Efficient Constructions of Pairing Based Accumulators

Ioanna Karantaidou
George Mason University
ikaranta@gmu.edu

Foteini Baldimtsi
George Mason University
foteini@gmu.edu

***Abstract*—Cryptographic accumulators are a crucial building block for a variety of applications where you need to represent a set of elements in a compact format while still being able to provide proofs of (non)membership. In this work, we give a number of accumulator constructions for the bilinear pairing setting in the trapdoor-based scenario, where a trusted manager maintains the accumulator. Using modular accumulator techniques, we first present the *first* optimally efficient (in terms of communication cost) dynamic, positive accumulators in the pairing setting. Additionally, we present a novel modular approach to construct universal accumulators that avoid costly non-membership proofs. We instantiate our generic construction and present the *first* universal accumulator in the bilinear pairing setting, that achieves constant parameter size, constant cost for element additions/deletions and witness generation by the manager, constant witness updates by the users and constant (non)membership verification. We finally show how our proposed universal accumulator construction can give rise to efficient ZK accumulators with constant non-membership witness updates.**

I. Introduction

Cryptographic accumulators, originally introduced in [9], allow the compact representation of an arbitrarily large set of elements S. Given a *membership witness* w for an accumulated element x and the current value of the accumulator v, it can be efficiently verified that the element x is a valid member of the accumulated set. The accumulated set can change over time as elements are added and deleted; accumulators that support both additions and deletions are called *dynamic*, while the ones that only support additions or deletions are called *additive* and *subtractive* respectively. Additionally, an accumulator is called *positive* if it only supports membership proofs, *negative* if it supports non-membership proofs, and *universal* if it supports both membership and non-membership proofs.

Accumulators are an important building block for a variety of applications where a compact representation of a set is needed while at the same time element (non)membership should be provable. Access control is a characteristic application where accumulators can be used to improve efficiency. Traditionally, a permitlist (or blocklist) is used to maintain the set of users with access to a system. Unfortunately though, the size of such lists is linear to the number of valid (or revoked) users. Dynamic accumulators, that support additions and deletions, can offer access control with much smaller costs. Similarly, for the case of anonymous access control (anonymous credentials), accumulators are commonly used to support more efficient anonymous proofs of membership [16], [4], [1], [15]. Beyond access control, common applications of accumulators include group signatures [35], [30] to maintain the list of valid group members, and anonymous cryptocurrencies [8], [38], [12] to compact the information posted on the ledger.

Related Work. Given their widespread use, a variety of accumulator constructions exist and can be categorized based on the underlying security assumptions, setup assumptions, offered functionalities (i.e dynamic, positive, universal etc.), as well as the security level. The most common types of accumulators, in terms of underlying assumptions, are the RSA-based constructions [9], [16], [29] and the bilinear pairing (BP) based constructions [34], [15], [20], [4]. More recently, there have also been proposals in the lattice-setting, such as [30], [26]. At the same time, the standard security properties of accumulators (correctness and soundness) have been formalized in [21], [37], [6], while a universally composable (UC) formalization was recently given in [7] and additional security properties such as zero-knowledge (ZK) were introduced in [23]. Vector commitment (VC) schemes can also be adapted and used as universal cryptographic accumulators, as shown in [19], and relevant constructions exist both in the RSA [19], [27] and BP setting [19], [27], [24], [39], [31].

Another important categorization is in regards to the required setup. There exist two main categories: "trapdoor"-based accumulators managed by a trusted party and "trapdoorless" or else "strong" accumulators. In a trapdoor-based accumulator [15], [6], a trusted party, often called *accumulator manager*, holds some secret information/trapdoor that allows it to efficiently add or remove elements and create witnesses. When a new element is added in the accumulator, the accumulator manager issues the corresponding membership witness w. The manager is responsible to issue an update message whenever a new element is added (or deleted) in the accumulator so that existing witness holders can update their witnesses. Trapdoor-based accumulators are commonly used in the anonymous credential scenario [15], [16], [1], [28], [22], where the credential issuer often serves as the accumulator manager, to allow for efficient revocation while preserving privacy. Trapdoor-based accumulators were also used in the first construction of zero-knowledge accumulators [23].

Trapdoorless, or else "strong" accumulators [14], [32] allow for public additions or deletions of elements without the

need for a trusted party. Users adding new elements to the accumulator can compute (and later update) their corresponding witnesses themselves. Most constructions of trapdoorless accumulators require a trusted setup phase [12], [4], [40]. Trapdoorless accumulators are mostly used in applications such as stateless blockchains [12], authenticated database queries [42], authenticated logs [40], etc., where multiple miners/cloud servers/CAs exist and need to perform updates and prove correctness of their operations on accumulated data that belong to different owners. Accumulators derived from vector commitment schemes typically belong in the trapdoorless setting.

The importance of trapdoor-based accumulators. In general, trapdoor-based accumulators are more efficient than trapdoorless ones, since the accumulator manager can utilize its secret trapdoor to allow for more efficient additions/deletions and witness creations. Thus, in application scenarios where a trusted party is already required (i.e. an issuer in anonymous credentials), the use of trapdoor-based accumulators can give rise to more efficient schemes. In this work, we focus on the trapdoor-based setting and include a trusted accumulator manager in our functionality. We note that trapdoorless accumulators which require a trusted setup phase, can be converted to trapdoor-based ones by replacing the trusted setup phase by an accumulator manager that holds the secret trapdoor.

In the trapdoor-based setting, a recent work by Baldimtsi et.al. [6] gives modular, generic constructions of accumulators and shows that the combination of accumulator schemes with different functionalities or security properties can yield to new constructions with improved efficiency or functionality sets. Beyond their generic modular framework, [6] explored the case of RSA-based accumulators and provided a new accumulator construction (called *Braavos*), which is the most efficient positive, RSA-based accumulator in terms of *communication cost of update messages*. The traditional trapdoor RSA-based accumulator required a communication cost of $|a+d|$ messages (where a and d are the numbers of added and deleted elements respectively). This communication cost corresponds to all the messages the manager has to broadcast to witness holders such that they can keep their witnesses up-to-date after accumulator updates, i.e. after other elements get added or deleted. The Braavos construction of [6] reduced the communication cost of $|d|$ which matches the lower bound for communication cost (for positive accumulators) given by Camacho [13][1]. We note that communication cost is an important metric of accumulator efficiency since in real world settings it is crucial to minimize the amount of information that needs to be sent between users and the accumulator manager.

The importance of BP accumulators. Although modular constructions turned out to provide efficiency gains for RSA based accumulators, the case of pairing based modular constructions remains unexplored. At a high level, the most popular BP accumulator[34] works as follows: to construct the accumulator value, one has to exponentiate a prime order group generator to the product of monomials $(x+a)$ where x are the accumulated elements and a is a secret trapdoor. The membership proof for element y is the remaining product in the exponent if we remove the term $(y+a)$.

Pairing accumulators are used as widely as RSA based ones. In particular, are commonly used on top of systems built on known-order groups which are known to have better group operation performance (smaller group sizes, shorter elements) and are supported by bilinear pairings. Such systems include anonymous credential systems, group signatures, authenticated logs, etc. [15], [2], [5], [18], [33], [35], [43], [42], [36], [38]. Finally, RSA based accumulators are limited to only being able to accumulate *prime* elements. Although one can apply a function that maps to primes (commonly implemented via a hash-to-prime operation), this makes ZK proofs of membership more complicated due to the non-algebraic format of the hash function. As an indication, [10] computes that proving membership for elements of 252 bits in an RSA accumulator blows from 54.51ms to 479.50ms when conversion to prime is needed (cases where the domain is not the prime numbers set). CRS size also grows from 86KB to 6852KB. Thus, a BP accumulator can also be preferable in applications where arbitrary elements are being accumulated.

A. Our Contributions

In this work, we deploy modular constructions to design efficient trapdoor-based bilinear accumulators.

We start by giving two constructions of *positive*, dynamic accumulators. As discussed above, positive accumulators allow users to create membership proofs, i.e. prove that an element belongs to the accumulated set S. Our constructions are the *first* in the bilinear pairing setting to achieve the optimal communication. Additionally, our constructions are the first to achieve the *join-revoke unlinkability* property [6] against users/witness holders in the bilinear setting. Our next constructions focus on optimizing both the parameter size and the computation costs in *universal* accumulators by taking advantage of the presence of a trusted manager. Our third construction is the *first* universal accumulator in the bilinear pairing setting, that achieves constant parameter size, constant cost additions/deletions and witness generation by the manager, constant witness updates by the users and constant (non)membership verification. Our final construction adds the set zero-knowledge property of [23] by not increasing any of the previously mentioned asymptotic costs.

Construction 1: Non-adaptively sound positive dynamic accumulator. We present our first construction in Section IV. Our goal is to achieve the optimal communication cost of witness updates which is $|d|$, i.e. linear to the number of deleted elements. In order to achieve this optimal communication cost we need to keep the accumulator value static when new elements are being added. Using the construction of [34] as

[1]In [13] it was shown that full batching of update messages in accumulators is impossible, since the lower communication bound for keeping a membership witness up-to-date equals $|d|$ (i.e. the number of deletions). [6] computed the equivalent bound for non-membership to be $|a|$ (i.e. the number of additions), which brings the lower communication bound for universal accumulators to $|a+d|$.

a starting point, we adapt the addition protocol so that on additions of new elements we keep the accumulator value fixed and instead we add the inverse of the added element x in the product (which describes the accumulator value in the bilinear pairing setting) in order to construct a witness for the newly added element. We note that such a change in the addition process makes the security analysis of our first construction quite different from [34].

Updates only during deletions keep the communication cost equal to the number of deletions which is optimal for a positive accumulator [13]. While achieving the optimal communication cost for pairing-based, positive, dynamic accumulators, our first construction is not adaptively sound, as opposed to [34]. Using this first construction as building block we then build an adaptively sound accumulator with the same properties.

Construction 2. Adaptively sound positive dynamic accumulator. In Section V, we take a modular construction approach and present a positive dynamic accumulator as a combination of two accumulators with different security properties (inspired by construction H, Figure 4 in [6]). More specifically, we combine our first construction of a non-adaptively sound accumulator with a positive, adaptively sound, additive accumulator. The final construction is a positive, adaptively sound, dynamic accumulator. The overall adaptively sound construction achieves the minimum communication bound for positive accumulators. This is true because the additive accumulator requires no updates for membership proof and the dynamic accumulator needs updates only in deletions as explained in the previous paragraph. This construction achieves join-revoke unlinkability against users/witness holders: given that users are not informed about additions, a revocation event is not linkable to a prior addition event.

We instantiate this generic construction in the pairing based setting by using the BB signature scheme of [11] (described in Section III-B) as a positive additive accumulator and, our Construction 1 as a non-adaptively sound positive dynamic accumulator. The resulting instantiation yields the *first optimally efficient*, in terms of communication cost, dynamic positive pairing based accumulator. Additionally, we also discuss why our construction supports efficient ZK proofs and thus can be used as a building block in privacy oriented applications.

Construction 3. Universal Dynamic Pairing Based Accumulator. In Section VI, we construct a *universal* accumulator as a modular construction of two positive accumulators. Existing constructions of traditional accumulators (both RSA and BP based) have better performance and shorter parameters in the trapdoor-based setting due to their limited expressiveness when compared to accumulator schemes derived from vector commitments. Their main overhead is in creating non-membership proofs since they are usually based on division properties and require non-constant computation for the accumulator manager. To overcome this issue we take a novel modular approach that completely saves us the cost of non-membership proofs. Specifically, we construct a universal accumulator by combining two positive accumulators where one keeps the accumulated elements and the second keeps the rest of the domain. We start by running a setup phase, where we add all possible elements in the domain are added in the second accumulator. Proving membership means that a user needs to show membership in the first accumulator, while proving non-membership again means that a user needs to show membership in the second accumulator. The resulting accumulator is universal (presented in Figure 3 as a generic construction) and we prove it to be adaptively sound.

We instantiate our generic construction for the bilinear pairing setting by using two instances of positive dynamic Nguyen accumulators [34]. As a result, by running a setup phase to accumulate all elements in the domain, we manage to reduce the high cost of non-membership operations of the original universal bilinear pairing construction of [4] from linear to the number of accumulated elements to constant. Our new modular construction technique could be also used beyond the bilinear setting.

In Table I of Section VII we provide an asymptotic comparison of our three first constructions with the state of the art pairing-based accumulators in the trapdoor-based setting. For completeness, in Table II we also compare with accumulator constructions that can be derived from vector commitment schemes which as discussed above generally suffer from higher parameter sizes.

