Client-Auditable Verifiable Registries

Nirvan Tyagi
Cornell University

Ben Fisch
Stanford University

Joseph Bonneau
New York University

Stefano Tessaro
University of Washington

Abstract. Verifiable registries allow clients to securely access a key-value mapping maintained by an untrusted server. Applications include distribution of public keys, routing information or software binaries. Existing proposals for verifiable registries rely on global invariants being audited whenever the registry is updated. Clients typically rely on trusted third-party auditors, as large registries become expensive to audit.

We propose several new protocols for *client-auditable* registries that enable efficient verification of many updates to the registry, removing the need for third-party auditors. Our solutions use incrementally-verifiable computation (IVC) and/or RSA accumulators. Our evaluation shows that our constructions meet practical throughput requirements (60 updates / second), which is $100\times$ faster than naive solutions using IVC. Clients save 100–$10^4\times$ bandwidth and computation costs over prior solutions requiring auditing every update.

1 Introduction

A number of systems have demonstrated the promise of *transparency* as a means to enhance security, most prominently the Certificate Transparency protocol first launched in 2013 [LLK13, Lau14]. The goal of transparency systems is to ensure that an authority's behavior can be monitored by users. Typically, misbehavior by the authority is not prevented but is detectable. The implicit assumption is that large, public-facing authorities are potentially malicious (or compromised) but are cautious: they are unwilling or at least extremely hesitant to carry out any attack that will leave evidence.

Certificate Transparency applies this philosophy to certification authorities (CAs) for the Internet's TLS public key infrastructure. By requiring that every valid certificate is included in a publicly available append-only log, Certificate Transparency enables genuine domain owners to scan for improperly issued certificates for their domain. This security model is reactive, weaker than that of many proactive proposals which aim to prevent rogue certificates from being accepted in the first place [CvO13]. However, it has proved to provide a useful balance between security, performance and flexibility. By comparison, HPKP [EPS15], a proactive protocol enabling domain owners to *pin* a limited set of keys to their domain, was deployed in 2015 but deprecated by 2019 due to lack of deployment and problems with overly strict pinning policies preventing legitimate key changes.

Transparency has been proposed in a number of other security contexts, including user-public key mappings in encrypted communication systems [Rya14, MBB+15], usage of cryptographic keys [YRC15], and distribution of software binaries [FDP+14, NKJ+17, AM18]. *Verifiable registries* [CDGM19, MKL+20] provide transparency for the key-value mappings required for all of these applications.

Key transparency. While our solution is general, our motivating challenge is public key distribution for secure messaging services. In addition to the importance of this application, it is the most demanding in terms of performance requirements, requiring support for a registry which contains billions of entries, is updated very frequently, and is audited by clients using mobile devices which are limited in bandwidth and computational power and frequently go offline.

Consider a modern secure messaging application such as WhatsApp or iMessage. These services have brought end-to-end encrypted communication to billions by hiding key management from users. Public-key distribution is centralized: a user Alice can send a message to Bob simply by entering his phone number. The phone number is used as a lookup key at a key directory server, which responds with Bob's public key. Of course, this is a major potential security hole as a malicious server could respond with an incorrect key for Bob and then decrypt all of Alice's communication to Bob. Currently, the only defense against malicious behavior by the key directory server is manual verification of key fingerprints by end users, an error-prone [DSB+16, TBB+17, VWO+17] process which the vast majority of users neither understand nor attempt [ASB+17].

With key transparency, the key directory server publicly commits to the mapping from users to keys using a verifiable registry. Users' devices can automatically monitor the key directory and alert users if any unexpected keys are mapped to their user name. This does not proactively prevent attacks: the server is still able to insert rogue keys which will be accepted and used. However, doing so will produce cryptographic evidence that users are able to see. In fact, it is typically a *requirement* that the server can unilaterally change a user's public key – a property needed for users to recover access if they lose their current device (and private keys) [BBG+20].[1]

The challenge of client auditability. Our goal is to support verifiable registries with billions of entries. Proofs of the current value of a mapping (used to prove the current value of a user's public key) should be small (e.g. less than a few kilobytes) and efficient for the server to compute. Most im-

[1] A minority of users may prefer a stronger, proactive security policy in which any key change must be authorized by signing with a key that is not under the servers' control, as discussed in CONIKS [MBB+15].

portantly, it must be efficient for the server to prove to a client what updates (if any) have been made to the client's key after an arbitrarily long period of disconnection. It is not enough for a client to check that the current value is unchanged since the last check, because the server may have changed the client's key to a malicious value and changed it back before the client came back online to check (a *ghost key* attack).

Previous solutions have either required clients to do auditing work linear in the number of updates to the registry (limiting update frequency in practice) [LLK13, Rya14, MBB+15] or relied on third-party auditors to verify global invariants at a cost linear in the entire size of the registry [MBB+15, MKL+20]. Our goal is to enable clients to audit the registry themselves efficiently both in asymptotic terms (e.g. in constant time with respect to the number of updates) and concrete terms (e.g. in milliseconds on a mobile device).

Contributions and roadmap. We propose several constructions for verifiable registries. While all of our constructions enable efficient client auditability, different techniques offer different trade-offs between server computation (key update throughput and key lookup latency), bandwidth costs, and client computation. We abstract the core of our constructions in terms of a new cryptographic primitive, which we call an *authenticated versioned dictionary* (AVD).

Merkle tree-based solutions. We first adapt constructions using Merkle trees in the trusted auditor setting [MBB+15], enabling efficient client auditing using *incrementally verifiable computation* (IVC) [Val08] via *recursive proofs* [BCCT13, BCTV14]. In each epoch, a succinct proof (in particular, a SNARK [Gro10, GGPR13]) is provided which proves the update from the previous epoch is correct and also recursively verifies a proof for the previous epoch.

