RSA-Based Dynamic Accumulator without Hashing into Primes

Victor Youdom Kemmoe and Anna Lysyanskaya

Brown University
`{vyoudomk, anna}@cs.brown.edu`

Abstract. A cryptographic accumulator is a compact data structure for representing a set of elements coming from some domain. It allows for a compact proof of membership and, in the case of a universal accumulator, non-membership of an element x in the data structure. A dynamic accumulator, furthermore, allows elements to be added to and deleted from the accumulator.

Previously known RSA-based dynamic accumulators were too slow in practice because they required that an element in the domain be represented as a prime number. Accumulators based on settings other than RSA had other drawbacks such as requiring a prohibitively large common reference string or a trapdoor, or not permitting deletions.

In this paper, we construct RSA-based dynamic accumulators that do not require that the accumulated elements be represented as primes. We also show how to aggregate membership and non-membership witnesses and batch additions and deletions. We demonstrate that, for 112-bit, 128-bit, and 192-bit security, the efficiency gains compared to previously known RSA-based accumulators are substantial, and, for the first time, make cryptographic accumulators a viable candidate for a certificate revocation mechanism as part of a WebPKI-type system. To achieve an efficient verification time for aggregated witnesses, we introduce a variant of Wesolowski's proof of exponentiation (Journal of Cryptology 2020) that does not require hashing into primes.

1 Introduction

A cryptographic accumulator [BdM94] is a compact data structure for representing a set of elements that allows for a compact proof of membership and, in the case of a universal accumulator, non-membership. This makes it attractive for certificate issue and revocation, especially in a distributed setting. The idea is that membership in a dynamically updated set $\mathcal{S}$ is determined by a single value acc (called *the accumulator value*); in order to demonstrate that $x \in \mathcal{S}$, one additionally needs a witness w_x, also of a small, fixed size. acc can be efficiently updated as values are added to and deleted from $\mathcal{S}$.

Benaloh and de Mare [BdM94] introduced cryptographic accumulators and gave the first construction, which was based on RSA. In it, the accumulator's public key is an RSA modulus $n = pq$[1]; an initial value, $\mathsf{acc}_\emptyset \leftarrow \mathbb{Z}_n^*$ that corresponds to the empty set is also picked. The value $\mathsf{acc}_\mathcal{S} = \mathsf{acc}_\emptyset^{\prod_{x\in\mathcal{S}} x}$ represents a set $\mathcal{S}$; for now, let us think of elements of $\mathcal{S}$ as positive integers. We say that $\mathsf{acc}_\mathcal{S}$ is the *accumulator* for $\mathcal{S}$. The witness w_x that $x \in \mathcal{S}$ is the value $w_x = \mathsf{acc}_\emptyset^{\prod_{x'\in\mathcal{S},x'\neq x} x'}$; to verify that $x \in \mathcal{S}$ using this witness, check that $(w_x)^x = \mathsf{acc}_\mathcal{S}$. To add y to the accumulator, Benaloh and de Mare let the value $\mathsf{acc}_{\mathcal{S}\cup\{y\}} = \mathsf{acc}_\mathcal{S}^y$ become the new accumulator value; publishing the value y makes it possible to update all the witnesses: $w_x := (w_x)^y$. (The original proposal did not provide for efficient deletion of elements.)

It is easy to see that this original proposal for a cryptographic accumulator requires some tweaking to achieve soundness, i.e., to ensure that no polynomial-time adversary could find a witness w_y for $y \notin \mathcal{S}$. For example, for a composite integer $x \in \mathcal{S}$, $x = x_1x_2$ for $x_1 > x_2 > 1$, $(w_x)^{x_1}$ will pass as the witness for x_2. A natural fix would be to parameterize by ℓ and require that $\mathcal{S} \subseteq \{2^\ell, \ldots, 2^{\ell+1} - 1\}$. This would rule out the possibility that both $x = x_1x_2$ and $x_1 < x$ can be in $\mathcal{S}$. However, unfortunately, this restriction is not sufficient[2]. Aware of this, Benaloh

[1] It is important that p and q be *safe primes*, i.e., $p = 2p' + 1$ and $q = 2q' + 1$ where p' and q' are both primes

[2] Let $x = x_1x_2$ and $y = y_1y_2$ where x_1, x_2, y_1, y_2 are all distinct and relatively prime to each other, and $2^\ell < x_1y_1 < 2^{\ell+1}$. For $z = x_1y_1$, we can compute w_z such that $w_z^z = \mathsf{acc}_\mathcal{S}$ from the values x, y, w_x, w_y. This is done by using the extended Euclidean GCD algorithm to find a, b such that

and de Mare argued that in their proposed applications, the value z for which the adversary would wish to provide a phony witness, will not be under the control of the adversary, but in fact will be chosen at random. They further argued (somewhat informally) that this would indeed result in a sound accumulator, i.e., one in which the polynomial-time adversary cannot compute w_z if $z \notin \mathcal{S}$. Formalizing this argument is one of the contributions of our paper.

Barić and Pfitzmann [BP97] showed that, if the domain of the accumulator is restricted to *prime* integers, then Benaloh and de Mare's construction is sound under the strong RSA assumption. Camenisch and Lysyanskaya [CL02] adapted this prime number accumulator construction so that the accumulator value can be efficiently updated not just when an element is added to the set, but also when one is deleted. Li, Li, and Xue [LLX07] further enhanced it to allow efficient witnesses not just of membership in the set represented by acc, but also of non-membership. Peng and Bao [PB10] proposed an attack on Li, Li, and Xue non-membership procedure that relies on obtaining the value $\mu = \prod_{s \in \mathcal{S}} s \bmod \phi(n)$, where $\mathcal{S}$ is the accumulated set. However, this attack is irrelevant to us since the value μ is never handed to an adversary.

However, in spite of the significant improvements in the functionality and security properties of RSA-based accumulators these subsequent works provided, RSA-based accumulators were considered impractical because of the requirement that $\mathcal{S} \subset \mathtt{PRIMES}$. In order to, for example, use them to handle certificate revocation, it was necessary to first represent a cryptographic certificate as a prime integer. In some limited applications this may not present a problem (for example, in CL anonymous credentials [CL01, CL03, Lys02], there is always a component of the credential that is already required to be a prime integer), but in general, one would need a hash function that maps its input domain to $\mathtt{PRIMES}$. That generally incurs an $O(\log N)$ overhead, where N is the upper bound of the integer interval from which primes are sampled. (See Gennaro, Halevi and Rabin [GHR99] for an analysis of how to efficiently hash to primes.)

Alternative constructions exist as well, but they have drawbacks, too. The bilinear-map-based construction of Nguyen [Ngu05] and follow-up work [ATSM09] handles deletions extremely efficiently, but requires either public parameters whose size is linear in the upper bound of the number of elements that can be added to acc, or that a trusted participant in possession of a trapdoor compute the accumulator value. Moreover, in the absence of a trusted party with the trapdoor, adding new elements to the set is as costly as computing the accumulator value from scratch. The Merkle-tree-based construction of Reyzin and Yakoubov [RY16] (and the earlier one by Crosby and Wallach [CW09]) has logarithmic (rather than constant) in the size of $\mathcal{S}$ witnesses, and also does not support deletions. These constructions can be combined to achieve efficient add updates and support deletions at the same time [BCD$^+$17, BKR24]; however, a prohibitively large common reference string (or a trusted third party with the trapdoor) is still needed to implement the combined construction.

Our contributions. We propose a random-oracle-based version of the RSA accumulator that does not require hashing to primes, and is therefore much more efficient in practice than previous RSA-based accumulators. As in prior work, the public key is an RSA modulus n, and the initial accumulator value is $\mathsf{acc}_\emptyset \leftarrow \mathbb{Z}_n^*$. The accumulator value corresponding to the set $\mathcal{S}$ is $\mathsf{acc}_\mathcal{S} = \mathsf{acc}_\emptyset^{\prod_{x \in \mathcal{S}} H(x)}$, where H is an appropriate hash function that we model as a random oracle in the security analysis, where we prove security under the strong RSA assumption. We show that this accumulator allows for dynamic additions (easy to see) and deletions (somewhat more complicated), and adapt Li, Li and Xue's techniques to show that, in addition to witnesses of membership, this accumulator allows for witnesses of non-membership.

In addition, we propose a version of Wesolowski's proof of exponentiation (PoE) [Wes20] that does not require hashing to primes. Using our PoE, we show that witnesses can be securely batched under the adaptive root assumption [Wes20] and the strong RSA assumption. In other words, an aggregated witness $w_{\mathcal{S}'}$ for the subset $\mathcal{S}' \subseteq \mathcal{S}$ is of size $|w_{\mathcal{S}'}| < \sum_{x \in \mathcal{S}'} |w_x|$; similarly, we can batch witnesses of non-membership such that the non-membership witness $\bar{w}_{\mathcal{S}^*}$ for the set $\mathcal{S}^*$ such that $\mathcal{S}^* \cap \mathcal{S} = \emptyset$ is of size $|\bar{w}_{\mathcal{S}^*}| < \sum_{x \in \mathcal{S}^*} |\bar{w}_x|$. Update information necessary for updating witnesses can be batched as well.

$ax + by = 1$ and using the trick due to Shamir: first, let $w = w_x^b w_y^a$. Note that $w^{xy} = (w_x^b w_y^a)^{xy} = w_x^{xyb} w_y^{xya} = \mathsf{acc}_\mathcal{S}^{yb} \mathsf{acc}_\mathcal{S}^{ax} = \mathsf{acc}_\mathcal{S}^{ax+by} = \mathsf{acc}_\mathcal{S}$. Thus $w_z = w^{x_2 y_2}$ will pass as the witness for $z = x_1 y_1$: $w_z^z = (w^{x_2 y_2})^{x_1 y_1} = w^{xy} = \mathsf{acc}_\mathcal{S}$.

Benefits to certificate revocation systems. Let us go over the promise that dynamic accumulators hold (but so far have not delivered on) when it comes to certificate revocation of a system such as WebPKI, which is the PKI our browsers rely on for TLS.

For simplicity, suppose that the certification authority (CA) responsible for issuing certificates is also responsible for revoking them; let us see how it would handle revocation using a dynamic accumulator. Let acc_t correspond to the accumulator value at time t; this accumulator value represents all of the current (unrevoked) certificates, and it is signed by the CA. In order to convince a verifier that its certificate x is still valid (i.e. has not been revoked), a web server needs a witness that its certificate is in the accumulator acc_t. This is a step that needs to be relatively practical, but a server can be reasonably expected to have the corresponding connectivity and computational resources.

Verification of the current validity of certificate x is the part conducted by a browser, on a potentially limited device, both from the computational and communication point of view, and therefore it is the part that needs to be optimized. If dynamic accumulators are to be used in this scenario, then this step would involve just checking that acc_t is fresh (e.g., the CA's signature on it includes a relatively recent time stamp) and that the server has presented the witness w_x that x is in the set corresponding to acc_t; no communication-intensive steps such as table lookups are needed for verification purposes. The fact that acc_t has a small size makes it a very attractive option for disseminating revocation information that addresses a real need: for example, in WebPKI, mobile browsers hardly even check for revocation information because this information is so unwieldy [LTZ+15]; the current front-runner alternative, CRLite [LCL+17] (put to use by the Mozilla family of browsers in 2020), still requires that a browser receive around 5Mb of data in order to be able to verify that a certificate has not been revoked.

A suitable cryptographic accumulator can potentially offer a significant improvement for WebPKI. Let us see why our proposed construction is up to the task. First, consider the client's side of the transaction, i.e., the step where the browser verifies that the server's certificate x has not been revoked. In addition to verifying the CA's signature on acc_t, the client in our construction needs to also verify that $w_x^{H(x)} = \mathsf{acc}_t$. This involves one application of a standard hash function and one modular exponentiation, which are both doable on browser-capable devices.

The fact that, in our construction, the hash function does not need to hash to primes makes an incredible difference: as we discuss in more detail in Section 7, hashing a 4KB input to even a small, 264-bit prime (which is a reasonable length needed to avoid collisions) takes about 13 ms of CPU time on a modern laptop, which can contribute to a significant overhead for a CA that must issue a very large number of certificates because it will have to hash each certificate into a prime. Eliminating this costly step can make the RSA accumulator a practical, viable candidate for use in this scenario. We show that we get the same level of security that one would get by hashing into a 264-bit prime at the expense of letting the length of $H(x)$ be 1704 bits. This necessitates a more involved modular exponentiation, but in our experiment comparing running times of the implementations of both approaches , i.e., hashing then performing modular exponentiation, it was still about 3 times faster than hashing into a 264-bit prime (see Section 7 for more details).

Next, we need to make sure that the costs to the CA and server are also reasonable. Here, we have two options: either the CA has a trapdoor to the accumulator (corresponding to an increased amount of trust the system places on the CA, which might not be a desirable design choice) or not. In the former case, a new element x can be added to the accumulator acc_t without needing to update it to a different value acc_{t+1} (see Section 5 for this flavor of the construction): using the trapdoor, the CA will compute w_x and communicate it to the server whose certificate is x. To handle deletions in the former case, and both additions and deletions in the latter case, the CA will have to update the accumulator from acc_t to acc_{t+1}, and publish some additional information that would allow each server to update its witness. As we show in Section 6.2, this information can be batched such that many certificates can be added (in the trapdoorless setting) or revoked (i.e. deleted from the accumulator) on a single update. The information necessary to update a server's membership witness would just be the list of the revoked certificates (note that each certificate can be represented just by a short hash) and a single element of the group $\mathbb{Z}_n^*$. Although this design would require that each server regularly download and process lists of previously revoked certificates, this type of load is comparable to WebPKI, and therefore not unrealistic in practice for a server, while offering clients vastly better efficiency, and requiring that the CAs do a comparable amount of work as in current systems.

Technical roadmap. Our first observation is that, if a set $\mathcal{S}$ consists of a polynomial number (in the security parameter ℓ) of odd integers drawn uniformly at random from the set $\{2^{\ell-1}, 2^{\ell}-1\}$ (i.e. they are random odd ℓ-bit integers whose most significant bit is 1; from now on, we will denote this set $\mathsf{Odds}(2^{\ell-1}, 2^{\ell}-1)$), the probability that for some $x \in \mathcal{S}$, $x \mid \prod_{x' \in \mathcal{S}, x' \neq x} x'$ (i.e., x divides the product of the rest of the elements of $\mathcal{S}$) is negligible in ℓ. More precisely, if x is chosen uniformly at random from $\mathsf{Odds}(2^{\ell-1}, 2^{\ell}-1)$, then with overwhelming probability, its largest prime factor's bit length is at least $k = \Omega(\ell^c)$ for a constant c; as we will see in Section 2.1, $c = 1/4$ is possible, and it translates into overwhelming probability $1 - 2^{\sqrt[4]{\ell}}$ of the largest prime factor of x having at least k bits. Since there are at least $2^{k-\log k}$ primes of length at least k, and a random number is a multiple of p with probability $1/p$, by the birthday bound, a super-polynomial $\Omega(2^{(k-\log k)/2})$ samples would have to be taken for the largest prime factor to repeat.

This observation, formalized in Section 2.1, yields a proof of security in the random oracle model for the following flavor of the RSA-based accumulator: $\mathsf{acc}_{\mathcal{S}} = \mathsf{acc}_{\emptyset}^{\prod_{x \in \mathcal{S}} H(x)}$, where $H : \{0,1\}^* \mapsto \mathsf{Odds}(2^{\ell-1}, 2^{\ell}-1)$ is a hash function that will be modeled as a random oracle in the proof of security.

We give a reduction from an adversary that breaks the soundness of this accumulator in the random-oracle model to solving the flexible RSA problem, contradicting the strong RSA assumption. In a nutshell, if the adversary provides a phony witness w_y for $y \notin \mathcal{S}$, then $d = \gcd(H(y), \prod_{x \in \mathcal{S}} H(x)) < H(y)$, by our observation. Since the witness verifies, $w_y^{H(y)/d} = \mathsf{acc}_{\emptyset}^{\prod_{x \in \mathcal{S}} H(x)/d}$; at the same time, $\gcd(H(y)/d, \prod_{x \in \mathcal{S}} H(x)/d) = 1$. Thus, we can use Shamir's trick (Lemma 1) to efficiently compute u such that $u^{H(y)/d} = \mathsf{acc}_{\emptyset}$, which breaks the flexible RSA problem where the challenge is $(n, \mathsf{acc}_{\emptyset})$.

It is easy to see that dynamic additions to this accumulator are possible: just as in the original Benaloh and de Mare construction, for $\mathsf{acc}_{\mathcal{S}} = \mathsf{acc}_{\emptyset}^{\prod_{s \in \mathcal{S}} H(s)}$, $\mathsf{acc}_{\mathcal{S} \cup \{y\}} = \mathsf{acc}_{\mathcal{S}}^{H(y)} = \mathsf{acc}_{\emptyset}^{H(y) \prod_{s \in \mathcal{S}} H(s)}$, the value $w'_x = w_x^{H(y)}$ is the witness that $x \in \mathcal{S} \cup \{y\}$ if w_x is the witness for $x \in \mathcal{S}$, since $(w'_x)^{H(x)} = w_x^{H(x)H(y)} = \mathsf{acc}_{\mathcal{S}}^{H(y)} = \mathsf{acc}_{\mathcal{S} \cup \{y\}}$. However, deletions are not as seamless: the Camenisch-Lysyanskaya observation that the Shamir trick can be used to update the witness for x after deleting y would require that $\gcd(H(x), H(y)) = 1$, which would be the case if we hashed to primes, but is not necessarily the case when we hash to $\mathsf{Odds}(2^{\ell-1}, 2^{\ell}-1)$.

In order to be able to efficiently update membership witnesses when a deletion has occurred, we generalize the notion of what counts as a membership witness in a way that still preserves soundness: Even though more values count as potential witnesses, the adversary will not be able to find any of them for a false statement. A valid membership witness will now consist of two components, $w_x = (\mathtt{w}, \mathtt{s})$ such that $\mathtt{s}$ is a multiset/tuple of small factors of $H(x)$ such that $\mathtt{w}^{\mathtt{x}} = \mathsf{acc}_{\mathcal{S}}$, where $\mathtt{x} = H(x)/\prod_{s \in \mathtt{s}} s$. By "small," we mean that each $s \in \mathtt{s}$ has bit length less than k. As we show in Section 4.2, generalizing witnesses this way does not detract from the soundness of the construction, but it allows efficient updates of membership witnesses. Suppose that $\gcd(H(x), H(y)) = s > 1$. Note that, if $w_x = (\mathtt{w}, \mathtt{s})$ is a valid witness for x, then so is $w'_x = (\mathtt{w}^s, \mathtt{s} \cup \{s\})$. Since $\gcd(H(x)/s, H(y)/s) = 1$, Shamir's trick works. For non-membership witnesses, notice that for an accumulated set $\mathcal{S}$ and $x \notin \mathcal{S}$, $H(x) \mid \prod_{x' \in \mathcal{S}} H(x')$ with negligible probability, which means that there exists $r \in \mathbb{Z}$ with a large prime factor of at least k-bit such that $r \mid H(x)$ and $\gcd(r, \prod_{x' \in \mathcal{S}} H(x')) = 1$. Using r, we can apply Li, Li and Xue technique to obtain a non-membership witness for x.

In Section 4, we present our universal dynamic accumulator that works over large odd integers in the random oracle model. In Section 5 we focus on the positive dynamic accumulator with a trapdoor, which allows the holder of the trapdoor to add elements to the accumulator without updating acc. This is an important use case to consider in view of the application to certificate revocation described above. In Section 6, we get to the question of batching witnesses. In this section, we first recall the proof of exponentiation (PoE) protocol due to Wesolowski [Wes20]. In this protocol, a prover capable of a long exponentiation convinces a verifier that $v^e = u$ where e is an integer so large that a verifier cannot carry out the exponentiation himself, and u and v are elements of a group of unknown order (such as $\mathbb{Z}_n^*$). Previously [Wes20, BBF19] it was known that this protocol can be made non-interactive in the random-oracle model by hashing into primes. In a contribution that is of independent interest, we show that this protocol can be adapted to drop the hash-to-primes requirement. Armed with PoE as a tool, we show that both membership and non-membership witnesses can be batched, and, moreover, accumulator updates can be batched as well. Finally, in Section 7, we experimentally compare prior work with our own.

2 Preliminaries

Notations. A function $f : \mathbb{N} \to [0,1]$ is negligible if $f(x) = o(x^{-c})$ for all $c \in \mathbb{N}$. We use $\mathsf{negl}(\cdot)$ to denote a negligible function. We denote the security parameter by λ. For $n \in \mathbb{N}$, we use $[n]$ to denote the set $\{1, \ldots, n\}$, and QR_n to denote the group of quadractic residues modulo n. For a finite set $\mathcal{S}$, we use $\#\mathcal{S}$ to denote its cardinality, $U(\mathcal{S})$ to denote the uniform distribution over $\mathcal{S}$, and $a \leftarrow\!\!\$\ \mathcal{S}$ to denote that a is sampled uniformly at random from $\mathcal{S}$. Let $\mathsf{Odds}(a,b) \overset{\text{def}}{=} \{a \le n \le b : n \equiv 1 \bmod 2\}$. For two functions $h, g : \mathbb{R} \to \mathbb{R}$, we use $h(x) \sim g(x)$ to denote that $\lim_{x\to\infty} h(x)/g(x) = 1$. Sometimes, we use bold character, $\mathbf{z}$, to denote a tuple, and for two tuples $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_m)$, we use $\mathbf{x}\|\mathbf{y}$ to denote their concatenation, i.e., $\mathbf{x}\|\mathbf{y} = (x_1, \ldots, x_n, y_1, \ldots, y_m)$. We say that $x \in \mathbf{x}$ if there exists $i \in [|\mathbf{x}|]$ such that $\mathbf{x}[i] = x$.

Definition 1 (Strong RSA assumption [BP97]). *For all $\lambda \in \mathbb{N}$ and probabilistic poly-time (ppt) adversary $\mathcal{A}$, given $n = pq$, where p and q are $\mathsf{poly}(\lambda)$-bit safe primes, and $u \in \mathbb{Z}_n^*$,*

$$\Pr\left[v^e \equiv u \bmod n \wedge e > 1 \middle| (v,e) \leftarrow \mathcal{A}(1^\lambda, u, n)\right] \le \mathsf{negl}(\lambda)$$

Remark 1. Barić and Pfitzmann [BP97] initially proposed a definition of the strong RSA assumption where p and q are $\mathsf{poly}(\lambda)$-bit primes and e is a prime. Clearly, their version is at least as hard as ours.