Construction 4. Efficient ZK Accumulators. For our last construction we focus on pairing based accumulators with additional properties. Specifically, we look at the Zero-Knowledge (ZK) accumulator of [23]. A ZK accumulator hides the information about the accumulated set from parties that get to see membership/non-membership witnesses and the accumulator value. We show that using the modular approach to create universal bilinear accumulators can also lead to more efficient ZK accumulator constructions in terms of computation. We present our construction and prove it secure in Section VIII.

Our observation is that the ZK-accumulator instantiation of [23] requires non-membership proof generation which as argued above can be particularly inefficient. To make matters worse, [23] introduces a new randomized non-membership witness update algorithm to achieve privacy. The idea is that whenever an update happens, non-membership witnesses need to be created from scratch with fresh randomness. While typically non-membership witness updates are constant, the approach of [23] increases the cost to linear to the size of the accumulated set. To overcome this issue we use the same approach we took in our third construction by carefully replacing non-membership proofs with membership proofs achieving constant non-membership witness creation for the accumulator manager. Join-revoke unlinkability in ZK-accumulators is achieved against external observes/verifiers, since they are not receiving broadcast messages and the type of the update (addition/deletion) is not reflected on the accumulator value.

II. Accumulator Preliminaries

We start by setting the notation used in the rest of the paper. By λ we denote the security parameter and by negl$(\cdot)$ the

property that a function is negligible in some parameter. By $\langle g \rangle$ we denote a group $\mathbb{G}$ with generator g.

A. Accumulator Definitions

Trapdoor-based Setting. Following the definitions of [6], we consider the following types of parties: the *accumulator manager* that possesses a secret key sk (often appears as auxiliary/trapdoor information), *witness holders* that are responsible for elements present or not in the accumulator and *third parties* that are usually interested in verifying membership and non-membership proofs.

Below we describe the algorithms run by every type of party in an accumulator setting. After running $Gen()$ we consider the set of params to be publicly available and default input to the rest of the algorithms[2]. Accumulator managers can send *update messages* which include update information that witness holders need in order to keep their witnesses up to date after additions or deletions happen on the accumulated set. The usual format of an update message is: upmsg="the element y was added/deleted at time t".

By S we denote the set of currently accumulated elements. One could also derive S by combining all update messages up to the current time t, i.e. $\mathsf{upmsg}_{[1\ldots t]}$. The accumulator manager keeps some local memory, denoted as m_t which includes the set of currently accumulated elements S.
The following algorithms are performed by the accumulator manager:

- **Gen**(1^λ, S_0)→(sk, v_0, m_0, params)**:** Takes as input the security parameter λ and an initial set of elements S_0 ($S_0 = \emptyset$ if the accumulator starts as empty) and outputs the manager's secret key sk and initial memory m_0, the accumulator's initial value v_0, and the scheme's public params. The accumulator's domain D, (i.e. the type of elements that can be accumulated) is described in params.
- **Add**(sk, v_t, m_t, x) →(v_{t+1}, m_{t+1}, w^x_{t+1}, upmsg_{t+1})**:** Takes as input the manager's secret key sk, the current accumulator value v_t, the current manager's memory m_t, and an element x which is being added to the accumulator. It outputs the updated accumulator value v_{t+1} and memory m_{t+1}, the membership witness w^x_{t+1} of element x and an update message upmsg_{t+1}.
- **Del**(sk,v_t, m_t, x) →(v_{t+1}, m_{t+1}, $\overline{w^x_{t+1}}$, upmsg_{t+1})**:** Takes as input the manager's secret key sk, the current accumulator value v_t, the current manager's memory m_t, and an element x and deletes it from the accumulator. It outputs the updated accumulator value v_{t+1}, the updated manager's memory m_{t+1}, the non-membership witness of element x $\overline{w^x_{t+1}}$ (for negative/universal accumulators) and an update message upmsg_{t+1}.

The following algorithms can be performed by any third party:

- **VerMem**(v_t, x, w^x_t)→ $\{0,1\}$**:** Takes as input the current accumulator value v_t, an element x and its membership witness w^x_t and outputs 1 if w^x_t is a valid witness for the fact that x is included in the accumulator, 0 otherwise.
- **VerNonMem**(v_t, x, $\overline{w^x_t}$)→ $\{0,1\}$**:** Takes as input the current accumulator value v_t, an element x and its non-membership witness $\overline{w^x_t}$ and outputs 1 if $\overline{w^x_t}$ is a valid witness for the fact that x is not included in the accumulator, 0 otherwise.

The following algorithms are performed by the witness holder of element x:

- **MemWitUpOnAdd**(v_t, v_{t+1}, x, w^x_t, upmsg_{t+1}) → w^x_{t+1}**:** It takes as input the old and updated accumulator values v_t and v_{t+1}, an element x and its obsolete membership witness w^x_t as well as an addition update message upmsg_{t+1} and outputs the updated membership witness for x, w^x_{t+1}.
- **MemWitUpOnDel**(v_t, v_{t+1}, x, w^x_t, upmsg_{t+1}) → w^x_{t+1}**:** It takes as input the old and updated accumulator values v_t and v_{t+1}, an element x and its obsolete membership witness w^x_t, as well as a deletion update message upmsg_{t+1} and outputs the updated membership witness for x, w^x_{t+1}.

For negative or universal accumulators, there are also the equivalent algorithms **NonMemWitUpOnAdd**, **NonMemWitUpOnDel**.

In case where an element x was not added using the **Add()** algorithm, but instead it was included in the initial set S_0 of elements accumulated, we also have the following algorithms for witness creation which can be executed by the manager to create a (non)membership witness for an element x:

- **MemWitCreate/ NonMemWitCreate** (sk, v_t, m_t, x) → w^x_t / $\overline{w^x_t}$**:** Takes as input the manager's secret key sk, the current accumulator value v_t, the manager's memory m_t (for **NonMemWitCreate** this is usually the set S), and an element x (included/not included in v_t). It outputs a membership witness for x, w^x_t/ a non-membership witness for x, $\overline{w^x_t}$.

Discussion on Definitional Choices. In our definitions we assume that the set of currently accumulated elements S is always part of the manager's memory m_t. When the **Add** algorithm is executed, the manager first checks S to verify that the element is not already in S, and when **Del** is executed, the manager checks that the element is already in S. While this is not a security concern, if not checked could lead to practical issues, such as accumulation of duplicates or deletion of non-existing elements.

Trapdoorless Setting. We also provide definitions for the trapdoorless (with trusted setup) setting, where parties can add and remove elements without the assistance of a trusted accumulator manager in Appendix A.

B. Accumulator Security Properties

We now present the security properties for a positive accumulator. An accumulator must be correct and sound. Informally, correctness implies that for every element in/not in the accumulator, an honest witness holder can always

[2]We note that in some constructions, i.e. bilinear-pairing based instantiations in the trapdoorless with trusted setup setting, the length of params indicates the capacity of the accumulator, i.e. the maximum number of elements that can be added such that public operations continue to be feasible.

prove membership/non-membership successfully. A formal definition is straightforward and given in [37].

Informally, soundness (sometimes called "collision resistance" in the accumulator setting) prevents a dishonest user from constructing a membership witness for an element not present in the accumulator or a non-membership witness for an element included in the accumulator. Soundness has two flavors: adaptive and non-adaptive. Adaptive soundness is a stronger notion that allows an adversary to pick elements adaptively and perform queries. We provide game-based definitions for both flavors of soundness:

Definition 1 (Adaptive Soundness for Dynamic Positive Accumulators)**.** *For security parameter λ, for all probabilistic polynomial time adversaries $\mathcal{A}$ with black-box access to* Add *and* Del *oracles on an accumulator for a dynamic set S with a changing value v:*

$$\Pr\left[\begin{array}{l}(sk, v_0, m_0, \mathsf{params}) \leftarrow Gen(1^\lambda, S_0);\\ (x, w^x) \leftarrow \mathcal{A}^{\mathsf{Add},\mathsf{Del}}(v);\\ x \notin S' :\\ VerMem(v, x, w^x) = 1\end{array}\right] \leq negl(\lambda)$$

Where S' is the updated accumulated set after the adversary's addition and deletion queries. Note that [6] allows an accumulator to hold duplicates of elements but based on most of related work that simulates a set, we allow an element x to be added only if it is not already a member.

Definition 2 (Non-Adaptive Soundness for Dynamic Positive Accumulators)**.** *For any λ, for all PPT adversaries $\mathcal{A}$ with black-box access to* Add *and* Del *oracles on an accumulator for a dynamic set S with a changing value v:*
Let L_D be the list of deletions that $\mathcal{A}$ submits ahead of time.

$$\Pr\left[\begin{array}{l}\{(x_1, ..., x_{|A|}) \in L_A,\\ (x_1, ..., x_{|D|}) \in L_D\} \leftarrow \mathcal{A}(\lambda);\\ (sk, v, m_0, \mathsf{params}) \leftarrow Gen(1^\lambda, S_0);\\ (x, w^x) \leftarrow \mathcal{A}^{\mathsf{Add},\mathsf{Del}}(v);\\ x \notin S' :\\ VerMem(v, x, w^x) = 1]\end{array}\right] \leq negl(\lambda)$$

Where S' is the updated accumulated set after the adversary's addition and deletion queries. Oracle Add returns nothing if $x \in S'$ or $x \notin L_A$ and Del returns nothing if $x \notin S'$ or $x \notin L_D$.

III. Bilinear Pairings and Pairing Accumulators Background

In this section we present the required building blocks and underlying assumptions to be used in our constructions. We start by giving the standard definition of a bilinear pairing, written in multiplicative notation. Pairings take as input two elements of known-order groups and are used by bilinear accumulators during the witness verification process.

Definition 3 (Bilinear Pairing)**.** *For multiplicative groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ with prime order p, a bilinear pairing is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, satisfying the following properties:*

- ***bilinearity:*** *$\forall u \in \mathbb{G}_1, v \in \mathbb{G}_2$: $e(u^a, v^b) = e(u, v)^{ab}$, for $a, b \in \mathbb{Z}$*
- ***non-degeneracy:*** *For g_1 being a generator of $\mathbb{G}_1$, g_2 being a generator of $\mathbb{G}_2$ holds that: $e(g_1, g_2) \neq 1$*

A bilinear pairing must be also efficiently computable.
A pairing instance is denoted as (p,$\mathbb{G}_1$,$\mathbb{G}_2$,$\mathbb{G}_T$,e,g_1,g_2). When $\mathbb{G}_1 = \mathbb{G}_2 = \langle g \rangle$, the pairing is called symmetric and is denoted as (p,$\mathbb{G}$,$\mathbb{G}_T$,e,g).

Now we give the q-Strong Diffie Hellman assumption for known-order groups. This is the main assumption used when proving security of bilinear accumulators [34], [4], [20] and will also be the basis for the security in our constructions.

Assumption 1 (q-Strong Diffie-Hellman q-SDH)**.** *Let G=$\langle g \rangle$ be a cyclic group of prime order p and $a \in \mathbb{Z}_p^*$. Any PPT algorithm $\mathcal{A}$, given the set $\{g^{a^i}, i \in [0, q]\}$ has negligible probability to compute pair $(x, g^{\frac{1}{x+a}}) \in \mathbb{Z}_p^* \times \mathbb{G}$.*

In the trapdoorless with a trusted setup setting, where entities being able to perform **AddPublic, DelPublic, (Non)MemWitCreatePublic** without knowing the accumulator secret key (here denoted as a), we treat the secret key a as a polynomial variable. In Equation 1 below we show how a group element raised to a product of degree one polynomials of unknown a (similar to accumulator value and element witness structure) can be computed when the element raised to powers of a is given as public information.

Multiplication of Polynomials in the Exponent. One can compute the evaluation of a degree k polynomial $p_k(x) = \sum_{i=0}^{i=k} c_i x^i$ at an unknown point a in the exponent using k elements $g^{a^i}, i = \{0, \ldots, k\}$ as follows:

For any group element g, it holds that

$$g^{\prod_{i=1}^{i=k} p_i(a)} = g^{\sum_{i=0}^{k} c_i a^i} = g^{c_0}(g^a)^{c_1} \cdot (g^{a^2})^{c_2} \cdot \cdots \cdot (g^{a^k})^{c_k} \quad (1)$$

Update: in order to multiply the exponent with the monomial $(y + x)$ evaluated at $x = a$, the whole polynomial gets re-computed as $p_k(x)(y+x) = \sum_{i=0}^{i=k+1} c'_i x^i$ and re-evaluated at a using equation 1 and the new coefficients c'_i.