However, state-of-the-art approaches for proof recursion incur significant practical overhead [BCTV14]. To overcome this limitation, we propose a novel approach to IVC using hierarchical SNARK aggregation [BMM+19] (which may be of independent interest). At epoch N, a client receives one aggregate proof for the first $N/2$ epochs, then the next $N/4$, $N/8$, and so on, resulting in logarithmic client overhead as well as amortized logarithmic server overhead to merge proofs after each epoch. Even so, replacing recursion-friendly SNARKs with standard SNARKs results in significant performance gains.

RSA accumulator-based solutions. Auditing an updated Merkle tree requires work linear in the number of updates. While parallelizable, computing a SNARK proof over many updates becomes a bottleneck at high throughput. To overcome this, we turn to RSA accumulators, whose algebraic structure enables sublinear batching of update proofs [BBF19]. We augment an RSA key-value accumulator [AR20] with efficient batched update proofs, producing an RSA-based verifiable registry with update proofs of constant size and apply the same two IVC approaches to achieve client auditability. Computing the update proof still requires work linear in the number of key updates, but only constant work done by the more costly SNARK prover, enabling higher update throughout than Merkle tree based solutions.

Additionally, we give an "algebraic" RSA construction that dispenses with the use of SNARKs entirely. Similar to our approach to IVC via aggregation, algebraic update proofs are provided over logarithmic number of ranges of increasing size. Instead of creating a succinct proof for each range by aggregating individual SNARKs, new range proofs are computed directly with a batched update proof over the range of epochs. Again, merging ranges can be done at relatively low amortized cost, leading to a client-auditable verifiable registry based solely on algebraic proofs.

While our RSA accumulator constructions enable high throughput of key updates, serving validity proofs for individual key-value mappings incurs higher latency, naively requiring work linear in the total size of the registry. We provide some deployment optimizations that help alleviate these costs with batching and caching, but ultimately this represents an essential performance tradeoff between the two approaches.

Checkpointing. To meet efficiency goals, clients will necessarily not be able to audit every update made to the registry. We introduce a powerful checkpointing technique that implicitly ensures a key update made by one client is represented in lookups made by another client. This holds as long as both clients individually perform cheap audits whenever looking up or updating a key.

Evaluation. We implement and evaluate our proposed constructions. In all of our constructions, clients save 100–$10^4\times$ bandwidth and computation costs over prior solutions requiring auditing every update; daily auditing requires downloading < 10 KB of data and < 100 ms verification time. On the server side, our RSA-based constructions surpass practical throughput requirements (60 updates / second), which is $100\times$ faster than baseline solutions using Merkle trees. However, the RSA constructions incur additional costs for key lookups.

2 Setting and Threat Model

A *verifiable registry* [CDGM19, MKL+20] consists of a list of key-value pairs (k, v) administered by a centralized[2] *server*. The server periodically signs and publishes, at each *epoch*, a commitment (or *digest*) d_i to the registry state D_i.

The server is not trusted and may arbitrarily deviate from the protocol. Our goal is not to prevent attacks, but to ensure that they are eventually detected. This is particularly suitable for an *honest-but-cautious* adversary. We do not attempt to

[2] It would be possible to use a semi-centralized model in which a set of semi-trusted servers collaboratively maintain the registry using techniques from distributed consensus and threshold cryptography.

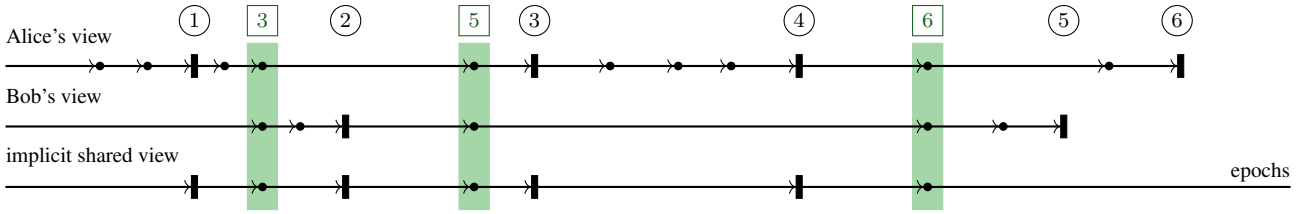

Figure 1: Eventual inconsistency detection for Alice's and Bob's view using shared checkpoints. Large ticks with circle labels indicate points in time where Alice or Bob perform a key lookup. When Alice or Bob make a new lookup, they audit the contiguous checkpoint subranges since their last lookup; the checkpoints are indicated by dots and the arrows indicate proofs that the versioned invariant holds between epoch digests. Checkpoints are chosen to guarantee that any two of Alice and Bob's overlapping lookup ranges will share at least one checkpoint, highlighted in green. Thus, the epoch digests of Alice and Bob's lookups are implicitly guaranteed to preserve the versioned invariant, up until their most recent shared checkpoint. The square label for the shared checkpoint indicates at what lookup the shared checkpoint first exists. The shared checkpoint lags behind the most recent lookups made by Alice and Bob, but will eventually catch up on future lookups.

guarantee availability against a malicious server, which can simply refuse to respond to any queries.

To prevent *split-view attacks*, i.e., attacks where different clients are provided diverging views of the registry, we assume some out-of-band mechanism for clients to communicate. This is necessary and in line with prior work on transparency systems [LLK13, MBB+15, CDGM19, MKL+20, LKMS04]. Concretely, we assume the availability of a public *bulletin board* that the server can append values to, but cannot otherwise tamper with or censor, a common assumption in cryptographic protocols in several contexts [Ben87, CBM15, CGJ+17]. This is a useful abstraction that admits different implementations – e.g., via a public blockchain [TD17] or using a gossip protocol [STV+16, MKL+20].