Lemma 1 (Shamir's trick [Sha81]). *For all $n, x, y \in \mathbb{N}$, $v, u \in \mathbb{Z}_n^*$ such that $v^x \equiv u^y \bmod n$ and $\gcd(x,y) = 1$, there exists $w \in \mathbb{Z}_n^*$ such that $w^x \equiv u \bmod n$.*

Proof. Since $\gcd(x,y) = 1$, there exists $\alpha, \beta \in \mathbb{Z}$ such that $\alpha x + \beta y = 1$. Let $w = u^\alpha v^\beta$. We have $w^x \equiv u^{\alpha x} v^{\beta x} \equiv u^{\alpha x} u^{\beta y} \equiv u \bmod n$. □

2.1 Number Theoretic Functions

Dickman-ρ function. Let $\rho : \mathbb{R}_{\ge 0} \to \mathbb{R}$ be the continuous solution to the differential equation $u\rho'(u) + \rho(u-1) = 0$ for $u > 1$ subjected to the initial condition $\rho(u) = 1$ for $0 \le u \le 1$. de Bruijn [dB51] proved that for $u > 1$, we have

$$\rho(u) = \exp\left(-u\left(\log u + \log\log u - 1 + O\left(\frac{\log\log u}{\log u}\right)\right)\right)$$

Therefore, $\rho(u) \sim (u \log u)^{-u}$ as $u \to \infty$.

Smooth numbers counting function. A function that will be important for us is a function that will allow us to count y-smooth numbers (numbers whose largest prime factor is less than or equal to y) in an arithmetic progression (sequences of the form $s_i = s_1 + (i-1)d$) define over an interval $[x]$, where $x \in \mathbb{N}$. To this end, let us consider the function

$$\Psi_{a,q}(x,y) \overset{\text{def}}{=} \#\{n \in [x] : (P^+(n) \le y) \wedge (n \equiv a \bmod q)\}$$

where $P^+(\cdot)$ is the function returning the largest prime factor of an integer. Based on the survey of Hildebrand and Tenenbaum [HT93], it follows that for $u = \log x / \log y$ and $a, q \in \mathbb{N}$ such that $\gcd(a,q) = 1$, if $u \ll (\log_2 x)^{1-\epsilon}$, with $\epsilon > 0$, we have

$$\Psi_{a,q}(x,y) = \frac{x\rho(u)}{q}\left(1 + O\left(\frac{1}{\sqrt{u \log y}}\right)\right)$$

Hence, $\Psi_{a,q}(x,y) \sim (x\rho(u))/q$ as $y \to \infty$.

Lemma 2. *Given a sufficiently large $\ell \in \mathbb{N}$, let $a \leftarrow\!\!\$ \ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$. For every $1 \le c \le \sqrt[4]{\ell}$,*

$$\Pr\left[P^+(a) \le 2^{c\sqrt{\ell}}\right] \le \left(\frac{\sqrt[4]{\ell}}{4}\log \ell\right)^{-\sqrt[4]{\ell}}$$

Proof. Let $1 \le c \le \sqrt[4]{\ell}$ and suppose $a \leftarrow\!\!\$ \ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$. Let η be the number of integers in $\mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ whose largest prime factor is less or equal to $2^{c\sqrt{\ell}}$. We have

$$\begin{aligned}
\eta &= \Psi_{1,2}(2^\ell - 1, 2^{c\sqrt{\ell}}) - \Psi_{1,2}(2^{\ell-1}, 2^{c\sqrt{\ell}}) \\
&= \frac{1}{2}\left((2^\ell - 1)\left(\frac{\sqrt{\ell}}{c}\log\left(\frac{\sqrt{\ell}}{c}\right)\right)^{-\sqrt{\ell}/c} - 2^{\ell-1}\left(\frac{\ell-1}{c\sqrt{\ell}}\log\left(\frac{\ell-1}{c\sqrt{\ell}}\right)\right)^{-(\ell-1)/c\sqrt{\ell}}\right) \\
&\approx 2^{\ell-2}\left(\frac{\sqrt{\ell}}{c}\log\left(\frac{\sqrt{\ell}}{c}\right)\right)^{-\sqrt{\ell}/c} \qquad \text{(since for large } \ell,\ \frac{2^\ell - 1}{2^\ell} \approx 1 \text{ and } \frac{\ell - 1}{\ell} \approx 1) \\
&\le 2^{\ell-2}\left(\frac{\sqrt[4]{\ell}}{4}\log \ell\right)^{-\sqrt[4]{\ell}} \qquad \text{(setting } c = \sqrt[4]{\ell})
\end{aligned}$$

Hence, $\Pr\left[P^+(a) \le 2^{c\sqrt{\ell}}\right] = \frac{\eta}{2^{\ell-2}} \le \left(\frac{\sqrt[4]{\ell}}{4}\log \ell\right)^{-\sqrt[4]{\ell}}$ □

Corollary 1. *Given a sufficiently large $\ell \in \mathbb{N}$, for every constant $1 \le c \le \sqrt[4]{\ell}$, $m \in \mathbb{N}$, and $a_1, a_2, \ldots, a_m \sim U\left(\mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)\right)$, let E be the event that there exists $i \in [m]$ such that $P^+(a_i) \mid \prod_{j \in [m]\setminus\{i\}} a_j$. Then,*

$$\Pr[\mathsf{E}] \le m^2\left(\frac{1}{2^{\ell^{3/4}}} + \left(\frac{\sqrt[4]{\ell}}{4}\log \ell\right)^{-\sqrt[4]{\ell}}\right)$$

Proof. Let ℓ be a large integer, and suppose $1 \le c \le \sqrt[4]{\ell}, m \in \mathbb{N}$ and $a_1, a_2, \ldots, a_m \sim U\left(\mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)\right)$. Let E_i be the even that $P^+(a_i)$ divides $\prod_{j \in [m]\setminus\{i\}} a_j$. Since $P^+(a_i)$ is a prime, it follows that E_i is exactly the event that there exists $j \in [m] \setminus \{i\}$ such that $P^+(a_i)$ divides a_j. We have

$$\begin{aligned}
\Pr[\mathsf{E}_i] &\le \sum_{j \in [m]\setminus\{i\}} \Pr\left[P^+(a_i) \text{ divides } a_j\right] \\
&\le \sum_{j \in [m]\setminus\{i\}} \Pr\left[P^+(a_i) \text{ divides } a_j \mid P^+(a_i) > 2^{c\sqrt{\ell}}\right] + \Pr\left[P^+(a_i) \le 2^{c\sqrt{\ell}}\right] \\
&\overset{(1)}{\le} (m-1)\left(\frac{1}{2^{c\sqrt{\ell}}} + \left(\frac{\sqrt[4]{\ell}}{4}\log \ell\right)^{-\sqrt[4]{\ell}}\right) \\
&\le (m-1)\left(\frac{1}{2^{\ell^{3/4}}} + \left(\frac{\sqrt[4]{\ell}}{4}\log \ell\right)^{-\sqrt[4]{\ell}}\right)
\end{aligned}$$

Inequality (1) follows from the fact that we have $\#\mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)/P^+(a_i)$ multiples of $P^+(a_i)$ in $\mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$. Since $\mathsf{E} = \cup_{i=1}^m \mathsf{E}_i$, it follows that

$$\Pr[\mathsf{E}] \le \sum_{i=1}^m \Pr[\mathsf{E}_i] \le m^2\left(\frac{1}{2^{\ell^{3/4}}} + \left(\frac{\sqrt[4]{\ell}}{4}\log \ell\right)^{-\sqrt[4]{\ell}}\right)$$

□

3 Cryptographic Accumulator

We recall the definition of universal and positive dynamic accumulators based on [RY16,BCD+17, DHS15, BKR24]. Our definition of non-membership witness creation is borrowed from the work of Baldimtsi, Karantaidou and Raghuraman [BKR24]. For a value a, we use $\hat{a}$ to say that a is optional. We use t to denote a discrete time counter.

Definition 2 (Universal Dynamic Accumulator). *A universal dynamic accumulator for a domain* $\mathcal{M}$ *is a tuple of poly-time algorithms* $\mathsf{UAcc} = (\mathsf{Gen}, \mathsf{Add}, \mathsf{Delete}, \mathsf{NonMemWitCreate}, \mathsf{MemWitUp}, \mathsf{NonMemWitUp}, \mathsf{MemVer}, \mathsf{NonMemVer})$ *with the following properties:*

– $\mathsf{Gen}(1^\lambda, \mathsf{aux}) \to (\mathsf{pp}, \widehat{\mathsf{sk}}, \mathsf{acc}_0)$*: This probabilistic algorithm takes as input the security parameter* 1^λ *and auxiliary information* aux*. It outputs the public parameter* pp*, the (optional) secret parameter* $\widehat{\mathsf{sk}}$*, and an initial accumulator* acc_0*.*
– $\mathsf{Add}(\mathsf{pp}, \widehat{\mathsf{sk}}, \mathsf{acc}_t, x) \to (\mathsf{acc}_{t+1}, w_{x,t+1}, \mathsf{upmsg}_{t+1})$*: This (probabilistic) algorithm takes as input the parameters* pp*,* $\widehat{\mathsf{sk}}$*, the accumulator* acc_t*, and an element* $x \in \mathcal{M}$*. It adds* x *to* acc_t*, producing a new accumulator* acc_{t+1}*, a membership witness* $w_{x,t+1}$ *for* x*, and update information* upmsg_{t+1} *that can be used to update the membership witnesses of other elements in the accumulator.*
– $\mathsf{Delete}(\mathsf{pp}, \widehat{\mathsf{sk}}, \mathsf{acc}_t, x, \widehat{w_{x,t}}) \to (\mathsf{acc}_{t+1}, \mathsf{upmsg}_{t+1})$*: This (probabilistic) algorithm takes as input the parameters* pp*,* $\widehat{\mathsf{sk}}$*, the accumulator* acc_t*, an element* x *that was previously added to* acc_t*, and an optional membership witness* $\widehat{w_{x,t}}$ *for* x *with respect to* acc_t*. It deletes* x *from* acc_t*, producing a new accumulator* acc_{t+1}*, and update information* upmsg_{t+1} *that can be used to update the membership witnesses of other elements in the accumulator.*
– $\mathsf{NonMemWitCreate}(\mathsf{pp}, x, \{\mathsf{upmsg}_i\}_{i=1}^t) \to \bar{w}_{x,t}$*: This (probabilistic) algorithm takes as input the parameter* pp*, an element* x*, a set of update information* $\{\mathsf{upmsg}_i\}_{i=1}^t$*. It returns a non-membership witness* $\bar{w}_{x,t}$ *for* x*.*
– $\mathsf{MemWitUp}(\mathsf{pp}, x, w_{x,t}, \mathsf{upmsg}_{t+1}) \to w_{x,t+1}$*: This (probabilistic) algorithm takes as input the parameter* pp*, an element* x*, a membership witness* $w_{x,t}$ *for* x*, and update information* upmsg_{t+1}*. It returns an updated membership witness* $w_{x,t+1}$ *for* x*.*
– $\mathsf{NonMemWitUp}(\mathsf{pp}, x, \bar{w}_{x,t}, \mathsf{upmsg}_{t+1}) \to \bar{w}_{x,t+1}$*: This (probabilistic) algorithm takes as input the parameter* pp*, an element* x*, a non-membership witness* $\bar{w}_{x,t}$ *for* x*, and update information* upmsg_{t+1}*. It returns an updated non-membership witness* $\bar{w}_{x,t+1}$ *for* x*.*
– $\mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}_t, x, w_{x,t}) \to 0/1$*: This deterministic algorithm takes as input the parameter* pp*, an accumulator* acc_t*, an element* x*, and a membership witness* $w_{x,t}$*, and returns* 1 *if* $w_{x,t}$ *is a witness for* x*'s membership in the set represented by* acc_t*, or* 0 *otherwise.*
– $\mathsf{NonMemVer}(\mathsf{pp}, \mathsf{acc}_t, x, \bar{w}_{x,t}) \to 0/1$*: This deterministic algorithm takes as input the parameter* pp*, an accumulator* acc_t*, an element* x*, and a non-membership witness* $\bar{w}_{x,t}$*, and returns* 1 *if* $\bar{w}_{x,t}$ *is a witness for* x*'s non-membership in the set represented by* acc_t*, or* 0 *otherwise.*

Remark 2. For our universal accumulator definition, we omitted the algorithm $\mathsf{MemWitCreate}$ that allows to create a membership witness for a previously accumulated element because we require the algorithm Add to return a membership witness every time it is invoked. In addition, for our universal accumulator construction (confer Section 4), $\mathsf{MemWitCreate}$ can easily be supported by using the history of update messages $\{\mathsf{upmsg}_i\}_{i=1}^t$ to recover the accumulated set $\mathcal{S}$ an compute the witness of any accumulated element.

Next, we need to define what it means for an accumulator to be correct. Ghosh et al. [GOP+16] provided a correctness definition for universal dynamic accumulators that omitted the witness update algorithms. Reyzin and Yakoubov [RY16] provided a definition for additive accumulators, i.e., those that support dynamic additions but not deletions. Baldimtsi, Canetti and Yakoubov [BCY20] provided a definition for universal dynamic accumulators that includes witness update algorithms. However, their definition was non-adaptive (the sequence of operations applied on an accumulator and the element for which the witness will be tested were selected before the accumulator's parameters were instantiated) and did not enforce a set structure on the sequence of operations applied on an accumulator. Following these works, we provide a correctness definition for positive dynamic accumulators that is adaptive, includes witness update algorithms, and enforces a set structure on the operations applied to an accumulator. The goal

```
$\mathsf{CorrectExp}^{\mathcal{A}}(1^\lambda)$ :
1 :  $\mathsf{pp}, \widehat{\mathsf{sk}}, \mathsf{acc}_0 \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{aux})$
     // $i$ and $j$ represent the times at which $x$ was added and $\bar{w}_y$ was created, respectively
2 :  $x, y, i, j \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Add,Delete}}}(1^\lambda, \mathsf{aux}, \mathsf{pp}, \mathsf{acc}_0)$
3 :  if $x = y \lor x \neq \mathbf{v}[i] \lor y \in \mathbf{v}$ : abort
4 :  $w_{x,t} \leftarrow \mathbf{mwit}[i]$
5 :  $\bar{w}_{y,j} \leftarrow \mathsf{NonMemWitCreate}(\mathsf{pp}, y, \{\mathbf{upmsgs}[1], \ldots, \mathbf{upmsgs}[j]\})$
6 :  for $k \in \{j+1, \ldots, t\}$ do :
7 :    $\bar{w}_{y,k} \leftarrow \mathsf{NonMemWitUp}(\mathsf{pp}, y, w_{y,k-1}, \mathbf{upmsgs}[k])$
8 :  return $\big(\mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}_t, x, w_{x,t}) = 0\big)$
            $\lor \big(\mathsf{NonMemVer}(\mathsf{pp}, \mathsf{acc}_t, y, \bar{w}_{y,t}) = 0\big)$

$\mathcal{O}_{\mathsf{Add,Delete}}(\mathsf{op}, v)$ :
1 :  Initialize $\mathbf{v} \leftarrow (), \mathbf{mwit} \leftarrow (), \mathbf{upmsgs} \leftarrow (), t \leftarrow 0$
2 :  if $v \notin \mathcal{M}$ : abort
3 :  if $\mathsf{op} = \mathsf{add}$ :
4 :    if $v \in \mathbf{v}$ : abort
5 :    $(\mathsf{acc}_{t+1}, w_{v,t+1}, \mathsf{upmsg}_{t+1}) \leftarrow \mathsf{Add}(\mathsf{pp}, \widehat{\mathsf{sk}}, \mathsf{acc}_t, v)$
6 :    $\mathbf{v} \leftarrow \mathbf{v} \| (v), \mathbf{mwit} \leftarrow \mathbf{mwit} \| (w_{v,t+1}), \mathbf{upmsgs} \leftarrow \mathbf{upmsgs} \| (\mathsf{upmsg}_{t+1})$
7 :    for $k_1 \in [t]$ do : // update membership witnesses of previously added elements
8 :      if $\mathbf{v}[k_1] \neq \bot$ : $\mathbf{mwit}[k_1] \leftarrow \mathsf{MemWitUp}(\mathsf{pp}, \mathbf{v}[k_1], \mathbf{mwit}[k_1], \mathsf{upmsg}_{t+1})$
9 :    $t \leftarrow t + 1$
10 :   return $\mathsf{acc}_{t+1}, w_{v,t+1}, \mathsf{upmsg}_{t+1}$
11 : if $\mathsf{op} = \mathsf{del}$ :
12 :   if $v \notin \mathbf{v}$ : abort
13 :   for $k_2 \in [t]$ do : // delete $v$ and its membership witness
14 :     if $\mathbf{v}[k_2] = v$ : $(\mathsf{acc}_{t+1}, \mathsf{upmsg}_{t+1}) \leftarrow \mathsf{Delete}(\mathsf{pp}, \widehat{\mathsf{sk}}, \mathsf{acc}_t, v, \widehat{\mathbf{mwit}[k_2]})$,
15 :                  $\mathbf{v}[k_2] \leftarrow \bot, \mathbf{mwit}[k_2] \leftarrow \bot$
       // append $\bot$ to $\mathbf{v}$ and $\mathbf{mwit}$ to ensure that their lenght matches $t + 1$
16 :   $\mathbf{v} \leftarrow \mathbf{v} \| (\bot), \mathbf{mwit} \leftarrow \mathbf{mwit} \| (\bot), \mathbf{upmsgs} \leftarrow \mathbf{upmsgs} \| (\mathsf{upmsg}_{t+1})$
17 :   for $k_3 \in [t]$ do : // update membership witnesses of previously added elements
18 :     if $\mathbf{v}[k_3] \neq \bot$ : $\mathbf{mwit}[k_3] \leftarrow \mathsf{MemWitUp}(\mathsf{pp}, \mathbf{v}[k_3], \mathbf{mwit}[k_3], \mathsf{upmsg}_{t+1})$
19 :   $t \leftarrow t + 1$
```

Fig. 1: Correctness Game for a universal dynamic accumulator

of a ppt adversary $\mathcal{A}$ that attacks the correctness of the system is to interact with the accumulator (with access to an oracle $\mathcal{O}_{\mathsf{Add,Delete}}$) and produce a correctness error, i.e., a value x in the accumulator whose membership witness fails to verify. More precisely, $\mathcal{A}$ will output elements x and y in the domain of the accumulator with respective up-to-date membership witness $w_{x,t}$ and non-membership witness $\bar{w}_{y,t}$ such that given the most recent value of the accumulator acc_t, either the pair $(x, w_{x,t})$ fails membership verification or the pair $(y, \bar{w}_{y,t})$ fails non-membership verification. The oracle $\mathcal{O}_{\mathsf{Add,Delete}}$ is in charge of executing add and del queries. It has access to the public and secret parameters $(\mathsf{pk}, \widehat{\mathsf{sk}})$ of the accumulator and is initialized with a discrete time counter t and tuples $\mathbf{v}$ that stores elements added to the accumulator, **mwit** that stores membership witnesses of elements in $\mathbf{v}$, and **upmsgs** that stores all update information produced after the execution of add and del queries. After each add or del query, $\mathcal{O}_{\mathsf{Add,Delete}}$ updates the all out-of-date membership witnesses stored in **mwit**, and it keeps track of the most recent value of the accumulator acc_t. We give the precise description of the correctness game in Fig. 1.

Definition 3 (Correctness). *A universal dynamic accumulator* UAcc *for a domain* $\mathcal{M}$ *is correct if for any* $x, y \in \mathcal{M}$ *such that* x *was added in the accumulator and* y *was not, up-to-date and well-formed membership witness* $w_{x,t}$ *and non-membership witness* $\bar{w}_{y,t}$ *pass membership and non-membership verification, respectively, with overwhelming probability. More specifically, for all* $\lambda \in \mathbb{N}$*, all* ppt *adversary* $\mathcal{A}$*,*

$$\Pr\left[\mathsf{CorrectGame}^{\mathcal{A}}(1^\lambda) = 1\right] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{CorrectGame}$ *is defined in Fig. 1.*

Definition 4 (Compactness). *A universal dynamic accumulator* UAcc *for a domain* $\mathcal{M}$ *is compact if for all* $\lambda \in \mathbb{N}$ *and element* $x \in \mathcal{M}$*, we have* $|\mathsf{acc}| = \mathsf{poly}(\lambda)$*, and* $|w_x| = |\bar{w}_x| = \mathsf{poly}(\lambda, |x|)$*.*

Remark 3. Although we require witnesses to have size $\mathsf{poly}(\lambda, |x|)$, which is the case of RSA-based schemes such as [BP97, CL02, LLX07] and Bilinear pairing-based schemes such as [Ngu05, CKS09, ATSM09], it should be noted that Merkle tree-based schemes such as [BLL02, CHKO12, RY16] support witnesses with size $\mathsf{poly}(\lambda, \log|\mathcal{S}|)$, where $\mathcal{S}$ is the set of elements that have been accumulated.

Definition 5 (Universal Dynamic Accumulator Security). *A universal dynamic accumulator* UAcc *for a domain* $\mathcal{M}$ *is secure if for all* ppt *adversary* $\mathcal{A}$ *with oracle access to* $\mathcal{O}_{\mathsf{Add,Delete}}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{r} \mathsf{pp}, \widehat{\mathsf{sk}}, \mathsf{acc}_0 \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{aux}); \\ x^*, w_{x^*,t}, \bar{w}_{x^*,t} \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Add,Delete}}}(1^\lambda, \mathsf{aux}, \mathsf{pp}, \mathsf{acc}_0) : \\ \mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}_t, x^*, w_{x^*,t}) = 1 \\ \wedge\ \mathsf{NonMemVer}(\mathsf{pp}, \mathsf{acc}_t, x^*, \bar{w}_{x^*,t}) = 1 \end{array}\right] \leq \mathsf{negl}(\lambda)$$

where acc_t *is output by* $\mathcal{O}_{\mathsf{Add,Delete}}$*. For this definition,* $\mathcal{O}_{\mathsf{Add,Delete}}$ *is defined as in Fig. 1 except for the following modifications: (1)* $\mathcal{O}_{\mathsf{Add,Delete}}$ *need not keep track of when elements are added or deleted, i.e., instead of storing the tuple* $\mathbf{v}$*, it just keeps track of the current set* $\mathcal{S}$ *representing the accumulated values; (2) it must receive a membership witness as part of a delete query.*

Remark 4. Our security definition for a universal dynamic accumulator is stronger than that of Li, Li and Xue (LLX) [LLX07]. In the LLX definition, the adversary takes as input $(1^\lambda, \mathsf{aux}, \mathsf{pp}, \mathsf{acc}_0)$ (same input as in our definition) and succeeds if it outputs a set $\mathcal{S}$, an element x, and witnesses w and $\bar{w}$ such that both $\mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}, x^*, w_{x^*,t}) = 1$ and $\mathsf{NonMemVer}(\mathsf{pp}, \mathsf{acc}, x^*, \bar{w}_{x^*,t}) = 1$; the difference from our definition is that in LLX, acc is the accumulator value that results in adding each element of S to the initial accumulator acc_0, while in our definition it is the value obtained via a series of queries to $\mathcal{O}_{\mathsf{Add,Delete}}$. Thus, an adversary breaking the LLX security game will also break ours (and will only need the Add queries to the oracle), meaning that if our security definition is satisfied, then so is LLX.

