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 \( S \) is represented by a single value \( \text{acc} \) (called the accumulator value); in order to demonstrate that \( x \in S \), one additionally needs a witness \( w_x \), also of a small, fixed size. \( \text{acc} \) can be efficiently updated as values are added to and deleted from \( S \).

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 \( n = p \ell \) \(^1\) an initial value, \( \text{acc}_0 \leftarrow \mathbb{Z}_n^* \) that corresponds to the empty set is also picked. The value \( \text{acc}_S = \text{acc}_0 \prod_{x \in S} x \) represents a set \( S \); for now, let us think of elements of \( S \) as positive integers. We say that \( \text{acc}_S \) is the accumulator for \( S \). The witness \( w_x \) that \( x \in S \) is the value \( w_x = \text{acc}_x \prod_{y \in S \setminus \{x\}} y^x \); to verify that \( x \in S \) using this witness, check that \( (w_x)^x = \text{acc}_S \). To add \( y \) to the accumulator, Benaloh and de Mare let the value \( \text{acc}_{S \cup \{y\}} = \text{acc}_S \) 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 \( y \notin S \). For example, for a composite integer \( x \in S \), \( x = x_1 x_2 \) for \( x_1 > x_2 > 1 \), \((w_x)^{x_2} \) will pass as the witness for \( x_2 \). A natural fix would be to parameterize by \( \ell \) and require that \( S \subseteq \{2^\ell, \ldots, 2^{\ell+1} - 1\} \). This would rule out the possibility that both \( x = x_1 x_2 \) and \( x_1 < x \) can be in \( S \). 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 \( x = x_1 x_2 \) and \( y = y_1 y_2 \) where \( x_1, x_2, y_1, y_2 \) are all distinct and relatively prime to each other, and \( 2^\ell < x_1 y_1 < 2^{\ell+1} \). For \( x = x_1 x_2 y_1 \), we can compute \( w_y \) such that \( w_y = \text{acc}_S \) 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 \( w = w_1 w_2 \). Note that \( w^{x_1 y_1} = (w_1^x w_2^y)^{x_1 y_1} = w_x^{x_1} w_y^{x_1 y_1} = \text{acc}_S^{x_1} \text{acc}_S^{x_1 y_1} = \text{acc}_S^{x_1 + y_1} = \text{acc}_S \). Thus \( w_z = w_x y_2 \) will pass as the witness for \( z = x_1 y_1 \): 

\( w_z = (w^{x_1} y_2)^{x_1 y_1} = w_1^{x_1} w_y^{x_1 y_1} = \text{acc}_S \).
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.

Baric 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 \( \text{acc} \), 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 \( \mathcal{S} \subseteq \text{PRIMES} \). 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 \( \text{PRIMES} \). 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, Hallevi 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 \( \text{acc} \), 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 Reyniz 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 \( \text{acc}_0 = Z^*_n \). The accumulator value corresponding to the set \( \mathcal{S} \) is \( \text{acc}_\mathcal{S} = \prod_{x \in \mathcal{S}} H(x) \), 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 of the set \( \mathcal{S} \) such that \( \mathcal{S}' \subseteq \mathcal{S} \) is of size \( |w_{\mathcal{S}'}| < \sum_{z \in \mathcal{S}'} |w_z| \); 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 \( |\bar{w}_{\mathcal{S}'}| < \sum_{z \in \mathcal{S}'} |\bar{w}_z| \). 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 \( \text{acc}_t \) 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
RSA-Based Dynamic Accumulator without Hashing into Primes

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 \( \text{acc} \). 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 \( \text{acc} \) 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 \( \text{acc} \): no communication-intensive steps such as table lookups are needed for verification purposes. The fact that \( \text{acc} \) 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 \( \text{acc} \), the client in our construction needs to also verify that \( w_x^{H(x)} = \text{acc} \). 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 \( \text{acc} \) without needing to update it to a different value \( \text{acc}_{t+1} \) (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 \( \text{acc}_t \) to \( \text{acc}_{t+1} \), 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 \( \mathbb{Z}_n^\ast \). 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 \( S \) consists of a polynomial number (in the security parameter \( \ell \)) of odd integers drawn uniformly at random from the set \( \{2^\ell - 1, 2^\ell - 1\} \) (i.e. they are random odd \( \ell \)-bit integers whose most significant bit is 1; from now on, we will denote this set \( \text{Odds}(2^\ell - 1, 2^\ell - 1) \)), the probability that for some \( x \in S, x \mid \prod_{i \in S, x \neq x_i} x_i \) (i.e., \( x \) divides the product of the rest of the elements of \( S \)) is negligible in \( \ell \). More precisely, if \( x \)
is chosen uniformly at random from $\text{Odds}(2^{\ell_1} - 1)$, then with overwhelming probability, its largest prime factor's bit length is at least $k = \Omega(\ell)$ for a constant $c$; as we will see in Section 2.1, $c = 1/4$ is possible, and it translates into overwhelming probability $1 - 2^{-\ell_1}$ 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: $\text{acc}_S = \text{acc}_\emptyset \prod_{x \in S} H(x)$, where $H : \{0, 1\}^* \mapsto \text{Odds}(2^{\ell_1} - 1)$ 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 $y \notin S$, then $d = \gcd(H(y), \prod_{x \in S} H(x)) < H(y)$, by our observation. Since the witness verifies, $w_y^{H(y)/d} = \text{acc}_\emptyset \prod_{x \in S} H(x)/d$; at the same time, $\gcd(H(y)/d, \prod_{x \in S} H(x)/d) = 1$. Thus, we can use Shamir’s trick (Lemma 1) to efficiently compute $u$ such that $u^{H(y)/d} = \text{acc}_\emptyset$, which breaks the flexible RSA problem where the challenge is $(n, \text{acc}_\emptyset)$.

It is easy to see that dynamic additions to this accumulator are possible: just as in the original Benaloh and de Mare construction, for $\text{acc}_S = \text{acc}_\emptyset \prod_{x \in S} H(x)$, $\text{acc}_{S \cup \{y\}} = \text{acc}_{\emptyset}^{H(y)} \prod_{x \in S} H(x)$, the value $w_x' = w_x^{H(y)}$ is the witness that $x \in S \cup \{y\}$ if $w_x$ is the witness for $x \in S$, since $(w_x')^{H(x)} = w_x^{H(x)H(y)} = \text{acc}_S^{H(y)} = \text{acc}_{S \cup \{y\}}$. 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 $\text{Odds}(2^{\ell_1} - 1)$.

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, $w_x = (u, s)$ such that $s$ is a multiset/tuple of small factors of $H(x)$ such that $w^* = \text{acc}_S$, where $x = H(x)/\prod_{s \in S} s$. By “small,” we mean that each $s \in S$ 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 $w_x = (u, s)$ is a valid witness for $x$, then so is $w_x' = (u^*, s \cup \{s\})$. Since $\gcd(H(x)/s, H(y)/s) = 1$, Shamir’s trick works. For details, see Section 3 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 $\text{acc}$. 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 Wesołowski [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 \( \text{neg}(\cdot) \) to denote a negligible function. We denote the security parameter by \( \lambda \). For \( n \in \mathbb{N} \), we use \([n]\) to denote the set \( \{1, \ldots, n\} \), and \( QR_n \) to denote the group of quadratic residues modulo \( n \). For a finite set \( S \), we use \#\( S \) to denote its cardinality, \( U(S) \) to denote the uniform distribution over \( S \), and \( a \leftrightarrow S \) to denote that \( a \) is sample uniformly at random from \( S \). Let \( \text{Odds}(a, b) \) denote that \( a \leq b : n \equiv 1 \mod 2 \). For two functions \( h, g : \mathbb{R} \to \mathbb{R} \), we use \( h(x) \sim g(x) \) to denote that \( \text{lim}_{x \to \infty} h(x)/g(x) = 1 \). Sometimes, we use bold character, \( \mathbf{x} \), to denote a tuple, and for two tuples \( \mathbf{x} = (x_1, \ldots, x_n) \) and \( \mathbf{y} = (y_1, \ldots, y_m) \), we use \( \mathbf{x} \parallel \mathbf{y} \) to denote their concatenation, i.e., \( \mathbf{x} \parallel \mathbf{y} = (x_1, \ldots, x_n, y_1, \ldots, y_m) \). We say that \( x \in \mathbf{x} \) if there exists \( i \in [|\mathbf{x}|] \) such that \( x[i] = x \).