Basic lookups and monitoring. Clients can interact with the server to query a key[3] k at epoch i and retrieve the associated value v, along with a proof π of validity with respect to d_i and some additional metadata (such as a version number). Clients perform lookups at the current epoch i to learn the authoritative value for a given key. The committing property of the registry ensure that, for a given digest d_i and key k, the server can only produce a valid proof π for one value v. For example, if Alice wants to encrypt a message to Bob, she queries Bob's username k_{Bob}, verifying the proof π to ensure she has received the correct value v_{Bob} of Bob's public key.

Clients also continually query their own username to ensure that the mapped value is correct, a process called *monitoring*. The additional metadata returned with monitoring queries can be used to quickly verify that a client's mapping has not changed during a long period of disconnectivity. For example, if a client queries its own key at digest d_i and the associated metadata indicates the version number has not changed since the last digest d_j which the client queried, this guarantees the mapping has not changed during this period.[4]

Invariants and client auditing. Associating metadata with each mapping enables efficient monitoring across many digests, but requires that every digest preserves certain invariants with respect to the prior digest. Past work has considered two such invariants. The *versioned* invariant [MBB+15, Bon16] associates with each key a version number that must be incremented whenever that key's value is updated. The *append-only* invariant [TBP+19] associates with each key an append-only list of the entire history of values for that key over the lifetime of the dictionary. Either invariant makes it easy to detect if a mapping has been modified; we focus exclusively on the simpler versioned invariant in this work, observing that the append-only property can be recovered by storing values in a chronological Merkle tree and versioning the Merkle root [MKL+20].

Our constructions improve upon prior work [MBB+15, CDGM19, TBP+19] in enabling *clients themselves* to efficiently audit any pair of epochs (i, j) — even if they are not consecutive — given digests d_i and d_j to ensure that D_j has been obtained from D_i without violating the invariant. Prior works[5] assume a third-party that audits the invariant property for each consecutive digest pair, (d_i, d_{i+1}), which requires work at least linear in the number of epochs to audit (in many cases, more: linear in the number of key updates). Because of this, client audits have not been practical due to both the computational burden required to verify every epoch proof and the bandwidth costs on the server to serve such proofs.

We rely on a bulletin board to prevent split-view attacks, in which a malicious server convinces Alice to accept digest d_i and Bob to accept digest $d'_i \neq d_i$ for the same epoch i. Both digests might be valid updates from a common ancestor d_j, but map a key to two distinct values. We assume that all digests $d_0, d_1, \ldots$ (i.e., one unique digest per epoch) are published by the server on the bulletin board, which clients will read from to maintain a consistent view.

[3] We use *key* to refer to the lookup key in a directory, e.g. a username. The *value* associated with that lookup key may itself be a cryptographic key in applications such as key transparency.

[4] If the version number has changed, the server can provide a proof that it changed at a specific digest d_k for $i < k < j$.

[5] An exception is the recent work of Chen et al. [CCDW20] which supports efficient auditing across epochs as does our work. We compare our approaches further in Section 9.

Efficient checkpointing. An important point is that an audit proof for epochs (i, j) only depends on d_i and d_j, and does not give any guarantee for the digests $d_{i+1}, \ldots, d_{j-1}$ that are on the bulletin board and other clients may access. (This is necessary to avoid work linear in the number of epochs.) Thus, a major challenge is that since clients are online at different times, they might never audit the same epochs. A malicious server can attempt an *oscillation attack* [MKL+20], serving clients interleaving sequences of digests where each sequence preserves the update invariant, but the two sequences interleaved do not preserve the invariant. If we want any two clients to detect such an attack, we need to ensure they at some point check digests for the same epoch—but we also want to avoid requiring clients to retrieve *all* digests, incurring work linear in the number of epochs.

We overcome this by introducing a new *checkpointing* technique, illustrated in Figure 1. For every audited range (i, j), clients audit the invariant for a logarithmic number of *checkpoint digests* corresponding to certain canonical epochs between i and j. Crucially, these checkpoints are chosen so that any two overlapping ranges will share at least one checkpoint. This implicitly guarantees that, for any two clients, the invariant is preserved through the sequence of digests in their interleaved view up to their latest common checkpoint, and any invariant deviance since then will eventually be detected on future audits. We note that clients *may* temporarily accept digest values that do not preserve the invariant. But the crucial point is that an oscillation attack is *guaranteed* to eventually be detected.

3 Preliminaries

Authenticated dictionaries. Our system for verifiable registries will use an *authenticated dictionary*. A dictionary D is a finite collection of key/value pairs $[(k_i, v_i)]_i$ where every key is unique; we denote $D[k] = v$ to map k to its unique value v. An authenticated dictionary enables committing to a dictionary with a digest, $d \leftarrow \mathsf{Commit}(D)$, and updating a value in the dictionary to produce a new digest, $(d', D') \leftarrow \mathsf{Update}(D, k, v)$. Lastly, it provides proofs for key lookups, $(v, \pi) \leftarrow \mathsf{Lookup}(D, k)$, that can be verified given the digest commitment, $0/1 \leftarrow \mathsf{VerLookup}(d, k, v, \pi)$. An authenticated dictionary must satisfy *value binding*, which means that it is infeasible to produce valid lookup proofs for key k to different values v and v'.

The most prevalent authenticated dictionary used in practice is a *Merkle tree* [Mer87]. Merkle trees admit lookup proofs and update proofs — proving that only a single key mapping was changed — of size and verification time $\mathcal{O}(\log N)$ for dictionaries of size N. We review these algorithms in Appendix A.