Here is an example of a universal accumulator that satisfies the LLX definition but does not satisfy ours. Suppose we are given a universal accumulator that satisfies our definition of security. We modify it as follows: to set it up, we generate acc_0 using Gen, and output $\mathsf{acc}'_0 = \mathsf{acc}_0 \| 0$. To add an element to the accumulator $\mathsf{acc}'_t = \mathsf{acc}_t \| b$, run Add to update acc_t to acc_{t+1}, and output $\mathsf{acc}'_{t+1} = \mathsf{acc}_{t+1} \| b$, i.e., addition does not change the bit b. But deletion does: the first time a deletion is performed, it flips it from 0 to 1. I.e., to delete from $\mathsf{acc}'_t = \mathsf{acc}_t \| b$, run Delete to update acc_t to acc_{t+1}, and output $\mathsf{acc}'_{t+1} = \mathsf{acc}_{t+1} \| 1$. Another modification is to $\mathsf{NonMemVer}$: if $\mathsf{acc}' = \mathsf{acc} \| 1$, it always accepts.

The adversary playing the LLX security game against this accumulator will not succeed, because it is not allowed to perform deletions, thus the last bit of acc will never be 1, and thus security follows from the security of the underlying universal accumulator. However, if the adversary plays our security game, in which deletion queries are allowed, then it wins easily.

Lipmaa's definition [Lip12] addressed this subtle issue with the LLX definition. His "undeniable" security game allows the adversary to output his own value of the accumulator acc; the adversary wins if he also outputs an element x and its membership witness w (accepted by MemVer) as well as its non-membership witness $\bar{w}$ (accepted by $\mathsf{NonMemVer}$) for acc. Our definition is weaker than Lipmaa's: the adversary in our security game is not free to produce any value for acc; instead, it must be obtained from acc_0 via a series of additions and deletions. Lipmaa's definition is unnecessarily strong for the WebPKI and similar applications in which the accumulator value is trusted to represent a set of valid certificates obtained via a sequence of additions and deletions, and is not up to an adversary to decide. Hence, we formulated our own definition.

Another reasonable formulation of the security game for the universal accumulator would be to run the adversary as in our game (i.e. with access to $\mathcal{O}_{\mathsf{Add,Delete}}$), but to modify the winning condition. We can declare the adversary successful if it produces either a valid membership witness for an element x that he had either never added, or added but subsequently deleted; or a valid non-membership witness for an element x that he has added to the accumulator and never deleted. This approach is known in the literature as collision-freeness [Lip12, DHS15]. We note that, for a universal accumulator satisfying our correctness definition, this formulation is equivalent to the one we give.

Definition 6 (Positive Dynamic Accumulator). *A positive dynamic accumulator for a domain $\mathcal{M}$ is a tuple of poly-time algorithms* $\mathsf{PAcc} = (\mathsf{Gen}, \mathsf{Add}, \mathsf{Delete}, \mathsf{MemWitUp}, \mathsf{MemVer})$ *whose properties are as elaborated in Definition 2.*

Remark 5. For our positive accumulator definition, we omitted the algorithm $\mathsf{MemWitCreate}$ that allows to create a membership witness for a previously accumulated element because we require the algorithm Add to return a membership witness every time it is invoked. In addition, for our positive accumulator construction (confer Section 5), $\mathsf{MemWitCreate}$ will amount to executing the algorithm Add.

The definitions of correctness and compactness for a positive dynamic accumulator PAcc are obtained from the definition of those notions for a universal dynamic accumulator (refer to definitions 3 and 4) by removing the parts regarding non-membership witnesses.

Definition 7 (Positive Dynamic Accumulator Security). *A positive dynamic accumulator* PAcc *for a domain $\mathcal{M}$ is secure if for all* ppt *adversary $\mathcal{A}$ with oracle access to $\mathcal{O}_{\mathsf{Add,Delete}}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{r} \mathsf{pp}, \widehat{\mathsf{sk}}, \mathsf{acc}_0 \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{aux}); \\ x^*, w_{x^*,t} \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Add,Delete}}}(1^\lambda, \mathsf{aux}, \mathsf{pp}, \mathsf{acc}_0) : \\ x^* \notin \mathcal{S} \wedge \mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}_t, x^*, w_{x^*,t}) = 1 \end{array}\right] \leq \mathsf{negl}(\lambda)$$

where acc_t *is output by $\mathcal{O}_{\mathsf{Add,Delete}}$. For this definition, $\mathcal{O}_{\mathsf{Add,Delete}}$ is defined as in Fig. 1 except for the following modifications: (1) $\mathcal{O}_{\mathsf{Add,Delete}}$ need not keep track of when elements are added or deleted, i.e., instead of storing the tuple $\mathbf{v}$, it just keeps track of the current set $\mathcal{S}$ representing the accumulated values; (2) it must receive a membership witness as part of a delete query.*

For the execution of a universal/positive dynamic accumulator, we consider three types of actors:

- an accumulator manager: it is in charge of executing the algorithms $\mathsf{Gen}, \mathsf{Add}$ and Delete.
- a user: it is in charge of managing witnesses by executing the algorithms $\mathsf{NonMemWitCreate}$, $\mathsf{MemWitUp}$, and $\mathsf{NonMemWitUp}$. It can also issue add and del requests to an accumulator manager.
- a verifier: it executes the algorithms MemVer and $\mathsf{NonMemVer}$. Note that since verification algorithms do not take secret parameters as input, a user can be a verifier.

For the rest of the paper, we are going to forgo the discrete time counter t and denote a new version of an accumulator acc with acc' and a new version of a membership witness w_x (resp. non-membership witness $\bar{w}_x$) for an element x with w'_x (resp. $\bar{w}'_x$).

```
– $\mathsf{Gen}(1^\lambda, \perp)$:
    1. Select primes $p, q, p', q'$ such that $p = 2p' + 1$, $q = 2q' + 1$, and $p', q'$ have a length that
       results in an RSA modulus with $\lambda$-bit security. As a result, $\log_2 p' \approx \log_2 q' \gg \lambda$.
    2. Compute $n \leftarrow pq$ and $u \leftarrow\$ \mathrm{QR}_n \setminus \{1\}$.
    3. Return $\mathsf{pp} = (n, u)$, $\mathsf{sk} = 4p'q'$, $\mathsf{acc} = u$.
– $\mathsf{Add}(\mathsf{pp}, \mathsf{acc}, x)$:
    1. Parse $\mathsf{pp}$ as $(n, u)$.
    2. Compute $\mathsf{acc}' \leftarrow \mathsf{acc}^{H(x)} \bmod n$.
    3. Let $\mathsf{s} = (1)$, $w_x = (\mathsf{acc}, \mathsf{s})$ and $\mathsf{upmsg} = (\mathsf{add}, H(x), 1, \mathsf{acc}, \mathsf{acc}')$.
    4. Return $\mathsf{acc}'$ , $w_x$, and $\mathsf{upmsg}$.
– $\mathsf{Delete}(\mathsf{pp}, \mathsf{sk}, \mathsf{acc}, x, w_x)$:
    1. Parse $\mathsf{pp}$ as $(n, u)$.
    2. If $w_x = \perp$ or $\mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}, x, w_x) = 0$, do:
        (a) Compute $\gamma \leftarrow 1/H(x) \bmod \mathsf{sk}$, and let $\delta = 1$.
        (b) Compute $\mathsf{acc}' \leftarrow \mathsf{acc}^\gamma \bmod n$.
    3. Else if $\mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}, x, w_x) = 1$, do:
        (a) Parse $w_x$ as $(\mathtt{w}, \mathsf{s})$, compute $\delta \leftarrow \prod_{i=1}^{|\mathsf{s}|} \mathsf{s}[i]$, and let $\mathsf{acc}' = \mathtt{w}$.
    4. Let $\mathsf{upmsg} = (\mathsf{del}, H(x), \delta, \mathsf{acc}, \mathsf{acc}')$.
    5. Return $\mathsf{acc}'$, and $\mathsf{upmsg}$.
```

Fig. 2: Accumulator Manager's algorithms

4 Universal Dynamic Accumulator Construction

In this section, we present our universal dynamic accumulator in the random oracle model. Our construction is based on [CL02,LLX07] with the exception that we work over large odd integers.

Let $H : \{0,1\}^* \to \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ be a random oracle such that $\ell = \mathsf{poly}(\lambda)$, and $\sqrt{\ell} \leq \tau \leq \ell^{3/4}$ be a value chosen such that for any $x \in \{0,1\}^*$, $P^+(H(x)) > 2^\tau$ with overwhelming probability.

Remark 6. The random oracle H can be instantiated by using a random oracle $H' : \{0,1\}^* \to 1\|\{0,1\}^{\ell-2}\|1$ such that for any $x \in \{0,1\}^*$, $H(x) = \mathsf{int}(H'(x))$, where $\mathsf{int}(\cdot)$ is the conventional function that maps bits to integers.

Our construction is presented in three figures: the algorithms executed by an accumulator manager are described in Fig. 2, those executed by users are described in Fig. 3, and finally, those executed by verifiers are described in Fig. 4. Each element $x \in \{0,1\}^*$ is first *hashed* into a large odd integer using the random oracle H such that the large odd integer admits a large prime factor with overwhelming probability. The presence of that large prime factor will help us ensure that membership witnesses can only be forged with negligible probability and non-membership witnesses do not exist with negligible probability.

Add an element to the accumulator. Let $\mathsf{acc} \in \mathrm{QR}_n$ be the current value of our accumulator. To add an element $x \in \{0,1\}^*$, we set $\mathsf{acc}' = \mathsf{acc}^{H(x)}$, $\mathtt{w} = \mathsf{acc}$, and let $w_x = (\mathtt{w}, \mathsf{s})$, where s is a tuple that will be used to store 2^τ-smooth factors of $H(x)$ and that is initialized to be the singleton (1). By construction, $\mathtt{w}$ is a modular $H(x)$ root of the new accumulator value acc'. Note that we can initialized s to $\perp$ or an empty tuple, but this will require checking whether s is *empty* during verification.

Delete an element from the accumulator. To delete x from the accumulator, we compute a modular $H(x)$-root acc' of acc and let acc' be the new accumulator value. However, if we have access to a valid witness $w_x = (\mathtt{w}, \mathsf{s})$ of x, we can set the new accumulator value to be $\mathtt{w}$, which is a modular r-root of acc, where r is a factor of $H(x)$ such that $P^+(r) = P^+(H(x)) > 2^\tau$. Using the witness during deletion allows us to save $O(\log^2(\mathsf{poly}(\lambda)))$ time since we do not have to compute $H(x)^{-1} \bmod \mathsf{sk}$ using the Extended Euclidean algorithm.

```
– $\mathsf{NonMemWitCreate}(\mathsf{pp}, x, \{\mathsf{upmsg}_i\}_{i=1}^m)$:
  1. Parse $\mathsf{pp}$ as $(n, u)$.
  2. Let $\mathcal{S} = \emptyset$, and $\mathbf{d} = (1)$.
  3. For each $\mathsf{upmsg} \in \{\mathsf{upmsg}_i\}_{i=1}^m$ do:
     (a) Parse $\mathsf{upmsg}$ as $(\mathsf{op}, v, \delta, \mathsf{acc}, \mathsf{acc}')$.
     (b) If $\mathsf{op} = \mathsf{add}$, set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$.
     (c) Else if $\mathsf{op} = \mathsf{del}$, set $\mathcal{S} \leftarrow (\mathcal{S} \setminus \{v\})$ and $\mathbf{d} \leftarrow \mathbf{d}||(\delta)$.
  4. Compute $\theta \leftarrow \prod_{y \in \mathcal{S}} y \cdot \prod_{i=1}^{|\mathbf{d}|} \mathbf{d}[i]$.
  5. Let $\mathtt{x} \leftarrow H(x)$ and $\mathbf{s} \leftarrow (1)$.
  6. While $\gcd(\theta, \mathtt{x}) \neq 1$, set $\mathtt{x} \leftarrow \mathtt{x} / \gcd(\theta, \mathtt{x}), \mathbf{s} \leftarrow \mathbf{s}||(\gcd(\theta, \mathtt{x}))$.
  7. Find $a, b \in \mathbb{Z}$ such that $a\theta + b\mathtt{x} = 1$.
  8. Compute $\mathtt{B} \leftarrow u^b \mod n$.
  9. Return $\bar{w}_x = (a, \mathtt{B}, \mathbf{s})$.

– $\mathsf{MemWitUp}(\mathsf{pp}, x, w_x, \mathsf{upmsg})$:
  1. Parse $\mathsf{pp}$ as $(n, u)$, $w_x$ as $(\mathtt{w}, \mathbf{s})$, and $\mathsf{upmsg}$ as $(\mathsf{op}, v, \delta, \mathsf{acc}, \mathsf{acc}')$.
  2. If $\mathsf{op} = \mathsf{add}$, compute $\mathtt{w}' \leftarrow \mathtt{w}^v \bmod n$, and let $w_x' = (\mathtt{w}', \mathbf{s})$.
  3. Else if $\mathsf{op} = \mathsf{del}$, do:
     (a) Compute $\mathtt{x} \leftarrow H(x) / \prod_{i=1}^{|\mathbf{s}|} \mathbf{s}[i]$ and $\mathtt{v} \leftarrow v / \delta$.
     (b) Compute $a, b \in \mathbb{Z}$ such that $a\mathtt{x} + b\mathtt{v} = \gcd(\mathtt{v}, \mathtt{x})$.
     (c) Compute $\mathtt{w}' \leftarrow (\mathsf{acc}')^a \mathtt{w}^b \bmod n$.
     (d) If $\gcd(\mathtt{v}, \mathtt{x}) \neq 1$, let $\mathbf{s}' \leftarrow \mathbf{s}||(\gcd(\mathtt{v}, \mathtt{x}))$. Otherwise, let $\mathbf{s}' \leftarrow \mathbf{s}$.
     (e) Let $w_x' = (\mathtt{w}', \mathbf{s}')$.
  4. Return $w_x'$.

– $\mathsf{NonMemWitUp}(\mathsf{pp}, x, \bar{w}_x, \mathsf{upmsg})$:
  1. Parse $\mathsf{pp}$ as $(n, u)$, $\bar{w}_x$ as $(\mathsf{a}, \mathtt{B}, \mathbf{s})$, and $\mathsf{upmsg}$ as $(\mathsf{op}, v, \delta, \mathsf{acc}, \mathsf{acc}')$.
  2. Compute $\mathtt{x} \leftarrow H(x) / \prod_{i=1}^{|\mathbf{s}|} \mathbf{s}[i]$, and $\mathtt{v} \leftarrow v / \delta$.
  3. If $\mathsf{op} = \mathsf{add}$, do:
     (a) Let $d \leftarrow 1$ and $\mathtt{x}' \leftarrow \mathtt{x}$.
     (b) While $\gcd(\mathtt{v}, \mathtt{x}') \neq 1$, set $\mathtt{x}' \leftarrow \mathtt{x}' / \gcd(\mathtt{v}, \mathtt{x}')$, and
         $d \leftarrow d \cdot \gcd(\mathtt{v}, \mathtt{x}')$.
     (c) Find $a, b \in \mathbb{Z}$ such that $a\mathtt{v} + b\mathtt{x}' = 1$.
     (d) Compute $\mathsf{a}' \leftarrow a\mathsf{a} \bmod \mathtt{x}'$.
     (e) Compute $z \leftarrow \lfloor a\mathsf{a}/\mathtt{x}' \rfloor \mathtt{v} + \mathsf{a}b$.
     (f) Compute $\mathtt{B}' \leftarrow \mathsf{acc}^z \mathtt{B}^d \bmod n$.
     (g) If $d \neq 1$, let $\mathbf{s}' \leftarrow \mathbf{s}||(d)$. Otherwise, $\mathbf{s}' \leftarrow \mathbf{s}$.
     (h) Let $\bar{w}_x' = (\mathsf{a}', \mathtt{B}', \mathbf{s}')$.
  4. Else if $\mathsf{op} = \mathsf{del}$, do:
     (a) Compute $\mathsf{a}' \leftarrow \mathsf{a}\mathtt{v} \bmod \mathtt{x}$.
     (b) Compute $z \leftarrow \lfloor \mathsf{a}\mathtt{v}/\mathtt{x} \rfloor$.
     (c) Compute $\mathtt{B}' \leftarrow (\mathsf{acc}')^z \mathtt{B} \bmod n$.
     (d) Let $\bar{w}_x' = (\mathsf{a}', \mathtt{B}', \mathbf{s})$.
  5. Return $\bar{w}_x'$.
```

Fig. 3: User's algorithms

Update membership witness. Let $x \in \{0,1\}^*$ be an element whose membership witness $w_x = (\mathtt{w}, \mathbf{s})$ needs to be updated and acc' be the new accumulator value. In the case acc' was generated by adding an element $y \in \{0,1\}^*$, we update w_x by computing $\mathtt{w}' = \mathtt{w}^{H(y)}$ and setting $w_x' = (\mathtt{w}', \mathbf{s})$. However, in the case acc' was generated after removing an element $y' \in \{0,1\}^*$, we need to be more ingenious. In Camenisch-Lysyanskaya [CL02], $\mathbf{s} = \perp$ and $\mathtt{w}$ is updated to $\mathtt{w}' \leftarrow (\mathsf{acc}')^a \mathtt{w}^b$, where $a, b \in \mathbb{Z}$ such that $aH(x) + bH(y') = \gcd(H(x), H(y')) = d$. For them, since H is a random-oracle that returns prime numbers, this operation is correct because $d = 1$ and $(\mathtt{w}')^{H(x)} = (\mathsf{acc}')^d = \mathsf{acc}'$. However, for us, given that H returns large odd numbers, $d > 1$ with non-negligible probability, but following Corollary 1, d is 2^τ-smooth with overwhelming probability. Therefore, to regain correctness, we add d to the tuple $\mathbf{s}$, which produces the new tuple $\mathbf{s}' = \mathbf{s}||(d)$. Now, during verification, we check that all components of s' are 2^τ-smooth and that $(\mathtt{w}')^{H(x)/\prod_{i=1}^{|\mathsf{s}'|}\mathsf{s}'[i]} = \mathsf{acc}'$. For instance, if $\mathsf{s}' = (1, d)$, we have $(\mathtt{w}')^{H(x)/\prod_{i=1}^{|\mathsf{s}'|}\mathsf{s}'[i]} = (\mathtt{w}')^{H(x)/d} = ((\mathsf{acc}')^d)^{1/d} = \mathsf{acc}'$.

We present a generalized version of this procedure in Fig. 3. Note that since all components of s are 2^τ-smooth factors of $H(x)$ whose product divide $H(x)$, we will need at most ℓ-bit to store s.

Remark 7. For any $x \in \{0,1\}^*$, a membership witness w_x carries a tuple s that stores 2^τ-smooth factors of $H(x)$, resulting from GCD computations, that are used to ensure correctness during $\mathsf{MemWitUp}$ and Delete with w_x as an argument. In addition, we require that each component of s is less than or equal to 2^τ to ensure that they are indeed 2^τ-smooth, which is acceptable since an integer greater than 2^τ will divide $H(x)$ with probability at most $2^{-\tau}$. Furthermore, as mentioned previously, we need at most ℓ-bit to store s, which is the same number of bits required to store $H(x)$. We argue that this is fine in practice since in some applications one needs to store x and $H(x)$ to ensure the integrity of x. Furthermore, for $\lambda = 128$, this amounts to storing 213 bytes (confer Section 7).

Before describing how we handle non-membership, we recall the key insights from Li, Li, and Xue's scheme [LLX07]. Li, Li, and Xue noted that if H returns prime numbers, then for an accumulated set $\mathcal{S}$ and an element $x \notin \mathcal{S}$, it follows that there exists $a, b \in \mathbb{Z}$ such that $a\prod_{x' \in \mathcal{S}} H(x') + bH(x) = 1$. This implies that $u^{a\prod_{x'\in\mathcal{S}} H(x') + bH(x)} = \mathsf{acc}^a \mathtt{B}^{H(x)} = u$, where $\mathtt{B} = u^b$. Therefore, $\bar{w}_x = (a, \mathtt{B}) \in \mathbb{Z} \times \mathbb{Z}_n^*$ is a non-membership witness of x. Li, Li, and Xue also showed how $\bar{w}_x$ can be efficiently updated. Suppose $y \in \{0,1\}^*$ was added to the accumulator, i.e., $\mathsf{acc}' = \mathsf{acc}^{H(y)}$, and let $a', b' \in \mathbb{Z}$ such that $a'H(y) + b'H(x) = 1$. By letting $\mathtt{B}' = \mathsf{acc}^{ab'}\mathtt{B}$, it follows that $\bar{w}'_x = (aa', \mathtt{B}')$ is a correct up-to-date non-membership witness since $(\mathsf{acc}')^{aa'}(\mathtt{B}')^{H(x)} = \mathsf{acc}^{a(a'H(y)+b'H(x))}\mathtt{B}^{H(x)} = u$. Now, instead of y being added, suppose $y' \in \{0,1\}^*$ was removed from the accumulator, i.e., $(\mathsf{acc}')^{H(y')} = \mathsf{acc}$. Then, it follows that $\bar{w}'_x = (aH(y'), \mathtt{B})$ is a correct up-to-date non-membership witness since $(\mathsf{acc}')^{aH(y')}\mathtt{B}^{H(x)} = \mathsf{acc}^a\mathtt{B}^{H(x)} = u$. It is possible to keep the first component of $\bar{w}_x$ constant-size by representing it as an element of $\mathbb{Z}_{H(x)}$.