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 \( \text{poly}(\lambda) \)-bit primes, and \( u \in \mathbb{Z}_n^* \),

\[
\Pr [v^e \equiv u \mod n \land e > 1 | (v, e) \leftarrow \mathcal{A}(1^\lambda, u, n)] \leq \text{neg}(\lambda)
\]

Remark 1. Barić and Pfitzmann \([BP97]\) initially proposed a definition of the strong RSA assumption where \( p \) and \( q \) are \( \text{poly}(\lambda) \)-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 \( v^x \equiv u^y \mod n \) and \( \gcd(x, y) = 1 \), there exists \( w \in \mathbb{Z}_n^* \) such that \( w^x \equiv u \mod n \).

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 \( w^x \equiv u^{\alpha x} v^{\beta x} \equiv u^x w^y \equiv u \mod n \).

2.1 Number Theoretic Functions

Dickman-\( \rho \) function. Let \( \rho : \mathbb{R}_{\geq 0} \to \mathbb{R} \) be the continuous solution to the differential equation \( u \rho'(u) + \rho(u) = 0 \) for \( u > 1 \) subjected to the initial condition \( \rho(u) = 1 \) for \( 0 \leq u \leq 1 \). 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 \( s_i = a + (i - 1)d \)) define over an interval \([x]\), where \( x \in \mathbb{N} \). To this end, let us consider the function

\[
\Psi_{a,q}(x, y) \overset{\text{def}}{=} \# \{ n \in [x] : (P^+(n) \leq y) \land (n \equiv a \mod q) \}
\]

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 \( 1 \leq c \leq \sqrt{\ell} \) and \( a \leftarrow \text{Odds}(2^{\ell-1}, 2^\ell - 1) \),

\[
\Pr \left[ P^+(a) \leq 2^c \sqrt{\ell} \right] \approx \left( \frac{\sqrt{\ell} \log \left( \frac{\sqrt{\ell}}{c} \right)}{c} \right)^{-\sqrt{\ell}/c}
\]
Odds optional. We use \( t \) 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.

**Proof.** Let \( 1 \leq c \leq \sqrt{\ell} \) and suppose \( a \leftarrow \text{Odds}(2^{\ell-1}, 2^{\ell} - 1) \). Let \( n \) be the number of integers in \( \text{Odds}(2^{\ell-1}, 2^{\ell} - 1) \) whose largest prime factor is less or equal to \( 2^{c\sqrt{\ell}} \). We have

\[
\eta = \psi_{1,2}(2^{\ell-1}, 2^{c\sqrt{\ell}}) - \psi_{1,2}(2^{\ell-1}, 2^{c\sqrt{\ell}})
\]

\[
= \frac{1}{2} \left( (2^{\ell} - 1) \left( \frac{\sqrt{\ell}}{c} \log \frac{\sqrt{\ell}}{c} \right)^{-\ell/c} \right)
\]

\[
\approx 2^{\ell-2} \left( \frac{\sqrt{\ell}}{c} \log \frac{\sqrt{\ell}}{c} \right)^{-\ell/c} \quad (\text{since for large } \ell, \frac{2^\ell - 1}{2^\ell} \approx 1 \text{ and } \frac{\ell - 1}{\ell} \approx 1)
\]

\[
\leq 2^{\ell-2} \left( \frac{\sqrt{\ell}}{4} \log \ell \right)^{-\frac{\sqrt{\ell}}{4}} \quad (\text{setting } c = \sqrt{\ell})
\]

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

**Corollary 1.** Given a sufficiently large \( \ell \in \mathbb{N} \), for every constant \( 1 \leq c \leq \sqrt{\ell} \), \( m \in \mathbb{N} \), and \( a_1, a_2, \ldots, a_m \sim U(\text{Odds}(2^{\ell-1}, 2^{\ell} - 1)) \), 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,

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

**Proof.** Let \( \ell \) be a large integer, and suppose \( 1 \leq c \leq \sqrt{\ell} \), \( m \in \mathbb{N} \) and \( a_1, a_2, \ldots, a_m \sim U(\text{Odds}(2^{\ell-1}, 2^{\ell} - 1)) \). 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

\[
\Pr[E_i] \leq \sum_{j \in [m] \setminus \{i\}} \Pr[P^+(a_i) \text{ divides } a_j]
\]

\[
\leq \sum_{j \in [m] \setminus \{i\}} \Pr[P^+(a_i) \text{ divides } a_j \mid P^+(a_i) > 2^{c\sqrt{\ell}}] + \Pr[P^+(a_i) \leq 2^{c\sqrt{\ell}}]
\]

\[
\overset{(1)}{\leq} (m-1) \left( \frac{1}{2^{c\sqrt{\ell}}} + \left( \frac{\sqrt{\ell}}{4} \log \ell \right)^{-\frac{\sqrt{\ell}}{4}} \right)
\]

\[
\leq (m-1) \left( \frac{1}{2^{c\sqrt{\ell}}} + \left( \frac{\sqrt{\ell}}{4} \log \ell \right)^{-\frac{\sqrt{\ell}}{4}} \right)
\]

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

\[
\Pr[E] \leq \sum_{i=1}^m \Pr[E_i] \leq m^2 \left( \frac{1}{2^{c\sqrt{\ell}}} + \left( \frac{\sqrt{\ell}}{4} \log \ell \right)^{-\frac{\sqrt{\ell}}{4}} \right)
\]

\[
\square
\]

### 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 $M$ is a tuple of polynomial time algorithms $\texttt{UAcc} = (\texttt{Gen}, \texttt{Add}, \texttt{Delete}, \texttt{NonMemWitCreate}, \texttt{MemWitUp}, \texttt{NonMemWitUp}, \texttt{MemVerify}, \texttt{NonMemVerify})$ with the following properties:

- $\texttt{Gen}(\lambda, \text{aux}) \rightarrow (pp, sk, acc_0)$: This probabilistic algorithm takes as input the security parameter $\lambda$ and auxiliary information aux. It outputs the public parameter pp, the (optional) secret parameter sk, and an initial accumulator $acc_0$.

- $\texttt{Add}(pp, sk, acc_i, x) \rightarrow (acc_{i+1}, w_{x,t+1}, \text{upmsg}_{i+1})$: This (probabilistic) algorithm takes as input the parameters pp, sk, the accumulator $acc_i$, and an element $x \in M$. It adds $x$ to $acc_i$, producing a new accumulator $acc_{i+1}$, a membership witness $w_{x,t+1}$ for $x$, and update information $\text{upmsg}_{i+1}$ that can be used to update the membership witnesses of other elements in the accumulator.

- $\texttt{Delete}(pp, sk, acc_i, x, \bar{w}_{x,t}) \rightarrow (acc_{i+1}, \text{upmsg}_{i+1})$: This (probabilistic) algorithm takes as input the parameters pp, sk, the accumulator $acc_i$, an element $x$ that was previously added to $acc_i$, and an optional membership witness $\bar{w}_{x,t}$ for $x$ with respect to $acc_i$. It deletes $x$ from $acc_i$, producing a new accumulator $acc_{i+1}$, and update information $\text{upmsg}_{i+1}$ that can be used to update the membership witnesses of other elements in the accumulator.

- $\texttt{NonMemWitCreate}(pp, x, \{\text{upmsg}_i\}_{i=1}^t) \rightarrow \bar{w}_{x,t}$: This (probabilistic) algorithm takes as input the parameter pp, an element $x$, a set of update information $\{\text{upmsg}_i\}_{i=1}^t$. It returns a non-membership witness $\bar{w}_{x,t}$ for $x$.

- $\texttt{MemWitUp}(pp, x, w_{x,t}, \text{upmsg}_{i+1}) \rightarrow w_{x,t+1}$: This probabilistic algorithm takes as input the parameter pp, an element $x$, a membership witness $w_{x,t}$ for $x$, and update information $\text{upmsg}_{i+1}$. It returns an updated membership witness $w_{x,t+1}$ for $x$.