RSA groups and key-value dictionaries. An *RSA group* is the multiplicative group of invertible integers modulo N (denoted $\mathbb{Z}_N^\times$), where N is the product of two secret primes. We define the *RSA quotient group* for N as $\mathbb{Z}_N^\times \setminus \{\pm 1\}$. The widely believed Strong RSA Assumption (Strong-RSA) asserts that it is computationally difficult to compute e^{th} roots of a non-trivial element of $\mathbb{Z}_N^\times$ for $e \geq 3$.

Recently, it was shown how to construct efficient authenticated key-value dictionaries based on the security of Strong-RSA [AR20, BBF19]. Our work builds on the KVaC construction [AR20] which we provide in Appendix E.

Lastly, proofs of integer discrete log [Wes19, BBF19] have been useful for batching insertions and membership proofs in RSA accumulators [CL02]. In such a proof, a prover convinces a verifier that for $u, v \in \mathbb{G}$ and $\alpha \in \mathbb{Z}$, the relation $v = u^\alpha$ holds, where $\mathbb{G}$ is an RSA quotient group. Importantly, the integer α can be much larger than $|\mathbb{G}|$, but the verifier's running time remains $\tilde{\mathcal{O}}(|\mathbb{G}|)$. Later, we will extend these techniques to apply to the RSA key-value dictionary.

SNARKs and incrementally-verifiable computation. A *non-interactive proof system* for a relation $\mathcal{R}$ over *statement-witness* pairs (x, w) enables producing a proof, $\pi \leftarrow \mathsf{Prove}(pk, x, w)$, that convinces a verifier $\exists w : (x; w) \in \mathcal{R}$, $0/1 \leftarrow \mathsf{Ver}(vk, \pi, x)$; pk and vk are proving and verification keys output by a setup, $(pk, vk) \leftarrow \mathsf{Setup}(\mathcal{R})$.

A *non-interactive argument of knowledge* further convinces the verifier not only that the witness w exists but also that the prover *knows* w. If π is succinct, i.e. of "small" size and verification time, with respect to x and $\mathcal{R}$, the protocol is further known as a (preprocessing) *SNARK* [Gro10, GGPR13]. We make use of SNARKs for relations of general circuit satisfiability, of which there exist many constructions [GGPR13, Gro10, GWC19, CHM+20, BBHR19, BFS20, Set20].

An *incrementally-verifiable computation* (IVC) [Val08] allows proving correctness of repeated application of a circuit computation. The predominant approach to IVC is use of *recursive SNARKs* [BCCT13, BCTV14, BCMS20, BGH19], in which the proof circuit for each intermediate state verifies one step of computation from the previous state *and* verifies correct computation from the initial state to the previous state by recursively verifying the proof for the previous state; such a proving circuit can be described because the recursive verification can be computed succinctly.

4 Authenticated Versioned Dictionaries

In this section we will define an *authenticated versioned dictionary* (AVD), the novel cryptographic primitive behind our verifiable registry system, and present several constructions of this primitive.

In a versioned dictionary D, every key k has both an associated value v and a version number u, denoted $D[k] = (u, v)$. The version number is incremented upon each update to the key. An ordered pair (D, D') of dictionary states preserve the *versioned invariant* if for every key k where $D[k] = (u, v)$

and $D'[k] = (u', v')$, either $v = v'$ or $u' > u$. An AVD provides the following algorithms to commit to the historical states of a versioned dictionary and also to compute proofs that they preserve the versioned invariant:

- $d_i \leftarrow \mathsf{Commit}(i, \mathsf{hist}_i)$ returns a digest for epoch i, a binding commitment to $\mathsf{hist}_i = (D_0, D_1 \ldots, D_i)$, which is an ordered list of dictionary states from epochs 0 to i.
- $(i+1, \mathsf{hist}_{i+1}, d_{i+1}) \leftarrow \mathsf{Update}(i, \mathsf{hist}_i, \{k_j, v_j\}_j)$ updates the dictionary values for input keys $\{k_j\}$ to the values $\{v_j\}$, increments the version number of each updated key, increments the epoch number i, and outputs a new digest d_{i+1} representing the new dictionary history.
- $(d_i, \rho) \leftarrow \mathsf{ProveEpoch}(i, d_j, \mathsf{hist}_j)$, for $j \geq i$, returns the digest d_i for epoch i along with a proof ρ that d_i is correct (i.e., consistent with d_j).
- $0/1 \leftarrow \mathsf{VerEpoch}(i, d_i, d_j, \rho)$ verifies the proof ρ that d_i is consistent with digest d_j.
- $(u, v, \pi) \leftarrow \mathsf{Lookup}(D, k)$ returns the value v and version number u along with a proof π that $D[k] = (u, v)$.
- $0/1 \leftarrow \mathsf{VerLookup}(d, k, u, v, \pi)$ verifies the proof π that $D[k] = (u, v)$ is consistent with d.
- $\pi_{\mathsf{Audit}} \leftarrow \mathsf{Audit}(i, d, \mathsf{chkpts}, \mathsf{hist})$ takes as input $\mathsf{hist} = (D_0, \ldots, D_i)$, the digest d for hist, and an ordered list of checkpoints $c_1, \ldots, c_k \in [i]$; it returns a proof that all pairs $(D_{c_j}, D_{c_{j+1}})$ for $j \in [1, k-1]$ respect the versioning invariant.
- $0/1 \leftarrow \mathsf{VerAudit}(i, d, \mathsf{chkpts}, \pi_{\mathsf{Audit}})$ is given the digest representing the dictionary history in epoch i along with an ordered list of checkpoints $c_1, \ldots, c_k \in [i]$ and verifies the proof π_{Audit}.