Create non-membership witness. In our case, since H does not *hash* to primes, we cannot directly apply the techniques developed by Li, Li, and Xue. However, from Corollary 1, it follows that for a set $\mathcal{S}$ and an element $x \notin \mathcal{S}$, $H(x) \nmid \prod_{x' \in \mathcal{S}} H(x')$. Furthermore, given that $P^+(H(x)) > 2^\tau$ with overwhelming probability, we can *factor* $H(x) = \mathtt{x} \cdot k$ such that $P^+(H(x)) = P^+(\mathtt{x})$, k is 2^τ-smooth and $\gcd(\mathtt{x}, \prod_{x' \in \mathcal{S}} H(x')) = 1$. Now, we can apply Li, Li, and Xue technique with respect to $\mathtt{x}$ and $\prod_{x' \in \mathcal{S}} H(x')$ to find $(a, \mathtt{B}) \in \mathbb{Z} \times \mathbb{Z}_n^*$. By letting $\bar{w}_x = (a, \mathtt{B}, \mathsf{s})$, where s is a tuple that stores factors of k less than or equal to 2^τ, i.e., $\prod_{i=1}^{|\mathsf{s}|}\mathsf{s}[i] = k$, it follows that $\bar{w}_x$ is a correct non-membership witness for x since $\mathsf{acc}^a\mathtt{B}^{H(x)/\prod_{i=1}^{|\mathsf{s}|}\mathsf{s}[i]} = u$. We provide a full description of this procedure in Fig. 3. Note that the total number of bits required to store s is upper bounded by ℓ, the output length of H, like in the case of membership witnesses.

Update non-membership witness. By carefully adapting Li, Li, and Xue techniques, we can efficiently update a non-membership witness $\bar{w}_x = (\mathsf{a}, \mathtt{B}, \mathsf{s})$ of an element x.

Let $\mathtt{x} = H(x)/\prod_{i=1}^{|\mathsf{s}|}\mathsf{s}[i]$ and suppose $y \in \{0,1\}^*$ was added to the accumulator. Given that $P^+(\mathtt{x}) > 2^\tau$ with overwhelming probability, it follows that $\mathtt{x} \nmid H(y)$. Therefore, we can *factor* $\mathtt{x} = \mathtt{x}' \cdot k'$ such that $P^+(\mathtt{x}) = P^+(\mathtt{x}')$, k' is 2^τ-smooth, and there exitst $a, b \in \mathbb{Z}$ such that $aH(y) + b\mathtt{x}' = 1$. By setting $\bar{w}'_x = (\mathsf{a}', \mathtt{B}', \mathsf{s}')$, where $\mathsf{a}' = a\mathsf{a} \bmod \mathtt{x}'$, $\mathtt{B}' = \mathsf{acc}^{\lfloor \mathsf{a}a/\mathtt{x}' \rfloor \mathsf{a}b}\mathtt{B}^{k'}$ and s' is a tuple obtained by concatenating factors of k' less than 2^τ to s, i.e., $\prod_{i=1}^{|\mathsf{s}'|}\mathsf{s}'[i] = kk'$, it follows that $(\mathsf{acc}')^{\mathsf{a}'}(\mathtt{B}')^{H(x)/\prod_{i=1}^{|\mathsf{s}'|}\mathsf{s}'[i]} = \mathsf{acc}^{\mathsf{a}(a'H(y)+b'\mathtt{x}')}\mathtt{B}^{k'\mathtt{x}'} = u$. Hence, $\bar{w}_x$ is a correct up-to-date non-membership witness.

Now, suppose that instead of adding y, we removed $y' \in \{0,1\}^*$ from the accumulator. From Section 5, after deletion of y', acc', the new accumulator value, is a modular r-root of acc, the old accumulator value, where $r \mid H(y')$, $P^+(r) = P^+(H(y'))$, and it can be recovered from upmsg. By letting $\bar{w}'_x = (\mathsf{a}', \mathtt{B}', \mathsf{s})$, where $\mathsf{a}' = \mathsf{a}r \bmod \mathtt{x}$ and $\mathtt{B}' = (\mathsf{acc}')^{\lfloor \mathsf{a}r/\mathtt{x} \rfloor}\mathtt{B}$, it follows that $(\mathsf{acc}')^{\mathsf{a}'}(\mathtt{B}')^{H(x)/\prod_{i=1}^{|\mathsf{s}|}\mathsf{s}[i]} = \mathsf{acc}^{\mathsf{a}}\mathtt{B}^{\mathtt{x}} = u$. We present a full description of this procedure in Fig. 3.

```
– MemVer(pp, acc, $x$, $w_x$):
    1. Parse pp as $(n, u)$, and $w_x$ as $(\mathtt{w}, \mathtt{s})$.
    2. For $i \in [|\mathtt{s}|]$, if $\mathtt{s}[i] > 2^\tau$, return 0.
    3. Compute $\mathtt{x} \leftarrow H(x)/\prod_{i=1}^{|\mathtt{s}|} \mathtt{s}[i]$.
    4. If $\mathtt{w}^{\mathtt{x}} \equiv \mathsf{acc} \bmod n$ return 1. Otherwise, return 0.
– NonMemVer(pp, acc, $x$, $\bar{w}_x$):
    1. Parse pp as $(n, u)$, and $\bar{w}_x$ as $(\mathtt{a}, \mathtt{B}, \mathtt{s})$.
    2. For $i \in [|\mathtt{s}|]$, if $\mathtt{s}[i] > 2^\tau$, return 0.
    3. Compute $\mathtt{x} \leftarrow H(x)/\prod_{i=1}^{|\mathtt{s}|} \mathtt{s}[i]$.
    4. If $\mathsf{acc}^{\mathtt{a}}\mathtt{B}^{\mathtt{x}} \equiv u \bmod n$ return 1. Otherwise, return 0.
```

Fig. 4: Verifier's algorithms

Remark 8. NonMemWitCreate has time complexity $\mathsf{poly}(m)$, where m is the size of the accumlated set, and that is inefficient in comparison to NonMemWitUp. Li, Li, and Xue [LLX07] proposed a method to reduce NonMemWitCreate's time complexity by computing $\mu = \prod_{x \in \mathcal{S}} H(x) \bmod \phi(n)$, where $\mathcal{S}$ is the accumlated set. Although this provide some savings during the execution of the Extended Euclidean algorithm, it still requires to go through all elements in $\mathcal{S}$. Furthermore, with that modification, only the accumulator manager is able to execute NonMemWitCreate since it requires the use of the secret key sk. We emphasize that μ is never handed to users. Mashatan and Vaudenay [MV13] proposed a technique that allows to create non-membership witnesses in Li-Li-Xue construction in a way that is independent of the size of $\mathcal{S}$, but it requires the use of an additive accumulator in which elements can only be added by the accumulator manager. In brief, suppose we have access to an additive accumulator Γ. To generate a non-membership witness for a non-accumlated element x, we sample $\mathsf{a} \in \mathbb{Z}_{H(x)}$, $\mathtt{B} \in \mathbb{Z}_n^*$, and then, compute $h = \mathsf{acc}^a \mathtt{B}^{H(x)}$ and add h to Γ, which generate a membership witness $w_{h,\Gamma}$. The non-membership witness of x is $\bar{w}_x = (\mathsf{a}, \mathtt{B}, h, w_{h,\Gamma})$. Clearly, both approaches can be applied to our construction to reduce the time complexity of NonMemWitCreate. However, as we previously mentioned, they both require a trusted accumulator manager to generate non-membership witnesses.

Batching addition. As in [CL02] and [LLX07], we note that the addition of multiple elements can be batched by adding the product of their H evaluations to the accumulator. In addition, after a batch addition, the (non-)membership witness for an element $x \in \{0,1\}^*$ can be updated by using the update information $\mathsf{upmsg}' = (\mathsf{add}, v', \delta', \mathsf{acc}, \mathsf{acc}')$ in conjunction with MemWitUp or NonMemWitUp, where v' represents the product of H evaluations of added or deleted elements, $\delta' = 1$, acc represents the last accumulator value for which the witness to be updated is valid and acc' represents the new accumulator's value.

4.1 Correctness and Compactness

In this section, we analyze the correctness and compactness of our universal dynamic accumulator construction.

Lemma 3. *Let n be the RSA modulus produced by $\mathsf{Gen}(1^\lambda)$, and suppose $x \in \{0,1\}^*$. Then, $H(x)^{-1} \bmod \phi(n)$ does not exist with probability at most $1/2^{\lambda-2}$.*

Proof. Given that n is an output of $\mathsf{Gen}(1^\lambda)$, it follows that $\phi(n) = 4p'q'$, where $\log_2 p' \approx \log_2 q' \gg \lambda$. For $x \in \{0,1\}^*$, we have $H(x) \sim U(\mathsf{Odds}(2^{\ell-1}, 2^\ell - 1))$. $H(x)^{-1} \bmod \phi(n)$ does not exist when $\gcd(H(x), \phi(n)) \neq 1$, which happens when p' or q' divides $H(x)$. Since $p' > 2^{\lambda-1}$ and $q' > 2^{\lambda-1}$, using the union bound we have $\Pr[(p' \mid H(x)) \vee (q' \mid H(x))] \leq \frac{2}{2^{\lambda-1}} = \frac{1}{2^{\lambda-2}}$ ☐

Corollary 2. Delete *fails with probability at most* $1/2^{\lambda-2}$.

Proof. This follows from Lemma 3. ☐

Lemma 4. $\mathsf{NonMemWitCreate}$ *fails to output correct a non-membership witness for an element that has not been accumulated with probability at most*

$$(q+1)^2\left(\frac{1}{2^{\ell^{3/4}}}+\left(\frac{\sqrt[4]{\ell}}{4}\log\ell\right)^{-\sqrt[4]{\ell}}\right)+\frac{1}{2^{\sqrt{\ell}}}$$

where $q \in \mathbb{N}$ represents the number of elements present in the accumulator.

Proof. Let acc be an accumulator produced by our construction and $\{\mathsf{upmsg}\}_{i=1}^{n}$ the set of update information produced after a series of Add and Delete operations that generated acc. Suppose $x \in \{0,1\}^*$ was not added to acc. From $\{\mathsf{upmsg}\}_{i=1}^{n}$, we can recover a set $\mathcal{S}$ that contains H evaluation of elements that are present in acc and a tuple $\mathbf{d}$ that contains products of 2^τ-smooth integers that divide previously deleted elements (in the description of $\mathsf{NonMemWitCreate}$, confer Fig. 3, those products are denoted by δ). Let $\theta = \prod_{y\in\mathcal{S}} y \prod_{i=1}^{|\mathbf{d}|}\mathbf{d}[i]$. Given that $H(x) \notin \mathcal{S}$ with overwhelming probability and from Corollary 1, $H(x) \nmid \theta$ with overwhelming probability, it follows that there exists $\mathtt{x}, k \in \mathbb{Z}$ such that $H(x) = k\mathtt{x}$, $k \mid \theta$, $P^+(\mathtt{x}) = P^+(H(x))$, and $\gcd(\theta, \mathtt{x}) = 1$. Let $a, b \in \mathbb{Z}$ such that $a\theta + b\mathtt{x} = 1$, and $\mathtt{B} = u^b \bmod n$. Since $u^\theta \equiv \mathsf{acc} \bmod n$, we have $\mathsf{acc}^a(\mathtt{B})^{\mathtt{x}} \equiv u \bmod n$.

Let $\mathbf{s}$ be a tuple that represents a factorization of k. Since $\mathtt{x}$ is computed in such a way that $k \mid \theta$ and the probability that an integer $c > 2^\tau$, with $\sqrt{\ell} \le \tau \le \ell^{3/4}$, divides both $H(x)$ and θ is less than or equal to $2^{-\sqrt{\ell}}$, it follows that each component of $\mathbf{s}$ is less than or equal to 2^τ with overwhelming probability.

Let Bad be the event that $\mathsf{NonMemWitCreate}$ fails, D the event that $H(x) \mid \theta$, E the event that there exists $j \in [|\mathbf{s}|]$ such that $\mathbf{s}[j] > 2^\tau$, and $q = \#\mathcal{S}$. We have

$$\begin{aligned}
\Pr[\mathsf{Bad}] &= \Pr[\mathsf{Bad}|\mathsf{D}]\Pr[\mathsf{D}] + \Pr[\mathsf{Bad}|\bar{\mathsf{D}}]\Pr[\bar{\mathsf{D}}]\\
&\le \Pr[\mathsf{D}] + \Pr[\mathsf{Bad}|\bar{\mathsf{D}},\mathsf{E}]\Pr[\mathsf{E}|\bar{\mathsf{D}}] + \Pr[\mathsf{Bad}|\bar{\mathsf{D}},\bar{\mathsf{E}}]\Pr[\bar{\mathsf{E}}|\bar{\mathsf{D}}]\\
&\overset{(1)}{\le} \Pr[\mathsf{D}] + \Pr[\mathsf{E}]\\
&\le (q+1)^2\left(\frac{1}{2^{\ell^{3/4}}}+\left(\frac{\sqrt[4]{\ell}}{4}\log\ell\right)^{-\sqrt[4]{\ell}}\right)+\frac{1}{2^{\sqrt{\ell}}}
\end{aligned}$$

Inequality (1) follows from the fact that E and D are independent. Therefore, unless Bad happens, $\bar{w}_x = (a, \mathtt{B}, \mathbf{s})$ is a valid non-membership witness for x. □

Lemma 5. $\mathsf{MemWitUp}$ *fails to output a correct updated membership witness with probability at most* $2^{-\sqrt{\ell}}$.

Proof. Let acc be an accumulator produced by our construction and acc' its update. Let $x \in \{0,1\}^*$ be an accumulated element, $w_x = (\mathtt{w}, \mathbf{s})$ its valid membership witness with respect to acc, and $w'_x = (\mathtt{w}', \mathbf{s}')$ its membership witness with respect to acc' generated by $\mathsf{MemWitUp}$. We show that w'_x is valid with probability at least $1 - 2^{-\sqrt{\ell}}$. Let $y \in \{0,1\}^*$, $\mathtt{x} = H(x)/\prod_{i=1}^{|\mathbf{s}|}\mathbf{s}[i]$, and $\mathtt{x}' = H(x)/\prod_{i=1}^{|\mathbf{s}'|}\mathbf{s}'[i]$. Without lost of generality, let us consider the following cases:

– Case 1: acc' was produced by adding y to acc, i.e., $\mathsf{acc}' = \mathsf{acc}^{H(y)}$. After executing $\mathsf{MemWitUp}$, we have $\mathtt{w}' = \mathtt{w}^{H(y)}$ and $\mathbf{s}' = \mathbf{s}$, so $\mathtt{x} = \mathtt{x}'$. Hence, $(\mathtt{w}')^{\mathtt{x}'} = (\mathtt{w}^{\mathtt{x}})^{H(y)} = \mathsf{acc}^{H(y)} = \mathsf{acc}'$. In addition, since w_x is valid, it follows that all components of $\mathbf{s}$ are less than 2^τ, and this is also the case for $\mathbf{s}'$.

– Case 2: acc' was produced by deleting y from acc. Let upmsg be the update message that was generated after deleting y from acc. Then, $\mathsf{upmsg} = (\mathsf{del}, H(y), \delta, \mathsf{acc}, \mathsf{acc}')$, where $\delta \ge 1$ is a 2^τ-smooth integer that divides $H(y)$. By setting $\mathtt{v} = H(y)/\delta$, it follows that $\mathsf{acc} = (\mathsf{acc}')^{\mathtt{v}}$. Let $d = \gcd(\mathtt{v}, \mathtt{x})$. After executing $\mathsf{MemWitUp}$, we have $\mathtt{w}' = (\mathsf{acc}')^a\mathtt{w}^b$, where $a, b \in \mathbb{Z}$ such that $a\mathtt{x} + b\mathtt{v} = d$, and $\mathsf{s}' = \mathsf{s}\|(d)$ if $d > 1$, else $\mathsf{s}' = \mathsf{s}$. Therefore, $\mathtt{x}' = \mathtt{x}/d$, and

$$\begin{aligned}
(\mathtt{w}')^{\mathtt{x}'} &= ((\mathsf{acc}')^a \mathtt{w}^b)^{\mathtt{x}/d} \\
&= ((\mathsf{acc}')^a \mathtt{w}^b)^{\mathtt{x}\mathtt{v}(1/\mathtt{v})(1/d)} \quad \text{(this follows from Lemma 3)} \\
&= \left(((\mathsf{acc}')^{\mathtt{v}})^{a\mathtt{x}} (\mathtt{w}^{\mathtt{x}})^{b\mathtt{v}}\right)^{(1/\mathtt{v})(1/d)} \\
&= (\mathsf{acc}^{a\mathtt{x}} \mathsf{acc}^{b\mathtt{v}})^{(1/\mathtt{v})(1/d)} \\
&= \mathsf{acc}^{1/\mathtt{v}} = \mathsf{acc}'
\end{aligned}$$

Each component of s is less than or equal to 2^τ since w_x is valid, and if $d = 1$, it follows that components of s' are also less than or equal to 2^τ. Otherwise, $\mathsf{s}' = \mathsf{s}\|(d)$, and given that d divides both $H(x)$ and $H(y)$ and $\sqrt{\ell} \le \tau \le \ell^{3/4}$, it follows that $d < 2^\tau$ with probability at least $1 - 2^{-\sqrt{\ell}}$. Therefore, each component of s' is less than or equal to 2^τ with overwhelming probability.

Notice that w'_x will be incorrect if there exists an index $j \in [|\mathsf{s}'|]$ such that $\mathsf{s}'[j] > 2^\tau$, and this happens with probability at most $2^{-\sqrt{\ell}}$. □

Lemma 6. $\mathsf{NonMemWitUp}$ *fails to output a correct updated non-membership witness with probability at most* $\left(\frac{\sqrt[4]{\ell}}{4}\log\ell\right)^{-\sqrt[4]{\ell}} + 2^{-\sqrt{\ell}}$.

Proof. Let acc be an accumulator produced by our construction and acc' its update. Let $x \in \{0,1\}^*$ be an element that was not added to the accumulator, $\bar{w}_x = (\mathsf{a}, \mathtt{B}, \mathsf{s})$ its valid non-membership witness with respect to acc, and $\bar{w}'_x = (\mathsf{a}', \mathtt{B}', \mathsf{s}')$ its non-membership witness with respect to acc' generated by $\mathsf{NonMemWitUp}$. We show that $\bar{w}'_x$ is incorrect with probability at most $\left(\frac{\sqrt[4]{\ell}}{4}\log\ell\right)^{-\sqrt[4]{\ell}} + 2^{-\sqrt{\ell}}$. Let $y \in \{0,1\}^*$, $\mathtt{x} = H(x)/\prod_{i=1}^{|\mathsf{s}|}\mathsf{s}[i]$, and $\mathtt{x}' = H(x)/\prod_{i=1}^{|\mathsf{s}'|}\mathsf{s}'[i]$. Without lost of generality, let us consider the following cases:

- Case 1: acc' resulted from adding y to acc, i.e., $\mathsf{acc}' = \mathsf{acc}^{H(y)}$. From Lemma 2, $P^+(H(x)) > 2^\tau$ with overwhelming probability. Since $\bar{w}_x$ is valid, it follows that $P^+(\mathtt{x}) = P^+(H(x))$, so $\mathtt{x} \nmid H(y)$ with overwhelming probability. As a result, there exists $r, d \in \mathbb{Z}$ such that $\mathtt{x} = dr$, $d \mid H(y)$, $P^+(r) = P^+(\mathtt{x})$ and $\gcd(H(y), r) = 1$. After executing $\mathsf{NonMemWitUp}$, $\mathtt{x}' = r$, and $\mathsf{s}' = \mathsf{s}$ if $d = 1$, else $\mathsf{s}' = \mathsf{s}\|(d)$. In addition, we have $\mathsf{a}' = a\mathsf{a} \bmod \mathtt{x}' = a\mathsf{a} - \lfloor a\mathsf{a}/\mathtt{x}'\rfloor \mathtt{x}'$, and $\mathtt{B}' = \mathsf{acc}^{z_1}\mathtt{B}^d$, where $z_1 = \lfloor a\mathsf{a}/\mathtt{x}'\rfloor H(y) + \mathsf{a}b$, and $a, b \in \mathbb{Z}$ such that $aH(y) + b\mathtt{x}' = 1$. Hence,
$$\begin{aligned}
(\mathsf{acc}')^{\mathsf{a}'}(\mathtt{B}')^{\mathtt{x}'} &= (\mathsf{acc}^{H(y)})^{a\mathsf{a} - \lfloor a\mathsf{a}/\mathtt{x}'\rfloor \mathtt{x}'} (\mathsf{acc}^{\lfloor a\mathsf{a}/\mathtt{x}'\rfloor H(y) + \mathsf{a}b}\mathtt{B}^d)^{\mathtt{x}'} \\
&= \mathsf{acc}^{\mathsf{a}(aH(y) + b\mathtt{x}')}\mathtt{B}^{d\mathtt{x}'} \\
&= \mathsf{acc}^{\mathsf{a}}\mathtt{B}^{\mathtt{x}} = u
\end{aligned}$$
In case $d = 1$, all components of s' are less than or equal to 2^τ since $\bar{w}_x$ is valid. Otherwise, $\mathsf{s}' = \mathsf{s}\|(d)$, and given that d divides both $H(x)$ and $H(y)$ and $\sqrt{\ell} \le \tau \le \ell^{3/4}$, it follows that $d < 2^\tau$ with overwhelming probability. Therefore, each component of s' is less than or equal to 2^τ with overwhelming probability.
Since in this case $\bar{w}'_x$ is correct if $P^+(H(x)) > 2^\tau$ and there does not exist $j \in [|\mathsf{s}'|]$ such that $\mathsf{s}'[j] > 2^\tau$, it follows that failure probability of $\mathsf{NonMemWitUp}$ is upper bounded by $\left(\frac{\sqrt[4]{\ell}}{4}\log\ell\right)^{-\sqrt[4]{\ell}} + 2^{-\sqrt{\ell}}$.
- Case 2: acc' was produced by deleting y from acc. Let upmsg be the update message that was generated after deleting y from acc. Then, $\mathsf{upmsg} = (\mathsf{del}, H(y), \delta, \mathsf{acc}, \mathsf{acc}')$, where $\delta \ge 1$ is a 2^τ-smooth integer that divides $H(y)$, and $\mathsf{acc} = (\mathsf{acc}')^{\mathtt{v}}$, with $\mathtt{v} = H(y)/\delta$. After executing $\mathsf{NonMemWitUp}$, we have $\mathsf{s}' = \mathsf{s}$, so $\mathtt{x}' = \mathtt{x}$. Also, $\mathsf{a}' = \mathsf{a}\mathtt{v} \bmod \mathtt{x} = \mathsf{a}\mathtt{v} - \lfloor \mathsf{a}\mathtt{v}/\mathtt{x}\rfloor \mathtt{x}$, and $\mathtt{B}' = (\mathsf{acc}')^{z_2}\mathtt{B}$, where $z_2 = \lfloor \mathsf{a}\mathtt{v}/\mathtt{x}\rfloor$. Hence,
$$\begin{aligned}
(\mathsf{acc}')^{\mathsf{a}'}(\mathtt{B}')^{\mathtt{x}'} &= (\mathsf{acc}')^{\mathsf{a}\mathtt{v} - \lfloor \mathsf{a}\mathtt{v}/\mathtt{x}\rfloor \mathtt{x}}((\mathsf{acc}')^{\lfloor \mathsf{a}\mathtt{v}/\mathtt{x}\rfloor}\mathtt{B})^{\mathtt{x}} \\
&= (\mathsf{acc}')^{\mathsf{a}\mathtt{v}}\mathtt{B}^{\mathtt{x}} \\
&= \mathsf{acc}^{\mathsf{a}}\mathtt{B}^{\mathtt{x}} = u
\end{aligned}$$

Since $\mathsf{s}' = \mathsf{s}$ and $\bar{w}_x$ is valid, it follows that all components of s' are less than or equal to 2^τ. Hence, in this case, $\bar{w}'_x$ is always correct.

