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 universal 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 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.

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 represented 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 $\mathsf{acc}_\mathcal{S} = \mathsf{acc}_\emptyset^{\prod_{x\in S} x}$ 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 $\mathcal{S} \subseteq \{2^\ell, \dots, 2^{\ell+1} - 1\}$. 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 $ax + by = 1$ 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 $w_z = w^{x_2y_2}$ will pass as the witness for $z = x_1y_1$: $w_z^z = (w^{x_2y_2})^{x_1y_1} = w^{xy} = \mathsf{acc}_\mathcal{S}$.

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.

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, BKR23]; 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.

We also show that, under the adaptive root assumption of Wesolowski [Wes20] and the strong RSA assumption, we can batch witnesses. 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.

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 6, hashing to even a small, 264-bit prime (which is the smallest reasonable length since we need to avoid collisions) takes about 0.02 seconds 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 makes 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 2048. This necessitates a more involved modular exponentiation, but in our experiment comparing running times of the implementations of both approaches, it was still about 2.3 times faster than hashing into a 264-bit prime. For larger parameters, the relative benefit is even bigger (see Section 6 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 7.2, 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 details, see Section 4, where we also show how to generalize non-membership witnesses of Li, Li and Xue to obtain a universal accumulator.

In Section 5 we focus on the positive (rather than universal) 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.

Finally, in Section 7, 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.

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 $\{1, \dots, n\}$, 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 $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_m)$, we use $\mathbf{x}\|\mathbf{y}$ to denote their concatenation, i.e., $\mathbf{x}\|\mathbf{y} = (x_1, \dots, x_n, y_1, \dots, y_m)$. 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 $\ell \in \mathbb{N}$, for every constant $1 \le c \le \sqrt[4]{\ell}$ and $a \leftarrow\!\$\ \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$,*

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

Proof. Let $1 \leq c \leq \sqrt[4]{\ell}$ 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

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

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

Corollary 1. *Given a sufficiently large $\ell \in \mathbb{N}$, for every constant $1 \leq c \leq \sqrt[4]{\ell}$, $m \in \mathbb{N}$, 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,*

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

Proof. Let ℓ be a large integer, and suppose $1 \leq c \leq \sqrt[4]{\ell}, m \in \mathbb{N}$ 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

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

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

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

□

3 Cryptographic Accumulator

We recall the definition of universal and positive dynamic accumulators based on [RY16,BCD+17, DHS15,BKR23]. Our definition of non-membership witness creation is borrowed from the work of Baldimtsi, Karantaidou and Raghuraman [BKR23]. 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 polynomial time algorithms* $\mathsf{UAcc} = (\mathsf{Gen}, \mathsf{Add}, \mathsf{Delete}, \mathsf{NonMemWitCreate}, \mathsf{MemWitUp}, \mathsf{NonMemWitUp}, \mathsf{MemVerify}, \mathsf{NonMemVerify})$ *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*.*
- $\mathsf{MemVerify}(\mathsf{pp}, \mathsf{acc}_t, x, w_{x,t}) \to 0/1$*: 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.*
- $\mathsf{NonMemVerify}(\mathsf{pp}, \mathsf{acc}_t, x, \bar{w}_{x,t}) \to 0/1$*: 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.*