- $\texttt{NonMemWitUp}(pp, x, \bar{w}_{x,t}, \text{upmsg}_{i+1}) \rightarrow \bar{w}_{x,t+1}$: 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 $\text{upmsg}_{i+1}$. It returns an updated non-membership witness $\bar{w}_{x,t+1}$ for $x$.

- $\texttt{MemVerify}(pp, acc_i, x, w_{x,t}) \rightarrow 0/1$: This deterministic algorithm takes as input the public parameter pp, the accumulator $acc_i$, an element $x$, and a membership witness $w_{x,t}$ for $x$. It returns 1 if $w_{x,t}$ is a witness for $x$'s membership in the set represented by $acc_i$, or $0$ otherwise.

- $\texttt{NonMemVerify}(pp, acc_i, x, \bar{w}_{x,t}) \rightarrow 0/1$: This deterministic algorithm takes as input the public parameter pp, the accumulator $acc_i$, an element $x$, and a non-membership witness $\bar{w}_{x,t}$ for $x$. It returns 1 if $\bar{w}_{x,t}$ is a witness for $x$'s non-membership in the set represented by $acc_i$, 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 $A$ that attacks the correctness of the system is to interact with the accumulator (with access to an oracle $O_{\text{Add,Delete}}$) 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, $A$ 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 $O_{\text{Add,Delete}}$ is in charge of executing $\texttt{add}$ and $\texttt{del}$ queries. It has access to the public and secret parameters $(pk, sk)$ of the accumulator and is initialized with a discrete time counter $t$ and tuples $v$ that stores elements added to the accumulator, $\texttt{mwit}$ that stores membership witnesses of elements in $v$, and $\text{upmsgs}$ that stores all update information produced after the execution of $\texttt{add}$ and $\texttt{del}$ queries. After each $\texttt{add}$ or $\texttt{del}$ query, $O_{\text{Add,Delete}}$ updates the all out-of-date membership witnesses stored in $\texttt{mwit}$, 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 $\texttt{UAcc}$ for a domain $M$ is correct if for any $x, y \in M$ such that $x$ was added in the accumulator and $y$ was not, up-to-date and well-formed membership witness $w_{x,t}$ and non-membership witness $\bar{w}_{y,t}$ pass membership
and non-membership verification, respectively, with overwhelming probability. More specifically, for all \( \lambda \in \mathbb{N} \), all ppt adversary \( A \),
\[
\Pr \left[ \text{CorrectGame}^A(1^\lambda) = 1 \right] \leq \text{negl}(\lambda)
\]
where CorrectGame is defined in Fig. 7.

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

**Remark 2.** Although we require witnesses to have size \( \text{poly}(\lambda, |x|) \), which is the case of RSA-based schemes such as [BP97, CL02, LLX07] and Bilinear pairing-based schemes such as [Ngu05].
it should be noted that Merkle tree-based schemes such as [BLL02,CHK012,Ry16] support witnesses with size $\text{poly}(\lambda, \log |S|)$, where $S$ is the set of elements that have been accumulated.

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

$$
\Pr \left[ \begin{array}{ll}
\text{pp, sk, acc}_0 & \text{Gen}(1^\lambda, \text{aux}) ;
\end{array} \right.
\left. 
x^*, w_{x^*}, t, \bar{w}_{x^*}, \bar{t} & \leftarrow A^{O_{\text{Add}\_\text{Delete}}}(1^\lambda, \text{aux}, \text{pp, acc}_0) ;
\text{MemVerify}(pp, acc_t, x^*, w_{x^*}, t) = 1 \\
\land \text{NonMemVerify}(pp, acc_t, x^*, \bar{w}_{x^*}, \bar{t}) = 1
\right] \leq \text{negl}(\lambda)
$$

where $acc_t$ is output by $O_{\text{Add}\_\text{Delete}}$. For this definition, $O_{\text{Add}\_\text{Delete}}$ is defined as in Fig. 1 with the exception that the tuple $v$ that stores elements added to the accumulator is replaced by a set $S$, 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 $\text{PA}_\text{Add}_\text{Delete}_\text{MemWitUp}_\text{MemVerify} = (\text{Gen}, \text{Add}, \text{Delete}, \text{MemWitUp}, \text{MemVerify})$ whose properties are as elaborated in Definition 2.

The definitions of correctness and compactness for a positive dynamic accumulator $\text{PA}_\text{Add}_\text{Delete}$ 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 $\text{PA}_\text{Add}_\text{Delete}_\text{MemWitUp}_\text{MemVerify}$ for a domain $\mathcal{M}$ is secure if for all ppt adversary $A$ with oracle access to $O_{\text{Add}\_\text{Delete}}$, there exists a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$
\Pr \left[ \begin{array}{ll}
\text{pp, sk, acc}_0 & \text{Gen}(1^\lambda, \text{aux}) ;
\end{array} \right.
\left. 
x^*, w_{x^*} \tau & \leftarrow A^{O_{\text{Add}\_\text{Delete}}}(1^\lambda, \text{aux}, \text{pp, acc}_0) ;
\text{MemVerify}(pp, acc_t, x^*, w_{x^*}, t) = 1 \\
x^* \not\in S \land \text{MemVerify}(pp, acc_t, x^*, \bar{w}_{x^*}, \bar{t}) = 1
\right] \leq \text{negl}(\lambda)
$$

where $acc_t$ is output by $O_{\text{Add}\_\text{Delete}}$. For this definition, $O_{\text{Add}\_\text{Delete}}$ is defined as in Fig. 1 with the exception that the tuple $v$ that stores elements added to the accumulator is replaced by a set $S$, 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 $\text{Gen, Add}$ and $\text{Delete}$. 

In this section, we present our universal dynamic accumulator in the random oracle model. Our
universal dynamic accumulator can be instantiated by using a random oracle \( H \) and 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
witness \( \bar{w} \) 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 the algorithms NonMemWitCreate, MemWitUp, and NonMemWitUp. It can also issue add and del requests to an accumulator manager.

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 \([\text{CL02}, \text{LLX07}]\) with the exception that we work over large odd integers.

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

Remark 5. The random oracle \( H \) can be instantiated by using a random oracle \( \ell' : \{0,1\}^* \rightarrow \{0,1\}^{\ell' - 2} \parallel 1 \) such that for \( x \in \{0,1\}^* \), \( H(x) = \text{int}(H'(x)) \), where \( \text{int}(\cdot) \) 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 \([\text{CL02}, \text{LLX07}]\), 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 \( \text{upmsg}' = (\text{add}, v', \delta', \text{acc}, \text{acc}') \) in conjunction with MemWitUp or NonMemWitUp, where \( v' \) represents the product of \( H \) evaluations of added or deleted elements,
Given that

Lemma 3.

Let

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

\[
\delta = 1, \quad \text{represents the new accumulator's value.}
\]

represents the last accumulator value for which the witness to be updated is valid

Fig. 3: User's algorithms

\[
\delta' = 1, \quad \text{acc represents the last accumulator value for which the witness to be updated is valid and acc'} \text{ represents the new accumulator's value.}
\]

4.1 Correctness and Compactness

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

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

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

\[\square\]

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

**Proof.** This follows from Lemma 3 \( \square \)

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

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

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

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

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

Let \( \text{Bad} \) be the event that NonMemWitCreate fails, \( \mathbf{D} \) the event that \( H(x) \not\in \mathcal{S} \) and \( \theta, \mathbf{E} \) the event that there exists \( j \in [s[i]] \) such that \( s[j] > 2^\ell \), and \( q = \#\mathcal{S} \). We have

\[
\Pr[\text{Bad}] = \Pr[\text{Bad}|\mathbf{D}] \Pr[\mathbf{D}] + \Pr[\text{Bad}|\mathbf{D}] \Pr[\mathbf{D}]
\leq \Pr[\mathbf{D}] + \Pr[\text{Bad}|\mathbf{D}, \mathbf{E}] \Pr[\mathbf{E}|\mathbf{D}] + \Pr[\text{Bad}|\mathbf{D}, \mathbf{E}] \Pr[\mathbf{E}|\mathbf{D}]
\leq (q + 1)^2 \left( \frac{1}{2^{3/4}} + \left( \frac{\sqrt{\ell}}{4} \log \ell \right)^{-2} \right) + \frac{1}{2^{\sqrt{\ell}}}
\]

Inequality (1) follows from the fact that \( \mathbf{E} \) and \( \mathbf{D} \) are independent. Therefore, unless \( \text{Bad} \) happens, \( \bar{w}_x = (a, \mathbf{B}, \mathbf{s}) \) is a valid non-membership witness for \( x \). \( \square \)
Lemma 5. MemWitUp fails to output a correct updated membership witness with probability at most $2^{-\sqrt{T}}$. 