A. Bilinear Accumulators Based on q-SDH

A bilinear positive accumulator on additive groups was first constructed by Nguyen [34], and it got extended to a universal accumulator in [20], [4]. Such constructions can be adapted to both trapdoor-based and trapdoorless settings.

We present the trapdoor-based version of the original universal BP [4] (in multiplicative notation), since it will be a building block for our construction.

- **Gen**(1^λ, $S_0 = \emptyset$)**:** Takes as input the security parameter and outputs a pairing instance (p, $\mathbb{G}$, $\mathbb{G}_T$, e, g), where $|p| = \lambda$ and p prime, the accumulator's domain $D = \mathbb{Z}_p - \{a\}$, the initial empty accumulator's value $v_0 = g^1$, and the manager's secret key $a \in \mathbb{Z}_p^*$. It also outputs the accumulator params=(pairing instance, g^a).
- **Add/Del**(sk, v_t, x)**:** In order to add x, the manager computes the new acc value as $v_{t+1} = v_t^{(x+a)}$ and returns the

membership witness $w^x_{t+1} = v_t$. In order to delete x, the manager sets $v_{t+1} = v_t{}^{\frac{1}{x+a}}$ and calls **NonMemWitCreate** $(\mathsf{sk}, v_{t+1}, m_t, x)$ to compute the non-membership witness $\overline{w^x_{t+1}}$. It outputs the updated accumulator value v_{t+1} and an update message upmsg_{t+1}.

- **MemWitCreate**(sk, v_t, x)**:** A witness for an element x can be created by manager as $w^x_t = v_t^{\frac{1}{x+a}}$.
- **MemWitUpOnAdd**$(v_t, x, w^x_t, \mathsf{upmsg} = y)$**:** Any witness holder of element x, when y is added, can update witness to $w^x_{t+1} = v_t(w^x_t)^{y-x}$.
- **MemWitUpOnDel**$(v_{t+1}, x, w^x_t, \mathsf{upmsg} = y)$**:** Any witness holder of element x, when y is deleted, can update witness to $w^x_{t+1} = (w^x_t)^{\frac{1}{y-x}} v_{t+1}^{\frac{1}{x-y}}$.
- **VerMem**(v_t, x, w^x_t)**:** A third party or the manager can verify that w_x is the witness of an element x that is a member of the accumulator with the following pairing comparison: $e(w^x_t, g^x g^a) = e(v_t, g)$.

Note that this construction is a universal accumulator so it also supports the following algotithms: **VerNonMem, NonMemWitCreate, NonMemWitUpOnAdd, NonMemWitUpOnDel**. We present the algorithm for non-membership witness creation to show how manager memory m_t and the set S (set of currently accumulated elements) play a role in non-membership witness creation:

- **NonMemWitCreate**$(\mathsf{sk}, v_t, m_t, x)$**:** Parse $m_t = S$ and find $u = \prod_{x_i \in S}(x_i + a) \in \mathbb{Z}$. Compute $d = u \bmod (a + x)$ and $c = g^{\frac{u-d}{x+a}}$. Output non-membership witness $\overline{w^x_{t+1}} = (c, d)$.

Non-membership witnesses can be updated by users in constant time as follows:

- **NonMemWitUpOnAdd**$(v_t, x, \overline{w^x_t} = (c_t, d_t), \mathsf{upmsg} = y)$**:** Any non-membership witness holder of element x, when y is added, can update witness to $w^x_{t+1} = (c_{t+1}, d_{t+1}) = (v_t {c_t}^{y-x}, d_t(y - x))$.
- **NonMemWitUpOnDel**$(v_{t+1}, x, \overline{w^x_t} = (c_t, d_t), \mathsf{upmsg} = y)$**:** Any non-membership witness holder of element x, when y is deleted, can update witness to $w^x_{t+1} = (c_{t+1}, d_{t+1}) = ((c_t v_{t+1})^{\frac{1}{y-x}}, \frac{d}{y-x})$.

Witnesses for non-membership can be verified by the following algorithm:

- **VerNonMem**$(v_t, x, \overline{w^x_t} = (c, d))$**:** If $d \neq 0$ and $e(c, g^x g^a)e(g, g)^d = e(v_t, g)$, return 1.

Trapdoorless bilinear pairing accumulator. For completeness, in Appendix B we present the algorithms that can be run by the public without knowledge of the secret in the trapdoorless (with a trusted setup) setting.

B. The BB Signature Construction of [11]

A signature scheme can serve as a positive additive accumulator. Essentially, the signature on a user's element x can serve as a membership witness and the signing public key and parameters serve as the accumulator value. A signature is an additive only accumulator (i.e. elements cannot be deleted) since an issued signature cannot be retracted.

For the case of pairing-friendly groups with the q-SDH assumption, we describe the signature construction of [11]. Recall that every signature scheme is described as a set of three algorithms: KeyGen for generation of parameters, Sign for signing a message under the signer's secret key and Verify for publicly verifying that a signature is valid.

- **KeyGen**(1^λ)**:** $(p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,\ e,\ g_1,\ g_2)$ with $|G_1| = |G_2| = p$ (p prime) and $|p| = \lambda$ and pick random $y_1, y_2 \in \mathbb{Z}^*_p$ and calculate $z = e(g_1, g_2) \in G_T$. Let $v_1 = {g_2}^{y_1}, v_2 = {g_2}^{y_2}$. pk=$(g_1, g_2, v_1, v_2, z)$, sk=$(g_1, y_1, y_2)$
- **Sign**(sk, x): For x in Domain $D = \mathbb{Z}_p$, pick a random $r \neq -(y_1+x)/y_2$ and compute $\sigma = (A = {g_1}^{\frac{1}{y_1+x+{y_2}r}}, r)$
- **Verify**(pk, x, σ): If $e(A, v_1 {g_2}^x {v_2}^r) = z$ return 1.

The scheme is existentially unforgeable for adaptive chosen message attacks under a variation of q-SDH assumption for groups $\mathbb{G}_1$, $\mathbb{G}_2$ (pairing input groups).

IV. A Communication Optimal Pairing Based Positive Dynamic Non-Adaptively Secure Accumulator

Here we give our first modular construction which uses the BP accumulator of [34] as a building block. Our construction is a positive, dynamic accumulator that satisfies non-adaptive soundness. We note that compared to [34], we do not support public updates and public witness creation. Depending on the underlying application such properties might not be required. Additionally, as we explain below, our first construction is non-adaptively secure (when [34] satisfies adaptive security). The advantage of our construction though, is that it does not have to update the accumulator value on additions (running algorithm **MemWitUpOnAdd** is not necessary). As a result this has an advantage on efficiency and *optimal* communication cost as we discuss in Section VII.

The basic intuition behind our construction is that it is static in additions. Typically, when a new element x is added in a dynamic accumulator, the membership witness for x is set to be the old accumulator value and a term relevant to x (depending on the exact construction) is added to the previous accumulator value. Our approach here is different: on additions we keep the accumulator value fixed and then, in order to construct a witness for the newly added element we add the inverse of the added element x in the product (the accumulator exponent in the BP case). Thus, no update witness needs to take place when a new element is added in the accumulator. Note that this calculation is feasible by the accumulator manager, as the knowledge of the secret key sk allows for witness creation of any possible element in the domain. This way, only the newly added member gets a witness and there is no need to deliver messages to the rest of the witness holders or for the witness holders to run any witness update algorithm, as opposed to before. Their previous witnesses remain valid after the addition. The verification process also remains the same (recall that verification consists of witness raised to the new term and a check if this equals to the current accumulator value).

Our technique is a modification of the idea that was informally introduced in a remark on page 12 of [16] but instead of applying to the RSA accumulator (as in [6]) we extend it to bilinear based accumulators. The idea describes how the accumulator value gets updated only during deletions for proving membership and how the accumulator manager is always capable of constructing a witness for any element.

Our simulation works for specially crafted initial accumulator values (such that queries don't give out $g^{\frac{1}{x+a}}$). Therefore, our construction is non-adaptively sound.

Deletion works the same as Construction of Section III-A. However, the membership witness now has to be created by the manager with the secret key, as opposed to the previous construction where the users could simulate all additions except the deleted element. This is only for the case when a user did not keep their initial witness or the accumulator was initiated as half full. If the users keep the witnesses given at the time they were added, they can update them efficiently in constant time with the use of upmsg.

We present the Pairing Based Positive Dynamic Non-Adaptively Secure Accumulator in Figure 1. The differences from Construction of Section III-A are marked with blue. After **Gen()** is run, manager's memory is kept fixed, so it is not given as input to other algorithms.

Security. Correctness of our construction is straightforward, for completeness we include it in Appendix B. For soundness we prove the following Theorem in Appendix C.

Theorem 1. *The accumulator construction of Figure 1 satisfies non-adaptive soundness as defined in Definition 2 under the q-SDH assumption.*

V. A Communication Optimal Pairing Based Positive Dynamic Adaptively Secure Accumulator

From [6] we know that if we combine a positive additive accumulator for authenticated additions with a positive dynamic accumulator to manage deletions, we get an *optimal* in terms of communication cost positive dynamic accumulator. We also know that if we combine an adaptively sound accumulator with a non-adaptively sound accumulator with elements being the outputs of a PRF, then we get an adaptively sound accumulator. These two ideas can be combined in order to construct a positive adaptively-sound dynamic accumulator by utilizing an adaptively-sound positive additive accumulator $\mathsf{ACC_A}$ and a non-adaptively sound positive dynamic accumulator $\mathsf{ACC_{NA}}$. This was already done in [6] and we recall the generic construction of such an accumulator in Figure 2. After the generic construction, and utilizing our new construction of Figure 1, we show how Figure 2 can be instantiated in the BP setting which would give as a result a positive adaptively-sound dynamic accumulator with *optimal* communication cost. *Security.* Correctness of our Figure 2 construction is self-evident and is derived by correctness of $\mathsf{ACC_{NA}}, \mathsf{ACC_A}$ as the witness is a conjuction of the two witnesses.

Adaptive security follows directly by the following Theorem which was stated and proved in [6]:

Gen(1^λ,$S = \emptyset$)
1) Generate a pairing instance (p, $\mathbb{G}$, $\mathbb{G}_T$, e, g), where $|p| = \lambda$ and p prime.
2) Pick random $a, u_0 \in \mathbb{Z}_p^*$
3) Set the initial accumulator value as $v_0 = g^{u_0} \in \mathbb{G}$, where u_0 is a random value instead of the identity element.
4) Set the domain $D = \mathbb{Z}_p - \{a\}$
5) Return $sk = a$, the initial accumulator value v_0, public params=$((p, \mathbb{G}, \mathbb{G}_T, e, g), g^a)$

Add($sk = a, v_t, x$)
1) If $x \notin D$, FAIL
2) Let $w_{t+1}^x = v^{\frac{1}{x+a}}$ and keep the same accumulator value $v_{t+1} = v_t$
3) Return w_{t+1}^x

Del($sk = a, v_t, y$)
1) If $y \notin D$, FAIL
2) Let $v_{t+1} = v_t^{\frac{1}{y+a}}$
3) Return v_{t+1}, upmsg $= y$

MemWitUpOnDel(v_{t+1}, x, w_t^x, upmsg $= y$)
1) Compute $w_{t+1}^x = w_t^{x\,\frac{1}{y-x}} \times v_{t+1}^{\frac{1}{x-y}}$
2) Return w_{t+1}^x

VerMem($sk = a, v_t, x, w_t^x$) **- accumulator manager verification**
1) If $v_t = (w_t^x)^{(x+a)}$, return 1
2) otherwise, return 0

VerMem(v_t, x, w_t^x) **- third party verification**
1) If e(g,v_t)=e($g^x g^a, w_t^x$), return 1 (note that g^a is in params).
2) otherwise, return 0

Fig. 1. A BP Based Positive Dynamic Non-Adaptively Secure Accumulator

Gen(1^λ, $\emptyset$)
1) ($\mathsf{ACC_{NA}}$.params,$\mathsf{ACC_{NA}}$.sk,$\mathsf{ACC_{NA}}.v$)$\leftarrow$ $\mathsf{ACC_{NA}}$.Gen(1^λ, $\emptyset$)
2) ($\mathsf{ACC_A}$.params,$\mathsf{ACC_A}$.sk,$\mathsf{ACC_A}$.v)$\leftarrow$ $\mathsf{ACC_A}$.Gen(1^λ,$\emptyset$)
3) Let F_s ($s \xleftarrow[R]{} \{0,1\}^\lambda$) be a PRF s.t. $F_s : D_{\mathsf{ACC_A}} \rightarrow D_{\mathsf{ACC_{NA}}}$
4) Return sk=($\mathsf{ACC_A}$.sk,$\mathsf{ACC_{NA}}$.sk,s), (params= $\mathsf{ACC_{NA}}$.params, $\mathsf{ACC_A}$.params), $\mathsf{ACC_{NA}}.v$, $\mathsf{ACC_A}.v$