Next, we need to define what it means for an accumulator to be correct. Reyzin and Yakoubov [RY16] were the first to give a formal definition of a correct accumulator (prior work omitted one); theirs applied only to an additive accumulator, i.e. one that supports dynamic additions but not deletions. Here, we use a similar approach to define correctness for a universal accumulator. The goal of a ppt adversary $\mathcal{A}$ that attacks the correctness of the system is to interact with the accumulator (with access to an oracle $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$) and produce a correctness error: either a value x in the accumulator whose membership witness fails to verify; or y not in the accumulator whose non-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 $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$ 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, $\mathbf{mwit}$ that stores membership witnesses of elements in $\mathbf{v}$, and $\mathbf{upmsgs}$ that stores all update information produced after the execution of add and del queries. After each add or del query, $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$ updates the all out-of-date membership witnesses stored in $\mathbf{mwit}$, 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*

```
$\mathsf{CorrectExp}^{\mathcal{A}}(1^\lambda)$ :
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 :  $x, y, i, j \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}}(1^\lambda, \mathsf{aux}, \mathsf{pp}, \mathsf{acc}_0)$
3 :  if $x = y \vee x \neq \mathbf{v}[i] \vee y \in \mathbf{v}$ : 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 $\big(\mathsf{MemVerify}(\mathsf{pp}, \mathsf{acc}_t, x, w_{x,t}) = 0\big)$
            $\vee \big(\mathsf{NonMemVerify}(\mathsf{pp}, \mathsf{acc}_t, y, \bar{w}_{y,t}) = 0\big)$

$\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}(\mathsf{op}, v)$ :
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

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 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 2. 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* $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*,*

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

where acc_t *is output by* $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$*. For this definition,* $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$ *is defined as in Fig. 1 with the exception that the tuple* $\mathbf{v}$ *that stores elements added to the accumulator is replaced by a set* $\mathcal{S}$*, and it does not store membership witnesses and update information. Furthermore, without loss of generality, adversaries are required to send witness memberships during delete requests.*

Remark 3. The definition of correctness (Definition 3) implies that for a given $x \in \mathcal{M}$, if x is present in the accumulator, then there exists a valid membership witness. Otherwise, there exists a valid non-membership witness. Therefore, if an adversary is able to produce valid membership and non-membership witnesses for the same element, it must be the case that one of them is a forgery.

Remark 4. Lipmaa [Lip12] proposed an alternative definition for the security of universal accumulators that is more suited for static settings where an accumulator does not need secret parameters to operate. In that definition, the goal of an adversary is to output an accumulator value acc^* in addition to outputting an element x with valid membership and non-membership witnesses w_x and $\bar{w}_x$, respectively. Lipmaa's definition was further extended to dynamic settings by Boneh, Bünz and Fisch [BBF19]. Without secret parameters, an adversary does not need access to an oracle to add and remove elements from an accumulator.

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

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* $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*,*

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

where acc_t *is output by* $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$*. For this definition,* $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$ *is defined as in Fig. 1 with the exception that the tuple* $\mathbf{v}$ *that stores elements added to the accumulator is replaced by a set* $\mathcal{S}$*, and it does not store membership witnesses and update information. Furthermore, without loss of generality, adversaries are required to send witness memberships during delete requests.*

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.

```
– $\mathsf{Gen}(1^\lambda, \perp)$:
    1. Select primes $p, q, p', q'$ such that $p = 2p' + 1$, $q = 2q' + 1$ and $\log_2 p' = \log_2 q' = \lambda$.
    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 $\mathsf{MemVerify}(\mathsf{pp}, \mathsf{acc}, x, w_x) = 0$, 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 $\mathsf{MemVerify}(\mathsf{pp}, \mathsf{acc}, x, w_x) = 1$, 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

– a user: it is in charge of managing witnesses by executing the algorithms NonMemWitCreate, MemWitUp, and NonMemWitUp. It can also issue add and del requests to an accumulator manager.
– a verifier: it executes the algorithms MemVerify and NonMemVerify. 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. $\bar{w}'_x$).

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 $\sqrt{\ell} \le \tau \le \ell^{3/4}$ be a value chosen such that for $x \in \{0,1\}^*$, $P^+(H(x)) > 2^\tau$ with overwhelming probability.

Remark 5. 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.