Therefore, $\mathsf{NonMemWitUp}$ fails with probability at most $\left(\frac{\sqrt[4]{\ell}}{4}\log\ell\right)^{-\sqrt[4]{\ell}} + 2^{-\sqrt{\ell}}$. □

Theorem 1. *Our construction is correct with a probability of at least* $1 - \mathsf{negl}(\lambda)$.

Proof. This follows from Corollary 2, and lemmas 4, 5 and 6.

Theorem 2. *Our construction is compact.*

Proof. Let $\lambda \in \mathbb{N}$ be a security parameter used as input to Gen, $\ell = \mathsf{poly}(\lambda)$ be the chosen bit-length of H's outputs, and $\sqrt{\ell} \leq \tau \leq \ell^{3/4}$ be a value chosen such that for any $x \in \{0,1\}^*$, $P^+(H(x)) > 2^\tau$ with overwhelming probability. From the description of our construction (confer Figs. 2, 3, and 4), $\mathsf{acc} \in \mathbb{Z}_n^*$, where $\log_2 n \approx 2(\lambda+1)$, so $|\mathsf{acc}| \leq 2(\lambda+2)$. For any $x \in \{0,1\}^*$ with membership witness $w_x = (\mathtt{w}, \mathsf{s})$, we have $\mathtt{w} \in \mathbb{Z}_n^*$ and s, which is a tuple whose components are 2^τ-smooth integers that divide $H(x)$ and their products also divide $H(x)$. Since we need less than ℓ bits to represent all components of s, we can conclude that $|w_x| < 2(\lambda+2) + \ell$. Finally, for any $y \in \{0,1\}^*$ with non-membership witness $\bar{w}_y = (\mathtt{a}, \mathtt{B}, \mathsf{s}')$, we have $\mathtt{a} \in \mathbb{Z}_{H(y)}$, $\mathtt{B} \in \mathbb{Z}_n^*$, and s' that is defined as s. Hence, $|\bar{w}_x| < 2(\lambda + 2 + \ell)$.

□

4.2 Security

Theorem 3. *Assume H is a random oracle. Under the strong RSA assumption, our construction is a secure universal dynamic accumulator.*

Proof. We proceed by contraposition. Let $\mathcal{A}$ be a ppt adversary that, given $(1^\lambda, \perp, \mathsf{pp}, \mathsf{acc}_0)$ as input, outputs $(x^*, w_{x^*}, \bar{w}_{x^*})$ with non-negligible probability $\varepsilon(\lambda)$ such that (x^*, w_{x^*}) and $(x^*, \bar{w}_{x^*})$ are both valid with respect to acc, where acc is the accumulator value generated after $\mathcal{A}$'s queries to $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$. Using $\mathcal{A}$, we construct a ppt adversary $\mathcal{B}$ that breaks the strong RSA assumption as follows:

1. $\mathcal{B}$ receives $(1^\lambda, v, n)$ as input from the Strong RSA challenger. Then, it computes $u = v^2 \bmod n$ and initialises an empty map $\mathsf{T} : \{0,1\}^* \to \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$, a set $\mathcal{S} = \emptyset$, and an integer $d = 1$.
2. $\mathcal{B}$ sets $\mathsf{pp} = (n, u)$, $\mathsf{acc} = \mathsf{acc}_0 = u$ and sends $(1^\lambda, \mathsf{pp}, \mathsf{acc}_0)$ to $\mathcal{A}$.
3. $\mathcal{B}$ simulates answers to $\mathcal{A}$'s oracle queries as follows:
   – For H queries, when $\mathcal{A}$ sends $x \in \{0,1\}^*$, $\mathcal{B}$ returns $\mathsf{T}[x]$ if $\mathsf{T}[x] \neq \perp$. Else, $\mathcal{B}$ samples $r \leftarrow\!\$\ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$, sets $\mathsf{T}[x] \leftarrow r$ and returns r.
   – For add queries, when $\mathcal{A}$ sends (add, x), $\mathcal{B}$ simulates H with x as input, but it does not return the output to $\mathcal{A}$. If $\mathsf{T}[x] \in \mathcal{S}$, $\mathcal{B}$ aborts. Otherwise, $\mathcal{B}$ updates $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathsf{T}[x]\}$, defines $\mathsf{s} \leftarrow (1)$ and $w_x \leftarrow (\mathsf{acc}, \mathsf{s})$ and computes $\mathsf{acc}' \leftarrow \mathsf{acc}^{\mathsf{T}[x]} \bmod n$. Next, $\mathcal{B}$ returns acc', w_x and $\mathsf{upmsg} = (\mathsf{add}, \mathsf{T}[x], 1, \mathsf{acc}, \mathsf{acc}')$. Finally, $\mathcal{B}$ sets $\mathsf{acc} \leftarrow \mathsf{acc}'$.
   – For del queries, when $\mathcal{A}$ sends (del, x, w_x), $\mathcal{B}$ simulates H with x as input, but it does not return the output to $\mathcal{A}$. If $\mathsf{T}[x] \notin \mathcal{S}$, $\mathcal{B}$ aborts. Next, $\mathcal{B}$ sets $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\mathsf{T}[x]\}$, and if $w_x \neq \perp$ and $\mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}, x, w_x) = 1$, $\mathcal{B}$ parses w_x as $(\mathtt{w}, \mathsf{s})$, computes $\delta \leftarrow \prod_{i=1}^{|\mathsf{s}|} \mathsf{s}[i]$, $d \leftarrow d \cdot \delta$, and $\mathsf{acc}' \leftarrow \mathtt{w}$. Otherwise, $\mathcal{B}$ sets $\delta \leftarrow 1$ and computes $\mathsf{acc}' \leftarrow u^{d \prod_{y \in \mathcal{S}} y} \bmod n$. After, $\mathcal{B}$ returns acc' and $\mathsf{upmsg} = (\mathsf{del}, \mathsf{T}[x], \delta, \mathsf{acc}, \mathsf{acc}')$. Finally, $\mathcal{B}$ sets $\mathsf{acc} \leftarrow \mathsf{acc}'$.
4. Once $\mathcal{A}$ outputs $(x^*, w_{x^*}, \bar{w}_{x^*})$, $\mathcal{B}$ does the following:
   (a) If $\mathsf{MemVer}(\mathsf{pp}, x^*, w_{x^*}, \mathsf{acc}) \neq 1$ or $\mathsf{NonMemVer}(\mathsf{pp}, x^*, \bar{w}_{x^*}, \mathsf{acc}) \neq 1$, abort.
   (b) Parse w_{x^*} as $(\mathtt{w}, \mathsf{s})$, $\bar{w}_{x^*}$ as $(\mathtt{a}, \mathtt{B}, \bar{\mathsf{s}})$.
   (c) Compute $\theta \leftarrow d \prod_{y \in \mathcal{S}} y$, $\mathtt{x} = \mathsf{T}[x^*] / \prod_{i=1}^{|\mathsf{s}|} \mathsf{s}[i]$ and $\bar{\mathtt{x}} = \mathsf{T}[x^*] / \prod_{i=1}^{|\bar{\mathsf{s}}|} \bar{\mathsf{s}}[i]$.
   (d) If $\mathsf{T}[x^*] \in \mathcal{S}$, it follows that $\bar{\mathtt{x}} \mid \theta$, so $\gcd(1 - \mathtt{a}\theta, \bar{\mathtt{x}}) = 1$. Furthermore, given that $\bar{\mathtt{x}}$ is odd, $\gcd(2(1 - \mathtt{a}\theta), \bar{\mathtt{x}}) = 1$. Since $(\mathsf{acc})^{\mathtt{a}} \mathtt{B}^{\bar{\mathtt{x}}} \equiv u^{\mathtt{a}\theta} \mathtt{B}^{\bar{\mathtt{x}}} \equiv u \equiv v^2 \bmod n$, we have $\mathtt{B}^{\bar{\mathtt{x}}} \equiv v^{2(1-\mathtt{a}\theta)} \bmod n$. By applying Lemma 1 with respect to $(\mathtt{B}, \bar{\mathtt{x}}, v, 2(1 - \mathtt{a}\theta))$, compute and output the $\bar{\mathtt{x}}$-root of v.

(e) Otherwise, If $\mathsf{T}[x^*] \notin \mathcal{S}$, from Corollary 1, it follows that $\mathsf{T}[x^*] \nmid \theta$ with overwhelming probability, and since w_{x^*} is valid, we have $P^+(\mathsf{T}[x]) = P^+(\mathtt{x})$. Hence, $\mathtt{x} \nmid \theta$. Let $\tilde{\mathtt{x}} = \mathtt{x}/\gcd(2\theta, \mathtt{x})$ and $\tilde{\theta} = 2\theta/\gcd(2\theta, \mathtt{x})$. Since $\mathtt{w}^{\mathtt{x}} \equiv \mathsf{acc} \equiv u^{\theta} \equiv v^{2\theta} \bmod n$ and from Lemma 3, $\gcd(2\theta, \mathtt{x})^{-1} \bmod \phi(n)$ exists with overwhelming probability, we have $\mathtt{w}^{\tilde{\mathtt{x}}} \equiv v^{\tilde{\theta}} \bmod n$ and $\gcd(\tilde{\theta}, \tilde{\mathtt{x}}) = 1$. By applying Lemma 1 with respect to $(\mathtt{w}, \tilde{\mathtt{x}}, v, \tilde{\theta})$, compute and output the $\tilde{\mathtt{x}}$-root of v. Note that if $\gcd(2\theta, \mathtt{x})^{-1} \bmod \phi(n)$ does not exist, then $\gcd(2\theta, \mathtt{x})$ admits a prime p' as a divisor such that $2p' + 1$ divides n, so $\mathcal{B}$ can factor n and easily solve the strong RSA challenge.

Note that during the execution of del queries, if $\mathcal{A}$ sends a tuple (del, y, w_y) such that $\mathsf{T}[y] \in \mathcal{S}$ and $w_y = (\mathtt{w}, \mathtt{s})$ is valid, it follows that $\mathtt{w} \equiv \mathsf{acc}^{1/\mathtt{y}} \equiv u^{(\mathsf{T}[y]/\mathtt{y}) \prod_{z \in \mathcal{S}\setminus\{\mathsf{T}[y]\}} z} \equiv u^{\delta \prod_{z \in \mathcal{S}\setminus\{\mathsf{T}[y]\}} z} \bmod n$, with $\delta \leftarrow \prod_{i=1}^{|\mathtt{s}|} \mathtt{s}[i]$ and $\mathtt{y} \leftarrow \mathsf{T}[y]/\delta$.

In addition, as long as $\mathcal{B}$ properly simulates $\mathcal{O}_{\mathsf{Add,Delete}}$, if $\mathcal{A}$ issues a forgery $(x^*, w_{x^*}, \bar{w}_{x^*})$ such that $\mathsf{T}[x^*] \in \mathcal{S}$, then $\mathcal{B}$ will solve the strong RSA challenge. Otherwise, as long $\mathsf{T}[x^*] \nmid \theta$ or $\gcd(2\theta, \mathtt{x})$ does not exist, $\mathcal{B}$ will still be able to solve the strong RSA challenge. $\mathcal{B}$ will fail to properly simulate $\mathcal{O}_{\mathsf{Add,Delete}}$ if it aborts during the execution of an add query for an element that was not accumulated or if it does not abort during the execution of a del query for an element that was not accumulated, and those events will happen only if there is a collision among $\mathcal{A}$'s queries to the random oracle H. Therefore, $\mathcal{B}$ succeeds with probability

$$\Pr[\mathcal{B} \text{ wins}] \geq \varepsilon(\lambda) \left(1 - \frac{\mathsf{q}_H^2}{2^{\ell-1}}\right)(1 - \nu)$$

where q_H represents the number of unique H's queries performed by $\mathcal{A}$ and
$\nu = q_H^2 \left(\frac{1}{2^{\ell^{3/4}}} + \left(\frac{\sqrt[4]{\ell}}{4} \log \ell\right)^{-\sqrt[4]{\ell}} \right) (1 - \frac{1}{2^{\lambda-2}})$.

□

Division Intractable Hash (DIH) Function. A hash family $\mathcal{H}$ is said to be division intractable if for any $h \in \mathcal{H}$ with input domain $\mathcal{D}$ and output domain $\mathcal{R}$, where $\mathcal{R}$ is an Euclidean domain, the probability that an adversary $\mathcal{A}$ outputs $x_1, \ldots, x_m, y \in \mathcal{D}$ such that $h(y) \mid \prod_{i=1}^{m} h(x_i)$ is negligible [GHR99]. Clearly, our random-oracle-based hash function H is an instance of a DIH. Although Lipmaa [Lip12] showed how to construct a static accumulator using a DIH, just assuming division intractability is not sufficient to obtain a dynamic accumulator. We need a DIH whose outputs contain a large prime factor with overwhelming probability. For instance, let H' be a DIH and suppose we can find a sequence $x_1, \ldots, x_m, y$ such that $H'(y) = 3^{k_1}$ and $\prod_{i=1}^{m} H'(x_i) = 3^{k_2} r$, where $k_1 > k_2$ and $3 \nmid r$, with non-negligible probability. Furthermore, suppose we use our positive accumulator construction to accumulate $x_1, \ldots, x_m$. Then, it will be possible to forge a valid membership witness $w_y = (\mathtt{w}_y, \mathtt{s}_y)$ for y using membership witnesses of $x_1, \ldots, x_m$ such that $H'(y)/\prod_{i=1}^{|\mathtt{s}_y|} \mathtt{s}[i] = 3^{k_2}$ and all components of $\mathtt{s}_y$ are smooth.

5 Positive Dynamic Accumulator Construction

In this section, we present a positive dynamic accumulator that is based on the CL-RSA-B construction of Baldimtsi et al. [BCD+17] and our universal accumulator presented in Section 4.

For this construction, we only present the algorithm Add in Fig. 5 because the algorithms Gen, Delete, MemVer are exactly the same algorithms presented in Section 4, and for $\mathsf{MemWitUp}$, only step 2, regarding the update of membership witnesses after Add operations, is removed. However, $\mathsf{pp} = n$ instead of (n, u), and acc, the old value of the accumulator, is removed from upmsg since it is not needed to update membership witnesses. Note that in Fig. 5, it is not necessary to return upmsg. However, we kept it for consistency. An advantage of this construction is its reduced communication complexity. The values of the accumulator and membership witnesses are only updated during the execution of Delete operations.

```
– $\mathsf{Add}(\mathsf{pp}, \mathsf{sk}, \mathsf{acc}, x)$:
    1. Parse $\mathsf{pp}$ as $n$.
    2. Compute $\gamma \leftarrow 1/H(x) \bmod \mathsf{sk}$.
    3. Compute $\mathtt{w} \leftarrow \mathsf{acc}^{\gamma} \bmod n$.
    4. Let $\mathsf{s} = (1)$, $w_x = (\mathtt{w}, \mathsf{s})$ and $\mathsf{upmsg} = (\mathsf{add}, H(x), 1, \perp)$.
    5. Return $\mathsf{acc}$ , $w_x$, and $\mathsf{upmsg}$.
```

Fig. 5: Description of the algorithm Add for the positive dynamic accumulator

Add an element to the accumulator. Let $\mathsf{acc} \in \mathrm{QR}_n$ be the initial value of our positive dynamic accumulator. To keep the accumulator's value unchanged during addition, we use the secret key sk to compute the witness of a newly added element. More specifically, to add an element $x \in \{0,1\}^*$, we use sk to compute a modular $H(x)$-root $\mathtt{w}$ of acc and let $w_x = (\mathtt{w}, \mathsf{s})$, where s is defined as in Section 4.

5.1 Security

Theorem 4. *Assume H is a random oracle. Under the strong RSA assumption, the above construction is a secure positive dynamic accumulator.*

Proof. We proceed by contraposition. Let $\mathcal{A}$ be a ppt adversary that, given $(1^\lambda, \perp, \mathsf{pp}, \mathsf{acc}_0)$ as input, after a total of q_H unique queries to H and a total of $\mathsf{q}_{\mathsf{del}}$ deletion queries to $\mathcal{O}_{\mathsf{Add,Delete}}$, outputs (x^*, w_{x^*}) with non-negligible probability $\varepsilon(\lambda)$ such that $x^* \notin \mathcal{S}$ and $\mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}, x^*, w_{x^*}) = 1$, where $\mathcal{S}$ is the set and acc is the accumulator generated after $\mathcal{A}$'s queries to $\mathcal{O}_{\mathsf{Add,Delete}}$. We build a ppt adversary $\mathcal{B}$ that uses $\mathcal{A}$ to that break the strong RSA assumption as follows:

1. $\mathcal{B}$ receives $(1^\lambda, v, n)$ as input from the Strong RSA Challenger. Then, it computes $u = v^2 \bmod n$ and initialises an empty map $\mathsf{T} : \{0,1\}^* \to \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ and a set $\mathcal{S} = \emptyset$.
2. $\mathcal{B}$ samples $\alpha_1, \ldots, \alpha_{\mathsf{q}_H} \leftarrow\$ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$, $j_1 \leftarrow\$ [\mathsf{q}_H]$, and $j_2 \leftarrow\$ \{0\} \cup [\mathsf{q}_{\mathsf{del}}]$ such that $\alpha_c \neq \alpha_e$ for $c, e \in [\mathsf{q}_H]$ and $c \neq e$.
3. $\mathcal{B}$ computes $\theta \leftarrow \alpha_{j_1}^{j_2} \prod_{i \in [\mathsf{q}_H], i \neq j_1} \alpha_i^{\mathsf{q}_{\mathsf{del}}}$ and $\mathsf{acc} = \mathsf{acc}_0 = u^\theta \bmod n$. Then, $\mathcal{B}$ sets $\mathsf{pp} \leftarrow n$, and initializes $k \leftarrow 1$.
4. $\mathcal{B}$ sends $(1^\lambda, \mathsf{pp}, \mathsf{acc}_0)$ to $\mathcal{A}$ and simulates answers to $\mathcal{A}$'s oracle queries as follows:
   – For H queries, when $\mathcal{A}$ sends $x \in \{0,1\}^*$, $\mathcal{B}$ returns $\mathsf{T}[x]$ if $\mathsf{T}[x] \neq \perp$. Otherwise, $\mathcal{B}$ sets $\mathsf{T}[x] \leftarrow \alpha_k$, $k \leftarrow k+1$, and returns $\mathsf{T}[x]$.
   – For add queries, when $\mathcal{A}$ sends (add, x), $\mathcal{B}$ simulates H with x as input, but it does not return the output to $\mathcal{A}$. If $\mathsf{T}[x] \in \mathcal{S}$, $\mathcal{B}$ aborts. Otherwise, $\mathcal{B}$ computes $\mathtt{w} \leftarrow u^{\theta/\mathsf{T}[x]} \bmod n$, initializes $\mathsf{s} \leftarrow (1)$, and sets $w_x \leftarrow (\mathtt{w}, \mathsf{s})$. Finally, $\mathcal{B}$ sets $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathsf{T}[x]\}$, and returns acc, w_x, and $\mathsf{upmsg} = (\mathsf{add}, \mathsf{T}[x], 1, \perp)$.
   – For del queries, when $\mathcal{A}$ sends (del, x, w_x), $\mathcal{B}$ simulates H with x as input, but it does not return the output to $\mathcal{A}$. If $\mathsf{T}[x] \notin \mathcal{S}$ or if $\mathsf{T}[x] = \alpha_{j_1}$ and $j_2 = 0$, $\mathcal{B}$ aborts. Next, $\mathcal{B}$ sets $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\mathsf{T}[x]\}$, and if $w_x \neq \perp$ and $\mathsf{MemVer}(\mathsf{pp}, \mathsf{acc}, x, w_x) = 1$, $\mathcal{B}$ parses w_x as $(\mathtt{w}, \mathsf{s})$, computes $\delta \leftarrow \prod_{i=1}^{|\mathsf{s}|} \mathsf{s}[i]$, $\theta \leftarrow \theta/(\mathsf{T}[x]/\delta)$, and $\mathsf{acc}' \leftarrow \mathtt{w}$. Otherwise, $\mathcal{B}$ sets $\delta \leftarrow 1$, computes $\theta \leftarrow \theta/\mathsf{T}[x]$, $\mathsf{acc}' \leftarrow u^\theta \bmod n$. In addition, if $\mathsf{T}[x] = \alpha_{j_1}$, $\mathcal{B}$ sets $j_2 \leftarrow j_2 - 1$. Finally, $\mathcal{B}$ returns acc' and $\mathsf{upmsg} = (\mathsf{del}, \mathsf{T}[x], \delta, \mathsf{acc}')$, and then sets $\mathsf{acc} \leftarrow \mathsf{acc}'$.
5. Once $\mathcal{A}$ outputs (x^*, w_{x^*}), $\mathcal{B}$ proceeds as follows:
   (a) If $\mathsf{T}[x^*] \neq \alpha_{j_1}$ or $j_2 \neq 0$ or $\mathsf{MemVer}(\mathsf{pp}, x^*, w_{x^*}, \mathsf{acc}) \neq 1$, $\mathcal{B}$ aborts.
   (b) Parse w_{x^*} as $(\mathtt{w}, \mathsf{s})$, and compute $\mathtt{x} = \mathsf{T}[x^*]/\prod_{i=1}^{[|\mathsf{s}|]} \mathsf{s}[i] = \alpha_{j_1}/\prod_{i=1}^{[|\mathsf{s}|]} \mathsf{s}[i]$.
   (c) Using a process similar to the one described in step 4e of the proof of Theorem 3, if $\gcd(2\theta, \mathtt{x})^{-1}\phi(n)$ exists, compute the $\tilde{\mathtt{x}}$-root of v, where $\tilde{\mathtt{x}} = \mathtt{x}/\gcd(2\theta, \mathtt{x})$. Otherwise, use $\gcd(2\theta, \mathtt{x})$ to factor n.