Add(sk,$\mathsf{ACC_{NA}}.v$**,** $\mathsf{ACC_A}.v$**,** x**)**
1) Set $r = F_s(x)$
2) $\mathsf{ACC_A}.w_{(x,r)} \leftarrow$ $\mathsf{ACC_A}$.Add(sk,$\mathsf{ACC_A}.v$, (x, r))
3) $\mathsf{ACC_{NA}}.w_r \leftarrow$ $\mathsf{ACC_{NA}}$.Add(sk,$\mathsf{ACC_{NA}}.v$, r)
4) Return w_x=($\mathsf{ACC_A}.w_{(x,r)}$, $\mathsf{ACC_{NA}}.w_r$)

Del(sk, $\mathsf{ACC_{NA}}.v$**,** x**)**
1) Set $r = F_s(x)$
2) ($\mathsf{ACC_{NA}}.v$,upmsg) $\leftarrow$ $\mathsf{ACC_{NA}}$.Del(sk,$\mathsf{ACC_{NA}}.v$, r)
3) Return $\mathsf{ACC_{NA}}.v$,upmsg

MemWitUpOnDel($\mathsf{ACC_{NA}}.v$**,** x **,**w_x**,** upmsg**)**
1) ($\mathsf{ACC_{NA}}.w_x$) $\leftarrow$ $\mathsf{ACC_{NA}}$.MemWitUpOnDel($\mathsf{ACC_{NA}}.v$, x, w_x, upmsg)
2) Return $\mathsf{ACC_{NA}}.w_x$

VerMem($\mathsf{ACC_{NA}}.v, \mathsf{ACC_A}.v, x, w_x$**)**
1) Set $r = F_s(x)$
2) $b_1 \leftarrow$ $\mathsf{ACC_A}$.VerMem($\mathsf{ACC_A}.v$, (x, r), $w_{(x,r)}$)
3) $b_2 \leftarrow$ $\mathsf{ACC_{NA}}$.VerMem($\mathsf{ACC_{NA}}.v, r, w_r$)
4) Return b_1 AND b_2

Fig. 2. A Generic Construction of a Dynamic Adaptively Secure Accumulator

Theorem 2. *A combination of accumulators* $\mathsf{ACC_A}$*, to accumulate and bind elements* (x, r)*, and* $\mathsf{ACC_{NA}}$ *to accumulate elements* r *where* $r = F_s(x)$*, is an adaptively sound positive dynamic accumulator, if* $\mathsf{ACC_A}$ *is an adaptively sound positive additive accumulator,* $\mathsf{ACC_{NA}}$ *is a non-adaptively sound positive dynamic accumulator, and* F_s *is a pseudorandom function.*

A. Instantiation

An instantiation of our adaptively sound construction can be given by using the BB signature scheme of [11] (described in Section III-B) as a positive additive accumulator and the positive dynamic accumulator from Figure 1 which we proved that is non-adaptively sound. The signature scheme of [11] is existentially unforgeable for adaptive chosen message attacks, which makes it an adaptively sound accumulator.

B. ZK Proof Of Membership in Modular Accumulator

In privacy-preserving applications (such as anonymous credentials), it is often desirable for the witness holders to prove their membership and at the same time to hide their element and witness. For the case of our generic construction of a dynamic adaptively secure accumulator (Figure 2), a zero-knowledge proof of membership ZKP consists of the conjunction (i.e. an AND-proof) of proofs of membership in $\mathsf{ACC_A}$ and $\mathsf{ACC_{NA}}$ and a proof that those members have the correct relationship. We can describe the ZK statement as follows (using Camenisch-Stadler [17] notation):

$$\mathsf{ZKP} = \left\{ \begin{array}{c} (x, r, \mathsf{ACC_A}, \mathsf{ACC_{NA}}) : \\ \mathsf{ACC_A.VerMem}(\mathsf{ACC_A}.v, (x, r), \mathsf{ACC_A}.w_{(x,r)}) \wedge \\ \mathsf{ACC_{NA}.VerMem}(\mathsf{ACC_{NA}}.v, r, \mathsf{ACC_{NA}}.w_r) \end{array} \right\} (\mathsf{ACC_A}.v, \mathsf{ACC_{NA}}.v)$$

The concrete instantiation of such a ZK statement depends on the concrete building blocks of the modular accumulator. In Appendix G we give an instantiation for the case where $\mathsf{ACC_A}$ is instantiated with strong Boneh-Boyen signatures III-B and $\mathsf{ACC_{NA}}$ is instantiated by our protocol in Figure 1.

VI. A Universal Dynamic Pairing Based Accumulator

All our previous constructions are of positive accumulators. In this Section we provide a third construction of a *universal* bilinear accumulator, i.e. an accumulator that supports both proofs of membership and non-membership, again utilizing modular construction techniques.

Typically, in universal (or negative) accumulators the highest cost comes by computing non-membership witnesses in the accumulator manager side (with the exception of some of the schemes derived by VC which as discussed can have higher setup and parameters costs). For both the state of the art works, RSA and BP, non-membership is computed with the use of the extended Euclidean algorithm. This requires the full exponent (without modular reduction) which is a product of terms that correspond to the currently added elements, therefore it grows linearly to the size of $|S|$. The computational cost of Euclidean algorithm also grows along with the full exponent product. Exponent division can also be used for the BP accumulator and has equivalent costs.

The goal of our third construction is to achieve creation of non-membership witnesses with constant cost while maintaining constant size of parameters. We take the following approach: we keep the added elements in a positive accumulator $\mathsf{ACC_1}$ and the not-yet-added/deleted/revoked elements in a second positive accumulator $\mathsf{ACC_2}$. By "not-yet-added" we mean all possible universe elements to be accumulated (i.e. the accumulator domain D). The second accumulator will start "full", i.e. having accumulated the whole domain D. The resulting accumulator is universal and we present this generic construction in Figure 3. A membership witness in the second accumulator $\mathsf{ACC_2}$ works as non-membership in a typical universal accumulator construction, while a membership witness in the first accumulator $\mathsf{ACC_1}$ serves as a membership witness in the overall universal accumulator. We note that such a generic construction might have further applications beyond the bilinear pairing setting.

Note 1. We note that the cost of the setup phase (i.e. that of creating the full accumulator) is equivalent to the size of the accumulator domain. This puts a polynomial bound in the domain which is reasonable for most real life applications, i.e. accumulating ID numbers, and is similar to that of accumulators derived from VC schemes as shown in Table II.

Gen(1^λ)**:**
1) $\mathsf{ACC_1}.Gen(1^\lambda, \emptyset)$
2) $\mathsf{ACC_2}.Gen(1^\lambda, D)$ with the same random coins for parameters, where D is the accumulator's domain

Add(x)**:**
1) $\mathsf{ACC_1}.\mathrm{Add}(x)$
2) $w_x = \mathsf{ACC_1}.\mathrm{MemWitCreate}(x)$
3) $\mathsf{ACC_2}.\mathrm{Del}(x)$

Del(x)**:**
1) $\mathsf{ACC_2}.\mathrm{Add}(x)$
2) $\overline{w_x}$=$\mathsf{ACC_2}$.MemWitCreate(x)
3) $\mathsf{ACC_1}.\mathrm{Del}(x)$

MemWitUpOnAdd/Del(y):
1) $\mathsf{ACC_1}$.MemWitUpOnAdd/Del(y)

NonMemWitUpOnAdd/Del (y):
1) $\mathsf{ACC_2}$.MemWitUpOnAdd/Del(y)

VerMem(x)**:**
1) $\mathsf{ACC_1}.\mathrm{VerMem}(x)$

VerNonMem(x)**:**
1) $\mathsf{ACC_2}.\mathrm{VerMem}(x)$

Fig. 3. Generic Construction for Universal Dynamic accumulator

Security. Correctness is straightforward as it is reduced to the correctness of individual schemes $\mathsf{ACC_1}, \mathsf{ACC_2}$ and is based on algorithms **MemWitCreate, VerMem, MemWitUpOnAdd/Del** that all refer to the same accumulator ($\mathsf{ACC_1}$ for membership and $\mathsf{ACC_2}$ for non-membership).

In Appendix D we now prove adaptive soundness of our construction as stated in the following theorem:

Theorem 3. *A combination of accumulators* $\mathsf{ACC}_1, \mathsf{ACC}_2$ *is a universal dynamic adaptively-sound accumulator if* $\mathsf{ACC}_1, \mathsf{ACC}_2$ *are positive dynamic adaptively-sound accumulators of domain* D *where the one is holding a set* $S \subset D$ *and the other one is holding* $\overline{S} \subset D$ *and public updates are not permitted*[3].

A. An Instantiation of our Universal BP Accumulator

We can instantiate our generic construction for universal dynamic accumulator (Figure 3) for the BP setting by using two instances of positive dynamic Nguyen accumulators [34]. The goal is to reduce the increasing cost of non-membership witness creation of the original universal pairing construction of [4]. Given that [34] is adaptively-secure, our resulting instantiation is also adaptively secure by Theorem 3.

VII. Efficiency Comparisons

In this Section we provide an asymptotic comparison of our results with the state of the art trapdoor-based, bilinear pairing accumulators, where we focus on the costs that occur in the accumulator manager side[4]. We expect concrete costs to be equivalent for all constructions as they refer to groups of same structure and size. Therefore, we use asymptotic notation to show the presence/absence of some specific operation or communication. We include two comparison tables, Table I compares our schemes with "traditional" BP accumulators and Table II compares our schemes with accumulators that can be derived from vector commitments. By **parameters** we denote the public keys and other public information that is created during the generation of the scheme and is kept fixed.

Comparison of Positive Accumulators. We compare our positive accumulator constructions with the positive bilinear accumulator of [15] and the positive accumulator of [34]. As shown in Table I we keep all operations constant and at the same time we require no witness updates for additions (**MemWitUpOnAdd**) which gives as the *optimal communication cost* of $|d|$ for positive accumulators as opposed to [34], [34] where communication cost is $|a| + |d|$.

We note that if we attempted a concrete comparison with [15] that has the same domain, then [15] is more efficient for constant operations (group exponentiations are replaced by modular multiplications), but it has parameters twice as long as its capacity (required also in the trapdoor-based setting i.e. for witness update by users).

Finally, we also note that public parameters of length q (that denotes capacity) used in the trapdoorless setting of [34] are not needed for the functionality in the trapdoor-based setting. They are used in the proof of our pairing based positive dynamic non-adaptively secure accumulator (Figure 1) but their absence favors security.

[3]This can be enforced by requiring the accumulator manager to always sign the most up to date value of the accumulator. Any attempt for a public update will not be verified since it will be missing the manager signature.

[4]In all comparisons of positive accumulators the algorithms related to non-membership are missing since this type of proofs is not supported.

	Positive				Universal	
	[15]	[34]	Constr. of Fig. 1	Instant. of Fig. 2	[4]	Instant. of Fig. 3
Add/Del	1	1	1	1	1	1
MemWitCreate	1	1	1	1	1	1
NonMemWitCreate	-	-	-	-	$\|S\|$	1
MemWitUpOnAdd	1	1	0	0	1	1
MemWitUpOnDel	1	1	1	1	1	1
NonMemWitUpOnAdd	-	-	-	-	1	1
NonMemWitUpOnDel	-	-	-	-	1	1
VerMem (3rd party)	1	1	1	1	1	1
VerNonMem	-	-	-	-	1	1
Storage						
Accumulator Size	1	1	1	1	1	1
Witness Size	1	1	1	1	1	1
parameters	2*q	1	1	1	1	1
Properties						
Dynamic?	✓	✓	✓	✓	✓	✓
Positive?	✓	✓	✓	✓	✓	✓
Negative?					✓	✓
Total communication	$\|a\|+\|d\|$	$\|a\|+\|d\|$	$\|d\|$	$\|d\|$	$\|a\|+\|d\|$	$\|a\| + \|d\|$
Adaptively Sound?	✓	✓		✓	✓	✓

TABLE I

An Asymptotic Comparison of Our Proposed Constructions with Traditional BP Accumulators. $|a|$ denotes the number of elements added to the accumulator, $|d|$ denotes the number of elements deleted from the accumulator, and $|S|$ denotes the total number of member elements currently accumulated. It holds that $|S| = |a| - |d|$. By $q < p - 1$ we denote the the bilinear accumulators' capacity that corresponds to accumulator's domain $\mathbb{Z}_p^*$ and depends on the desired published parameters size.