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,

```
– $\mathsf{NonMemWitCreate}(\mathsf{pp}, x, \{\mathsf{upmsg}_i\}_{i=1}^m)$:
  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 $\theta \leftarrow \prod_{y \in \mathcal{S}} y \prod_{i=1}^{|\mathbf{d}|} \mathbf{d}[i]$.
  5. Let $\mathtt{x} \leftarrow H(x)$ and $\mathtt{s} \leftarrow (1)$.
  6. While $\gcd(\theta, \mathtt{x}) \neq 1$, set $\mathtt{x} \leftarrow \mathtt{x} / \gcd(\theta, \mathtt{x}), \mathtt{s} \leftarrow \mathtt{s} \| (\gcd(\theta, \mathtt{x}))$.
  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 $\bar{w}_x = (a, \mathtt{B}, \mathtt{s})$.

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

Fig. 3: User's algorithms

$\delta' = 1$, 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 and of our 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 $\log_2 p' = \log_2 q' = \lambda$. 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

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

Fig. 4: Verifier's algorithms

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 $\Pr\left[(p' \mid H(x)) \vee (q' \mid H(x))\right] \leq \frac{2}{2^{\lambda-1}} = \frac{1}{2^{\lambda-2}}$ □

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

Proof. This follows from Lemma 3. □

Lemma 4. 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 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 $\sqrt{\ell} \leq \tau \leq \ell^{3/4}$, 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 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

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

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, $w_x = (\mathtt{w}, \mathsf{s})$ its valid membership witness with respect to acc, and $w'_x = (\mathtt{w}', \mathsf{s}')$ 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 $y \in \{0,1\}^*$, $\mathtt{x} = H(x)/\prod_{i=1}^{|\mathsf{s}|} \mathsf{s}[i]$, 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' 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 $\mathsf{s}' = \mathsf{s}$, 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 s are less than 2^τ, and this is also the case for 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 $\delta \geq 1$ 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 $\sqrt{\ell} \leq \tau \leq \ell^{3/4}$, 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 $\sqrt{\ell} \le \tau \le \ell^{3/4}$ 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 $|\mathsf{acc}| \le 2(\lambda + 2)$. 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 $\bar{w}_y = (\mathsf{a}, \mathtt{B}, \mathsf{s}')$, we have $\mathsf{a} \in \mathbb{Z}_{H(y)}$, $\mathtt{B} \in \mathbb{Z}_n^*$, 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 $(1^\lambda, \bot, \mathsf{pp}, \mathsf{acc}_0)$ 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 $\mathsf{T}[x] \ne \bot$. 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 $\mathsf{MemVerify}(\mathsf{pp}, \mathsf{acc}, x, w_x) = 1$, $\mathcal{B}$ 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 $\mathsf{MemVerify}(\mathsf{pp}, x^*, w_{x^*}, \mathsf{acc}) \neq 1$ or $\mathsf{NonMemVerify}(\mathsf{pp}, x^*, \bar{w}_{x^*}, \mathsf{acc}) \neq 1$, abort.
   (b) Parse w_{x^*} as $(\mathtt{w}, \mathsf{s})$, $\bar{w}_{x^*}$ as $(\mathsf{a}, \mathtt{B}, \bar{\mathsf{s}})$.
   (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 $\gcd(1 - \mathsf{a}\theta, \bar{\mathtt{x}}) = 1$. Furthermore, given that $\bar{\mathtt{x}}$ is odd, $\gcd(2(1 - \mathsf{a}\theta), \bar{\mathtt{x}}) = 1$. Since $(\mathsf{acc})^{\mathsf{a}} \mathtt{B}^{\bar{\mathtt{x}}} \equiv u^{\mathsf{a}\theta} \mathtt{B}^{\bar{\mathtt{x}}} \equiv u \equiv v^2 \bmod n$, we have $\mathtt{B}^{\bar{\mathtt{x}}} \equiv v^{2(1-\mathsf{a}\theta)} \bmod n$. By applying Lemma 1 with respect to $(\mathtt{B}, \bar{\mathtt{x}}, v, 2(1 - \mathsf{a}\theta))$, 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 $w_y = (\mathtt{w}, \mathsf{s})$ 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 $\delta \leftarrow \prod_{i=1}^{|\mathsf{s}|} \mathsf{s}[i]$ 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 $\nu = q_H^2 \left(\frac{1}{2^{\ell^{3/4}}} + \left(\frac{\sqrt[4]{\ell}}{4} \log \ell \right)^{-\sqrt[4]{\ell}} \right) \left(1 - \frac{1}{2^{\lambda-2}}\right)$.

□

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, $\mathsf{MemVerify}$ 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. 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.