Note that during the execution of del queries, if $\mathcal{A}$ sends a tuple (del, y, w_y) such that $\mathsf{T}[y] \in \mathcal{S}$ and $w_y = (\mathtt{w}, \mathsf{s})$ is valid, it follows that $\mathtt{w} \equiv \mathsf{acc}^{1/\mathtt{y}} \equiv u^{\theta/\mathtt{y}} \bmod n$, where $\mathtt{y} \leftarrow \mathsf{T}[y]/\prod_{i=1}^{|\mathsf{s}|} \mathsf{s}[i]$.

Since $\alpha_1, \ldots, \alpha_{\mathsf{q}_H}$ are random and distinct from each other, if $\mathcal{B}$ correctly guessed j_2, then it will correctly simulate $\mathcal{O}_{\mathsf{Add,Delete}}$'s answers for add and del queries. In step 5, if $\mathcal{B}$ correctly guessed j_1, then as long as $\gcd(2\theta, \mathtt{x})^{-1} \mod \phi(n)$ exists and α_{j_1} does not divide θ or $\gcd(2\theta, \mathtt{x})^{-1} \mod \phi(n)$ does not exists, $\mathcal{B}$ will be able to break the strong RSA assumption with probability

$$\Pr[\mathcal{B} \text{ wins}] \geq \frac{\varepsilon(\lambda)}{\mathsf{q}_H(\mathsf{q}_{\mathsf{del}}+1)} \left(1 - \nu + \frac{\nu}{2^{\lambda-2}}\right)$$

where $\nu = q_H^2 \left(\frac{1}{2^{\ell^{3/4}}} + \left(\frac{\sqrt[4]{\ell}}{4} \log \ell \right)^{-\sqrt[4]{\ell}} \right)$. □

6 PoE without Primes and Witness Aggregation

In this section, we introduce a variant of Wesolowski's Proof of Exponentiation (PoE) [Wes20] that does not necessitate *hashing* into primes. In addition, we show how the techniques presented by Boneh, Bünz, and Fisch [BBF19] to aggregate (non-)membership witnesses for accumulators defined over primes can be generalized to our setting and how to use the variant of Wesolowski's PoE to reduce the verification time of aggregated (non-)membership witnesses.

6.1 Proof of Exponentiation

Definition 8 (Hidden Order Group Sampler [BHR$^+$21, BBF19]). *A hidden order group sampler is a* ppt *algorithm* GGen *that takes as input a security parameter* 1^λ *and outputs an abelian group* $\mathbb{G}$ *whose order is at most* $2^{\mathsf{poly}(\lambda)}$ *and a trapdoor* sk *that can be used to efficiently compute the exact order of* $\mathbb{G}$*. In addition, for all* ppt *adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[g^e = 1_{\mathbb{G}} \wedge e \neq 0 \;\middle|\; \begin{array}{r} (\mathbb{G}, \mathsf{sk}) \leftarrow \mathsf{GGen}(1^\lambda) \\ g \leftarrow\!\!\$\; \mathbb{G} \\ e \leftarrow \mathcal{A}(1^\lambda, \mathbb{G}, g) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

Definition 9 (Adaptive Root assumption [Wes20]). *A hidden order group sampler* GGen *satisfies the adaptive root assumption with respect to a challenge space* $\mathcal{C} \subset \mathbb{Z}$ *if for all* ppt *adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[v^c = u \neq 1_{\mathbb{G}} \;\middle|\; \begin{array}{r} (\mathbb{G}, \mathsf{sk}) \leftarrow \mathsf{GGen}(1^\lambda) \\ (u, \mathsf{state}) \leftarrow \mathcal{A}_1(1^\lambda, \mathbb{G}) \\ c \leftarrow\!\!\$\; \mathcal{C} \\ v \leftarrow \mathcal{A}_2(u, \mathsf{state}, c) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

Remark 9. Special care must be taken when selecting the challenge space $\mathcal{C}$. For instance, if $\#\mathcal{C} = \mathsf{poly}(\lambda)$, then the adversary $\mathcal{A}$ can have $\mathcal{A}_1$ sample $h \leftarrow\!\!\$\; \mathbb{G}$, compute and output $u = h^{\prod_{i \in \mathcal{C}} i}$, which will guarantee that $\mathcal{A}$ always wins by having $\mathcal{A}_2$ compute and output $v = h^{\prod_{i \in \mathcal{C}, i \neq c} i}$ for any challenge $c \in \mathcal{C}$. One might think that picking $\mathcal{C}$ such that $\#\mathcal{C} = 2^{\mathsf{poly}(\lambda)}$ might be enough to ensure that $\mathcal{A}$ wins with at most negligible probability, but Boneh, Bünz, and Fisch [BBF24] noted that if $a \leftarrow\!\!\$\; \mathcal{C}$ is B-smooth with non-negligible probability and $\#\mathtt{PRIMES} \cap [B] = \mathsf{poly}(\lambda)$, then even though the challenge space is of size $\Theta(2^{\mathsf{poly}(\lambda)})$, $\mathcal{A}$ can win with non-negligible probability by having $\mathcal{A}_1$ compute and output $u = (h')^{\prod_{p \in \mathtt{PRIMES} \cap [B]} p^k}$, where $h' \in \mathbb{G}$ and k is a large integer.

For a pair of interactive Turing machines M and N, let $\langle \mathsf{M}, \mathsf{N} \rangle(x)$ denote N's output after its interaction with M on a common input x.

Definition 10 (Proof of Exponentiation [Wes20, BBF19]). *For* $\lambda \in \mathbb{N}$*, let* $(\mathbb{G}, \mathsf{sk}) \leftarrow \mathsf{GGen}(1^\lambda)$*, and consider the language* $\mathcal{L}_{\mathsf{PoE},\mathbb{G}} = \{(v, u, e) \in \mathbb{G}^2 \times \mathbb{Z} : v^e = u\}$*. A proof of exponentiation (PoE) for* $\mathcal{L}_{\mathsf{PoE},\mathbb{G}}$ *is an interactive protocol (argument) between a* ppt *prover* P *and a* ppt

Initialization:

1. Run $(\mathbb{G}, \mathsf{sk}) \leftarrow \mathsf{GGen}(1^\lambda)$. Then, set $\ell = \mathsf{poly}(\lambda)$ and $\sqrt{\ell} \leq \tau \leq \ell^{3/4}$ such that for $a \leftarrow\$ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$, $P^+(a) > 2^\tau$ with high probability. Finally, send $(1^\lambda, \mathbb{G}, \ell)$ to prover P and verifier V.
2. **Statement:** $(v, u, e) \in \mathbb{G}^2 \times \mathbb{Z}$**.**

Interaction:

1. V samples $c \leftarrow\$ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ and sends it to P.
2. P computes $\pi \leftarrow v^{\lfloor e/c \rfloor}$ and sends it to V.
3. V computes $r \leftarrow e \bmod c$. Then, it outputs 1 if $\pi^c v^r = u$. Otherwise, it outputs 0.

Fig. 6: SimPoE: Wesolowski PoE without primes.

verifier V *such that on a common input* $(v', u', e') \in \mathbb{G}^2 \times \mathbb{Z}$, V *outputs* 1 *after its interaction with* P *if it is* convinced *that* $(v', u', e') \in \mathcal{L}_{\mathsf{PoE},\mathbb{G}}$. *Otherwise,* V *outputs* 0. *In addition, a PoE for* $\mathcal{L}_{\mathsf{PoE},\mathbb{G}}$ *must satisfy the following properties:*

- ***Completeness****: for all* $(v, u, e) \in \mathcal{L}_{\mathsf{PoE},\mathbb{G}}$,

$$\Pr[\langle \mathsf{P}, \mathsf{V} \rangle(v, u, e) = 1] = 1$$

- ***Soundness****: for all* $(v, u, e) \notin \mathcal{L}_{\mathsf{PoE},\mathbb{G}}$,

$$\Pr[\langle \mathsf{P}, \mathsf{V} \rangle(v, u, e) = 1] \leq \mathsf{negl}(\lambda)$$

Remark 10. For a PoE to be useful, a verifier V should perform less than $O(\log e)$ group operations for an input $(v, u, e) \in \mathbb{G}^2 \times \mathbb{Z}$, especially if e is large, because it is always possible to check if $(v, u, e) \in \mathcal{L}_{\mathsf{PoE},\mathbb{G}}$ using $O(\log e)$ group operations via repeated squaring.

Wesolowski [Wes20] constructed a PoE for $\mathcal{L}_{\mathsf{PoE},\mathbb{G}}$ where for a statement $(v, u, e) \in \mathbb{G}^2 \times \mathbb{Z}$, e is a power of 2. To prove that $(v, u, e) \in \mathcal{L}_{\mathsf{PoE},\mathbb{G}}$, V samples a random prime $c \leftarrow\$ \mathtt{PRIMES}(2\lambda)$, where $\mathtt{PRIMES}(2\lambda)$ represents the set of $2^{2\lambda}$ first positive primes, and sends c to P. Next, P computes $\pi \leftarrow v^{\lfloor e/c \rfloor}$ and sends it to V. Finally, V computes $r \leftarrow e \bmod c$ and outputs 1 if $\pi^c v^r = u$. Wesolowski proved that his PoE is sound under the adaptive root assumption with challenge space $\mathcal{C} = \mathtt{PRIMES}(2\lambda)$. In addition, since V's message is completely random, Wesolowski PoE can be converted into a non-interactive protocol via the Fiat-Shamir heuristic [FS87] in the random-oracle model. However, in practice, converting Wesolowski PoE into a non-interactive protocol incurs an $O(\lambda)$ overhead because we will need to hash into primes in the set $[N]$, where $\log_2 N \approx 2\lambda + \log_2 \lambda$. Boneh, Bünz, and Fisch [BBF19] further generalized Wesolowski's protocol by allowing e to be any integer (rather than a power of 2 as in Wesolowski's work).

In Fig. 6, we present a variant of Wesolowski PoE that does not require hashing to primes, which we call *SimPoE*, because it is more simple. The message issued by V is sampled from $\mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$, where $\ell = \mathsf{poly}(\lambda)$ is selected such that for $a \leftarrow\$ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$, $P^+(a) > 2^{\sqrt{\ell}}$ with overwhelming probability (Lemma 2). Since an integer sampled uniformly at random from $\mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ is not smooth with overwhelming probability and $\#\mathsf{Odds}(2^{\ell-1}, 2^\ell - 1) = 2^{\ell-2}$, it follows that the adaptive root assumption with the challenge space $\mathcal{C} = \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ should hold for a hidden order group $\mathbb{G}$ because it will be hard for a ppt adversary to execute the strategies mentioned in Remark 9.

Theorem 5. *Assume* GGen *is a hidden order group sampler, and for* $\lambda \in \mathbb{N}$, *let* $(\mathbb{G}, \mathsf{sk}) \leftarrow \mathsf{GGen}(1^\lambda)$. *Let* $\ell = \mathsf{poly}(\lambda)$ *and* $\sqrt{\ell} \leq \tau \leq \ell^{3/4}$ *such that for* $a \leftarrow\$ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$, $P^+(a) > 2^\tau$ *with overwhelming probability* $1 - \left(\frac{\sqrt[4]{\ell}}{4} \log \ell\right)^{-\sqrt[4]{\ell}}$. *Under the adaptive root assumption with challenge space* $\mathcal{C} = \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$, *SimPoE (Fig. 6) is sound.*

Proof. We proceed by contraposition. Let P^* be a ppt prover that, given $(1^\lambda, \mathbb{G}, \ell)$ as input, makes a verifier V output 1 after executing SimPoE on a statement $(u, v, e) \notin \mathcal{L}_{\mathsf{PoE},\mathbb{G}}$ with non-negligible probability $\varepsilon(\lambda)$. We use P^* to build a ppt adversary $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ that breaks the adaptive root assumption with challenge space $\mathcal{C} = \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ as follows:

```
– Setup(1^λ):
    1. Execute $(\mathbb{G}, \mathsf{sk}) \leftarrow \mathsf{GGen}(1^\lambda)$, and publish $\mathbb{G}$.
– Prove(u, v, e):
    1. Compute $c \leftarrow H(u, v, e)$, and then $\pi \leftarrow v^{\lfloor e/c \rfloor}$.
    2. Return $\pi$.
– Verify(u, v, e, π):
    1. Compute $c \leftarrow H(u, v, e)$, and then $r \leftarrow e \mod c$.
    2. Return 1 if $\pi^c v^r = u$. Otherwise, return 0.
```

Fig. 7: NI-SimPoE.

1. After receiving the statement (u, v, e), $\mathcal{B}_1$ sends u/v^e to the adaptive root challenger, which replies with a challenge $c \leftarrow\!\!\$\ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$. Note that since $(u, v, e) \notin \mathcal{L}_{\mathsf{PoE},\mathbb{G}}$, we have $v^e \neq u$, and so $u/v^e \in \mathbb{G} \setminus \{1\}$.
2. Next, $\mathcal{B}_1$ sends c to P^* and $(u/v^e, c)$ to $\mathcal{B}_2$.
3. After receiving c, P^* computes and sends π to $\mathcal{B}_2$. If $\pi^c v^{e-\lfloor e/c \rfloor c} \neq u$, $\mathcal{B}_2$ aborts. Otherwise, it sends $\pi/v^{\lfloor e/c \rfloor}$ to the adaptive root challenger.

If P^*'s message is correct, then $(\pi/v^{\lfloor e/c \rfloor})^c = u/v^e$. Therefore, $\Pr[\mathcal{B} \text{ wins}] = \varepsilon(\lambda)$. □

For completeness sake, in Fig. 7, we provide a description of a non-interactive version of SimPoE, called NI-SimPoE, in the random oracle model. It uses a random oracle $H : \mathbb{G}^2 \times \mathbb{Z} \rightarrow \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$. Note that H can be instantiated via a random oracle $H' : \{0, 1\}^* \rightarrow \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ such that for $(u, v, e) \in \mathbb{G}^2 \times \mathbb{Z}$, $H(u, v, e) = H'(\mathsf{bin}(u, v, e))$, where $\mathsf{bin}(\cdot, \cdot, \cdot)$ is a function that efficiently maps elements of $\mathbb{G}^2 \times \mathbb{Z}$ to binary strings.

Candidate for GGen. As mentioned in [Wes20, BBF24, BHR+21], a candidate for GGen is a ppt algorithm that samples a random RSA group where the modulus is a product of safe primes, i.e., it takes as input 1^λ and output $n = p.q$ and $\mathsf{sk} = (p - 1)(q - 1)$, where p and q are $O(\lambda)$-bit safe primes. However, Boneh, Bünz, and Fisch [BBF19] noted that over $\mathbb{Z}_n^*$, the soundness of Wesolowski PoE does not hold because with $(v, u, e) \in \mathcal{L}_{\mathsf{PoE},\mathbb{Z}_n^*}$ we can generate a valid interaction for $(v, -u, e)$ by having P reply with $\pi = -1 \cdot v^{\lfloor e/c \rfloor}$ after receiving c from V. Therefore, the subgroup QR_n or the quotient group $\mathbb{Z}_n^*/\{-1, 1\}$ is preferred. In the case of QR_n, testing for membership is *hard* and that makes it an unpractical choice in general, but for our application, QR_n will be enough.

6.2 Aggregating Witnesses

We extend our universal dynamic accumulator construction presented in Section 4 to allow users to aggregate (non-)membership witnesses. This allows a user with a collection of elements and their respective (non-)membership witnesses to generate a witness that can be used to prove membership or non-membership of all the elements in the collection.

First, we extend the definition of universal dynamic accumulator to account for witness aggregation. Remember that we use t to denote a discrete time counter and $\hat{a}$ to denote that the value a is optional.

Definition 11 (Accumulator with Witness Aggregation). *A universal dynamic accumulator* UAcc *for a domain $\mathcal{M}$ supports witness aggregation if it satisfies definitions 2, 3, 4, and 5 and supports the following* ppt *algorithms:*

– $\mathsf{MemWitAggr}(\mathsf{pp}, \widehat{\mathsf{acc}}_t, \{(x_i, w_{x_i,t})\}_{i=1}^m) \rightarrow w_{(x_1,\ldots,x_m),t}$*: This (probabilistic) algorithm takes as input the public parameter* pp*, an optional accumulator value* $\widehat{\mathsf{acc}}_t$*, a set of element and membership witness pairs* $\{(x_i, w_{x_i,t})\}_{i=1}^m$*. It outputs a membership witness* $w_{(x_1,\ldots,x_m),t}$ *that can be used to attest the membership of* $\{x_1, \ldots, x_m\}$*.*

– $\mathsf{NonMemWitAggr}(\mathsf{pp}, \widehat{\mathsf{acc}_t}, \{(x_i, \bar{w}_{x_i,t})\}_{i=1}^m) \rightarrow \bar{w}_{(x_1,\ldots,x_m),t}$: *This (probabilistic) algorithm takes as input the public parameter* pp*, an optional accumulator value* $\widehat{\mathsf{acc}_t}$*, and a set of element and non-membership witness pairs* $\{(x_i, \bar{w}_{x_i,t})\}_{i=1}^m$*. It outputs a non membership witness* $\bar{w}_{(x_1,\ldots,x_m),t}$ *that can be used to attest the non-membership of* $\{x_1, \ldots, x_m\}$*.*
– $\mathsf{MemAggrVer}(\mathsf{pp}, \mathsf{acc}_t, \{x_i\}_{i=1}^m, w_{(x_1,\ldots,x_m),t}) \rightarrow 0/1$: *This deterministic algorithm takes as input the public parameter* pp*, an accumulator* acc_t*, a set of elements* $\{x_1, \ldots, x_m\}$ *and the aggregation of their membership witnesses* $w_{(x_1,\ldots,x_m),t}$*. It returns* 1 *if* $w_{(x_1,\ldots,x_m),t}$ *certifies that* $\{x_1, \ldots, x_m\}$ *is a subset of the set represented by* acc_t*. Otherwise, it returns* 0*.*
– $\mathsf{NonMemAggrVer}(\mathsf{pp}, \mathsf{acc}_t, \{x_i\}_{i=1}^m, \bar{w}_{(x_1,\ldots,x_m),t}) \rightarrow 0/1$: *This deterministic algorithm takes as input the public parameter* pp*, an accumulator* acc_t*, a set of elements* $\{x_1, \ldots, x_n\}$ *and the aggregation of their non-membership witnesses* $\bar{w}_{(x_1,\ldots,x_m),t}$*. It returns* 1 *if* $\bar{w}_{(x_1,\ldots,x_m),t}$ *certifies that* $\{x_1, \ldots, x_m\}$ *is disjointed from the set represented by* acc_t*. Otherwise, it returns* 0*.*

Remark 11. Boneh, Bünz, and Fisch [BBF19] proposed a mechanism to aggregate (non-)membership witnesses for RSA-based accumulators defined over primes. However, they did not provide a clear syntax. Srinivasan et al. [SKBP22] proposed a definition for trapdoorless accumulators in the batching setting, i.e., elements accumulated are sets. However, their proposed syntax includes algorithms to aggregate (non-)membership witnesses of singletons.

We do not provide a formal definition of correctness as it can be obtained by modifying the game presented in Definition 3 (confer Fig. 1) to allow a ppt adversary $\mathcal{A}$ to choose a set of accumulated elements $\{x_1, \ldots, x_m\}$ and a point of time t_1 at which the oracle $\mathcal{O}_{\mathsf{Add,Delete}}$ should aggregate the membership witnesses of those elements. In addition, $\mathcal{A}$ can choose a set of elements $\{y_1, \ldots, y_{m'}\}$ not in the accumulator and a point of time t_2 at which their non-membership witnesses should be aggregated. $\mathcal{A}$ wins if $(\{x_1, \ldots, x_m\}, w_{(x_1,\ldots,x_m),t_1})$ fails $\mathsf{MemAggrVer}$ with respect to acc_{t_1} or $(\{y_1, \ldots, y_{m'}\}, \bar{w}_{(y_1,\ldots,y_{m'}),t_2})$ fails $\mathsf{NonMemAggrVer}$ with respect to acc_{t_2}. We say that a universal dynamic accumulator with witness aggregation supports is correct if $\mathcal{A}$ wins with negligible probability.

Note that aggregated witnesses should also satisfy the definition of compactness (Definition 4). More specifically, for a set $\{x_1, \ldots, x_m\}$, it must be the case that $|w_{(x_1,\ldots,x_m),t}| = |\bar{w}_{(x_1,\ldots,x_m),t}| = \mathsf{poly}(\lambda, |x_1|, \ldots, |x_m|)$.

Definition 12 (Witness Aggregation Security [BBF19]). *A universal dynamic accumulator* UAcc*, for a domain* $\mathcal{M}$*, that supports witness aggregation is secure if for all* ppt *adversary* $\mathcal{A}$ *with oracle access to* $\mathcal{O}_{\mathsf{Add,Delete}}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{r} \mathsf{pp}, \widehat{\mathsf{sk}}, \mathsf{acc}_0 \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{aux}); \\ \mathcal{X}, w_{\mathcal{X},t}, \mathcal{Y}, \bar{w}_{\mathcal{Y},t} \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Add,Delete}}}(1^\lambda, \mathsf{aux}, \mathsf{pp}, \mathsf{acc}_0) : \\ \mathsf{MemAggrVer}(\mathsf{pp}, \mathsf{acc}_t, \mathcal{X}, w_{\mathcal{X},t}) = 1 \\ \wedge\ \mathsf{NonMemAggrVer}(\mathsf{pp}, \mathsf{acc}_t, \mathcal{Y}, \bar{w}_{\mathcal{Y},t}) = 1 \wedge\ \mathcal{X} \cap \mathcal{Y} \neq \emptyset \end{array}\right] \leq \mathsf{negl}(\lambda)$$

where acc_t *is output by* $\mathcal{O}_{\mathsf{Add,Delete}}$*, which is defined as in Definition 5.*

In Figs. 8 and 9, we present the algorithms needed to enable witness aggregation for our universal dynamic accumulator construction. They are based on the work of Boneh, Bünz, and Fisch [BBF19], except that (1) they apply to our accumulator rather than that of Li, Li and Xue [LLX07]; and (2) the Wesolowski challenge c need not be prime. We use a random oracle $H : \{0,1\}^* \rightarrow \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ whose properties are defined as in Section 4. Without PoE, for a set $\{y_1, \ldots, y_m\}$, verifying its (non-)membership using an aggregated witness would require $O(m\ell)$ group operations in $\mathbb{Z}_n^*$. However, by having users prepare PoE proofs using $\mathsf{NI\text{-}SimPoE}$ (confer Fig. 7) and include those proofs in the aggregated witnesses, we are able to reduce the number of group operations to $O(\ell)$, eliminating dependence on m. In addition, since PoE proofs are performed with elements in QR_n (witnesses contain components in QR_n and the accumulator value acc belongs to QR_n), we do not suffer from the PoE soundness issue that arises by working over $\mathbb{Z}_n^*$ as mentioned in Section 6.1.