Comparison of Universal Accumulators. In the last two columns of Table I we compare the instantiation of our generic construction for universal dynamic accumulator (Figure 3) with the universal accumulator of [4].

Overall, communication cost for our construction remains the same as in [4] (optimal for universal accumulators) and constant asymptotic costs for operations remain constant.

Our main advantage is in the cost of **NonMemWitCreate**. In [4] this cost is linear to $|S|$ since as described in Section III-A when a non-membership proof is computed by running **NonMemWitCreate**, the accumulator manager uses the set S to compute the exponent u which requires a multiplication of all elements in S. In our modular construction, we bring non-membership cost from $|S|$ down to constant by only requiring membership proofs to be computed even for proving non-membership.

We note that in [4], the manager could alternatively keep a copy of the last used full exponent of the accumulator, u_{t-1}. If we consider the simple case where one element y was added since time $t-1$, the manager, in order to run **NonMemWitCreate** has to compute $u_t = u_{t-1}(y + a)$. u_{t-1} is of size $|S| - 1$ and the whole integer multiplication has cost $|S|$. Since this cost (of keeping an up-to-date exponent) is not captured in any other algorithm and is only used in **NonMemWitCreate**, we include it here which results to $|S|$.

Finally, we note that in Table I, we only include functions of repeated use (i.e. we don't include algorithm **Gen**, which is only run once). Our universal dynamic accumulator construction (Figure 3) includes a generation algorithm with computation time that depends on domain D, i.e. each element

that could be added. The rest of concrete costs remain almost the same as [4] with the exception of **NonMemWitCreate** where we have asymptotic level gains.

Vector commitment schemes as trapdoor-based accumulators. We also compare our universal accumulator constructions with those derived by vector commitments. The main idea on how to get an accumulator from a position-binding VC scheme has as follows: to add an element i, one can commit to a vector of value 1 in position i (commitment to vector $(v_1, ..., v_i = 1, ...)$). A proof of opening v_i to 1 translates to a proof of membership for i and a proof of opening v_i to 0 translates to a proof of non-membership. Such accumulators can be converted to trapdoor-based ones by allowing for a manager that holds the setup trapdoor. Typically, by holding the trapdoor, a manager can update a commitment to v_i into v_i' or create a proof of opening at position i faster. This yields to trapdoor-based accumulators with better efficiency than their trapdoor-less counterparts. In all our comparisons below, the published parameters are the minimum parameters in the trapdoor based setting, required by users to update their witnesses and verifiers to run the verification algorithm.

Vector commitment schemes in the bilinear pairing environment that can be converted to trapdoor-based accumulators are [19], [27], [24], [39], [31]. The schemes derived by [19], [27] and [24] require public parameters of length $\mathcal{O}(|D|^2)$ and $\mathcal{O}(|D|)$ respectively, where D is the accumulator domain. Note, that in the VC case, the parameter size would only affect the vector's length -not its domain- but when converted to an accumulator, the length of parameters translates to the maximum number of possible positions, i.e. the accumulator domain. The schemes derived by [39], [31], have the advantage of smaller parameters (of length $\mathcal{O}(|q|)$ where q is the accumulator capacity) when compared to [19], [27], [24] but are more expensive (non-constant) when it comes to other operations such as addition/deletion of elements and non-membership creation computation costs.

For the instantiation of our generic construction of Figure 3 (and later in our ZK accumulator construction of Figure 4), we used the "traditional" BP accumulator of [4] instead of a VC derived one, given that [4] has constant parameters and constant updates and membership witness creation cost in the trapdoor-based setting. This way, although we require $\mathcal{O}(|D|)$ setup time (for our universal constructions), there is no need to publish longer parameters for verification and witness update as in the VC derived constructions.

In Table II we summarize the efficiency costs of our universal accumulators when compared to accumulators derived by VC schemes. By params we denote the public information that has to be available in order to be used by the rest of the entities, i.e. the witness holders/users and the verifier. As opposed to [4], some of these schemes can offer constant **NonMemWitCreate** as in our constructions, but at the cost of rather large public parameters (compared to constant).

The ZK column of Table II, describes the notion of zero-knowledge in set queries which also extends to accumulators as shown in [23] and [21]. In our construction (Fig. 4) we consider the stronger notion of [23], i.e. a verifier or a third party cannot check the correctness of their guess about elements in the set (we refer to Section VIII for details).

	|params|	Add/Del	NonMem-WitCreate	Setup time	ZK
[19],[27]	$\|D\|^2$	1	1	$\|D\|^2$	
[24]	$\|D\|$	1	1	$\|D\|$	
[39]	q	$\|S\|$	$\|S\|$	q	
[31]	q	1	$\|S\|$	q	✓[21]
Instant. of Fig. 3	1	1	1	$\|D\|$	
Fig. 4	1	1	1	$\|D\|$	✓[23]

TABLE II
AN ASYMPTOTIC COMPARISON OF OUR UNIVERSAL CONSTRUCTIONS WITH UNIVERSAL ACCUMULATORS DERIVED BY BP VC SCHEMES IN THE TRAPDOOR-BASED SETTING. $|S|$ DENOTES THE TOTAL NUMBER OF MEMBER ELEMENTS CURRENTLY ACCUMULATED. Q DENOTES THE ACCUMULATOR CAPACITY AND D THE ACCUMULATOR DOMAIN.

On applying batching techniques. Our constructions can further benefit from batching approaches and we discuss it more in Appendix F.

VIII. ZERO-KNOWLEDGE ACCUMULATORS

In a recent work Ghosh et al. [23], introduced the notion of Zero-Knowledge (ZK) accumulators. Their setting assumes an outsourced storage environment and the following entities: (a) the data owner, who possesses the accumulator's secret key and is in charge of accumulator updates, (b) a server that gets the accumulator value and some auxiliary information and constructs/delivers element witnesses for proving (non)membership (i.e. performs most of the computational work) and (c) clients (or else verifiers) that see the accumulator value and query the server asking for (non)membership witnesses of specific elements which they can locally verify.

Apart from correctness and soundness, ZK in the setting of [23] assures that a client that gets to see only witnesses, learns nothing about the accumulated set, apart from the elements she queried since the last accumulator update. This is due to the lack of auxiliary information that prevents the client from reconstructing witnesses and check their guess until they achieve a whole set enumeration. Moreover, in the case the accumulator value is a merge of different accumulated sets (from multiple owners), a ZK accumulator with updates on auxiliary information in all sets, can prevent the client from locating the source of the update, by turning all previously obtained witnesses obsolete.

Generalizing the ZK setting of [23] beyond outsourced storage. The notion of ZK accumulators can also refer to the more general setting of set membership (outside the narrow application of outsourced storage). In that setting, as presented in Section II-A, initial witness creation happens with the use of the secret but by a trusted entity, the manager. Later witnesses are computed by users/element holders using public information. Soundness should be preserved at all times. Finally, interaction takes place between the user and a verifier. The role of the verifier is the same in both settings: verifiers get to see the updated accumulator value and verify (non)membership witnesses.

In this more general setting, we still consider the ZK notion of [23]: the verifier should not infer any information about the set S, apart from the information gained by the witnesses seen after the last update. This implies, that after an accumulator update, the verifier cannot locally update the witnesses received earlier and has no way to check if these elements are still in the accumulator or not. Moreover, no further information is leaked to the verifier, like the size of the set or the type of the updates taking place (addition or deletion). We note, that prior to [23], some weaker accumulator indistinguishability notion was considered in [21] however, [21] does not satisfy ZK as defined in [23], since the witness updates are deterministic which makes it possible for a verifier (who can see witnesses) to check whether a specific element was added in the accumulated set upon an update. We finally note that this is a different notion compared to the one discussed in Section V-B where an element holder could prove (non)membership without revealing its element.

The high level idea, in order to construct accumulators that satisfy our ZK notion, is to include a *randomization process*. Concretely, randomness is added during **Setup**$(1^\lambda, S_0)$ (i.e. in the case where the accumulator is initialized with added elements) and fresh randomness is added with every call to **Add()/Del()**. Randomness is included in auxiliary information aux, known to the parties that perform accumulator updates and hold witnesses. The key to preserving ZK against the verifier is (i) verification does not use aux as input (as long as the same randomness is included in the accumulator value and the randomized witness) and (ii) witness updates (that used to be feasible to compute with public information and a guess about the added/deleted element) are not longer feasible to compute without knowledge of aux.

A. Definition of ZK Accumulators

A ZK accumulator, as defined in [23] can be adjusted to work in our setting with some changes to the algorithms of Section II-A in order to use some auxiliary information. More specifically, the **Gen()** algorithm now returns some auxiliary information, aux, as output. Usually, aux belongs to the same domain as the accumulator elements because its role is to obfuscate the exact update that happened and/or the elements already in the set (also derived by updates in the clear). aux is updated to aux' via **Add(),Del()**. Finally, aux is given as input to **MemWitCreate(), NonMemWitCreate()**, while **MemWitUpOnAdd(), MemWitUpOnDel(), NonMemWitUpOnAdd(), NonMemWitUpOnDel()** receive $\mathsf{aux}, \mathsf{aux}'$ (i.e. aux in time t and aux' in time $t+1$).

Below we recall the definition of the ZK property of an accumulator as given in [23] modified for our setting: a witness is honestly generated with the use of a secret and soundly updated by users with public information. Witnesses are submitted by the users to the verifier. Soundness for ZK accumulator is defined as in Definition 1.

Definition 4 (ZK property in accumulators)**.** *Let a D be a function that for a given element x and a set X, $D(query, x, X) = 1$ if x belongs in X and 0 otherwise. Also, $D(update, x, add/del, X) = 1$ if an update, for example addition of x in X is valid (not already a member). Let $\mathsf{Real}_{\mathsf{Adv}}(1^\lambda)$ be the game between an adversary $\mathcal{A}$ and the Challenger and $\mathsf{Ideal}_{\mathsf{Adv},\mathsf{Sim}}(1^\lambda)$ be the the game between $\mathcal{A}$ and a simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$*

- $\mathsf{Real}_{\mathsf{Adv}}(1^\lambda)$
  - ***Setup:*** *The Challenger generates the key and publishes the* params *to $\mathcal{A}$. $\mathcal{A}$ picks a set S_0 and gives it to the Challenger who initializes the accumulator to include the set and returns back the value $acc.v_0$ (and keeps $X = S_0, acc.v = acc.v_0, \mathsf{aux}$).*
  - ***Queries:*** *$\mathcal{A}$ can query on updates or witness queries.*
    - *If $\mathcal{A}$ queries for a witness on x, if $x \in S$ then Challenger runs MemWitCreate(sk,$acc.v$,x, aux) and returns w_x and its type (membership), otherwise it runs NonMemWitCreate(sk,$acc.v$,x, aux) and returns $\overline{w_x}$ and its type (non-membership).*
    - *If $\mathcal{A}$ asks for an update (addition/deletion) on x, Challenger runs Add/Del($\mathsf{sk}, v, x, \mathsf{aux}$) and returns $acc.v$ to $\mathcal{A}$ (and keeps $acc.v', S' = S \cup \{x\}/S - \{x\}, \mathsf{aux}'$)*
- $\mathsf{Ideal}_{\mathsf{Adv},\mathsf{Sim}}(1^\lambda)$
  - ***Setup:*** *On input 1^λ, Sim_1 forwards to $\mathcal{A}$ the accumulator* params*. $\mathcal{A}$ chooses a set S_0 and Sim_1 responds with $acc.v_0$ without seeing S_0. It also keeps $S = S_0, acc.v = acc.v_0$ and same* $\mathsf{state}_\mathsf{S}$ *for all variables it has seen so far.*
  - ***Queries:*** *$\mathcal{A}$ can query on updates or witness queries.*
    - *If $\mathcal{A}$ queries for a witness on x, simulator runs $Sim_2(acc.v, x, \mathsf{state}_\mathsf{S}, D(query, x, S))$, gets a witness and its type (membership/non-membership) and returns it to $\mathcal{A}$.*
    - *If $\mathcal{A}$ asks for an update (addition/deletion) on x, simulator runs $Sim_2(acc.v, \mathsf{state}_\mathsf{S}, D(update, x, add/del, S))$. If output is 1 and update is of type add, then $S = S \cup \{x\}$ and if update is of type del, then $S = S - \{x\}$. Sim responds to $\mathcal{A}$ with $acc.v'$ and sets $acc.v = acc.v'$.*

$\mathsf{Real}_{\mathsf{Adv}}(1^\lambda), \mathsf{Ideal}_{\mathsf{Adv},\mathsf{Sim}}(1^\lambda)$ *should be indistinguishable except with negligible probability.*

1) The BP Universal ZK Accumulator of [23]: The ZK accumulator of [23] builds on top of the Nguyen accumulator which we described in Section III-A. Below, we briefly present the construction of [23]. Note that public updates of the accumulator value are not available, as the manager controls the randomization process. An extra parameter aux stores the product of all random integers used so far (we ignore the size of the accumulated set as updatable variable because all needed parameters are already published by **Gen()**).