```
– Add(pp, sk, acc, $x$):
    1. Parse 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 acc , $w_x$, and upmsg.
```

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

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 $\mathsf{MemVerify}(\mathsf{pp}, \mathsf{acc}, x^*, w_{x^*}) = 1$, 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 $\mathcal{B}$, using $\mathcal{A}$, 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 $\mathsf{T} : \{0,1\}^* \to \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ 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 $\mathsf{MemVerify}(\mathsf{pp}, \mathsf{acc}, x, w_x) = 1$, $\mathcal{B}$ 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 $\mathsf{MemVerify}(\mathsf{pp}, x^*, w_{x^*}, \mathsf{acc}) \neq 1$, $\mathcal{B}$ 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 $\nu = q_H^2 \left(\frac{1}{2^{\ell^{3/4}}} + \left(\frac{\sqrt[4]{\ell}}{4} \log \ell \right)^{-\sqrt[4]{\ell}} \right)$. □

λ	$\mathsf{H}_{\mathsf{prime}}$ length	$\mathsf{H}_{\mathsf{prime}}$ time (s)	$\mathsf{H}_{\mathsf{Odds}}$ length	$\mathsf{H}_{\mathsf{Odds}}$ time(s)
256	264	0.01718	2^{11}	0.00010
512	521	0.23695	2^{13}	0.00028
1024	1034	1.1978	2^{14}	0.00060

Table 1: $\mathsf{H}_{\mathsf{prime}}$ versus $\mathsf{H}_{\mathsf{Odds}}$. $\mathsf{H}_{\mathsf{prime}}$ length is the binary output length of $\mathsf{H}_{\mathsf{prime}}$ and is set such that we have $O(\lambda/2)$ security. $\mathsf{H}_{\mathsf{Odds}}$ length is the binary output length of $\mathsf{H}_{\mathsf{Odds}}$ and is set such that its 3/4-root is greater than or equal to $\mathsf{H}_{\mathsf{prime}}$ output length. The times listed are averages of 5 trials.

λ	$\mathsf{H}_{\mathsf{prime}}$ length	$\mathsf{H}_{\mathsf{prime}} + \mathbb{Z}_n^*$ time (s)	$\mathsf{H}_{\mathsf{Odds}}$ length	$\mathsf{H}_{\mathsf{Odds}} + \mathbb{Z}_n^*$ time(s)
256	264	0.01833	2^{11}	0.00781
512	521	0.23912	2^{13}	0.03048
1024	1034	1.20190	2^{14}	0.06117

Table 2: $\mathsf{H}_{\mathsf{prime}}$ plus modular exponentiation versus $\mathsf{H}_{\mathsf{Odds}}$ plus modular exponention. bit-lengths are set as in Table 1. The times listed are averages of 5 trials.

6 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.

More specifically, let $\mathsf{H}_{\mathsf{prime}}$ be a hash function that hashes to primes and $\mathsf{H}_{\mathsf{Odds}}$ be a hash function that hashes to large odds integers. Assuming that for a collision resistant hash function whose outputs are λ bits we have $O(\lambda/2)$ security, i.e., it takes $O(2^{\lambda/2})$ time to find a collision, we are interested in comparing the time it takes to execute a $\mathsf{H}_{\mathsf{prime}}$ such that we have $O(\lambda/2)$ security with the time it takes to execute a $\mathsf{H}_{\mathsf{Odds}}$ such that its outputs' largest prime factors have rougly the same size as the outputs of $\mathsf{H}_{\mathsf{prime}}$. From the prime number theorem, to get $O(\lambda/2)$ security from $\mathsf{H}_{\mathsf{prime}}$, we will need to hash to prime in the set $[N]$ where $\log_2(N) \approx \lambda + \log_2(\lambda)$, and so, for $x \in \{0,1\}^*$ we will need $\log_2(P^+(\mathsf{H}_{\mathsf{Odds}}(x))) \geq \lambda + \log_2(\lambda)$. Note that from Lemma 2, if outputs of $\mathsf{H}_{\mathsf{Odds}}$ are ℓ-bit length, then $\log_2(P^+(\mathsf{H}_{\mathsf{Odds}}(x))) > \ell^{3/4}$ with overwhelming probability. In addition, we compare the time it takes to execute $\mathsf{H}_{\mathsf{prime}}$ and accumulate its output with the time it takes to execute $\mathsf{H}_{\mathsf{Odds}}$ and accumulate its output. The results are compiled in Tables 1 and 2.

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. We used Blake2s with a 32-byte digest from the PyCryptodome library version 3.19.0 [pyc] to instantiate $\mathsf{H}_{\mathsf{Odds}}$. $\mathsf{H}_{\mathsf{prime}}$ was instantiated using the construction of Barić and Pfitzmann [BP97], and the underlying collision resistant hash function was instantiated using Blake2s with a 32-byte digest. We used 10-bit inputs for both $\mathsf{H}_{\mathsf{prime}}$ and $\mathsf{H}_{\mathsf{Odds}}$, and all modular exponentiation were performed over an RSA modulus of 4096-bits.

7 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.

7.1 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* $\lambda \in \mathbb{N}$,

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

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* $\lambda \in \mathbb{N}$,

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

Remark 6. 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 [BBF18] 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.

Given a pair of interactive Turing machines M and N, let $\langle \mathsf{M}, \mathsf{N} \rangle(x)$ denote the output of N 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 *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 7. 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

<u>Initialization:</u>

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:** $(v, u, e) \in \mathbb{G}^2 \times \mathbb{Z}$.

<u>Interaction:</u>

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.

$\log_2 N \approx 2\lambda + \log_2 \lambda$. 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 6.

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:

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 $H : \mathbb{G}^2 \times \mathbb{Z} \to \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$. Note that H can be instantiated via a random oracle $H' : \{0,1\}^* \to \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ 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, BBF18, 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.