Proof. Let $\text{acc}$ be an accumulator produced by our construction and $\text{acc}'$ its update. Let $x \in \{0,1\}^*$ be an accumulated element, $w_x = (x,s)$ its valid membership witness with respect to $\text{acc}$, and $w'_x = (w',s')$ its membership witness with respect to $\text{acc}'$ generated by MemWitUp. We show that $w'_x$ is valid with probability at least $1 - 2^{-\sqrt{T}}$. Let $y \in \{0,1\}^*$, $x = H(x)/\prod_{i=1}^{\|s\|} s[i]$, and $x' = H(x)/\prod_{i=1}^{\|s'\|} s'[i]$. Without loss of generality, let us consider the following cases:

- Case 1: $\text{acc}'$ was produced by adding $y$ to $\text{acc}$, i.e., $\text{acc}' = \text{acc}H(y)$. After executing MemWitUp, we have $w' = wH(y)$ and $s' = s$, so $x = x'$. Hence, $(w')^x = (s^x)H(y) = \text{acc}H(y) = \text{acc}'$. In addition, since $w_x$ is valid, it follows that all components of $s$ are less than $2^\tau$, and this is also the case for $s'$. 
- Case 2: $\text{acc}'$ was produced by deleting $y$ from $\text{acc}$. Let $\text{upmsg}$ be the update message that was generated after deleting $y$ from $\text{acc}$. Then, $\text{upmsg} = (\text{del}, H(y), \delta, \text{acc}, \text{acc}')$, where $\delta \geq 1$ is a $2^\tau$-smooth integer that divides $H(y)$. By setting $v = H(y)/\delta$, it follows that $\text{acc} = (\text{acc}')^v$. Let $d = \gcd(v,x)$. After executing MemWitUp, we have $w' = (\text{acc}')^v w$, where $a, b \in \mathbb{Z}$ such that $ax + bv = d$, and $s' = s||d$ if $d > 1$, else $s' = s$. Therefore, $x' = x/d$, and

\[
(w')^{x/d} = ((\text{acc}')^v w)^{x/d} = ((\text{acc}')^a w)^{x(v/\delta)(1/d)} \quad \text{(this follows from Lemma 2)}
\]
\[
= ((\text{acc}')^a w)^{x(v/\delta)}((1/\delta)(1/d))
\]
\[
= (\text{acc}'^a \text{acc}^{br}x^{(1/\delta)(1/d)})
\]
\[
= \text{acc}^v = \text{acc}'.
\]

Each component of $s$ is less than or equal to $2^\tau$ since $w_x$ is valid, and if $d > 1$, then components of $s'$ are also less than or equal to $2^\tau$. Otherwise, $s' = s||d$ if $d > 1$, and given that $d$ divides both $H(x)$ and $H(y)$ and $\sqrt{T} \leq \delta \leq \delta^{3/4}$, it follows that $d < 2^\tau$ with probability at least $1 - 2^{-\sqrt{T}}$. Therefore, each component of $s'$ is less than or equal to $2^\tau$ with overwhelming probability.

Notice that $w'_x$ will be incorrect if there exists an index $j \in [\|s'\|]$ such that $s'[j] > 2^\tau$, and this happens with probability at most $2^{-\sqrt{T}}$. 

\[\square\]

Lemma 6. NonMemWitUp fails to output a correct updated non-membership witness with probability at most $\left(\frac{2T}{3} \log l\right)^{-\sqrt{T}} + 2^{-\sqrt{T}}$. 

Proof. Let $\text{acc}$ be an accumulator produced by our construction and $\text{acc}'$ its update. Let $x \in \{0,1\}^*$ be an element that was not added to the accumulator, $w_x = (a,b,s)$ its valid non-membership witness with respect to $\text{acc}$, and $w'_x = (a',b',s')$ its non-membership witness with respect to $\text{acc}'$ generated by NonMemWitUp. We show that $w'_x$ is incorrect with probability at most $\left(\frac{2T}{3} \log l\right)^{-\sqrt{T}} + 2^{-\sqrt{T}}$. Let $y \in \{0,1\}^*$, $x = H(x)/\prod_{i=1}^{\|s\|} s[i]$, and $x' = H(x)/\prod_{i=1}^{\|s'\|} s'[i]$. Without loss of generality, let us consider the following cases:

- Case 1: $\text{acc}'$ resulted from adding $y$ to $\text{acc}$, i.e., $\text{acc}' = \text{acc}H(y)$. From Lemma 2, $P^+(H(x)) > 2^\tau$ with overwhelming probability. Since $w_x$ is valid, it follows that $P^+(x) = P^+(H(x))$, so $x \notin H(y)$ with overwhelming probability. As a result, there exists $r, d \in \mathbb{Z}$ such that $x = dr$, $d \mid H(y)$, $P^+(r) = P^+(x)$ and $\gcd(H(y),r) = 1$. After executing NonMemWitUp, $x' = r$, and $s' = s$ if $d = 1$, else $s' = s||d$. In addition, we have $a' = a a \mod x' = a a - [a a/x']x'$, and $b' = accz: B^d$, where $z_1 = [a a/x'] H(y) + ab$, and $a, b \in \mathbb{Z}$ such that $a H(y) + b x' = 1$. Hence,

\[
(\text{acc}')^{x'} = (\text{acc})^{H(y)} (a a - [a a/x']x') (acc1 a a/x') (H(y) + ab x')^{x'}
\]
\[
= acc^b H(y + b x') B^{dx'}
\]
\[
= acc^{b^2} = u
\]
In case $d = 1$, all components of $s'$ are less than or equal to $2^\tau$ since $\tilde{w}_x$ is valid. Otherwise, $s' = s \parallel (d)$, and given that $d$ divides both $H(x)$ and $H(y)$ and $\sqrt{\tau} \leq \ell \leq \ell^{3/4}$, it follows that $d < 2^\tau$ with overwhelming probability. Therefore, each component of $s'$ is less than or equal to $2^\tau$ with overwhelming probability.

Since in this case $\tilde{w}_x'$ is correct if $P^+(H(x)) > 2^\tau$ and there does not exist $j \in [\lfloor s' \rfloor]$ such that $s'[j] > 2^\tau$, it follows that failure probability of $\text{NonMemWitUp}$ is upper bounded by

$$\left(\frac{3\tau}{4} \log \ell\right)^{-\sqrt{\tau}} + 2^{-\sqrt{\tau}}.$$

- Case 2: $\text{acc}'$ was produced by deleting $y$ from $\text{acc}$. Let $\text{upmsg}$ be the update message that was generated after deleting $y$ from $\text{acc}$. Then, $\text{upmsg} = (\text{del}, H(y), \delta, \text{acc}, \text{acc}')$, where $\delta \geq 1$ is a $2^\tau$-smooth integer that divides $H(y)$, and $\text{acc} = (\text{acc}')^x$, with $v = H(y)/\delta$. After executing $\text{NonMemWitUp}$, we have $s' = s$, so $x' = x$. Also, $a' = av \mod x = av - [av/x]x$, and $B' = (\text{acc}')^{\frac{x}{av}} B$, where $z_2 = [av/x]$. Hence,

$$\begin{align*}
(\text{acc}')^s ((B')^x &= (\text{acc}')^{av - [av/x]x}((\text{acc}')^{[av/x]}B)^x \\
&= (\text{acc}')^{avB^x} = \text{acc}^\ast B^x = u
\end{align*}$$

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

Therefore, $\text{NonMemWitUp}$ fails with probability at most $\left(\frac{3\tau}{4} \log \ell\right)^{-\sqrt{\tau}} + 2^{-\sqrt{\tau}}$. \hfill \square