- **Gen**$(1^\lambda,\ S_0)$**:** The same as Gen() algorithm of section III-A, but now v_0 is raised to r_0 and $\mathsf{aux}_0 = r_0$.

- **Add/Del**($\mathsf{sk} = a, v_t, x, \mathsf{aux}$)**:** In order to add x, manager parses $\mathsf{aux}_t = r$ chooses a random $r' \in \mathbb{Z}_p^*$ and computes the new acc value as $v_{t+1} = v_t^{(x+a)r'}$ and for deletion as $v_{t+1} = v^{\frac{r'}{x+a}}$. It also outputs the updated accumulator value v_{t+1}, an update message and the updated information $\mathsf{aux}_{t+1} = rr'$.
- **MemWitCreate**($\mathsf{sk} = a, v_t, x, \mathsf{aux}$)**:** $w_t^x = v_t^{\frac{1}{x+a}}$
- **NonMemWitCreate**(v_t, x, S, aux)**:** The algorithm is the same for secret key holders/users with auxiliary information and is also called by **NonMemWitUpOnAdd()/NonMemWitUpOnDel()**. Using the extended Euclidean algorithm, compute polynomials $q_1(y), q_2(y)$ s.t. $q_1(y)\prod(x_i + y) + q_1(y)(x + y), x_i \in S$. Pick random $\gamma \in \mathbb{Z}^*{}_p$ and set $q_1' = q_1(y) + \gamma(x + y)$ and $q_2'(y) = q_2(y) - \gamma\prod(x_i + y)$. Set $\overline{w_t^x} = (g^{q_1'(a)r^{-1}}, g^{q_2'(a)}) = (w_1, w_2)$. Note that w_i can be calculated by a user with the use of Equation 1 and all calculations can be performed by the manager in integers (due to knowledge of $\mathsf{sk} = a$).
- **MemWitUpOnAdd**($v_t, x, w_x, \mathsf{upmsg} = y, \mathsf{aux}_t, \mathsf{aux}_{t+1}$)**:** Any witness holder of element x, when y is added, can retrieve r' by $\mathsf{aux}_{t+1}/\mathsf{aux}_t$ and update witness to $w_{t+1}^x = v_t(w_t^x)^{(y-x)r'}$.
- **MemWitUpOnDel**($v_{t+1}, x, w_x, \mathsf{upmsg} = y, \mathsf{aux}_t, \mathsf{aux}_{t+1}$)**:** Any witness holder of element x, when y is deleted, can retrieve r' by $\mathsf{aux}_{t+1}/\mathsf{aux}_t$ and update witness to $w_{t+1}^x = (w_t^x)^{\frac{r'}{y-x}} v_{t+1}^{\frac{1}{x-y}}$.
- **VerMem**(v_t, x, w_t^x)**:** Outputs 1 if $e(v_t, g) = e(w_t^x, g^x g^a)$.
- **VerNonMem**($v_t, x, \overline{w_t^x} = (w_1, w_2)$)**:** Outputs 1 if $e(w_1, v_t)e(w_2, g_x g_a) = e(g, g)$.

B. Our Bilinear Universal ZK-Accumulator

The ZK-accumulator instantiation of [23] (inspired by the construction of Section III-A) is noticeably less efficient in the case of non-membership proof generation (manager/user) and witness updates. This is due to the complexity of the algorithm **NonMemWitCreate()**, which involves $|S|$ degree polynomials for users or integers increasing with the size of S for the manager. Non-membership witness update is no more constant, as opposed to construction of Section III-A, as it requires a fresh execution of **NonMemWitCreate()** and comes with the same cost. This is the main efficiency overhead, considering that **NonMemWitUpOnAdd(), NonMemWitUpOnDel()** are run $|D| - |S|$ times, i.e. for every element not present in the accumulator, in every update. Moreover, in the case where users run it, they need to store all elements of S (S can be derived by update messages).

To keep the cost of all operations constant and achieve the property of a ZK universal accumulator, we use our construction of Section VI to replace non-membership with membership proofs. The frequency of communication remains the same ($|a| + |d|$ lower bound for universal accumulators), with a small overhead in every message.

Compared to construction of Section III-A, we add the auxiliary information and we also separate accumulator updates from witness creation to simulate the stand-alone witness queries. Our instantiation is presented in Figure 4.

Security. Below we discuss why our proposed construction satisfies the properties of Soundness and Zero-Knowledge. Correctness is straightforward and derives from the correctness of the generic construction (generic construction for universal dynamic accumulator (Figure 3)).

- **Gen**(1^λ, S_0)$\rightarrow$(sk,$acc.v_0$,params, aux_0) **:** Publish the pairing instance, create the accumulator's secret key a and publish the accumulator's public parameters (as in construction of section III-A). Pick random $r_1, r_2 \in \mathbb{Z}_p^*$. Set $acc_1.v_0 = g^{r_1\prod(x_i+a)}, x_i \in S$, $acc_2.v_0 = g^{r_2\prod y_i+a}, y_i \in D - S$ and $acc.v_0 = (acc_1.v_0, acc_2.v_0)$
- **Add**($\mathsf{sk} = a$, $acc.v_t$, x, aux_t) $\rightarrow$($acc.v_{t+1}$, $\mathsf{upmsg}_{t+1}, \mathsf{aux}_{t+1}$) **:** Pick random $r_1', r_2' \in \mathbb{Z}_p^*$. Set $acc_1.v_{t+1} = acc_1.v_t{}^{r_1'(x+a)}$, $acc_2.v_{t+1} = acc_2.v_t{}^{r_2'/(x+a)}$ and $acc.v_{t+1} = (acc_1.v_{t+1}, acc_2.v_{t+1})$. $\mathsf{aux}_t = (r_1, r_2)$, $\mathsf{aux}_{t+1} = (r_1r_1', r_2r_2')$
- **Del**($\mathsf{sk} = a$, v_t, m_t, x, aux_t) $\rightarrow$($acc.v_{t+1}$, $\mathsf{upmsg}_{t+1}, \mathsf{aux}_{t+1}$) **:** Pick random $r_1', r_2' \in \mathbb{Z}_p^*$. Set $acc_2.v_{t+1} = acc_2.v_t^{r_2'(x+a)}$, $acc_1.v_{t+1} = acc_1.v_t^{r_1'/(x+a)}$ and $acc.v_{t+1} = (acc_1.v_{t+1}, acc_2.v_{t+1})$. $\mathsf{aux}_t = (r_1, r_2)$, $\mathsf{aux}_{t+1} = (r_1r_1', r_2r_2')$
- **MemWitUpOnAdd/NonMemWitOnAdd**($acc.v_t$, $acc.v_{t+1}$, x, w_t^x, upmsg_{t+1}, $\mathsf{aux}_t, \mathsf{aux}_{t+1}$) $\rightarrow$ $w_{t+1}^x/\overline{w_{t+1}^x}$ **:** Parse $\mathsf{upmsg}_{t+1} = y$, $\mathsf{aux}_t = (r_1, r_2)$, $\mathsf{aux}_{t+1} = (r_1r_1', r_2r_2')$ and divide coefficients to get r_1', r_2'. $w_{t+1}^x = acc_1.v_t w_t^{x(x-y)r_1'}$ / $\overline{w_{t+1}^x} = acc_2.v_{t+1}^{\overline{y-x}}(\overline{w_t^x})^{\frac{r_2'}{x-y}}$
- **MemWitUpOnDel/NonMemWitOnDel**($acc.v_t$, $acc.v_{t+1}$, x, w_t^x, upmsg_{t+1}, $\mathsf{aux}_t, \mathsf{aux}_{t+1}$) $\rightarrow$ $w_{t+1}^x/\overline{w_{t+1}^x}$ **:** Parse $\mathsf{upmsg}_{t+1} = y$, $\mathsf{aux}_t = (r_1, r_2)$, $\mathsf{aux}_{t+1} = (r_1r_1', r_2r_2')$ and divide coefficients to get r_1', r_2'. $w_{t+1}^x = acc_1.v_{t+1}^{\frac{1}{y-x}} w_t^{x\frac{r_1'}{x-y}}$ / $\overline{w_{t+1}^x} = acc_2.v_t(w_t^x)^{(x-y)r_2'}$
- **MemWitCreate/ NonMemWitCreate** ($\mathsf{sk} = a$, $acc.v_t$, x) $\rightarrow w_t^x$ / $\overline{w_t^x}$**:** $w_t^x = acc_1.v_t^{\frac{1}{x+a}}/\overline{w_t^x} = acc_2.v_t^{\frac{1}{x+a}}$
- **VerMem**($acc.v_t$, x, w_t^x)$\rightarrow$ $\{0,1\}$ **:** Run **VerMem**($acc_1.v_t$, x, w_t^x) (section III-A) and return output.
- **VerNonMem**($acc.v_t$, x, $\overline{w_t^x}$)$\rightarrow$ $\{0,1\}$ **:** Run **VerMem**($acc_2.v_t$, x, $\overline{w_t^x}$) (section III-A) and return output.

Fig. 4. Universal bilinear ZK-Accumulator by two positive accumulators

Soundness. The soundness adversary has the same privileges as a witness holder, i.e. receives the auxiliary information after every update. The bilinear instantiation of [23] (acc_1, acc_2 of Figure 4) is adaptively sound under the q-Strong Bilinear Diffie Hellman assumption which can be reduced to the q-SDH assumption, according to [20]. Therefore, the soundness of our universal bilinear ZK-Accumulator by two positive accumulators (Figure 4) is adaptively-sound according to Theorem 3.

Zero-Knowledge. In Appendix E we prove zero-knowledge of our construction as stated in the following theorem:

Theorem 4. *The universal bilinear ZK-Accumulator (acc_1, acc_2) of Figure 4 that instantiates the generic universal*

modular construction of Figure 3 (where $acc_i, i \in \{1,2\}$ is the positive bilinear instantiation of [23] with auxiliary information r_i) satisfies the ZK property of Definition 4.

Efficiency of our ZK Accumulator. Our construction maintains all constant operations and decreases non-membership witness creation from $\mathcal{O}(|S|)$ to $\mathcal{O}(1)$. It also decreases non-membership witness updates from $\mathcal{O}(|S|)$ to $\mathcal{O}(1)$ and requires from users to only keep the last two update messages in order to update their witness (as opposed to the ZK bilinear instantiation of [23] that requires all of the update messages in order to recreate the set S in every update). The communication cost per update remains $\mathcal{O}(1)$ and happens in every addition and deletion as indicated by the optimal communication bound for universal accumulators.

ACKNOWLEDGEMENTS.

This work has been funded by NSF 1717067, an IBM Faculty award and a Facebook Faculty Award.

REFERENCES

[1] Tolga Acar and Lan Nguyen. Revocation for delegatable anonymous credentials. In *PKC*, 2011.

[2] Man Ho Au, Willy Susilo, and Yi Mu. Practical anonymous divisible e-cash from bounded accumulators. In *FC*, 2008.

[3] Man Ho Au, Willy Susilo, and Yi Mu. Proof-of-knowledge of representation of committed value and its applications. In *ACISP*, 2010.

[4] Man Ho Au, Patrick P Tsang, Willy Susilo, and Yi Mu. Dynamic universal accumulators for ddh groups and their application to attribute-based anonymous credential systems. In *CT-RSA*, 2009.

[5] Man Ho Au, Qianhong Wu, Willy Susilo, and Yi Mu. Compact e-cash from bounded accumulator. In *CT-RSA*, 2007.

[6] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. In *EuroS&P*. IEEE, 2017.