```
– Setup($1^\lambda$):
    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($u, v, e, \pi$):
    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.

Therefore, the subgroup QR_n or the quotient group $\mathbb{Z}_n^*/\{-1,1\}$ is prefered. 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.

7.2 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:*

- $\mathsf{MemWitAggr}(\mathsf{pp}, \widehat{\mathsf{acc}}_t, \{(x_i, w_{x_i,t})\}_{i=1}^m) \to w_{(x_1,\ldots,x_m),t}$*: 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 $\{x_1, \ldots, x_m\}$.*
- $\mathsf{NonMemWitAggr}(\mathsf{pp}, \widehat{\mathsf{acc}}_t, \{(x_i, \bar{w}_{x_i,t})\}_{i=1}^m) \to \bar{w}_{(x_1,\ldots,x_m),t}$*: 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 $\{x_1, \ldots, x_m\}$.*
- $\mathsf{MemAggrVerify}(\mathsf{pp}, \mathsf{acc}_t, \{x_i\}_{i=1}^m, w_{(x_1,\ldots,x_m),t}) \to 0/1$*: 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.*
- $\mathsf{NonMemAggrVerify}(\mathsf{pp}, \mathsf{acc}_t, \{x_i\}_{i=1}^m, \bar{w}_{(x_1,\ldots,x_m),t}) \to 0/1$*: 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 8. 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 $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$ 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 $\mathsf{MemAggrVerify}$ with respect to acc_{t_1} or $(\{y_1, \ldots, y_{m'}\}, \bar{w}_{(y_1,\ldots,y_{m'}),t_2})$ fails $\mathsf{NonMemAggrVerify}$ 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* $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*,*

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

$\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$ *is defined as in Definition 5, and* acc_t *is the accumulator value managed by* $\mathcal{O}_{\mathsf{Add},\mathsf{Delete}}$*.*

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 e need not be prime. We use a random oracle $H : \{0,1\}^* \to \mathsf{Odds}(2^{\ell-1}, 2^\ell - 1)$ 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 for 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 7.1.