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

**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 $\text{Gen}$, $\ell = \text{poly}(\lambda)$ be the chosen bit-length of $H$'s outputs, and $\sqrt{\tau} \leq \tau \leq \ell^{3/4}$ be a value chosen such that for any $x \in \{0, 1\}^*$, $P^+(H(x)) > 2^\tau$ with overwhelming probability. From the description of our construction (confer Figs. 1, 2, and 3), $\text{acc} \in \mathbb{Z}_n^*$, where $\log_2 n \approx 2(\lambda + 1)$, so $|\text{acc}| \leq 2(\lambda + 2)$. For any $x \in \{0, 1\}^*$ with membership witness $w_x = (u, s)$, we have $u \in \mathbb{Z}_n^*$ and $s$, which is a tuple whose components are $2^\tau$-smooth integers that divide $H(x)$ and their products also divide $H(x)$. Since we need less than $\ell$ 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 $\tilde{w}_y = (a, B, s')$, we have $a \in \mathbb{Z}_{H(y)}, B \in \mathbb{Z}_n^*$, and $s'$ that is defined as $s$. Hence, $|w_x| < 2(\lambda + 2) + \ell$. \hfill \square

### 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 $A$ be a ppt adversary that, given $(1^\lambda, \bot, \text{pp}, \text{acc}_0)$ as input, outputs $(x^*, w_{x^*}, \tilde{w}_{x^*})$ with non-negligible probability $\varepsilon(\lambda)$ such that $(x^*, w_{x^*}, \tilde{w}_{x^*})$ are both valid with respect to $\text{acc}$, where $\text{acc}$ is the accumulator value generated after $A$'s queries to $\text{CAddDelete}$. Using $A$, we construct a ppt adversary $B$ that breaks the strong RSA assumption as follows:

1. $B$ receives $(1^\lambda, v, n)$ as input from the Strong RSA challenger. Then, it computes $u = v^2 \mod n$ and initialises an empty map $T : \{0, 1\}^* \to \text{Odds}(2^{\ell - 1}, 2^{\ell} - 1)$, a set $S = \emptyset$, and an integer $d = 1$.
2. $B$ sets $\text{pp} = (u, u)$, $\text{acc} = \text{acc}_0 = u$ and sends $(1^\lambda, \text{pp}, \text{acc}_0)$ to $A$.
3. $B$ simulates answers to $A$’s oracle queries as follows:
   - For $H$ queries, when $A$ sends $x \in \{0, 1\}^*$, $B$ returns $T[x]$ if $T[x] \neq \bot$. Else, $B$ samples $r \leftarrow \text{Odds}(2^{\ell - 1}, 2^{\ell} - 1)$, sets $T[x] \leftarrow r$ and returns $r$. 
- For add queries, when $A$ sends $(\text{add}, x)$, $B$ simulates $H$ with $x$ as input, but it does not return the output to $A$. If $T[x] \in S$, $B$ aborts. Otherwise, $B$ updates $S \leftarrow S \cup \{T[x]\}$, defines $s \leftarrow 1$ and $w_x \leftarrow (\text{acc}, s)$ and computes $\text{acc}' \leftarrow \text{acc}[x] \mod n$. Next, $B$ returns $\text{acc}'$, $w_x$ and $\text{upmsg} = (\text{add}, T[x], 1, \text{acc}')$. Finally, $B$ sets $\text{acc} \leftarrow \text{acc}'$.

- For del queries, when $A$ sends $(\text{del}, x, w_x)$, $B$ simulates $H$ with $x$ as input, but it does not return the output to $A$. If $T[x] \notin S$, $B$ aborts. Next, $B$ sets $S \leftarrow S \setminus \{T[x]\}$, and if $w_x \neq \bot$ and $\text{MemVerify}(\text{pp}, \text{acc}, x, w_x) = 1$, $B$ parses $w_x$ as $(v, s)$, computes $\delta \leftarrow \prod_{i=1}^{s} s[i]$, $d \leftarrow d \cdot \delta$, and $\text{acc}' \leftarrow w$. Otherwise, $B$ sets $\delta \leftarrow 1$ and computes $\text{acc}' \leftarrow u^d \prod_{s \in S} s \mod n$.

After, $B$ returns $\text{acc}'$ and $\text{upmsg} = (\text{del}, T[x], \delta, \text{acc}', \text{acc})$. Finally, $B$ sets $\text{acc} \leftarrow \text{acc}'$.

4. Once $A$ outputs $(x^*, w_{x^*}, \bar{w}_{x^*})$, $B$ does the following:

(a) If $\text{MemVerify}(\text{pp}, x^*, w_{x^*}, \text{acc}) \neq 1$ or $\text{NonMemVerify}(\text{pp}, x^*, \bar{w}_{x^*}, \text{acc}) \neq 1$, abort.

(b) Parse $w_{x^*}$ as $(v, s)$, $\bar{w}_{x^*}$ as $(a, B, s)$.

(c) Compute $\theta \leftarrow d \prod_{y \in S} y$, $x = T[x^*] / \prod_{i=1}^{s} s[i]$ and $\bar{x} = T[x^*] / \prod_{i=1}^{s} \bar{s}[i]$.

(d) If $T[x^*] \notin S$, it follows that $\bar{x} \parallel \theta$, so $\gcd(1 - a\theta, \bar{x}) = 1$. Furthermore, given that $x$ is odd, $\gcd(2(1 - a\theta), \bar{x}) = 1$. Since $(\text{acc})^a B^x \equiv u^{a\theta} B^x \equiv u \equiv v^2 \mod n$, we have $B^x \equiv v^2 (1 - a\theta) \mod n$. By applying Lemma 1 with respect to $(B, x, v, 2(1 - a\theta))$, compute and output the $x$-root of $v$.

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

Note that during the execution of the del queries, if $A$ sends a tuple $(\text{del}, y, w_y)$ such that $T[y] \in S$ and $w_y = (v, s)$ is valid, it follows that $w \equiv \text{acc}[y/v] \equiv u^{(T[y]/y)} \prod_{\zeta \in \zeta \setminus \{T[y]\}} \zeta \equiv u^B \prod_{\zeta \in S \setminus \{T[y]\}} \zeta \mod n$, with $\delta \leftarrow \prod_{i=1}^{s} \bar{s}[i]$ and $\gamma \leftarrow T[y]/\delta$.

In addition, as long as $B$ properly simulates $O_{\text{Add/Delet}}$, if $A$ issues a forgery $(x^*, w_{x^*}, \bar{w}_{x^*})$ such that $T[x^*] \in S$, then $B$ will solve the strong RSA challenge. Otherwise, as long $T[x^*] \parallel \theta$ or $\gcd(2B, x)$ does not exist, $B$ will still be able to solve the strong RSA challenge. $B$ will fail to properly simulate $O_{\text{Add/Delet}}$ 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 $A$’s queries to the random oracle $H$. Therefore, $B$ succeeds with probability

$$\Pr[B \text{ wins}] \geq \varepsilon(\lambda) \left(1 - \frac{q_H}{2^{c-1}}\right) (1 - \nu)$$

where $q_H$ represents the number of unique $H$’s queries performed by $A$ and

$$\nu = q_H^2 \left(\frac{1}{\log \ell} + \left(\frac{7}{2\log \ell} \log \ell\right) - \frac{7}{2}\right) \left(1 - \frac{1}{\log \ell}\right).$$

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, MemVerify are exactly the same algorithms presented in Section 4 and for MemWithUp, only step 2, regarding the update of membership witnesses after Add operations, is removed. However, $pp = n$ 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.
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 $A$ be a ppt adversary that, given $(1^λ, x, n)$ as input, after a total of $q_H$ unique queries to $H$ and a total of $q_{del}$ deletion queries to $O_{Add,Delete}$, outputs $(x^*, w_x)$ with non-negligible probability $ε(λ)$ such that $x^* \notin S$ and $\text{MemVerify}(pp, acc, x^*, w_x) = 1$, where $S$ is the set and $acc$ is the accumulator generated after $A$’s queries to $O_{Add,Delete}$. We build a ppt adversary $B$, using $A$, that breaks the strong RSA assumption as follows:

1. $B$ receives $(1^λ, x, n)$ as input from the Strong RSA Challenger. Then, it computes $u = x^* \mod n$ and initializes an empty map $T : \{0, 1\}^* \mapsto \text{Odds}(2^{λ−1}, 2^{λ−1})$ and a set $S = ∅$.