[7] Foteini Baldimtsi, Ran Canetti, and Sophia Yakoubov. Universally composable accumulators. In *CT-RSA*, 2020.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE S&P*, 2014.

[9] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Eurocrypt*, 1993.

[10] Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. Cryptology ePrint Archive, Report 2019/1255, 2019. https://eprint.iacr.org/2019/1255.

[11] Dan Boneh and Xavier Boyen. Short signatures without random oracles and the sdh assumption in bilinear groups. *Journal of Cryptology*, 21(2):149–177, 2008.

[12] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *CRYPTO*, 2019.

[13] Philippe Camacho and Alejandro Hevia. On the impossibility of batch update for cryptographic accumulators. In *LATINCRYPT*, 2010.

[14] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In *ICISC*, 2008.

[15] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *PKC*. Springer, 2009.

[16] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, 2002.

[17] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In *CRYPTO*, 1997.

[18] Sébastien Canard and Aline Gouget. Multiple denominations in e-cash with compact transaction data. In *FC*. Springer, 2010.

[19] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *PKC*, 2013.

[20] Ivan Damgård and Nikos Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. Cryptology ePrint Archive, Report 2008/538, 2008. https://eprint.iacr.org/2008/538.

[21] David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *CT-RSA*, 2015.

[22] Yevgeniy Dodis, Aggelos Kiayias, Antonio Nicolosi, and Victor Shoup. Anonymous identification in ad hoc groups. In *Eurocrypt*, 2004.

[23] Esha Ghosh, Olga Ohrimenko, Dimitrios Papadopoulos, Roberto Tamassia, and Nikos Triandopoulos. Zero-knowledge accumulators and set algebra. In *Asiacrypt*, 2016.

[24] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. In *ACM-CCS*, 2020.

[25] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In *Eurocrypt*, 2008.

[26] Mahabir Prasad Jhanwar and Reihaneh Safavi-Naini. Compact accumulator using lattices. In *Security, Privacy, and Applied Cryptography Engineering*, 2015.

[27] Russell WF Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *CRYPTO*, 2019.

[28] Jorn Lapon, Markulf Kohlweiss, Bart De Decker, and Vincent Naessens. Performance analysis of accumulator-based revocation mechanisms. In *IFIP SEC*, 2010.

[29] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In *ACNS*, 2007.

[30] Benoît Libert, San Ling, Khoa Nguyen, and Huaxiong Wang. Zero-knowledge arguments for lattice-based accumulators: logarithmic-size ring signatures and group signatures without trapdoors. In *Eurocrypt*, 2016.

[31] Benoît Libert, Somindu Ramanna, and Moti Yung. Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In *ICALP*, 2016.

[32] Helger Lipmaa. Secure accumulators from euclidean rings without trusted setup. In *ACNS*, 2012.

[33] Patrick Märtens. Practical compact e-cash with arbitrary wallet size. *IACR Cryptology ePrint Archive*, 2015:86, 2015.

[34] Lan Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA*, 2005.

[35] Lan Nguyen and Rei Safavi-Naini. Efficient and provably secure trapdoor-free group signature schemes from bilinear pairings. In *Asiacrypt*, 2004.

[36] Dimitrios Papadopoulos, Stavros Papadopoulos, and Nikos Triandopoulos. Taking authenticated range queries to arbitrary dimensions. In *ACM-CCS*. ACM, 2014.

[37] Leonid Reyzin and Sophia Yakoubov. Efficient asynchronous accumulators for distributed pki. In *SCN*, 2016.

[38] Shi-Feng Sun, Man Ho Au, Joseph K Liu, and Tsz Hon Yuen. Ringct 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero. In *ESORICS*, 2017.

[39] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. Cryptology ePrint Archive, Report 2020/527, 2020. https://eprint.iacr.org/2020/527.

[40] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *ACM-CCS*, 2019.

[41] Giuseppe Vitto and Alex Biryukov. Dynamic universal accumulator with batch update over bilinear groups. Cryptology ePrint Archive, Report 2020/777, 2020. https://eprint.iacr.org/2020/777.

[42] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. An expressive (zero-knowledge) set accumulator. In *EuroS&P*. IEEE, 2017.

[43] Qingji Zheng and Shouhuai Xu. Verifiable delegated set intersection operations on outsourced encrypted data. In *2015 IEEE International Conference on Cloud Engineering*, pages 175–184. IEEE, 2015.

APPENDIX

A. Algorithm definitions in the trapdoorless setting

Here we provide definitions for the trapdoorless (with trusted setup) setting, where parties can add and remove elements without the assistance of a trusted accumulator manager. Here, users need to know the set S of all currently accumulated elements.

- **AddPublic/DelPublic**(v_t, x, S) $\rightarrow$(v_{t+1}, w^x_{t+1}, upmsg_{t+1}) **:** Takes as input the current accumulator value v_t, an element x included/not included in the accumulator and the set S. It outputs the updated accumulator value v_{t+1} after addition/deletion of x, and a membership witness w^x_{t+1}/non-membership witness $\overline{w^x_{t+1}}$. To notify other parties about the update, it outputs a message upmsg_{t+1}. The witness $w^x_{t+1}/\overline{w^x_{t+1}}$ is the output of a call to **MemWitCreatePublic/ NonMemWitCreatePublic** for public witness creation (which we describe below).
- **MemWitCreatePublic/ NonMemWitCreatePublic** (v_t, x, S) $\rightarrow$ w^x_t **/** $\overline{w^x_t}$ **:** Takes as input the current accumulator value v_t, an element x (included/not included in the accumulator) and the currently accumulated set S. It outputs a membership witness for x, w^x_t/ a non-membership witness for x, $\overline{w^x_t}$.

Notice that in the trapdoorless setting the algorithms performed for verification (**VerMem,VerNonMem**) and for witness updates (**MemWitUpOnAdd/Del, NonMemWitUpOnAdd/Del**) are the same as before. Finally, we note that **MemWitCreatePublic/ NonMemWitCreatePublic** can also be useful in the trapdoor-based scenario, in order for users to create their current witness (i.e. in case they lost w), without the help of the accumulator manager, as long as they know the current set of accumulated elements S.

B. Algorithms of [4] in the trapdoorless setting

We present the algorithms that can be run by the public without knowledge of the secret in the trapdoorless with a trusted setup setting. First, the **Gen()** algorithm (run by a trusted entity) needs to additionally output the following parameters: a tuple $\{g^{a^j}\}$, $j \in [2, capacity]$. Also, set S (i.e. the set of accumulated elements) is used:

- **AddPublic/DelPublic**(v_t, x, S)**:** In order to add x, a third party computes the new acc value as $v_{t+1} = (g^{\prod_{x_i \in S}(x_i+a)})^{(x+a)}$ (and Equation 1) and for deletion as $v_{t+1} = g^{\prod_{x_i \in S}(x_i+a)}$, where $x \notin S$. Then it can call **MemWitCreatePublic** (v_{t+1}, x, S)/ **NonMemWitCreatePublic**(v_{t+1}, x, S) to return a membership/non-membership witness for x. It also outputs the updated accumulator value v_{t+1} and a message upmsg_{t+1}.
- **MemWitCreatePublic**(v_t, x, S)**:** A witness for element x can be created by a third party as $w^x_t = g^{\prod_{x_i \in S}(x_i+a)}$, where $x \notin S$ (and Equation 1).
- **NonMemWitCreatePublic**(v_t, x, S)**:** Compute the polynomial $v(a) = \prod_{x_i \in S}(x_i + a)$. Perform a polynomial division of $v(a)$ by $(x + a)$ as $v(a) = c(a)(x + a) + d$. Expand $c(a) = \sum_{i=0}^{|S|-1}(u_i a^i)$ according to Equation 1. Compute $c = g^{c(a)=\prod_{i=0}^{|S|-1} g^{u_i}}$. Output non-membership witness $\overline{w^x_{t+1}} = (c, d)$.

Correctness in accumulators ensures that a valid witness will verify via the **VerMem** algorithm. A membership witness for an element x is equal to w_x. Recall that in our construction when new elements are added to the accumulator the witnesses for rest of the elements remain the same. When a witness holder proves membership in the accumulator at time t, she needs to present a witness w_x that satisfies the equation $w^x_t = v_t^{\frac{1}{x+a}}$. The verifier (seeing the current accumulator value v_t) can either check if $w_t^{x(x+a)} = v_t$ holds (in the case she holds the secret information a) or equivalently if $e(w^x_t, g^x g^a) = e(v_t, g)$ (using public information). As long as issued witness was valid and satisfied the equations above, the membership proof will go through. Our construction preserves the relationship between the membership witness and the accumulator value of the original scheme [34], [4] presented in Section III-A and also the verification and witness update algorithm that we are analyzing below.

In case that a witness holder wishes to prove membership after a number of deletions has happened, they would have to update their witness first.

However, we can see that the above relationship is preserved since **MemWitUpOnDel** (Section III-A) works as follows:

It can be verified in integers that

$$\frac{y+a}{(x+a)(y-x)} + \frac{1}{x-y} = \frac{y+a}{(x+a)(y-x)} - \frac{x+a}{(y-x)(x+a)}$$
$$= \frac{(y+a)-(x+a)}{(y-x)(x+a)} = \frac{y-x}{(y-x)(x+a)} = \frac{1}{(x+a)}$$

MemWitUpOnDel performs the following operations

$w^x_{t+1} = w^x_t{}^{\frac{1}{y-x}} v_{t+1}{}^{\frac{1}{x-y}} = [v_{t+1}{}^{\frac{y+a}{x+a}}]^{\frac{1}{y-x}} (v_{t+1})^{\frac{1}{x-y}}$

which equals $v_{t+1}^{\frac{1}{x+a}}$ as shown above.

C. Soundness proof of our pairing based positive dynamic non-adaptively secure accumulator (Figure 1)

We now provide the proof of Theorem 1.

Proof. Let $\mathcal{R}$ be a reduction that receives as input a pairing instance and the public params of the accumulator. $\mathcal{R}$ leverages an adversary $\mathcal{A}$ which can break the non-adaptive soundness of the accumulator. $\mathcal{A}$ is allowed to submit accumulator update queries to $\mathcal{R}$. Queries are of the form Query(update, add/del, element to be added/deleted). The goal of $\mathcal{R}$ is to break the q-SDH assumption. The reduction works as follows:

- $\mathcal{A}$ submits the attack information:
  In the first step, $\mathcal{A}$ is going to send a list of additions and deletions, L_A= (x_1,...,$x_{|A|}$), L_D=(x_1,...,$x_{|D|}$).
- $\mathcal{R}$ initializes the accumulator:
  $\mathcal{R}$ picks an index $j \in [1, |D|]$
  $\mathcal{R}$ picks a counter $c \in [0, |D|]$, saves $c_i = |D|$ for $i \neq j$ and $c_j = c$

$\mathcal{R}$ initializes accumulator value as $v = g^{\prod_{i=1}^{|D|}(x_i+a)^{c_i}} = g^{\sum_{k=0}^{C} b_k a^k}$, where $C = \sum c_i$, $x_i \in L_D$ the elements to be deleted and b_k the coefficients of the polynomial in $\mathbb{Z}_p[a]$ that comes from extending the product in exponent. Note that $\mathcal{R}$ does not know a but given the public params knows g^a thus can compute v according to Equation 1. $\mathcal{R}$ returns v to $\mathcal{A}$ along with accumulator params

- $\mathcal{A}$ submits Query(update, add, x):
  if in L_A continue, else abort
  $\mathcal{R}$ computes $c_r = c_r - 1$ and $w_{x_r} = g^{\prod_{i=1, i\neq r}^{|D|}(x_i+a)^{c_i}} = g^{\sum_{k=0}^{C} b_k a^k}$
- $\mathcal{A}$ submits Query(update, del, x)
  If $x_r \in L_D$ continue, else abort
  $\mathcal{R}$ computes $c_r = c_r - 1$ and $v = g^{\prod_{i=1}^{|D|}(x_i+a)^{c_i}} = g^{\sum_{k=0}^{C} b_k a^k}$ and sends upmsg
  Add/Del queries can repeat. If at any point the number of additions or deletions of x_j is greater than c, FAIL.
- $\mathcal{A}$ sends its forgery (x^*, w^*):
  If $x^* \notin L_D$, abort
  If $\mathcal{A}$'s witness $w^* = w_{x_i}, i \neq j$ FAIL
  If number of deletions of x_j does not equal c ($c_j \neq c$), abort
  Else check if $\mathcal{A}$'s witness is valid: w^* s.t. $e(g, v) = e(g^{x^*} g^a, w^*)$

In case the reduction does not abort: $\mathcal{R}$ has a computed value $u = \prod_{x_i \in L_D, x_i \neq x^*}(x_i + a)^{c_i} = \sum_{k=0}^{C} b_k a^k$, $C = \sum c_i$
$(x^*+a) \nmid u$ (because they share no binomials of degree one as factors, which makes them co-prime) so there is a polynomial $k(a)$ with $\deg(k(a)) < C$ and a positive integer r s.t. $w^{*(x^*+a)} = g^u = g^{k(a)(x^*+a)+r}$ and $g^{\frac{1}{x^*+a}} = [w^* g^{-k(a)}]^{1/r}$. $\mathcal{R}$ breaks q-SDH as $g^{-k(a)}$ is computable by $\mathcal{R}$ in a known order group.