In Fig. 8, we describe two helper algorithms $\mathsf{Mem2Aggr}$ and $\mathsf{NonMem2Aggr}$ that are used to compute the aggregation of two membership and non-membership witnesses, respectively. Both take as input a public parameter pp, an accumulator value acc, two elements x_1, x_2 with their respective membership witnesses w_{x_1}, w_{x_2} in the case of $\mathsf{Mem2Aggr}$ or their respective non-membership witnesses $\bar{w}_{x_1}, \bar{w}_{x_2}$ in the case of $\mathsf{NonMem2Aggr}$, and a value isDone that is either equal to 0 or 1 and is used to determine whether $\mathsf{NI\text{-}SimPoE}$ proofs should be computed and included in the aggregated witness. In Fig. 9, we use $\mathsf{Mem2Aggr}$ in the description of $\mathsf{MemWitAggr}$ and $\mathsf{NonMem2Aggr}$ in the description of $\mathsf{NonMemWitAggr}$ to show how we can aggregate the (non-)membership witnesses of $m > 2$ elements by recursively aggregating the (non-)membership witnesses of two elements.

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

– Disaggregating $w_{x_1,\ldots,x_m}$: From the description of $\mathsf{MemWitAggr}$ in Fig. 9, we have $w_{x_1,\ldots,x_m} = (\mathtt{w}_{x_1,\ldots,x_m}, \mathbf{s}_1, \ldots, \mathbf{s}_m, \pi_{x_1,\ldots,x_m})$. To recover a membership witness for $x_i \in \{x_1, \ldots, x_m\}$, compute

$$\mathtt{w}_{x_i} \leftarrow \mathtt{w}_{x_1,\ldots,x_m}^{\prod_{j=1,j\neq i}^{m}\left(H(x_j)/\prod_{k=1}^{|\mathbf{s}_{x_j}|}\mathbf{s}_{x_j}[k]\right)}$$

and set $w_{x_i} = (\mathtt{w}_{x_i}, \mathbf{s}_{x_i})$.

```
– $\mathsf{Mem2Aggr}(\mathsf{pp}, \mathsf{acc}, x, w_x, y, w_y, \mathsf{isDone})$:
  1. Parse $\mathsf{pp}$ as $(n, u)$, $w_x$ as $(\mathtt{w}_x, \mathsf{s}_x)$, and $w_y$ as $(\mathtt{w}_y, \mathsf{s}_y)$.
  2. Compute $\mathtt{x} \leftarrow H(x)/\prod_{i=1}^{|\mathsf{s}_x|} \mathsf{s}_x[i]$ and $\mathtt{y} \leftarrow H(y)/\prod_{i=1}^{|\mathsf{s}_y|} \mathsf{s}_y[i]$.
  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 $\mathsf{s}'_y \leftarrow \mathsf{s}_y \| (\gcd(\mathtt{x}, \mathtt{y}))$.
  5. If $\mathsf{isDone} = 0$, return $w_{x,y} = (\mathtt{w}_{x,y}, \mathsf{s}_x, \mathsf{s}'_y)$.
  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 $w_{x,y} = (\mathtt{w}_{x,y}, \mathsf{s}_x, \mathsf{s}'_y, \pi_{x,y})$.

– $\mathsf{NonMem2Aggr}(\mathsf{pp}, \mathsf{acc}, x, \bar{w}_x, y, \bar{w}_y, \mathsf{isDone})$:
  1. Parse $\mathsf{pp}$ as $(n, u)$, $\bar{w}_x$ as $(\mathsf{a}_x, \mathtt{B}_x, \mathsf{s}_x)$, and $\bar{w}_y$ as $(\mathsf{a}_y, \mathtt{B}_y, \mathsf{s}_y)$.
  2. Compute $\mathtt{x} \leftarrow H(x)/\prod_{i=1}^{|\mathsf{s}_x|} \mathsf{s}_x[i]$ and $\mathtt{y} \leftarrow H(y)/\prod_{i=1}^{|\mathsf{s}_y|} \mathsf{s}_y[i]$.
  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 $\mathsf{s}'_y \leftarrow \mathsf{s}_y \| (\gcd(\mathtt{x}, \mathtt{y}))$.
  5. Compute $\gamma \leftarrow \mathsf{a}_x b\mathtt{y}' + \mathsf{a}_y a\mathtt{x}'$ and $\mathsf{a}_{x,y} \leftarrow \gamma \bmod \mathtt{x}\mathtt{y}'$.
  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 $\bar{w}_{x,y} = (\mathsf{a}_{x,y}, \mathtt{B}_{x,y}, \mathsf{s}_x, \mathsf{s}_y)$.
  8. Else if $\mathsf{isDone} = 1$:
     (a) Compute $\mathtt{C} \leftarrow \mathsf{acc}^{\mathsf{a}_{x,y}} \bmod n$ and $\mathtt{D} \leftarrow \mathtt{B}_{x,y}^{\mathtt{x}\mathtt{y}'} \bmod n$.
     (b) Compute $\pi_{\mathtt{C},(x,y)} \leftarrow \mathsf{NI\text{-}SimPoE.Prove}(\mathsf{acc}, \mathtt{C}, \mathsf{a}_{x,y})$.
     (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 $\bar{w}_{x,y} = (\mathsf{a}_{x,y}, \mathtt{B}_{x,y}, \mathsf{s}_x, \mathsf{s}_y, \pi_{\mathtt{C},(x,y)}, \pi_{\mathtt{D},(x,y)}, \mathtt{C}, \mathtt{D})$.
```