2. $B$ samples $α_1, \ldots, α_{q_H} \leftarrow \text{Odds}(2^{λ−1}, 2^{λ−1})$ and a set $S = ∅$. Let $c = 1$, and sets $\text{acc} = \text{acc}_0 = u^θ \mod n$. Then, $B$ sends $pp \leftarrow n$, and initializes $k \leftarrow 1$.

3. $B$ computes $θ = α_1^{q_{del}} \prod_{i \in [q_H], i \neq 1} α_i^{q_{del}}$ and $\text{acc} = \text{acc}_0 = u^θ \mod n$. Then, $B$ sets $pp \leftarrow n$, and initializes $k \leftarrow 1$.

4. $B$ sends $(1^λ, pp, acc_0)$ to $A$ and simulates answers to $A$’s oracle queries as follows:

- For $H$ queries, when $A$ sends $x \in \{0, 1\}^∗$, $B$ returns $T[x]$ if $T[x] \neq ⊥$. Otherwise, $B$ sets $T[x] \leftarrow α_k$, $k \leftarrow k + 1$, and returns $T[x]$.

- For add queries, when $A$ sends $(add, x)$, $B$ simulates $H$ with $x$ as input, but it does not return the output to $A$. If $T[x] ∈ S$, $B$ aborts. Otherwise, $B$ computes $v = u^θ/T[x] \mod n$, initializes $s \leftarrow 1$, and sets $w_x \leftarrow (v, s)$. Finally, $B$ sets $S = S ∪ \{T[x]\}$, and returns $\text{acc}$, $w_x$, and $\text{upmsg} = (add, T[x], 1, ⊥)$.

- For del queries, when $A$ sends $(del, x, w_x)$, $B$ simulates $H$ with $x$ as input, but it does not return the output to $A$. If $T[x] \notin S$ or if $T[x] = α_j$, and $j_2 = 0$, $B$ aborts. Next, $B$ sets $S = S \setminus \{T[x]\}$, and if $w_x \neq ⊥$ and $\text{MemVerify}(pp, acc, x, w_x) = 1$, $B$ parses $w_x$ as $(v, s)$, computes $δ = \prod_{i = 1}^{|s|} s[i]$, $θ = θ/(T[x]/δ)$, and $\text{acc} \leftarrow v$. Otherwise, $B$ sets $δ \leftarrow 1$, computes $θ = θ/T[x]$, $\text{acc} = u^θ \mod n$. In addition, if $T[x] = α_j$, $B$ sets $j_2 = j_2 + 1$. Finally, $B$ returns $\text{acc} \prime$ and $\text{upmsg} = (del, T[x], δ, acc\prime)$, and then sets $\text{acc} \leftarrow \text{acc} \prime$.

5. Once $A$ outputs $(x^*, w_x)$, $B$ proceeds as follows:

(a) If $T[x] = α_j$, or $j_2 = 0$ or $\text{MemVerify}(pp, x^*, w_x, acc) \neq 1$, $B$ aborts.

(b) Parse $w_x$ as $(v, s)$, and compute $x = T[x]^{q_{del}}/\prod_{i = 1}^{|s|} s[i] = α_j/\prod_{i = 1}^{|s|} s[i]$. (c) Using a process similar to the one described in step 4c of the proof of Theorem 3, if $\text{gcd}(2θ, x)^{-1}φ(n)$ exists, compute the $\bar{x}$-root of $v$, where $\bar{x} = x/\gcd(2θ, x)$. Otherwise, use $\text{gcd}(2θ, x)$ to factor $n$.

Note that during the execution of del queries, if $A$ sends a tuple $(del, y, w_y)$ such that $T[y] ∈ S$ and $w_y = (v, s)$ is valid, it follows that $u = acc^{q_{del}/θ} = v^{θ/γ} \mod n$, where $y = T[y]/\prod_{i = 1}^{|s|} s[i]$.

Since $α_1, \ldots, α_{q_H}$ are random and distinct from each other, if $B$ correctly guesses $j_2$, then it will correctly simulate $O_{Add,Delete}$’s answers for add and del queries. In step 4c, if $B$ correctly guessed $j_1$, then as long as $\text{gcd}(2θ, x)^{-1} \mod φ(n)$ exists and $α_j$ does not divide $θ$ or $\text{gcd}(2θ, x)^{-1} \mod φ(n)$ does not exist, $B$ will be able to break the strong RSA assumption with probability

$$\Pr[B \text{ wins}] \geq \frac{ε(λ)}{q_H (q_{del} + 1)} \left(1 − \frac{ν}{2λ−2}\right)$$

where $ν = q_H^2 \left(\frac{1}{2(2^λ+1)} + \frac{λ}{4^λ \log ℓ} - \frac{5}{7}\right)$. □

Fig. 5: Description of the algorithm Add for the positive dynamic accumulator.
RSA-Based Dynamic Accumulator without Hashing into Primes

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>$H_{\text{prime}}$ length</th>
<th>$H_{\text{prime}}$ time (s)</th>
<th>$H_{\text{Odds}}$ length</th>
<th>$H_{\text{Odds}}$ time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>264</td>
<td>0.01718</td>
<td>$2^{11}$</td>
<td>0.00010</td>
</tr>
<tr>
<td>512</td>
<td>521</td>
<td>0.23695</td>
<td>$2^{13}$</td>
<td>0.00028</td>
</tr>
<tr>
<td>1024</td>
<td>1034</td>
<td>1.1978</td>
<td>$2^{14}$</td>
<td>0.00060</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>$H_{\text{prime}}$ length</th>
<th>$H_{\text{prime}} + Z^*_n$ time (s)</th>
<th>$H_{\text{Odds}}$ length</th>
<th>$H_{\text{Odds}} + Z^*_n$ time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>264</td>
<td>0.01833</td>
<td>$2^{11}$</td>
<td>0.00310</td>
</tr>
<tr>
<td>512</td>
<td>521</td>
<td>0.23912</td>
<td>$2^{13}$</td>
<td>0.03048</td>
</tr>
<tr>
<td>1024</td>
<td>1034</td>
<td>1.20190</td>
<td>$2^{14}$</td>
<td>0.06117</td>
</tr>
</tbody>
</table>

Table 2: $H_{\text{prime}}$ plus modular exponentiation versus $H_{\text{Odds}}$ 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 $H_{\text{prime}}$ be a hash function that hash to primes and $H_{\text{Odds}}$ be a hash function that hash to large odds integers. Assuming that for a collision resistant hash function whose outputs are $\lambda$ 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 $H_{\text{prime}}$ such that we have $O(\lambda/2)$ security with the time it takes to execute a $H_{\text{Odds}}$ such that its outputs’ largest prime factors have roughly the same size as the outputs of $H_{\text{prime}}$. From the prime number theorem, to get $O(\lambda/2)$ security from $H_{\text{prime}}$, 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 $\log_2(\mathcal{P}^{+}(H_{\text{Odds}}(x))) \geq \lambda + \log_2(\lambda)$. Note that from Lemma 2 if outputs of $H_{\text{Odds}}$ are $\ell$-bit length, then $\log_2(\mathcal{P}^{+}(H_{\text{Odds}}(x))) > \ell^{3/4}$ with overwhelming probability. In addition, we compare the time it takes to execute $H_{\text{prime}}$ and accumulate its output with the time it takes to execute $H_{\text{Odds}}$ 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 Subsystem 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 $H_{\text{Odds}}$. $H_{\text{prime}}$ was instantiated using the construction of Baric 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 $H_{\text{prime}}$ and $H_{\text{Odds}}$, 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 $1^\lambda$ and outputs an...
abelian group $G$ whose order is at most $2^{poly(\lambda)}$ and a trapdoor $sk$ that can be used to efficiently compute the exact order of $G$. In addition, for all ppt adversary $A$, there exists a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\Pr \left[ g^c = 1_G \wedge c \neq 0 \mid (G, sk) \leftarrow \text{GGen}(1^\lambda), g \leftarrow \text{G}, c \leftarrow A(1^\lambda, G, g) \right] \leq \text{negl}(\lambda)$$

**Definition 9 (Adaptive Root assumption [Wes20]).** A hidden order group sampler $\text{GGen}$ satisfies the adaptive root assumption with respect to a challenge space $C \subset \mathbb{Z}$ if for all ppt adversary $A = (A_1, A_2)$, there exists a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\Pr \left[ v^c = u \neq 1_G \mid (G, sk) \leftarrow \text{GGen}(1^\lambda), (u, \text{state}) \leftarrow A_1(1^\lambda, G), c \leftarrow C, v \leftarrow A_2(u, \text{state}, c) \right] \leq \text{negl}(\lambda)$$

