (\bmod n_1)$. This translates to an incoherent witness for the PDU accumulator of Li *et al.* [7].

– If the incoherent witness W is of the form $(\mathrm{nonmem}, a, d, h, w)$. Then, both $c_1^a \equiv d^x h (\bmod n_1)$ and $\mathrm{Verify}_2(K_p^2, c_2, w, h)$ must hold for it to pass the verification step. This translates to either an incoherent witness for the PDU accumulator of Li *et al.* [7] or an incoherent proof for the WD accumulator of Section 3.

Hence, an adversary who can find incoherent witnesses for our FDU accumulator is capable of producing incoherent proofs for the WD accumulator with arbitrary domain as defined in Section 3 or the PDU accumulator of Li *et al.* [7], both of which are based on the Strong RSA assumption.

□

Note that setting $g = h$ reduces the structure of our non-membership proofs to that of Li *et al.* [7].

We point out that the efficacy of our proof structure allows the authority to perform efficient batch updates (with the secret key) for a given value x. The authority first checks to see whether x is a member of the accumulated set or not. If a member, then using the same procedure as in the scheme of Li *et al.* the authority can efficiently update the witness. This is not incompatible with the impossibility of batch update without the secret key [3]. However, the scheme of Li *et al.* did not offer such a mechanism for a non-member element. In our scheme, we can create a new non-membership proof deploying the mechanism for declaration.

5. CONCLUSIONS AND FUTURE WORK

We constructed the first *fully dynamic* universal accumulator, based on the Strong RSA assumption, by providing a new proof structure for the non-membership witnesses. Moreover, this new structure of non-membership proofs allows our scheme to be the first of its kind to offer an *efficient batch update* mechanism to the authority, for both members and non-members. We obtained our fully dynamic universal accumulator by means of deploying a weak dynamic accumulator with arbitrary domain, which we showed how to obtain from a weak dynamic accumulator with a domain of certain form.

ACKNOWLEDGEMENTS

We would like to thank Nokia for initiating this work, as well as Rafik Chaabouni who contributed.

A. EXTRA DEFINITIONS

Definition 11 (Dynamic Accumulators). A dynamic accumulator DAcc, *with a domain* P*, a set* $X \subseteq P$*, and values* $x \in X$ *to be accumulated, consists of the following algorithms.*

– A setup probabilistic algorithm $\mathrm{KeyGen}(1^k) \to (K_s, K_p)$*, where* K_s *is only used by the authority and* K_p *is public.*

– An algorithm $\mathrm{AccVal}(K_s, K_p, X) \to c$*, which computes an accumulator value* c*.*

– An algorithm $\mathrm{UpdEle}(K_s, K_p, c, \mathrm{op}, x) \to (c', \mathrm{extra})$*, where* $\mathrm{op} = +$ *or* $\mathrm{op} = -$*, which computes the accumulator* c' *for* $X\mathrm{op}\{x\}$ *from the accumulator* c *for* X*. When* $\mathrm{op} = +$*, we must have* $x \notin X$ *and we say that* x *is inserted into* X*. When* $\mathrm{op} = -$*, we must have* $x \in X$ *and we say that* x *is deleted from* X*. The algorithm also returns some extra information* extra*, which might be needed for dynamic witness update.*

– An algorithm $\mathrm{WitGen}(K_s, c, X, x) \to W$ *to generate a proof of membership for the value* x *with respect to accumulator* c *of* X*.*

– An algorithm $\mathrm{UpdWit}(K_p, c, c', \mathrm{extra}, \mathrm{op}, x, W, y) \to W'$ *to generate a proof* W' *for* y *in accumulator* c' *from a proof* W *for* y *in accumulator* c*, where* $\mathrm{UpdEle}(K_s, K_p, c, \mathrm{op}, x) \to (c', \mathrm{extra})$*. It must be the case that* $x \neq y$*.*