Fig. 8: Helper Algorithms for Witness Aggregation.

– Disaggregating $\bar{w}_{x_1,\ldots,x_m}$: From the description of $\mathsf{NonMemWitAggr}$ in Fig. 9, $\bar{w}_{x_1,\ldots,x_m} = (\mathsf{a}_{x_1,\ldots,x_m}, \mathtt{B}_{x_1,\ldots,x_m}, \mathsf{s}_1, \ldots, \mathsf{s}_m, \pi_{\mathtt{C},(x_1,\ldots,x_m)}, \pi_{\mathtt{D},(x_1,\ldots,x_m)}, \mathtt{C}, \mathtt{D})$.
  To recover a non-membership witness for $x_i \in \{x_1, \ldots, x_m\}$, first compute $\mathtt{x}_i \leftarrow H(x_i)/\prod_{i=1}^{|\mathsf{s}_{x_i}|} \mathsf{s}_{x_i}[i]$. Then, compute $\mathsf{a}_{x_i} \leftarrow \mathsf{a}_{x_1,\ldots,x_m} \bmod \mathtt{x}_i$ and

$$\mathtt{B}_{x_i} \leftarrow \mathsf{acc}^{\lfloor \mathsf{a}_{x_1,\ldots,x_m}/\mathtt{x}_i \rfloor} \mathtt{B}_{x_1,\ldots,x_m}^{\prod_{j=1,j\neq i}^{m} \left(H(x_j)/\prod_{k=1}^{|\mathsf{s}_{x_j}|} \mathsf{s}_{x_j}[k] \right)}$$

  Finally, set $\bar{w}_{x_i} = (\mathsf{a}_{x_i}, \mathtt{B}_{x_i}, \mathsf{s}_{x_i})$.