**Remark 6.** Special care must be taken when selecting the challenge space $C$. For instance, if $\#C = \text{poly}(\lambda)$, then the adversary $A$ can have $A_1$ sample $h \leftarrow G$, compute and output $u = h^{\prod_{c \in C} c}$, which will guarantee that $A$ always wins by having $A_2$ compute and output $v = h^{\prod_{c \in C} |c|}$ for any challenge $c \in C$. One might think that picking $C$ such that $\#C = 2^{\text{poly}(\lambda)}$ might be enough to ensure that $A$ wins with at most negligible probability, but Boneh, Bitz, and Fisch [BBF18] noted that if $a \leftrightarrow C$ is $B$-smooth with non-negligible probability and $\#\text{PRIMES} \cap [B] = \text{poly}(\lambda)$, then even though the challenge space is of size $\Theta(2^{\text{poly}(\lambda)})$, $A$ can win with non-negligible probability by having $A_1$ compute and output $u = (h')^{\prod_{c \in \text{PRIMES} \cap [B]} c^2}$, where $h' \in G$ and $k$ is a large integer.

Given a pair of interactive turing machines $M$ and $N$, let $(M, N)$ 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 $(G, sk) \leftarrow \text{GGen}(1^\lambda)$, and consider the language $L_{\text{PoE}, G} = \{(v, u, e) \in G^2 \times \mathbb{Z} : v^e = u\}$. A proof of exponentiation (PoE) for $L_{\text{PoE}, G}$ is an interactive protocol (argument) between a ppt prover $P$ and a ppt verifier $V$ such that on a common input $(v', u', e') \in G^2 \times \mathbb{Z}$, $V$ outputs 1 after its interaction with $P$ if it is convinced that $(v', u', e') \in L_{\text{PoE}, G}$. Otherwise, $V$ outputs 0. In addition, a PoE for $L_{\text{PoE}, G}$ must satisfy the following properties:

- **Completeness:** for all $(v, u, e) \in L_{\text{PoE}, G}$,
  $$\Pr[\langle P, V \rangle(v, u, e) = 1] = 1$$

- **Soundness:** for all $(v, u, e) \notin L_{\text{PoE}, G}$,
  $$\Pr[\langle P, V \rangle(v, u, e) = 1] \leq \text{negl}(\lambda)$$

**Remark 7.** For a PoE to be useful, a verifier $V$ should perform less than $O(\log e)$ group operations for an input $(v, u, e) \in G^2 \times \mathbb{Z}$, especially if $e$ is large, because it is always possible to check if $(v, u, e) \in L_{\text{PoE}, G}$ using $O(\log e)$ group operations via repeated squaring.

Wesolowski [Wes20] constructed a PoE for $L_{\text{PoE}, G}$ where for a statement $(v, u, e) \in G^2 \times \mathbb{Z}$, $e$ is a power of 2. To prove that $(v, u, e) \in L_{\text{PoE}, G}$, $V$ samples a random prime $c \leftarrow \text{PRIMES}(2^k)$, where $\text{PRIMES}(2^k)$ represents the set of $2^{2^k}$ first positive primes, and sends $c$ to $P$. Next, $P$ computes $\pi \leftarrow 2^{\log_{2^k} e}$ and sends it to $V$. Finally, $V$ computes $r \leftarrow e \mod c$ and outputs 1 if $\pi^r v^r = u$. Wesolowski proved that his PoE is sound under the adaptive root assumption with challenge space $C = \text{PRIMES}(2^k)$. 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
Initialization:
1. Run $(G, sk) \leftarrow \text{GGen}(1^\lambda)$. Then, set $\ell = \text{poly}(\lambda)$ and $\sqrt{\ell} \leq \tau \leq \ell^{3/4}$ such that for a $\leftrightarrow \text{Odds}(2^{\ell-1}, 2^\ell - 1)$, $P^*(a) > 2^{\ell}$ with overwhelming probability. Finally, send $(1^\lambda, G, \ell)$ to prover $P$ and verifier $V$.
2. Statement: $(v, u, e) \in G^2 \times \mathbb{Z}$.

Interaction:
1. $V$ samples $c \leftrightarrow \text{Odds}(2^{\ell-1}, 2^\ell - 1)$ and sends it to $P$.
2. $P$ computes $\pi \leftrightarrow \frac{u}{v^c}$ and sends it to $V$.
3. $V$ computes $r \leftrightarrow e \mod c$. Then, it outputs $1$ if $\pi^r v^r = u$. 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 $\text{Odds}(2^{\ell-1}, 2^\ell - 1)$, where $\ell = \text{poly}(\lambda)$ is selected such that for a $\leftrightarrow \text{Odds}(2^{\ell-1}, 2^\ell - 1)$, $P^*(a) > 2^{\ell}$ with overwhelming probability (Lemma 2). Since an integer sampled uniformly at random from $\text{Odds}(2^{\ell-1}, 2^\ell - 1)$ is not smooth with overwhelming probability and $\#\text{Odds}(2^{\ell-1}, 2^\ell - 1) = 2^{2\ell}\ell$, it follows that the adaptive root assumption with the challenge space $C = \text{Odds}(2^{\ell-1}, 2^\ell - 1)$ should hold for a hidden order group $G$ because it will be hard for a ppt adversary to execute the strategies mentioned in Remark 6.

**Theorem 5.** Assume $\text{GGen}$ is a hidden order group sampler, and for $\lambda \in \mathbb{N}$, let $(G, sk) \leftarrow \text{GGen}(1^\lambda)$. Let $\ell = \text{poly}(\lambda)$ and $\sqrt{\ell} \leq \tau \leq \ell^{3/4}$ such that for a $\leftrightarrow \text{Odds}(2^{\ell-1}, 2^\ell - 1)$, $P^*(a) > 2^\ell$ with overwhelming probability $1 - \left(\frac{3\tau^2}{4} \log \ell\right) - \frac{\ell^3}{\ell}$.

Under the adaptive root assumption with challenge space $C = \text{Odds}(2^{\ell-1}, 2^\ell - 1)$, SimPoE (Fig. 6) is sound.

**Proof.** We proceed by contraposition. Let $P^*$ be a ppt prover that, given $(1^\lambda, G, \ell)$ as input, makes a verifier $V$ output $1$ after executing SimPoE on a statement $(u, v, e) \notin \mathcal{L}_{\text{PoE,G}}$ with non-negligible probability $\varepsilon(\lambda)$. We use $P^*$ to build a ppt adversary $B = (B_1, B_2)$ that breaks the adaptive root assumption with challenge space $C = \text{Odds}(2^{\ell-1}, 2^\ell - 1)$ as follows:

1. After receiving the statement $(u, v, e)$, $B_1$ sends $u/v^e$ to the adaptive root challenger, which replies with a challenge $c \leftrightarrow \text{Odds}(2^{\ell-1}, 2^\ell - 1)$. Note that since $(u, v, e) \notin \mathcal{L}_{\text{PoE,G}}$, we have $v^e \neq u$, and so $u/v^e \in G \setminus \{1\}$.
2. Next, $B_1$ sends $c$ to $P^*$ and $(u/v^e, c)$ to $B_2$.
3. After receiving $c$, $P^*$ computes and sends $\pi$ to $B_2$. If $\pi^c v^c \neq u$, $B_2$ aborts. Otherwise, it sends $\pi/v^c$ to the adaptive root challenger.