An authenticated versioned dictionary must satisfy *history binding*, which means that it is infeasible to produce any i, d, d_i, d_i' and ρ, ρ' such that $\mathsf{VerEpoch}(i, d_i, d, \rho) = 1$ and $\mathsf{VerEpoch}(i, d_i', d, \rho') = 1$ or proofs π, π' for version/value pairs $(u, v) \neq (u', v')$ such that $\mathsf{VerLookup}(d, k, u, v, \pi) = 1$ and $\mathsf{VerLookup}(d, k, u', v', \pi') = 1$. In addition, the prover algorithm Audit and verifier algorithm VerAudit must satisfy the security definition of a non-interactive proof system for the relation $\mathcal{R}_{\mathsf{Audit}}$ over statement/witness pairs $(x = (d, \mathsf{chkpts}), w = \mathsf{hist})$, where $d = \mathsf{Commit}(\mathsf{hist})$ and for every checkpoint $c_j \in [i]$ the pair of dictionary states $(D_{c_j}, D_{c_{j+1}})$ from the list $\mathsf{hist} = (D_1, \ldots, D_i)$ respect the versioned invariant. Note that proving all historical states respect the versioned invariant is a special case where the checkpoints include all prior epochs.

4.1 Constructions: SNARK Recursion vs Aggregation

We begin with discussing general methods for constructing an AVD from any underlying authenticated dictionary, and we will later present several concrete constructions of AVDs starting from either sparse Merkle trees or the RSA key-value commitment KVaC [AR20]. We distinguish the AVD algorithms from the authenticated dictionary algorithms Commit, Update, Lookup, and VerLookup of the same name using the notation $\mathsf{AVD}.xxx$.

History commitment. The version number of a key is treated as part of the dictionary value, i.e. $D[k] = (u, v)$. The commitment AVD.Commit combines a dynamic append-only Merkle tree (or more generally any dynamic vector commitment) with the algorithm Commit. The digest for the first epoch with no history $d_0 \leftarrow \mathsf{AVD.Commit}(0, D_0)$ is set to the digest $\delta_0 \leftarrow \mathsf{Commit}(D_0)$. Given $\mathsf{hist}_i = (D_0, ..., D_i)$ and the digests $d_0, ..., d_{i-1}$ for the first $i-1$ epochs, the ith epoch digest $d_i \leftarrow \mathsf{AVD.Commit}(i, \mathsf{hist}_i)$ is a pair of values $d_i = (\mathsf{rt}_i, \delta_i)$ where rt_i is the root of a Merkle tree over leaf values $(d_0, ..., d_{i-1})$ and $\delta_i \leftarrow \mathsf{Commit}(D_i)$.

Batched updates. $\mathsf{AVD.Update}(i, \mathsf{hist}_i, \{k_j, v_j\}_j)$ first retrieves the current version number u_j of each k_j and then applies the underlying update algorithm $\mathsf{Update}(D_i, k_j, (u_j + 1, v_j))$ iteratively for each (k_j, v_j) pair, returning the final (δ_{i+1}, D_{i+1}) pair. The epoch number is directly incremented to $i+1$, is appended to hist_i to get hist_{i+1} D_{i+1}, and $d_{i+1} = (\mathsf{rt}_{i+1}, \delta_{i+1})$ where rt_{i+1} is the root of the Merkle tree over leaves $(d_0, ..., d_{i-1})$.

Audit proofs. The first step is to construct an audit proof for a single pair of authenticated dictionary digests. This is a non-interactive proof system for a relation $\mathcal{R}_{\mathsf{Inv}}$ over pairs $(x = (\delta, \delta'), w = (D, D'))$ such that $\delta \leftarrow \mathsf{Commit}(D)$, $\delta' \leftarrow \mathsf{Commit}(D')$, and the transition from dictionary state D to D' respects the versioned invariant. Since our ultimate goal is to build a registry that clients can audit efficiently we optimize for the proof size and verification time (i.e., we would like the proof system to be a SNARK for the relation $\mathcal{R}_{\mathsf{Inv}}$). We will denote this abstractly as a pair of algorithms $\pi_{\mathsf{Inv}} \leftarrow \mathsf{ProveInv}(D, D')$ and $0/1 \leftarrow \mathsf{VerInv}(\delta, \delta', \pi_{\mathsf{Inv}})$. Any general purpose SNARK scheme suffices, however, in our concrete constructions we obtain specialized proofs of this relation that are more efficient that general purpose SNARKs.

The second step is to construct a proof system for auditing a series of history checkpoints, starting from the algorithms ProveInv and VerInv. Naively, the prover could create proofs for all pairs of adjacent epoch digests in hist_i using ProveInv and return all of these proofs. This is a valid proof for $\mathcal{R}_{\mathsf{Audit}}$ irrespective of the specified checkpoints (in fact, it proves the entire history of digests respects the invariant). This has the advantage that the server managing the versioned dictionary can compute a single proof for the most recent digest pair after each update and does not need to recompute any proofs upon a client audit. On the other hand, the proof for $\mathcal{R}_{\mathsf{Audit}}$ is linear in the number of epochs and thus too expensive for client-auditing. Alternatively, the audit proof could consist of individual invariant proofs only for each pair of checkpoint digests. This is linear only in the number of checkpoints, but the server would have to compute these proofs on the fly during an audit, incurring a higher latency. We have two

different approaches to addressing these challenges: the first method uses SNARK recursion and the second uses SNARK aggregation.