Now, we give a description of $\mathsf{Mem2Aggr}$ and $\mathsf{NonMem2Aggr}$, which are two helper algorithms presented in Fig. 8 that are used to compute the aggregation of two membership and non-membership witnesses, respectively.

```
– $\mathsf{Mem2Aggr}(\mathsf{pp}, \mathsf{acc}, x, w_x, y, w_y, \mathsf{isDone})$:
  1. Parse $\mathsf{pp}$ as $(n, u)$, $w_x$ as $(\mathtt{w}_x, \mathtt{s}_x)$, and $w_y$ as $(\mathtt{w}_y, \mathtt{s}_y)$.
  2. Compute $\mathtt{x} \leftarrow H(x)/\prod_{i=1}^{|\mathtt{s}_x|} \mathtt{s}_x[i]$ and $\mathtt{y} \leftarrow H(y)/\prod_{i=1}^{|\mathtt{s}_y|} \mathtt{s}_y[i]$.
  3. Find $a, b \in \mathbb{Z}$ such that $a\mathtt{x} + b\mathtt{y} = \gcd(\mathtt{x}, \mathtt{y})$.
  4. Compute $\mathtt{w}_{x,y} \leftarrow \mathtt{w}_x^b \mathtt{w}_y^a$, $\mathtt{y}' \leftarrow \mathtt{y}/\gcd(\mathtt{x}, \mathtt{y})$, and set $\mathtt{s}'_y \leftarrow \mathtt{s}_y \| (\gcd(\mathtt{x}, \mathtt{y}))$.
  5. If $\mathsf{isDone} = 0$, return $w_{x,y} = (\mathtt{w}_{x,y}, \mathtt{s}_x, \mathtt{s}'_y)$.
  6. Else if $\mathsf{isDone} = 1$:
     (a) Compute $\pi_{x,y} \leftarrow \mathsf{NI\text{-}SimPoE.Prove}(\mathtt{w}_{x,y}, \mathsf{acc}, \mathtt{x}\mathtt{y}')$.
     (b) Return $w_{x,y} = (\mathtt{w}_{x,y}, \mathtt{s}_x, \mathtt{s}'_y, \pi_{x,y})$.

– $\mathsf{NonMem2Aggr}(\mathsf{pp}, \mathsf{acc}, x, \bar{w}_x, y, \bar{w}_y, \mathsf{isDone})$:
  1. Parse $\mathsf{pp}$ as $(n, u)$, $\bar{w}_x$ as $(\mathtt{a}_x, \mathtt{B}_x, \mathtt{s}_x)$, and $\bar{w}_y$ as $(\mathtt{a}_y, \mathtt{B}_y, \mathtt{s}_y)$.
  2. Compute $\mathtt{x} \leftarrow H(x)/\prod_{i=1}^{|\mathtt{s}_x|} \mathtt{s}_x[i]$ and $\mathtt{y} \leftarrow H(y)/\prod_{i=1}^{|\mathtt{s}_y|} \mathtt{s}_y[i]$.
  3. Find $a, b \in \mathbb{Z}$ such that $a\mathtt{x} + b\mathtt{y} = \gcd(\mathtt{x}, \mathtt{y})$.
  4. Let $\mathtt{x}' = \mathtt{x}/\gcd(\mathtt{x}, \mathtt{y})$, $\mathtt{y}' = \mathtt{y}/\gcd(\mathtt{x}, \mathtt{y})$, and set $\mathtt{s}'_y \leftarrow \mathtt{s}_y \| (\gcd(\mathtt{x}, \mathtt{y}))$.
  5. Compute $\gamma \leftarrow \mathtt{a}_x b\mathtt{y}' + \mathtt{a}_y a\mathtt{x}'$ and $\mathtt{a}_{x,y} \leftarrow \gamma \bmod \mathtt{x}\mathtt{y}'$.
  6. Compute $\mathtt{B}_{x,y} \leftarrow \mathsf{acc}^{\lfloor \gamma/\mathtt{x}\mathtt{y}' \rfloor} \mathtt{B}_x^b \mathtt{B}_y^a \bmod n$.
  7. If $\mathsf{isDone} = 0$, retun $\bar{w}_{x,y} = (\mathtt{a}_{x,y}, \mathtt{B}_{x,y}, \mathtt{s}_x, \mathtt{s}_y)$.
  8. Else if $\mathsf{isDone} = 1$:
     (a) Compute $\mathtt{C} \leftarrow \mathsf{acc}^{\mathtt{a}_{x,y}} \bmod n$ and $\mathtt{D} \leftarrow \mathtt{B}_{x,y}^{\mathtt{x}\mathtt{y}'} \bmod n$.
     (b) Compute $\pi_{\mathtt{C},(x,y)} \leftarrow \mathsf{NI\text{-}SimPoE.Prove}(\mathsf{acc}, \mathtt{C}, \mathtt{a}_{x,y})$.
     (c) Compute $\pi_{\mathtt{D},(x,y)} \leftarrow \mathsf{NI\text{-}SimPoE.Prove}(\mathtt{B}_{x,y}, \mathtt{D}, \mathtt{x}\mathtt{y}')$.
     (d) Return $\bar{w}_{x,y} = (\mathtt{a}_{x,y}, \mathtt{B}_{x,y}, \mathtt{s}_x, \mathtt{s}_y, \pi_{\mathtt{C},(x,y)}, \pi_{\mathtt{D},(x,y)}, \mathtt{C}, \mathtt{D})$.
```

Fig. 8: Helper Algorithms for Witness Aggregation.

Aggregating two membership witnesses ($\mathsf{Mem2Aggr}$). Let (x, w_x) and (y, w_y) be pairs of element-membership witness with respect to an accumulator value acc. The objective is to produce a membership witness $w_{x,y}$ of size less than $|w_x| + |w_y|$ that can be used to prove x and xy membership in a time that is asymptotically equal to the time it takes to verify the membership of one element. Note that $w_x = (\mathtt{w}_x, \mathtt{s}_x)$ and $w_y = (\mathtt{w}_y, \mathtt{s}_y)$. Let $\mathtt{x} = H(x)/\prod_{i=1}^{|\mathtt{s}_x|} \mathtt{s}_x[i]$ and $\mathtt{y} = H(y)/\prod_{i=1}^{|\mathtt{s}_y|} \mathtt{s}_y[i]$. By tweaking Shamir's trick(Lemma 1), we can compute $\mathtt{w}_{x,y}$ such that $(\mathtt{w}_{x,y})^{\mathtt{x}\mathtt{y}} = \mathsf{acc}^d$, where $d = \gcd(\mathtt{x}, \mathtt{y})$ is 2^τ-smooth with overwhelming probability since, following Lemma 2, $P^+(\mathtt{x}) > 2^\tau$ and $P^+(\mathtt{y}) > 2^\tau$ with overwhelming probability. If $\gcd(\mathtt{x}, \mathtt{y}) = 1$, then we can use $\mathtt{w}_{x,y}$ as one of the component of our aggregated membership witness. However, this is not necessarily the case. To fix the issue, we append d to $\mathtt{s}_y$, generating $\mathtt{s}'_y = \mathtt{s}_y \| (d)$. This insures that we can recover $\mathtt{y}' = \mathtt{y}/d$ such that $(\mathtt{w}_{x,y})^{\mathtt{x}\mathtt{y}'_2} = \mathsf{acc}$. Checking that $\mathtt{w}_{x,y}$ is indeed a modular $\mathtt{x}\mathtt{y}'$-root of acc requires approximately 2ℓ group operations in $\mathbb{Z}_n^*$, which is the same number of group operations needed to check the membership of x and y separately. To make the number of group operations equivalent to the number of group operations needed to prove the membership of one element, we use NI-SimPoE to generate a proof $\pi_{x,y}$ for the statement $(\mathtt{w}_{x,y}, \mathsf{acc}, \mathtt{x}\mathtt{y}')$. Finally, our aggregated membership witness is $w_{x,y} = (\mathtt{w}_{x,y}, \mathtt{s}_x, \mathtt{s}'_y, \pi_{x,y})$. In Fig. 8, $\mathsf{Mem2Aggr}$ takes as argumement a boolean variable isDone, in addition to the tuple (x, w_x, y, w_y), that is used to determine whether the $\mathsf{NI\text{-}SimPoE}$ proof $\pi_{x,y}$ should be computed and included in the aggregated witness. isDone will necessary to generalize the aggregation of membership witnesses for $m \geq 2$ (confer Fig. 9).

Aggregating two non-membership witnesses ($\mathsf{NonMem2Aggr}$). Let $(x, \bar{w}_x)$ and $(y, \bar{w}_y)$ be pairs of element-non-membership witness with respect to an accumulator value acc, and let $u \in \mathbb{Z}_n^*$ be the starting value of the accumulator. The objective is to produce $\bar{w}_{x,y}$ with properties similar to an aggregated membership witness. Note that $\bar{w}_x = (\mathtt{a}_x, \mathtt{B}_x, \mathtt{s}_x)$ and $\bar{w}_y = (\mathtt{a}_y, \mathtt{B}_y, \mathtt{s}_y)$. Let $\mathtt{x} = H(x)/\prod_{i=1}^{|\mathtt{s}_x|} \mathtt{s}_x[i]$ and $\mathtt{y} = H(y)/\prod_{i=1}^{|\mathtt{s}_y|} \mathtt{s}_y[i]$. Let $a, b \in \mathbb{Z}$ such that $a\mathtt{x} + b\mathtt{y} = \gcd(\mathtt{x}, \mathtt{y}) =$

```
– $\mathsf{MemWitAggr}(\mathsf{pp}, \mathsf{acc}, \{(x_1, w_{x_1}), \ldots, (x_m, w_{x_m})\})$:
  1. If $m = 2$, return $\mathsf{Mem2Aggr}(\mathsf{pp}, \mathsf{acc}, x_1, w_{x_1}, x_2, w_{x_2}, 1)$.
  2. Else if $m > 2$ do:
     (a) Compute $w_{x_1,x_2} \leftarrow \mathsf{Mem2Aggr}(\mathsf{pp}, \mathsf{acc}, x_1, w_{x_1}, x_2, w_{x_2}, 0)$.
     (b) For $i = 3$ to $m-1$, do:
         $w_{x_1,\ldots,x_i} \leftarrow \mathsf{Mem2Aggr}(\mathsf{pp}, \mathsf{acc}, (x_j)_{j=1}^{i-1}, w_{x_1,\ldots,x_{i-1}}, x_i, w_i, 0)$.
     (c) Return $\mathsf{Mem2Aggr}(\mathsf{pp}, \mathsf{acc}, (x_j)_{j=1}^{m-1}, w_{x_1,\ldots,x_{m-1}}, x_m, w_m, 1)$.

– $\mathsf{NonMemWitAggr}(\mathsf{pp}, \mathsf{acc}, \{(x_1, \bar{w}_{x_1}), \ldots, (x_m, \bar{w}_{x_m})\})$:
  1. If $m = 2$, return $\mathsf{NonMem2Aggr}(\mathsf{pp}, \mathsf{acc}, x_1, \bar{w}_{x_1}, x_2, \bar{w}_{x_2}, 1)$.
  2. Else if $m > 2$ do:
     (a) Compute $w_{x_1,x_2} \leftarrow \mathsf{NonMem2Aggr}(\mathsf{pp}, \mathsf{acc}, x_1, \bar{w}_{x_1}, x_2, \bar{w}_{x_2}, 0)$.
     (b) For $i = 3$ to $m-1$, do:
         $w_{x_1,\ldots,x_i} \leftarrow \mathsf{NonMem2Aggr}(\mathsf{pp}, \mathsf{acc}, (x_j)_{j=1}^{i-1}, \bar{w}_{x_1,\ldots,x_{i-1}}, x_i, \bar{w}_i, 0)$.
     (c) Return $\mathsf{NonMem2Aggr}(\mathsf{pp}, \mathsf{acc}, (x_j)_{j=1}^{m-1}, \bar{w}_{x_1,\ldots,x_{m-1}}, x_m, \bar{w}_m, 1)$.

– $\mathsf{MemAggrVer}(\mathsf{pp}, \mathsf{acc}, \{x_i\}_{i=1}^{m}, w_{x_1,\ldots,x_m})$:
  1. Parse $\mathsf{pp}$ as $(n, u)$, $w_{x_1,\ldots,x_m}$ as $(\mathtt{w}_{x_1,\ldots,x_m}, \mathbf{s}_{x_1}, \ldots, \mathbf{s}_{x_m}, \pi_{x_1,\ldots,x_m})$.
  2. For $i \in [m]$ do:
     (a) For $j \in [|\mathbf{s}_{x_i}|]$, if $\mathbf{s}_{x_i}[j] > 2^{\tau}$, return 0.
     (b) Compute $\mathtt{x}_i \leftarrow H(x_i)/\prod_{k=1}^{|\mathbf{s}_{x_i}|} \mathbf{s}_{x_i}[k]$.
  3. Return $\mathsf{NI\text{-}SimPoE.Verify}(\mathtt{w}_{x_1,\ldots,x_m}, \mathsf{acc}, \prod_{i=1}^{n} \mathtt{x}_i, \pi_{x_1,\ldots,x_m})$.

– $\mathsf{NonMemAggrVer}(\mathsf{pp}, \mathsf{acc}, \{x_i\}_{i=1}^{m}, \bar{w}_{x_1,\ldots,x_m})$ :
  1. Parse $\mathsf{pp}$ as $(n, u)$, $\bar{w}_{x_1,\ldots,x_m}$ as $(\mathsf{a}_{x_1,\ldots,x_m}, \mathtt{B}_{x_1,\ldots,x_m}, \mathbf{s}_{x_1}, \ldots, \mathbf{s}_{x_m},$
     $\pi_{\mathtt{C},(x_1,\ldots,x_m)}, \pi_{\mathtt{D},(x_1,\ldots,x_m)}, \mathtt{C}, \mathtt{D})$.
  2. For $i \in [m]$ do:
     (a) For $j \in [|\mathbf{s}_{x_i}|]$, if $\mathbf{s}_{x_i}[j] > 2^{\tau}$, return 0.
     (b) Compute $\mathtt{x}_i \leftarrow H(x_i)/\prod_{k=1}^{|\mathbf{s}_{x_i}|} \mathbf{s}_{x_i}[k]$.
  3. Return $\mathsf{NI\text{-}SimPoE.Verify}(\mathsf{acc}, \mathtt{C}, \mathsf{a}_{x_1,\ldots,x_m}, \pi_{\mathtt{C},(x_1,\ldots,x_m)})$
     $\wedge$ $\mathsf{NI\text{-}SimPoE.Verify}(\mathtt{B}_{x_1,\ldots,x_m}, \mathtt{D}, \prod_{i=1}^{n} \mathtt{x}_i, \pi_{\mathtt{D},(x_1,\ldots,x_m)})$ $\wedge$ $\mathtt{CD} \equiv u \bmod n$.
```

Fig. 9: Witness Aggregation Algorithms.

d. Let $\mathtt{x}' = \mathtt{x}/d$ and $\mathtt{y}' = \mathtt{y}/d$. Given that $u = \mathsf{acc}^{\mathsf{a}_x}\mathtt{B}_x^{\mathtt{x}} = \mathsf{acc}^{\mathsf{a}_y}\mathtt{B}_y^{\mathtt{y}}$, it follows that

$$\begin{aligned} u = u^{a\mathtt{x}'+b\mathtt{y}'} &= \left(\mathsf{acc}^{\mathsf{a}_x}\mathtt{B}_x^{\mathtt{x}}\right)^{b\mathtt{y}'}\left(\mathsf{acc}^{\mathsf{a}_y}\mathtt{B}_y^{\mathtt{y}}\right)^{a\mathtt{x}'} \\ &= \mathsf{acc}^{\mathsf{a}_x b\mathtt{y}' + \mathsf{a}_y a\mathtt{x}'}\left(\mathtt{B}_x^{b}\mathtt{B}_y^{a}\right)^{\mathtt{x}\mathtt{y}'} \\ &= \mathsf{acc}^{\gamma \bmod \mathtt{x}\mathtt{y}'}\left(\mathsf{acc}^{\lfloor \gamma/(\mathtt{x}\mathtt{y}')\rfloor}\mathtt{B}_x^{b}\mathtt{B}_y^{a}\right)^{\mathtt{x}\mathtt{y}'} \end{aligned}$$

where $\gamma = \mathsf{a}_x b\mathtt{y}' + \mathsf{a}_y a\mathtt{x}'$.

Let $\mathsf{a}_{x,y} = \gamma \bmod \mathtt{x}\mathtt{y}'$, $\mathtt{B}_{x,y} = \mathsf{acc}^{\lfloor \gamma/(\mathtt{x}\mathtt{y}')\rfloor}\mathtt{B}_x^{b}\mathtt{B}_y^{a}$, and $\mathbf{s}'_y = \mathbf{s}_y \| (d)$. Clearly, $\bar{w}_{x,y} = (\mathsf{a}_{x,y}, \mathtt{B}_{x,y}, \mathbf{s}_x, \mathbf{s}'_y)$ is an aggregated non-membership witness for x and y, but its verification requires approximately the same number of group operations needed to check the non-membership of x and y separately. To solve this issue, we use NI-SimPoE to compute the proofs $\pi_{\mathtt{C},(x,y)}$ and $\pi_{\mathtt{D},(x,y)}$ for the statements $(\mathsf{acc}, \mathtt{C}, \gamma)$ and $(\mathtt{B}_{x,y}, \mathtt{D}, \mathtt{x}\mathtt{y}')$, respectively, with $\mathtt{C} = \mathsf{acc}^{\gamma}$ and $\mathtt{D} = \mathtt{B}_{x,y}^{\mathtt{x}\mathtt{y}'}$. Finally, we let $\bar{w}_{x,y} = (\mathsf{a}_{x,y}, \mathtt{B}_{x,y}, \mathbf{s}_x, \mathbf{s}'_y, \pi_{\mathtt{C},(x,y)}, \pi_{\mathtt{D},(x,y)}, \mathtt{C}, \mathtt{D})$. In Fig. 8, NonMem2Aggr takes as argumement a boolean variable isDone, in addition to the tuple $(x, \bar{w}_x, y, \bar{w}_y)$, that is used to determine whether the values $\mathtt{C}$, $\mathtt{D}$, and the NI-SimPoE proofs $\pi_{\mathtt{C},(x,y)}$, $\pi_{\mathtt{D},(x,y)}$ should be computed and included in the aggregated witness. isDone will necessary to generalize the aggregation of non-membership witnesses for $m \geq 2$ (confer Fig. 9).

In Fig. 9, we use Mem2Aggr in the description of MemWitAggr and NonMem2Aggr in the description of NonMemWitAggr to show how we can aggregate the (non-)membership witnesses of $m > 2$ elements by recursively aggregating the (non-)membership witnesses of two elements. We present a graphical representation of the process in Fig. 10 for $m = 3$ membership witnesses.

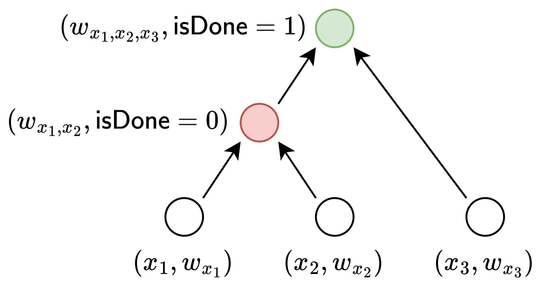
$(w_{x_1,x_2,x_3}, \mathsf{isDone} = 1)$
$(w_{x_1,x_2}, \mathsf{isDone} = 0)$
(x_1, w_{x_1})
(x_2, w_{x_2})
(x_3, w_{x_3})

Fig. 10: A depiction of $\mathsf{MemWitAggr}$'s calls to $\mathsf{Mem2Aggr}$ to aggregate $m = 3$ membership witnesses. The process is similar for $\mathsf{NonMemWitAggr}$ and $\mathsf{NonMem2Aggr}$.

Witness disaggregation. For a set $\{x_1, \dots, x_m\}$ with either aggregated membership witness $w_{x_1,\dots,x_m}$ or aggregated non-membership witness $\bar{w}_{x_1,\dots,x_m}$, it is possible to disaggregate the aggregated witness and obtain a witness for each $x_i \in \{x_1, \dots, x_m\}$:

- Disaggregating $w_{x_1,\dots,x_m}$: From the description of $\mathsf{MemWitAggr}$ in Fig. 9, we have $w_{x_1,\dots,x_m} = (\mathtt{w}_{x_1,\dots,x_m}, \mathbf{s}_1, \dots, \mathbf{s}_m, \pi_{x_1,\dots,x_m})$. To recover a membership witness for $x_i \in \{x_1, \dots, x_m\}$, compute
$$\mathtt{w}_{x_i} \leftarrow \mathtt{w}_{x_1,\dots,x_m}^{\prod_{j=1,j\neq i}^{m}\left(H(x_j)/\prod_{k=1}^{|\mathbf{s}_{x_j}|}\mathbf{s}_{x_j}[k]\right)}$$
and set $w_{x_i} = (\mathtt{w}_{x_i}, \mathbf{s}_{x_i})$.
- Disaggregating $\bar{w}_{x_1,\dots,x_m}$: From the description of $\mathsf{NonMemWitAggr}$ in Fig. 9, $\bar{w}_{x_1,\dots,x_m} = (\mathtt{a}_{x_1,\dots,x_m}, \mathtt{B}_{x_1,\dots,x_m}, \mathbf{s}_1, \dots, \mathbf{s}_m, \pi_{\mathtt{C},(x_1,\dots,x_m)}, \pi_{\mathtt{D},(x_1,\dots,x_m)}, \mathtt{C}, \mathtt{D})$.
To recover a non-membership witness for $x_i \in \{x_1, \dots, x_m\}$, first compute $\mathtt{x}_i \leftarrow H(x_i)/\prod_{i=1}^{|\mathbf{s}_{x_i}|}\mathbf{s}_{x_i}[i]$. Then, compute $\mathtt{a}_{x_i} \leftarrow \mathtt{a}_{x_1,\dots,x_m} \bmod \mathtt{x}_i$ and
$$\mathtt{B}_{x_i} \leftarrow \mathsf{acc}^{\lfloor \mathtt{a}_{x_1,\dots,x_m}/\mathtt{x}_i \rfloor}\mathtt{B}_{x_1,\dots,x_m}^{\prod_{j=1,j\neq i}^{m}\left(H(x_j)/\prod_{k=1}^{|\mathbf{s}_{x_j}|}\mathbf{s}_{x_j}[k]\right)}$$
Finally, set $\bar{w}_{x_i} = (\mathtt{a}_{x_i}, \mathtt{B}_{x_i}, \mathbf{s}_{x_i})$.