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

For completeness sake, in Fig. 6 we provide a description of a non-interactive version of SimPoE, called NI-SimPoE, in the random oracle model. It uses a random oracle $H : G^2 \times \mathbb{Z} \rightarrow \text{Odds}(2^{\ell-1}, 2^\ell - 1)$. Note that $H$ can be instantiated via a random oracle $H^* : \{0,1\}^* \rightarrow \text{Odds}(2^{\ell-1}, 2^\ell - 1)$ such that for $(u, v, e) \in G^2 \times \mathbb{Z}$, $H(u, v, e) = H^*(\text{bin}(u, v, e))$, where $\text{bin}(\cdot, \cdot, \cdot)$ is a function that efficiently maps elements of $G^2 \times \mathbb{Z}$ 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 $1^\lambda$ and output $n = p \cdot q$ and $sk = (p - 1)(q - 1)$, where $p$ and $q$ are $O(\lambda)$-bit safe primes. However, Boneh, Bünz, and Fisch [BBF19] noted that over $\mathbb{Z}_n^\lambda$, the soundness of Wesolowski PoE does not hold because with $(v, u, e) \in \mathcal{L}_{\text{PoE,Z}_n^\lambda}$, we can generate a valid interaction for $(v, -u, e)$ by having $P$ reply with $\pi = -1 \cdot v^c/\ell$ after receiving $c$ from $V$. 
Remark 8. Boneh, Bünz, and Fisch [BBF19] proposed a mechanism to aggregate (non-)membership game presented in Definition 3 (confer Fig. 1) to allow a \( \text{ppt} \) batching setting, i.e., elements accumulated are sets. However, their proposed syntax includes syntax. Srinivasan et al. [SKBP22] proposed a definition for trapdoorless accumulators in the 20 Victor Youdom Kemmoe and Anna Lysyanskaya

Therefore, the subgroup \( QR_n \) or the quotient group \( \mathbb{Z}_n^* / \{−1, 1\} \) is preferred. In the case of \( QR_n \), testing for membership is hard and that makes it an unpractical choice in general, but for our application, \( QR_n \) will be enough.

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 \( a \) to denote that the value \( a \) is optional.

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

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

- \( \text{NonMemWitAggr}(pp, \hat{\text{acc}}_i, \{(x_i, \bar{w}_{x_i}, t_i)\}_{i=1}^m) \rightarrow \bar{w}(x_1, \ldots, x_m), t \): This (probabilistic) algorithm takes as input the public parameter \( pp \), an optional accumulator value \( \hat{\text{acc}}_i \), and a set of element and non-membership witness pairs \( \{(x_i, \bar{w}_{x_i}, t_i)\}_{i=1}^m \). It outputs a non-membership witness \( \bar{w}(x_1, \ldots, x_m), t \) that can be used to attest the non-membership of \( \{x_1, \ldots, x_m\} \).

- \( \text{MemAggrVerify}(pp, \text{acc}_i, \{x_i\}_{i=1}^m, w(x_1, \ldots, x_m), t) \rightarrow 0/1 \): This deterministic algorithm takes as input the public parameter \( pp \), an accumulator \( \text{acc}_i \), 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 \( \text{acc}_i \). Otherwise, it returns 0.

- \( \text{NonMemAggrVerify}(pp, \text{acc}_i, \{x_i\}_{i=1}^m, \bar{w}(x_1, \ldots, x_m), t) \rightarrow 0/1 \): This deterministic algorithm takes as input the public parameter \( pp \), an accumulator \( \text{acc}_i \), a set of elements \( \{x_1, \ldots, x_m\} \) 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 \( \text{acc}_i \). 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 \( \text{ppt} \) adversary \( A \) to choose a set of
accumulated elements \( \{x_1, \ldots, x_m\} \) and a point of time \( t_1 \) at which the oracle \( O_{\text{Add.Delete}} \) should aggregate the membership witnesses of those elements. In addition, \( A \) 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. \( A \) wins if \( \{x_1, \ldots, x_m\}, w_{[x_1, \ldots, x_m], t_1} \) fails \( \text{MemAggrVerify} \) with respect to \( acc_t \) or \( \{\{y_1, \ldots, y_m\}, \bar{w}_{[y_1, \ldots, y_m], t_2} \) fails \( \text{NonMemAggrVerify} \) with respect to \( acc_t \). We say that a universal dynamic accumulator with witness aggregation supports is correct if \( A \) 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 \( |w_{[x_1, \ldots, x_m], t}| = |\bar{w}_{(x_1, \ldots, x_m), t}| = \text{poly}(\lambda, |x_1, \ldots, |x_m|) \).

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

\[
\text{Pr}
\begin{bmatrix}
\text{pp, sk, acc}_0 \leftarrow \text{Gen}(1^\lambda, \text{aux}); \\
X, x_{X,t}, Y, y_{Y,t} \leftarrow A_{\text{Add.Delete}}(1^\lambda, \text{aux}, \text{pp}, \text{acc}_0); \\
\text{MemAggrVerify}(\text{pp, acc}_t, X, x_{X,t}) = 1
\end{bmatrix}
\wedge \text{NonMemAggrVerify}(\text{pp, acc}_t, Y, y_{Y,t}) = 1 \land X \cap Y \neq \emptyset \\
\leq \text{negl}(\lambda)
\]

\( O_{\text{Add.Delete}} \) is defined as in Definition 5, and \( acc_t \) is the accumulator value managed by \( O_{\text{Add.Delete}} \).

In Figs. 8 and 9 we present the algorithms needed to enabled witness aggregation for our universal dynamic accumulator construction. They are based on the work of Boneh, Bitz, 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 \( H : \{0, 1\}^* \rightarrow \text{Odds}(2^{\ell - 1}, 2^\ell - 1) \) 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 \( O(ml) \) group operations in \( \mathbb{Z}_n^* \). However, by having users prepare PoE proofs using NI-SimPoE (confer Fig. 7) and include those proofs in the aggregated witnesses, we are able to reduce the number of group operations to \( O(l) \), eliminating dependence on \( m \). In addition, since PoE proofs are performed for elements in \( QR \) (witnesses contain components in \( QR \) and the accumulator value \( acc \) belongs to \( QR \)), we do not suffer from the PoE soundess issue that arises by working over \( \mathbb{Z}_n \) as mentioned in Section 7.1.

In Fig. 8 we describe two helper algorithms \( \text{Mem2Aggr} \) and \( \text{NonMem2Aggr} \) 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 \( \text{Mem2Aggr} \) or their respective non-membership witnesses \( \bar{w}_{x_1}, \bar{w}_{x_2} \) in the case of \( \text{NonMem2Aggr} \), and a value \( \text{isDone} \) that is either equal to 0 or 1 and is used to determine whether NI-SimPoE proofs should be computed and included in the aggregated witness. In Fig. 9 we use \( \text{Mem2Aggr} \) in the description of \( \text{MemWitAggr} \) and \( \text{NonMem2Aggr} \) in the description of \( \text{NonMemWitAggr} \) 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 \( \text{MemWitAggr} \) in Fig. 6 we have \( w_{x_1, \ldots, x_m} = (w_{x_1, \ldots, x_m}, s_1, \ldots, s_m, \pi_{x_1, \ldots, x_m}) \). To recover a membership witness for \( x_i \in \{x_1, \ldots, x_m\} \), compute

\[
\mathcal{W}_{x_i} = \prod_{j=1}^m (H(x_j)/\prod_{k=1}^{s_{x_j}} s_{x_j}[k])
\]

and set \( w_{x_i} = (\mathcal{W}_{x_i}, s_{x_i}) \).
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 \( |\bar{w}_{y_1, \ldots, y_{m'}}| < 10(\lambda + 2 + 2m'\ell) \).
This work was supported by NSF grants 2312241, 2154170, and 2247305.

Assume Theorem 8.

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

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

Proof. This follows from Theorem 4 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 $A$ be a ppt adversary that, on input $(1^\lambda, \bot, pp, acc_0)$, where $(pp, acc_0)$ are output of Gen$(1^\lambda, \bot)$, can output $(X, w_X, Y, w_Y)$ with probability $\varepsilon(\lambda)$ such that $w_X$ and $w_Y$ are valid, $X \cap Y = \emptyset$, and $\varepsilon(\lambda)$ is non-negligible. We use $A$ to either break the strong RSA assumption or the adaptive root assumption as follows:

- If there exists $x \in X \cap Y$ such that after extracting its membership witness $w_x$ from $w_X$ and its non-membership $\bar{w}_x$ from $w_Y$, $w_x$ and $\bar{w}_x$ are valid, then from Theorem 5 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 $\pi_X$ associated to $w_X$ or the PoE proofs $\pi_{C,Y}$ and $\pi_{B,Y}$ associated to $w_Y$ 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


RSA-Based Dynamic Accumulator without Hashing into Primes


pyc. Pycryptodome.