SNARK recursion. Using a standard approach for incrementally-verifiable computation based on recursive proofs [BCCT13, BCTV14], the server can generate a single proof attesting to the entire history of digests upon each update. Each incremental SNARK for the ith epoch proves that d_i for epoch i and d_{i-1} for epoch $i-1$ respect the invariant, and also proves the existence of a similarly valid SNARK proof for epoch $i-1$. Recursively, this attests that the invariant is preserved from d_0 through d_i. The complexity of this recursive relation is proportional to the combined complexity of the SNARK verification algorithm and $\mathcal{R}_{\mathsf{Inv}}$. As an optimization, we can replace $\mathcal{R}_{\mathsf{Inv}}$ (which is linear in the size of the dictionary) with the relation $\mathcal{R}_{\mathsf{VerInv}}$ over pairs $(x = (d, d'), w = \pi_{\mathsf{Inv}})$ where $d = (\mathsf{rt}, \delta)$ and $d' = (\mathsf{rt}', \delta')$ such that $\mathsf{VerInv}(\delta, \delta', \pi) = 1$. In other words, instead of proving the invariant directly using the SNARK, the prover computes the basic invariant proof using $\mathsf{ProveInv}(D, D')$ to get the witness π_{Inv}, and then uses the SNARK to prove its existence along with the rest of the recursive statement. The complexity of $\mathcal{R}_{\mathsf{VerInv}}$ is proportional to the complexity of the verification algorithm VerInv. This saves the prover work when the algorithm $\mathsf{ProveInv}$ is substantially more efficient than the generic SNARK prover algorithm, which is true for all of our concrete constructions of AVDs.

Finally, the server can compute the next incremental SNARK each time it updates the AVD. The latest SNARK is returned as the audit proof. Computing a recursive SNARK upon each update has a relatively high prover overhead compared to computing a normal invariant proof for each new digest. However, it achieves a constant size proof and verification time, independent of the epoch or number of checkpoints. It also still achieves better latency than the naive solution of computing an invariant proof on the fly for each pair of checkpoints. Moreover, computing the SNARK once per epoch amortizes this work over all client audit requests.

SNARK aggregation. An alternative approach is to lower the cost of producing invariant proofs for all checkpoints on the fly via a technique called SNARK aggregation. This is a bootstrapping technique, where the server will maintain a cache of precomputed invariant proofs for epoch digest pairs, and then will aggregate these proofs to get the audit proofs for the checkpoint intervals. [BMM+19] present a protocol that aggregates N individual SNARKs into a single proof of size $\mathcal{O}(\log N)$ in $\mathcal{O}(N)$ time. They further show that if the statements being proved by the SNARKs are "sequential", that is, the statement for SNARK at index i is $x_i = (a_i, b_i)$ where $a_i = b_{i-1}$ and $b_i = a_{i+1}$, then the aggregated proof can be verified efficiently in $\mathcal{O}(\log N)$ time. This version of the aggregated proof proves knowledge of the intermediate statements x_i, given the starting and ending statements, x_1 and x_N. This is indeed the case when aggregating invariant proofs for a contiguous list of adjacent epoch digest pairs (full details provided in Appendix D).

Producing the aggregated proof is still linear in the number of proofs being aggregated. Unfortunately, this means that constructing an aggregated proof for even a small number of checkpoints will in the worst case require work linear in the total number of epochs. Fortunately, for the specific checkpoints that clients in our system will audit there is an efficient proof caching and aggregation strategy that achieves amortized efficiency. We will discuss this in Section 6.

4.2 Merkle Tree Constructions

We construct several variations of AVDs based on a sparse Merkle tree as the underlying authenticated dictionary. The first uses SNARK recursion to build the audit proofs and the second uses SNARK aggregation.

The sparse Merkle tree authenticated dictionary algorithms $(\mathsf{Commit}, \mathsf{Lookup}, \mathsf{VerLookup})$ works as follows. Let $H : \mathcal{K} \to [N]$ denote a one-to-one mapping of keys in the dictionary keyspace to integers in $[N]$.[6] The algorithm $\mathsf{Commit}(D)$ returns the root d of a sparse Merkle tree over N leaves where for each $k \in \mathcal{K}$ the leaf at index $H(k)$ is assigned the value $v = D[k]$. The algorithm $\mathsf{Lookup}(D, k)$ returns $v = D[k]$ and the Merkle path π for the leaf at index $H(k)$. The algorithm $\mathsf{VerLookup}(d, k, v, \pi)$ verifies the Merkle path π for the index $H(k)$ and leaf value v.[7]

The sparse Merkle tree authenticated dictionary is transformed into an AVD using the generic procedure described in Section 4.1 above. However, as discussed, there are several ways to construct the audit proofs with varying tradeoffs. The most straightforward way to prove the invariant relation $\mathcal{R}_{\mathsf{Inv}}$ for a pair of digests (d, d') is to provide the complete set of key/value pairs that differ between the two dictionary states along with a Merkle update proof (Section 3) for this complete set of updates. The verifier checks that the update proof is valid and also checks the invariant directly: for each key k updated from $D[k] = (u, v)$ to $D'[k] = (u', v')$, it checks that $u' > u$. (This is the audit proof used in CONIKS [MBB+15]). The algorithm Audit for a sequence of checkpoints returns a proof of this form for each adjacent pair of checkpoint dictionary states. Constructing this audit proof is efficient for the prover, but inefficient for the verifier (i.e., linear in the total sum of updates between all checkpoint pairs). We will refer to this construction as **MT-Linear**.

To reduce the size and verification complexity of audit proofs, we apply either the generic SNARK recursion method or the SNARK aggregation method described in Section 4.1.

[6] N need not exceed 2^{256}. If the dictionary keyspace is larger than 2^{256}, then H can be a collision-resistant hash function instead of a bijective function.

[7] The Merkle tree is not an explicit input to the algorithm $\mathsf{Lookup}(D, k)$, but the server managing the authenticated dictionary would have this cached and would not need to reconstruct the Merkle tree path.