Batching deletion. Remember that to delete an element $x \in \{0,1\}^*$ from our accumulator, we need to provide x and its membership witness w_x. Now, with the possibility to aggregate membership witnesses, we can batch the deletion of multiple elements by providing the product of their H evaluations and an aggregation of their membership witnesses. After a batch deletion of elements $\{x_1, \dots, x_m\}$, the (non-)membership witness of an element x' can be updated by executing $\mathsf{MemWitUp}$ or $\mathsf{NonMemWitUp}$ with the update information upmsg', where upmsg' is formed as follows:

- Compute $v' \leftarrow \prod_{i=1}^{m} H(x_i)$.
- If the H evaluations of $x_i \in \{x_1, \dots, x_m\}$ was used during the batch deletion, set $\delta' = 1$. Otherwise, if $w_{x_1,\dots,x_m}$ was used, compute $\delta' \leftarrow \prod_{i=1}^{m}\prod_{j=1}^{|\mathbf{s}_{x_i}|}\mathbf{s}_{x_i}[j]$.
- Finally, set $\mathsf{upmsg}' = (\mathsf{del}, v', \delta', \mathsf{acc}, \mathsf{acc}')$, where acc is the old accumulator value for which the witness to be updated is valid and acc' is the new accumulator value.

Theorem 6. *Our universal dynamic accumulator with support for witness aggregation is compact.*

Proof. For a set $\{x_1, \dots, x_m\}$ with membership witness $w_{x_1,\dots,x_m}$, we have $|w_{x_1,\dots,x_m}| < 4(\lambda + 2) + m\ell$, and for a set $\{y_1, \dots, y_{m'}\}$ with non-membership witness $\bar{w}_{y_1,\dots,y_{m'}}$, we have $|\bar{w}_{y_1,\dots,y_{m'}}| < 10(\lambda + 2) + 2m'\ell$. $\square$

λ	$\mathsf{H}_{\mathsf{Prime}}$ length	$\mathsf{H}_{\mathsf{Prime}}$ time (ms)	# Primality test	Primality test time w/ composite input (ms)	Primality test time w/ prime input (ms)	$\mathsf{H}_{\mathsf{Odd}}$ length	$\mathsf{H}_{\mathsf{Odd}}$ time (ms)
112	232	10.65	158.33	0.04	4.03	1440	0.48
128	264	13.62	177.37	0.04	4.95	1704	0.60
192	393	31.9	274.91	0.06	13.03	2896	1.07
256	521	52.31	345.69	0.08	22.21	4208	1.56

Table 1: $\mathsf{H}_{\mathsf{Prime}}$ versus $\mathsf{H}_{\mathsf{Odd}}$. **# Primality test** represents the average number of primality test performed during an evaluation of $\mathsf{H}_{\mathsf{Prime}}$, **Primality test time w/ composite input** represents the average time it takes to check if a number is a composite, and **Primality test time w/ prime input** represents the average time it takes to check if a number is prime. The time required to compute $\mathsf{H}_{\mathsf{Prime}}$ is dominated by the product of **# Primality test** and **Primality test time w/ composite input** plus **Primality test time w/ prime input** since we stop the execution once we find a prime.

λ	$\mathsf{Add}^{(\mathsf{H}_{\mathsf{Prime}},\mathsf{sk})}$ time (ms)	$\mathsf{Add}^{(\mathsf{H}_{\mathsf{Odd}},\mathsf{sk})}$ time (ms)	$\mathsf{Add}^{\mathsf{H}_{\mathsf{Prime}}}$ time (ms)	$\mathsf{Add}^{\mathsf{H}_{\mathsf{Odd}}}$ time (ms)
112	12.83	2.73	11.06	1.96
128	20.37	7.38	14.27	3.98
192	99.48	68.84	35.34	25.10
256	456.10	402.71	65.97	110.6

Table 2: Comparison of different Add algorithms. $\mathsf{Add}^{(\mathsf{H},\mathsf{sk})}$ represents the addition procedure that uses the secret key sk and H as the underlying hash function, and $\mathsf{Add}^{\mathsf{H}}$ represents the addition procedure that is performed without sk using H as the underlying hash function.

Theorem 7. *Our universal dynamic accumulator with support for witness aggregation is correct.*

Proof. This follows from Theorem 1 and by inspecting how aggregated (non-)membership witnesses are computed. ☐

Theorem 8. *Assume H is a random oracle. Under the strong RSA assumption and the adaptive root assumption, our universal dynamic accumulator with support for witness aggregation is secure.*

Proof. We proceed by contraposition. Let $\mathcal{A}$ be a ppt adversary that, on input $(1^\lambda, \perp, \mathsf{pp}, \mathsf{acc}_0)$, where $(\mathsf{pp}, \mathsf{acc}_0)$ are output of $\mathsf{Gen}(1^\lambda, \perp)$, can output $(\mathcal{X}, w_{\mathcal{X}}, \mathcal{Y},$
$\bar{w}_{\mathcal{Y}})$ with probability $\varepsilon(\lambda)$ such that $w_{\mathcal{X}}$ and $\bar{w}_{\mathcal{Y}}$ are valid, $\mathcal{X} \cap \mathcal{Y} \neq \emptyset$, and $\varepsilon(\lambda)$ is non-negligible. We use $\mathcal{A}$ to either break the strong RSA assumption or the adaptive root assumption as follows:

– If there exists $x \in \mathcal{X} \cap \mathcal{Y}$ such that after extracting its membership witness w_x from $w_{\mathcal{X}}$ and its non-membership $\bar{w}_x$ from $\bar{w}_{\mathcal{Y}}$, w_x and $\bar{w}_x$ are valid, then from Theorem 3, it follows that we can use $x, w_x, \bar{w}_x$ to break the strong RSA assumption.
– If it is not the case, then either the PoE proof $\pi_{\mathcal{X}}$ associated to $w_{\mathcal{X}}$ or the PoE proofs $\pi_{\mathsf{C},\mathcal{Y}}$ and $\pi_{\mathsf{D},\mathcal{Y}}$ associated to $\bar{w}_{\mathcal{Y}}$ are forgeries, and from Theorem 5, we can use them to break the adaptive root assumption.

Hence, with probability at least $\varepsilon(\lambda)$, we can either break the strong RSA assumption or the adaptive root assumption in polynomial time. ☐

7 Experimentation

In this section, we compare the time it takes to hash to a prime integer with the time it takes to hash to a large odd integer, and consequences for the efficiency of previous RSA-based accumulator constructions compared to ours.

λ	$\mathsf{Delete}^{(\mathsf{H}_{\mathsf{Prime}},\mathsf{sk})}$ time (ms)	$\mathsf{Delete}^{(\mathsf{H}_{\mathsf{Odd}},\mathsf{sk})}$ time (ms)	$\mathsf{MemVer}^{\mathsf{H}_{\mathsf{Prime}}}$ time (ms)	$\mathsf{MemVer}^{\mathsf{H}_{\mathsf{Odd}}}$ time (ms)
112	6.22	2.19	4.44	1.97
128	12.71	6.81	6.01	4.02
192	82.76	68.44	17.18	25.19
256	424.45	403.27	36.47	109.73

Table 3: Comparison of Delete algorithms using the secret key sk, and MemVer algorithms. We can delete an element without sk if we have a valid membership witness for it, and the time required to execute Delete that way is dominated by the time it takes to execute MemVer.

λ	Exponent size (bit)	Modular exp time (ms)	Exponent size (bit)	Modular exp time (ms)
112	232	0.26	1440	1.47
128	264	0.6	1704	3.67
192	393	3.74	2896	26.49
256	521	14.62	4208	111.70

Table 4: Modular exponentiation with an exponent sampled from $\mathsf{H}_{\mathsf{Prime}}$'s output domain versus modular exponentiation with an exponent sampled from $\mathsf{H}_{\mathsf{Odd}}$'s output domain.

More specifically, let $\mathsf{H}_{\mathsf{Prime}}$ be a hash function that hashes to primes and $\mathsf{H}_{\mathsf{Odd}}$ be a hash function that hashes to large odd integers.

First, we need to establish the range of $\mathsf{H}_{\mathsf{Prime}}$ and $\mathsf{H}_{\mathsf{Odd}}$ that would give us λ bits of security, i.e. the same amount of security as a random function f with 2λ-bit outputs. (Why compare to such a random function? By the birthday bound, it takes $O(2^{\lambda})$ time to find a collision in f, giving us λ bits of security.) Suppose the function $\mathsf{H}_{\mathsf{Prime}}$ used in previous accumulator constructions returns a random prime integer in the set $[2^n]$. In order to achieve the same level of security (i.e. collision-resistance that the previous constructions require of $\mathsf{H}_{\mathsf{Prime}}$) as f, we need to set $n = 2\lambda + \log_2(2\lambda)$. This is because, by the prime number theorem, there are asymptotically exactly $2^{2\lambda}$ primes less than 2^n. In other words, to get λ bits of security, $\mathsf{H}_{\mathsf{Prime}}$ must return a random prime from the set $[2^{2\lambda+\log_2(2\lambda)}]$. In our experiments, we took the same approach to computing $\mathsf{H}_{\mathsf{Prime}}$ as prior work [BP97,GHR99]: to compute $\mathsf{H}_{\mathsf{Prime}}(x)$, compute $y = H(x)$, where H is a hash function with $n/2$-bit outputs, and then for $r \in \{0, \ldots, 2^{n/2} - 1\}$, if $z = 2^{n/2}y + r$ is prime, stop and return (z, r). Note that, with this approach, given x and the witness w, to verify that x is in the accumulator it is not necessary to redo the rejection sampling when we execute $\mathsf{H}_{\mathsf{Prime}}$: if r is appended to the witness w, then it is sufficient to check that (1) $z' = 2^{n/2}H(x) + r$ is prime and (2) $w^{z'} = \mathsf{acc}$; thus the penalty incurred from hashing to primes can be minimized.

Next, how should we set up the output length of our function $\mathsf{H}_{\mathsf{Odd}}$ to achieve λ bits of security? First, note that, the security of our construction requires that it be hard for the adversary's to find two inputs x_1 and x_2 such that $P^{+}(\mathsf{H}_{\mathsf{Odd}}(x_1))|\mathsf{H}_{\mathsf{Odd}}(x_2)$. So to be comparable to the security of $\mathsf{H}_{\mathsf{Prime}}$, we need $\log_2(P^{+}(\mathsf{H}_{\mathsf{Odd}}(x))) \geq 2\lambda + \log_2(2\lambda)$ for any input x with overwhelming probability. From Lemma 2, if $\mathsf{H}_{\mathsf{Odd}}$'s outputs are ℓ-bit long, then $\log_2(P^{+}(\mathsf{H}_{\mathsf{Odd}}(x))) > \ell^{3/4}$ with overwhelming probability. So to have at least the same (λ-bit) security as $\mathsf{H}_{\mathsf{Prime}}$, we need $\ell^{3/4} \geq 2\lambda + \log_2(2\lambda)$, i.e., we set $\ell \geq (2\lambda + \log_2(2\lambda))^{4/3}$.

We are now ready to present our performance results. Table 1 compares the time it takes to compute $\mathsf{H}_{\mathsf{Prime}}$ with λ bits of security with the time it takes to compute $\mathsf{H}_{\mathsf{Odd}}$ with the same security.

In Table 2, we compare the Add algorithms. Recall that, in prior accumulators, in order to add an element x, one first needed to compute $\mathsf{H}_{\mathsf{Prime}}(x)$, and then either update the accumulator A to $A^{\mathsf{H}_{\mathsf{Prime}}(x)} \bmod n$ (in the event that the addition is performed without access to the secret key, i.e. the factorization of n), or compute the witness $w_x = A^{1/\mathsf{H}_{\mathsf{Prime}}(x)}$ (in the event that the addition is performed with access to the secret key). In our construction, instead of computing $\mathsf{H}_{\mathsf{Prime}}(x)$, one needs to just compute $\mathsf{H}_{\mathsf{Odd}}(x)$; which, as we see from Table 1, is a much more efficient operation. However, the drawback is that the output of $\mathsf{H}_{\mathsf{Odd}}$ is longer, meaning that the cost of exponentiation is increased (as elaborated on in Table 4). As a result, our approach is

λ	Mem wit size $\mathsf{H}_{\mathsf{Prime}}$ (byte)	Mem wit size $\mathsf{H}_{\mathsf{Odd}}$ (byte)	Non-mem wit size $\mathsf{H}_{\mathsf{Prime}}$ (byte)	Non-mem wit size $\mathsf{H}_{\mathsf{Odd}}$ (byte)
112	256	436	285	616
128	384	597	417	810
192	960	1322	1010	1684
256	1920	2448	1986	2972

Table 5: Comparison of witness sizes between an accumulator using $\mathsf{H}_{\mathsf{Prime}}$ and an accumulator using $\mathsf{H}_{\mathsf{Odd}}$.

λ	PoE prove$^{\mathsf{H}_{\mathsf{Prime}}}$ time(s)	PoE prove$^{\mathsf{H}_{\mathsf{Odd}}}$ time(s)	PoE verify$^{\mathsf{H}_{\mathsf{Prime}}}$ time(ms)	PoE verify$^{\mathsf{H}_{\mathsf{Odd}}}$ time(ms)
112	1	0.98	4.92	3.02
128	2.24	2.15	6.55	7.27
192	9.34	9.29	21.15	52.97
256	27.45	27.34	51.30	224.80

Table 6: PoE using $\mathsf{H}_{\mathsf{Prime}}$ versus PoE using $\mathsf{H}_{\mathsf{Odd}}$. We used an exponent of 128 KB.

significantly less costly for the values of λ used in practice, such as $\lambda = 112$ or $\lambda = 128$; however, as λ increases, resulting in longer RSA moduli, our approach is not as beneficial. In Table 3, we make a similar comparison for Delete and MemVer; here, we are more generous to the previous schemes that use $\mathsf{H}_{\mathsf{Prime}}$: when deleting an element x, we know that $\mathsf{H}_{\mathsf{Prime}}(x)$ has already been computed, which would allow one to recompute it more efficiently than doing it from scratch (as described above). In Table 5 we compare the witness sizes of the two approaches. Table 6 compares our PoE approach with prior work [Wes20].

We performed our experiments for four realistic values for the security parameter λ. The lowest one, $\lambda = 112$, is the security level that, according to NIST [Bar20], corresponds to the security of 2048-bit RSA, which is a popular parameter choice. We also consider setting λ to 128, 192 and 256, because those security levels are also NIST recommended.

We performed our experimentation on a laptop equipped with an Intel Core i7-11800H 2.30 GHz CPU and 16 GB of RAM running Ubuntu 22.04.03 LTS via Windows Subsytem for Linux. We used SageMath version 9.5 to implement our prototype. To instantiate $\mathsf{H}_{\mathsf{Prime}}$, we used the construction of Barić and Pfitzmann [BP97] with Blake2b from the PyCryptodome library version 3.19.0[3]as the underlying collision resistant hash function with a digest of 224-bit for $\lambda = 112$, 256-bit for $\lambda = 128$, 384-bit for $\lambda = 192$, and 512-bit digest for $\lambda = 256$. For primality testing, we used Miller-Rabin primality test from PyCryptodome with an error probability of 10^{-30}, which translates to 50 rounds of Miller-Rabin test's execution per primality check. To instantiate $\mathsf{H}_{\mathsf{Odd}}$ we concatenated multiple outputs of Blake2b with a digest of 224-bit for $\lambda = 112$, 256-bit for $\lambda = 128$, 384-bit for $\lambda = 192$, and 512-bit for $\lambda = 256$. Then, we flipped the most and least significant bits to 1. All group operations were performed over an RSA modulus of 2048 bits for $\lambda = 112$, 3072 bits for $\lambda = 128$, 7680 bits for $\lambda = 192$, and 15360 bits for $\lambda = 256$ following NIST recommendations [Bar20].

For Tables 1 to 3, we used 4 KB inputs for both $\mathsf{H}_{\mathsf{Prime}}$ and $\mathsf{H}_{\mathsf{Odd}}$. In addition, for Tables 1 to 4, the times and numbers of primality tests listed are averages from 500 trials. However, for Table 6, due to resource constraints, we reduced the number of trials to 100.

Acknowledgments

We thank the anonymous reviewers of ACM CSS 2024 for their comments, which improved the experimentation section. We also thank Peihan Miao for her feedback on our experimentations. This work was supported by NSF grants 2312241, 2154170, and 2247305.

[3] `https://github.com/Legrandin/pycryptodome`

References

ATSM09. 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 Marc Fischlin, editor, *Topics in Cryptology – CT-RSA 2009*, pages 295–308, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

Bar20. Elaine Barker. *Recommendation for key management:: part 1 - general.* May 2020.

BBF19. Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 561–586, Cham, 2019. Springer International Publishing.

BBF24. Dan Boneh, Benedikt Bünz, and Ben Fisch. A survey of two verifiable delay functions using proof of exponentiation. *IACR Communications in Cryptology*, 1(1), 2024.

BCD$^+$17. Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 301–315, 2017.

BCY20. Foteini Badimtsi, Ran Canetti, and Sophia Yakoubov. Universally composable accumulators. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, pages 638–666, Cham, 2020. Springer International Publishing.

BdM94. Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 274–285, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

BHR$^+$21. Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Time- and space-efficient arguments from groups of unknown order. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, pages 123–152, Cham, 2021. Springer International Publishing.

BKR24. Foteini Baldimtsi, Ioanna Karantaidou, and Srinivasan Raghuraman. Oblivious accumulators. In *Public-Key Cryptography – PKC 2024: 27th IACR International Conference on Practice and Theory of Public-Key Cryptography, Sydney, NSW, Australia, April 15–17, 2024, Proceedings, Part II*, page 99–131, Berlin, Heidelberg, 2024. Springer-Verlag.

BLL02. Ahto Buldas, Peeter Laud, and Helger Lipmaa. Eliminating counterevidence with applications to accountable certificate management. *J. Comput. Secur.*, 10(3):273–296, sep 2002.

BP97. Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 480–494, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

CH10. Philippe Camacho and Alejandro Hevia. On the impossibility of batch update for cryptographic accumulators. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *Progress in Cryptology – LATINCRYPT 2010*, pages 178–188, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

CHKO12. Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. *International Journal of Information Security*, 11(5):349–363, Oct 2012.

CKS09. Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Stanisław Jarecki and Gene Tsudik, editors, *Public Key Cryptography – PKC 2009*, pages 481–500, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

CL01. Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045, pages 93–118. Springer Verlag, 2001.

CL02. Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 61–76, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

CL03. Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In *SCN 2002*, volume 2576, pages 268–289, 2003.

CW09. Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. SSYM'09, page 317–334, USA, 2009. USENIX Association.

dB51. Nicolaas G de Bruijn. The asymptotic behaviour of a function occuring in the theory of primes. *Journal of the Indian Mathematical Society. New Series*, 15:25–32, 1951.

DHS15. David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In Kaisa Nyberg, editor, *Topics in Cryptology — CT-RSA 2015*, pages 127–144, Cham, 2015. Springer International Publishing.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

GHR99. Rosario Gennaro, Shai Halevi, and Tal Rabin. Secure hash-and-sign signatures without the random oracle. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 123–139, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

GOP$^+$16. Esha Ghosh, Olga Ohrimenko, Dimitrios Papadopoulos, Roberto Tamassia, and Nikos Triandopoulos. Zero-knowledge accumulators and set algebra. In *Proceedings, Part II, of the 22nd International Conference on Advances in Cryptology — ASIACRYPT 2016 - Volume 10032*, page 67–100, Berlin, Heidelberg, 2016. Springer-Verlag.

HT93. Adolf Hildebrand and Gerald Tenenbaum. Integers without large prime factors. *Journal de théorie des nombres de Bordeaux*, 5(2):411–484, 1993.

LCL$^+$17. James Larisch, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. CRLite: A Scalable System For Pushing All TLS Revocations To Browsers. In *IEEE Symposium on Security and Privacy*, San Jose, California, USA, May 2017.

Lip12. Helger Lipmaa. Secure accumulators from euclidean rings without trusted setup. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security*, pages 224–240, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

LLX07. Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 253–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

LTZ$^+$15. Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An End-to-end Measurement Of Certificate Revocation In The Web's PKI. In *ACM Internet Measurement Conference*, Tokyo, Japan, October 2015.

Lys02. Anna Lysyanskaya. *Signature schemes and applications to cryptographic protocol design.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 2002.

MV13. Atefeh Mashatan and Serge Vaudenay. A fully dynamic universal accumulator. *Proceedings Of The Romanian Academy Series A-Mathematics Physics Technical Sciences Information Science*, 14:269–285, 2013.

Ngu05. Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, pages 275–292, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

PB10. Kun Peng and Feng Bao. Vulnerability of a non-membership proof scheme. *2010 International Conference on Security and Cryptography (SECRYPT)*, pages 1–4, 2010.

RY16. Leonid Reyzin and Sophia Yakoubov. Efficient asynchronous accumulators for distributed pki. In Vassilis Zikas and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 292–309, Cham, 2016. Springer International Publishing.

Sha81. Adi Shamir. On the generation of cryptographically strong pseudo-random sequences. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming*, pages 544–550, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

SKBP22. Shravan Srinivasan, Ioanna Karantaidou, Foteini Baldimtsi, and Charalampos Papamanthou. Batching, aggregation, and zero-knowledge proofs in bilinear accumulators. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2719–2733, New York, NY, USA, 2022. Association for Computing Machinery.

Wes20. Benjamin Wesolowski. Efficient verifiable delay functions. *Journal of Cryptology*, 33(4):2113–2147, Oct 2020.

A Communication Lower Bound for Batch updates

Camacho and Hevia [CH10] showed that for a secure dynamic accumulator with deterministic witness update procedures and deterministic witness verification procedures, after m additions or deletions, the update information upmsg will need to be of size $\Omega(m)$. For our batch update procedures (for addition and deletion) presented in Sections 2 and 6.2, we have $\mathsf{upmsg} = (\mathsf{op}, v, \delta, \mathsf{acc}, \mathsf{acc}')$ with v representing the product of the hash evaluation of the elements that were added or removed. Therefore, after m additions or deletions, $|\mathsf{upmsg}| = \Omega(m)$.