– A predicate $\mathrm{Verify}(K_p, c, x, W)$ *to check a proof.*

Definition 12 (Universal Accumulators). A universal accumulator *scheme* UAcc*, with a domain* P *a set* $X \subseteq P$*, and values* $x \in X$ *to be accumulated, consists of the following algorithms.*

– A setup probabilistic algorithm $\mathrm{KeyGen}(1^k) \to (K_s, K_p)$*, where* K_s *is only used by the authority and* K_p *is public.*

– An algorithm $\mathrm{AccVal}(K_s, K_p, X) \to c$*, which computes an accumulator value* c*.*

– An algorithm $\mathrm{MemWitGen}(K_s, c, X, x) \to W$ *to generate a proof of membership for* $x \in X$*.*

– An algorithm $\mathrm{NonMemWitGen}(K_s, c, X, x) \to W$ *to generate a proof of non-membership for* $x \in P \setminus X$*.*

– A predicate $\mathrm{IsMem}(W)$ *telling whether* W *is a proof of membership (true case) or a proof of non-membership (false case).*

– A predicate $\mathrm{Verify}(K_p, c, x, W)$ *to check a proof.*

REFERENCES

1. Josh Cohen Benaloh and Michael De Mare, *One-way accumulators: A decentralized alternative to digital sinatures (extended abstract)*, in EUROCRYPT 1993, pp. 274–285, 1993.
2. Niko Barić and Birgit Pfitzmann, *Collision-free accumulators and fail-stop signature schemes without trees*, in Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques, EUROCRYPT 1997, pp. 480–494, Berlin, Heidelberg, 1997. Springer-Verlag.
3. Philippe Camacho and Alejandro Hevia, *On the Impossibility of Batch Update for Cryptographic Accumulators*, LATINCRYPT 2010, volume 6212 of Lecture Notes in Computer Science, pp. 178–188, 2010.

4. Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente, *An accumulator based on bilinear maps and efficient revocation for anonymous credentials* Stanislaw Jarecki and Gene Tsudik, editors, PKC 2009, volume 5443 of Lecture Notes in Computer Science, pp. 481–500, Springer, 2009.
5. Jan Camenisch and Anna Lysyanskaya, *Dynamic accumulators and application to efficient revocation of anonymous credentials*, in Moti Yung, editor, CRYPTO 2002, volume 2442 of Lecture Notes in Computer Science, pp. 61–76. Springer, 2002.
6. Nelly Fazio and Antonio Nicolosi, *Cryptographic Accumulators: Definitions, constructions and applications*, Manuscript, 2003.
7. Jiangtao Li and Ninghui Li and Rui Xue, *Universal Accumulators with efficient nonmembership proofs*, Jonathan Katz and Moti Yung, editors, ACNS 2007, volume 4521 of Lecture notes in Computer Science, pp. 253–269. Springer, 2007.
8. Lan Nguyen, *Accumulators from bilinear pairings and applications*, Alfred Menezes, editor, CT-RSA 2005, volume 3376 of Lecture Notes in Computer Science, pp. 275–292. Springer, 2005.
9. Kaisa Nyberg, *Fast Accumulated Hashing*, FSE 1996, volume 1039 of Lecture Notes in Computer Science, pages 83-87. Springer, 1996.
10. Kun Peng and Feng Bao, *Vulnerability of a non-membership proof scheme*, SECRYPT 2010, pages 419-422. SciTePress, 2010.
11. Peishun Wang, Huaxiong Wang, and Josef Pieprzyk, *A new dynamic accumulator for batch updates*, Sihan Qing, Hideki Imai, and GuilinWang, editors, ICICS 2007, volume 4861 of Lecture Notes in Computer Science, pp. 98–112. Springer, 2007.
12. Peishun Wang, Huaxiong Wang, and Josef Pieprzyk, *Improvement of a dynamic accumulator at icics 07 and its application in multi-user keyword-based retrieval on encrypted data*, in APSCC 2008, pp. 1381–1386. IEEE, 2008.

Received July 15, 2013