Here, the complexity of the relation $\mathcal{R}_{\mathsf{VerInv}}$ is quasi-linear in the number of updates between the two epoch states because it is proportional to the verification of the Merkle update paths described above. We will refer to the resulting construction from SNARK recursion as **MT-Recurse** and the construction from SNARK aggregation as **MT-Aggr**.

In Appendix B, we present an optimization to reduce the length of the Merkle paths in the relation $\mathcal{R}_{\mathsf{VerInv}}$.

4.3 RSA Accumulator Constructions

Our second family of constructions use a key-value authenticated dictionary called KVaC [AR20], which is based on the classical RSA accumulator. KVaC directly incorporates version numbers as well. Key-value pairs are committed to with the following digest, where u represents a version number for the key, H is a collision-resistant hash function mapping keys to primes, and g is a member of an RSA quotient group:

$$d \leftarrow \left(g^{\left(\prod_i \mathsf{H}(k_i)^{u_i}\right)\cdot\left(\sum_i v_i/\mathsf{H}(k_i)\right)}, g^{\prod_i \mathsf{H}(k_i)^{u_i}}\right)$$

To update a key's value from v to $v+\delta$, the new digest $d' = (d_1^{\mathsf{H}(k)} d_2^{\delta}, d_2^{\mathsf{H}(k)})$ is computed, where the previous digest $d = (d_1, d_2)$.

Batching updates. When updating the values associated with many keys, we observe that instead of applying each update in sequence, all updates $[k,\delta]_i$ can be applied at once by the following:

$$Z \leftarrow \textstyle\prod_i \mathsf{H}(k_i) \qquad \Delta \leftarrow \left(\prod_i \mathsf{H}(k_i)\right)\cdot\left(\sum_i \delta_i/\mathsf{H}(k_i)\right).$$

Then the batched update follows the same form as before, $d' = (d_1^Z d_2^{\Delta}, d_2^Z)$. We will take advantage of this form to construct audit proofs for the versioned invariant.

Proving the versioned invariant. To build an AVD from KVaC, we start with constructing a proof that the versioned invariant is preserved between two digests. One way to do this is to prove that d' is the result of correctly applying the batch update procedure to d, but it turns out that proving a weaker statement suffices. The prover constructs a proof of knowledge for the following relation between $d = (X_1, X_2)$ and updated digest $d' = (Y_1, Y_2)$:

$$\mathcal{R}_{\mathsf{KVaC}} = \left\{((X_1, X_2, Y_1, Y_2); (\alpha, \beta)) : Y_1 = X_1^{\alpha} X_2^{\beta} \wedge Y_2 = X_2^{\alpha}\right\}.$$

We show in Appendix E that it is computationally infeasible to produce a valid proof for this relation if the versioned invariant is violated. This is a somewhat surprising result, as we do not enforce any extra structure on α and β, such as matching the structure of (Z, Δ). Rather, simply proving knowledge of *any* α and β ensures that either the underlying pair of dictionary states do not violate the versioned invariant *or* that the prover has solved a computational problem related to factoring, breaking the Strong-RSA assumption.

We use the generalized knowledge of integer discrete log proof system from [BBF19] (Figure 12, Appendix E) as the non-interactive proof of knowledge for $\mathcal{R}_{\mathsf{KVaC}}$. Importantly, this proof system, which leverages the algebraic structure of the RSA group, has a constant-time verification algorithm and constant-sized proof. This is a significant improvement over other Merkle-based [MBB+15, MKL+20] and bilinear pairing-based [TBP+19, LGG+20] constructions of authenticated dictionaries with append-only proofs.

Unfortunately, computing membership and non-membership proofs for keys from scratch is expensive – on the order of the combined number of keys with non-null values and number of past updates to the dictionary. Given a (non-)membership proof for a previous epoch, the proof can be updated to be valid for the current epoch in time linear in the number of key updates that have since occurred. However, even these updates can be expensive for the provider if many epochs have passed since a key's last query date. We provide some optimizations to alleviate these costs in Section 6.

Checkpoint auditing. Recall that there are two "naive" ways to implement the audit proofs for multiple checkpoints using the invariant proofs above, as discussed in Section 4.1. The first sends the verifier proofs for every single adjacent digest in the history (starting from the first checkpoint and ending at the last) and the second sends the verifier invariant proofs for every adjacent pair of checkpoint digests. The first is more computationally efficient for the prover, while the second achieves smaller proofs and verification.[8] We call this first method **RSA-Linear** and the second method **RSA-Alg**. Normally, this second method incurs a higher latency to respond to audits and overall requires significantly more server computation (up to quadratic in the number of epochs). However, it turns out that **RSA-Alg** performs well in our system for verifiable registries (Section 5) due to how checkpoints are chosen (see Section 6).

Finally, we can achieve better tradeoffs between the prover and verifier's work by applying either of the generic transformations using SNARK recursion or SNARK aggregation, as described in Section 4.1. We call the resulting constructions **RSA-Recurse** and **RSA-Aggr** respectively.

5 Client Auditing using AVDs

We show here how to use an AVD as described above to build a client-auditable verifiable registry. We consider a single server that maintains and updates a sequence of dictionary states $\mathsf{hist}_i = (D_0, D_1, \ldots, D_i)$ – each dictionary state D_{i+1} is obtained by running the Update procedure on hist_i and new key-value pairs to be included in the next epoch $i+1$. The server also regularly publishes, on a public bulletin board, a (signed) pair (d_i, i), where $d_i \leftarrow \mathsf{Commit}(i, \mathsf{hist}_i)$ –

[8] Note that for the constructions using Merkle trees, the basic invariant proof we presented was linear in the number of updates between the two epoch states, and thus this second strategy would *not* achieve a smaller proof size. In contrast, the invariant proofs for the RSA KVaC dictionary are constant size, independent of the number of updates between the epochs.

as discussed in Section 2, we assume that all clients have a consistent view of this bulletin board.