Let $\mathcal{A}$ perform a successful forgery with probability ϵ. Then the probability of $\mathcal{R}$ breaking q-SDH is $\frac{1}{|D|}\frac{1}{|D|+1}\epsilon$, where $\frac{1}{|D|}$ is the probability of a correct guess of x^* and $\frac{1}{|D|+1}$ is the probability of a correct guess of the number of deletions on x^*. □

Discussion. Our security proof is based on the fact that at any point, for accumulator value $v = g^u$ and $x \notin acc$: $(x+a) \nmid u$. Initialization of acc as $v_0 = g^{u_0}$, where u_0 random element of $\mathbb{Z}_p^*$ cannot be proved to satisfy this condition for all $x \in D$. Our construction is proven secure against a non-adaptive adversary, therefore it uses an arbitrary initial exponent u_0 according to the attacker's queries and a guess for the attack. Random elements can be safely accumulated in a non-adaptively sound accumulator, since those elements are chosen without using any information about the accumulator, according to [6].

D. Soundness proof of our generic construction for universal dynamic accumulator (Figure 3)

We provide the proof of Theorem 3.

Proof. Let $\mathcal{A}$ be an adversary that breaks the adaptive soundness of $acc = (\mathsf{ACC}_1, \mathsf{ACC}_2)$. Then, there is a reduction $\mathcal{R}$ that breaks the adaptive soundness of acc_R, where acc_R is an adaptively-sound positive accumulator.
$\mathcal{A}$ is allowed to submit accumulator update queries to $\mathcal{R}$ and $\mathcal{R}$ to a challenger Ch. Queries are of the form accumulator.Query(update, add/del, element to be added/deleted).

Ch runs Gen($1^\lambda, \emptyset$) and returns $acc_R.v_0$, params to $\mathcal{R}$. $\mathcal{R}$ initializes $\mathsf{ACC}_1.v_0 = acc_R.v_0$, and asks for acc_R.Query(update, add,x), $\forall x \in D$ and gets back $acc_R.v_D$. It also keeps a set L for all the currently accumulated elements in acc_R, S for all elements in ACC_1 and $\overline{S}$ for ACC_2. R sets $L, S = \emptyset, \overline{S} = D$, $\mathsf{ACC}_2.v_0 = acc_r.v_D$ and publishes $acc.v_0 = (\mathsf{ACC}_1.v_0, \mathsf{ACC}_2.v_0)$, params to $\mathcal{A}$, who can run the following queries polynomial times:

- $\mathcal{A}$ performs acc.Query(update,add,x_1) ($t = t_1$):
  $\forall y \in L$, $\mathcal{R}$ asks for acc_R.Query(update, del, y).
  $\forall x \in S$, $\mathcal{R}$ asks for acc_R.Query(update, add, x).
  $\mathcal{R}$ asks for acc_R.Query(update, add, x_1) and returns $\mathsf{ACC}_1.v_{t_1} = acc_R.v, w_{x_1}$, upmsg to $\mathcal{A}$.
  $\mathcal{R}$ sets $L = S \cup \{x_1\}$ and $\forall y \in L$, $\mathcal{R}$ asks for acc_R.Query(update, del, y).
  $\mathcal{R}$ sets $L = D - S$ and $\forall y \in L$, $\mathcal{R}$ asks for acc_R.Query(update, add, y).
  $\mathcal{R}$ asks for acc_R.Query(update, del, x_1) and returns $\mathsf{ACC}_2.v_{t_1} = acc_R.v$, upmsg to $\mathcal{A}$.
  $\mathcal{R}$ sets $S = S \cup \{x_1\}$ and $L = D - S$
- $\mathcal{A}$ performs acc.Query(update,del,y_2) ($t = t_2$):
  $\forall y \in L$, $\mathcal{R}$ asks for acc_R.Query(update, del, y).
  $\forall x \in S$, R asks for acc_R.Query(update, add, x).
  $\mathcal{R}$ asks for acc_R.Query(update, del, x_2) and returns $\mathsf{ACC}_1.v_{t_2} = acc_R.v$, upmsg to $\mathcal{A}$.
  $\mathcal{R}$ sets $L = S - \{x_2\}$ and $\forall y \in L$, $\mathcal{R}$ asks for acc_R.Query(update, del, y).
  $\mathcal{R}$ sets $L = D - S$ and $\forall y \in L$, $\mathcal{R}$ asks for acc_R.Query(update, add, y).
  $\mathcal{R}$ asks for acc_R.Query(update, add, x_2) and returns $\mathsf{ACC}_2.v_{t_2} = acc_R.v, \overline{w_{x_2}}$, upmsg to $\mathcal{A}$.
  $\mathcal{R}$ sets $S = S - \{x_2\}$ and $L = D - S$

At $t = t^*$, $\mathcal{A}$ performs a valid forgery (x^*, w^*) to acc. If $w^* = w_{x*}$, $x^* \notin acc$ and VerMem($\mathsf{ACC}_1.v_{t^*}, x^*, w^*$)=1: $\mathcal{R}$ asks for deletion of L and addition of S to acc_R and achieves a forgery (x^*, w^*) to acc_R with the same probability. Otherwise, if $w^* = \overline{w_{x*}}$, $x^* \in acc$ and VerMem($\mathsf{ACC}_2.v_{t^*}, x^*, w^*$)=1: $\mathcal{R}$ asks for deletion of L and addition of $D - S$ to acc_R and achieves a forgery (x^*, w^*) to acc_R. □

E. Zero-Knowledge property of our universal bilinear ZK-Accumulator by two positive accumulators (Figure 4)

Here we provide the proof of Theorem 4.

Proof. Sim is defined to work as follows:

- Sim generates the pairing instance/parameters. It then picks a random $a \in \mathbb{Z}_p^*$ and returns the public params to $\mathcal{A}$. $\mathcal{A}$ picks a set S. Then Sim picks random $r_1, r_2 \in \mathbb{Z}_p^*$, stores them and sends back to $\mathcal{A}$ the accumulator which consists of two pieces (g^{r_1}, g^{r_2}) and initializes an empty list C of triples (to keep track of the witnesses).

- $\mathcal{A}$ asks for an update add(x)/del(x):
  Sim runs D(update,x,add/del,S) and if it outputs 1, Sim picks random $r'_1, r'_2 \in \mathbb{Z}_p^*$ and returns to $\mathcal{A}$ the accumulator values $(g^{r'_1}, g^{r'_2})$. Then updates stored values $r_1 = r'_1$, $r_2 = r'_2$, $C = \emptyset$ and S.
- $\mathcal{A}$ queries element x:
  Sim checks if $(x, -, -) \in C$. If yes returns $\overline{w_x}/w_x$. Otherwise, it queries D(query,x,S). If gets back 1, it computes $w_x = g^{r_1(x+a)^{-1}}$ and sends it to $\mathcal{A}$. Otherwise, it computes $\overline{w_x} = g^{r_2(x+a)^{-1}}$ and sends it to $\mathcal{A}$. It also appends (x,non-membership/membership,$\overline{w_x}/w_x$) to C.

The returned values are distributed the same as in the real game. For more details, we refer to the corresponding proof of [23] for the case of the distribution of membership witnesses and the accumulator value in the presence of updates. □

F. Applying batching techniques to our constructions

Recent results for RSA accumulators [12], suggest that batching witnesses in one aggregated witness along with a proof of correct aggregation can bring the size of the witness for multiple elements and total verification time (in terms of group operations) of a (non)membership proof down to constant and could potentially be applied to our constructions as well. Interestingly though, the construction of a batched non-membership proof for the RSA accumulator depends on both the number of batched elements and S. If we apply such an aggregation technique for non-membership proofs to our generic construction for universal dynamic accumulator (Figure 3), S would not affect the proof cost (since we skip non-membership proofs all-together).

Additionally [12] proposes a technique to batch accumulator value updates. In the trapdoorless setting, they show how an untrusted entity can add or delete a bunch of elements and prove correctness of the accumulator update that corresponds to these elements with reduced communication cost for the proof. Note that this does not reduce the communication cost of update messages, as it would break the communication bound [13]. If other users have elements in the accumulator they need to update their witnesses with individual update messages. Therefore, our communication gain for instantiation of Figure 2 still holds in an environment of batched updates. Alternatively, it has been proposed that a central entity recreates and redistributes all witnesses after a batched update. By applying this technique to our instantiation of Figure 2 we can get rid of the redistribution communication cost in the case of an update that consists only of additions or achieve less total computation time before distributions of witnesses ($n|d|$ for n witness holders). The same holds for batch witness updates in the trapdoor-based setting.

In [41] the manager's broadcast message is computed using the trapdoor, in order for users to update their witnesses more efficiently. The communication cost is still linear to the update. In this case, instead of the individual elements, the update message is a common polynomial where individual witness holders can use their element as an input. The number of coefficients of the polynomial (as well as its degree) equals the number of updates, therefore follows the lower communication bound. Applying such techniques on instantiation of Figure 2 can benefit the computation time for witness holders while still saving from communication after batch additions (no need to update/broadcast).

G. Instantiations for Zero-Knowledge Proofs of Membership in Bilinear Accumulators

Instantiation for Figure 2: We give an instantiation for the case where $\mathsf{ACC_A}$ is instantiated with strong Boneh-Boyen signatures III-B and $\mathsf{ACC_{NA}}$ is instantiated by our protocol in Figure 1.

For the signature scheme of Section III-B, a user can commit to an element m that satisfies the **Verify()** algorithm. The proof is an adaptation of [3], where instead of proving knowledge of a signature on a committed message m and revealing the message, we prove knowledge of a signature on m along with a commitment to m.

For the case of $\mathsf{ACC_{NA}}$ (instantiated with protocol of Figure 1), the following proofs, adapted from [4], [38], prove in zero-knowledge that the user knows the opening to a commitment to an element r and also knowledge of a witness w that satisfies the membership verification equation of the construction if Section III-A (this is $\mathsf{ZKP_1}$ below).

Let $D = \mathbb{G}_q = \langle h_i \rangle, i = 0, 1, 2$, $\mathbb{G} = \langle g_i \rangle, i = 0, 1$ be publicly known generators to be used for commitments.

$$\mathsf{ZPK_1} = \left\{ \ (w, r, \tilde{r}) : e(w, g_0^r g_0^a) = e(v, g_0) \wedge C = g_0^r g_1^{\tilde{r}} \ \right\}$$

which can be converted to the following proof that hides the witness w:

$$\mathsf{ZPK'_1} = \left\{ \begin{array}{c} (r_1, r_2, r, \delta_1, \delta_2, \tilde{r}) : w_1 = g_0^{r_1} g_1^{r_2} \wedge w_1^r = g_0^{\delta_1} g_1^{\delta_2} \\ \wedge \ \frac{e(w_2, g_0^a)}{e(v, g_0)} = e(w_2, g_0)^{-r} e(g_1, g_0)^{\delta_1} e(g_1, g_0^a)^{r_1} \\ \wedge C = g_0^r g_1^{\tilde{r}} \end{array} \right\}$$

where $r_1, r_2 \leftarrow \mathbb{Z}_p$, $w_1 = g_0^{r_1} g_1^{r_2}$, $w_2 = w g_1^{r_1}$, $\delta_1 = r_1 r$, $\delta_2 = r_2 r$.

Instantiation for Figures 3 and 4: The above proof can also be used in the positive accumulator of [34], as well as in our bilinear instantiations for our universal modular accumulators of Figures 3 and 4.

Instantiating ZKP for membership verification (that is done in the bilinear pairing environment) can be done more efficiently using the Groth-Sahai system [25] when there is no need for the user to hide the element accumulated (corresponds to non-bilinear group arithmetic) but still needs to hide the witness.