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 $\{x_1, \ldots, x_m\}$, 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 $x_i \in \{x_1, \ldots, x_m\}$ was used during the batch deletion, set $\delta' = 1$. Otherwise, if $w_{x_1,\ldots,x_m}$ was used, compute $\delta' \leftarrow \prod_{i=1}^{m} \prod_{j=1}^{|\mathsf{s}_{x_i}|} \mathsf{s}_{x_i}[j]$.
– 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 $\{x_1, \ldots, x_m\}$ with membership witness $w_{x_1,\ldots,x_m}$, we have $|w_{x_1,\ldots,x_m}| < 4(\lambda + 2) + m\ell$, and for a set $\{y_1, \ldots, y_{m'}\}$ with non-membership witness $\bar{w}_{y_1,\ldots,y_{m'}}$, we have $|\bar{w}_{y_1,\ldots,y_{m'}}| < 10(\lambda + 2) + 2m'\ell$. □

```
– $\mathsf{MemWitAggr}(\mathsf{pp}, \mathsf{acc}, \{(x_1, w_{x_1}), \dots, (x_m, w_{x_m})\})$:
  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:
         $w_{x_1,\dots,x_i} \leftarrow \mathsf{Mem2Aggr}(\mathsf{pp}, \mathsf{acc}, (x_j)_{j=1}^{i-1}, w_{x_1,\dots,x_{i-1}}, x_i, w_i, 0)$.
     (c) Return $\mathsf{Mem2Aggr}(\mathsf{pp}, \mathsf{acc}, (x_j)_{j=1}^{m-1}, w_{x_1,\dots,x_{m-1}}, x_m, w_m, 1)$.

– $\mathsf{NonMemWitAggr}(\mathsf{pp}, \mathsf{acc}, \{(x_1, \bar{w}_{x_1}), \dots, (x_m, \bar{w}_{x_m})\})$:
  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:
         $w_{x_1,\dots,x_i} \leftarrow \mathsf{NonMem2Aggr}(\mathsf{pp}, \mathsf{acc}, (x_j)_{j=1}^{i-1}, \bar{w}_{x_1,\dots,x_{i-1}}, x_i, \bar{w}_i, 0)$.
     (c) Return $\mathsf{NonMem2Aggr}(\mathsf{pp}, \mathsf{acc}, (x_j)_{j=1}^{m-1}, \bar{w}_{x_1,\dots,x_{m-1}}, x_m, \bar{w}_m, 1)$.

– $\mathsf{MemAggrVerify}(\mathsf{pp}, \mathsf{acc}, \{x_i\}_{i=1}^{m}, w_{x_1,\dots,x_m})$:
  1. Parse $\mathsf{pp}$ as $(n, u)$, $w_{x_1,\dots,x_m}$ as $(\mathtt{W}_{x_1,\dots,x_m}, \mathbf{s}_{x_1}, \dots, \mathbf{s}_{x_m}, \pi_{x_1,\dots,x_m})$.
  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 $\mathsf{NI\text{-}SimPoE.Verify}(\mathtt{W}_{x_1,\dots,x_m}, \mathsf{acc}, \prod_{i=1}^{n} \mathtt{x}_i, \pi_{x_1,\dots,x_m})$.

– $\mathsf{NonMemAggrVerify}(\mathsf{pp}, \mathsf{acc}, \{x_i\}_{i=1}^{m}, \bar{w}_{x_1,\dots,x_m})$ :
  1. Parse $\mathsf{pp}$ as $(n, u)$, $\bar{w}_{x_1,\dots,x_m}$ as $(\mathtt{a}_{x_1,\dots,x_m}, \mathtt{B}_{x_1,\dots,x_m}, \mathbf{s}_{x_1}, \dots, \mathbf{s}_{x_m},$
     $\pi_{\mathtt{C},(x_1,\dots,x_m)}, \pi_{\mathtt{D},(x_1,\dots,x_m)}, \mathtt{C}, \mathtt{D})$.
  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 $\mathsf{NI\text{-}SimPoE.Verify}(\mathsf{acc}, \mathtt{C}, \mathtt{a}_{x_1,\dots,x_m}, \pi_{\mathtt{C},(x_1,\dots,x_m)})$
     $\wedge$ $\mathsf{NI\text{-}SimPoE.Verify}(\mathtt{B}_{x_1,\dots,x_m}, \mathtt{D}, \prod_{i=1}^{n} \mathtt{x}_i, \pi_{\mathtt{D},(x_1,\dots,x_m)})$ $\wedge$ $\mathtt{CD} \equiv u \bmod n$.
```

Fig. 9: Witness Aggregation Algorithms.

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 $\pi_{\mathtt{C},\mathcal{Y}}$ and $\pi_{\mathtt{D},\mathcal{Y}}$ 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. □

Acknowledgments

This work was supported by NSF grants 2312241, 2154170, and 2247305.

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.

BBF18. Dan Boneh, Benedikt Bünz, and Ben Fisch. A survey of two verifiable delay functions. Cryptology ePrint Archive, Paper 2018/712, 2018. `https://eprint.iacr.org/2018/712`.

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.

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.

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.

BKR23. Foteini Baldimtsi, Ioanna Karantaidou, and Srinivasan Raghuraman. Oblivious accumulators. Cryptology ePrint Archive, Paper 2023/1001, 2023. `https://eprint.iacr.org/2023/1001`.

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.

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.

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.

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.

pyc. Pycryptodome.

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.