Client look-up and auditability. Whenever a client looks up a value for k in epoch i, it retrieves[9] the latest commitment d_i from the bulletin board, and requests k from the server. The server then computes $(u, v, \pi) \leftarrow \mathsf{Lookup}(D_i, k)$, and sends (u, v, π) to the client. The client verifies π by running $\mathsf{VerLookup}(d_i, k, u, v, \pi)$.

Before accepting the value v, the client needs to ensure that the states $D_0, D_1, \ldots, D_i$ do not violate the versioned invariant. To solve this, the client also requests from the server an audit proof for a carefully chosen set of checkpoints $\mathsf{chkpts} \subseteq [j, i]$, where j is the epoch of the previous lookup (or $j = 0$ if this is the first lookup) and receives $\pi_{\mathsf{Audit}} \leftarrow \mathsf{Audit}(i, d_i, \mathsf{chkpts}, \mathsf{hist}_i)$ from the server, which is verified. The client also receives proofs $(d_c, \rho_c) \leftarrow \mathsf{ProveEpoch}(c, d_i, \mathsf{hist}_i)$ for all $c \in \mathsf{chkpts}$, and verifies the ρ_c's are correct. Finally, the client also checks that d_c is indeed the digest for epoch c on the bulletin board, for all $c \in \mathsf{chkpts}$.

If the checkpoints are $j \leq c_1 < c_2 < \cdots < c_\ell \leq i$, this guarantees that $(D_{c_k}, D_{c_{k+1}})$ for all $k \in [1, \ell - 1]$ respect the versioned invariant. This is however not enough to prevent oscillation attacks – imagine two clients accessing the same key, but using disjoint sets of checkpoints. One way out of this is to use $\mathsf{chkpts} = (j, j+1, \ldots, i-1, i)$, but this makes the overall workload linear in the number of epochs.

Efficient checkpointing. We avoid linear cost by carefully choosing a subset of checkpoints of *logarithmic* size, inspired by the deterministic skiplist approach of [MB02]. The checkpoint epochs for a range (a, c) have the property that at least one of the checkpoints will be shared with the checkpoints of an overlapping range (b, d), where $a \leq b \leq c \leq d$. This already ensures that if two clients are served different values $v \neq v'$ for the same key and with the same version at epochs $i < j$, respectively, then a violation of the invariant will be caught, at the latest, when the client that made an access at i makes its first lookup at epoch $i' \geq j$. (The same is true if they are served values with out-of-order version numbers.)

To see why this is true, assume that Client 1 makes two consecutive lookups at epochs a and c, whereas Client 2 makes two consecutive lookups at epochs b and d, where $a < b < c < d$. Thus, they audit overlapping epoch ranges (a, c) and (b, d) with shared checkpoint γ where $a \leq b \leq \gamma \leq c \leq d$. Then, the clients cannot be served different values $v \neq v'$ for the same key k and version u at epochs b and c, respectively – this is because Client 2 verified the invariant is preserved between b and γ, whereas Client 1 verified the invariant is preserved between γ and c – combined, these imply that the invariant is also preserved in (b, c), and this contradicts the fact that the views diverge at b and c. In contrast, note that it is possible for the clients to obtain inconsistent values at c and d – however, the violation of the invariant will be caught in a future audit. It is clear that this argument can be generalized to any two lookups made by a pair of clients, and also to guarantee that the later lookup cannot return a value older than the earlier lookup. (Also see Figure 1 for an illustration.)

[9] We abstract away here from the fact that, depending on the implementation of the bulletin board, it may be convenient for the client to obtain the commitment d_i from the server first, and then check consistency with the bulletin board later on.

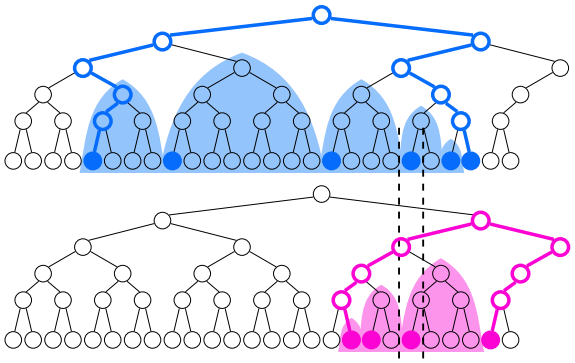

Figure 2: Checkpoint epochs for a range are chosen by the logarithmic number of subtrees that span the range. Two overlapping ranges are guaranteed to share a checkpoint, indicated by the dashed lines. Invariant proofs are only needed for ranges between checkpoints, which correspond only to complete subtrees; this insight is used to improve prover efficiency.

We define the checkpoint epochs for a range as follows. Consider the binary tree imposed over all epochs with each epoch making up a leaf, populated from left to right such that the left subtree of the tree is complete (see Figure 2). Any range of epochs of length M can be divided into $\mathcal{O}(\log M)$ consecutive subranges of complete subtrees, defined by the path in the tree from the start epoch to the end epoch. The $\mathcal{O}(\log M)$ checkpoint epochs are chosen as the start epoch of each of these subtrees. We prove that overlapping ranges are guaranteed a shared checkpoint in Appendix C. Intuitively, the paths in the tree connecting the start and end epochs for overlapping ranges intersect at some point; the nodes at which the two paths intersect identifies a shared starting point for a subtree.

6 Further Optimizations

We describe two optimizations to improve server efficiency for (1) computing checkpoint audit proofs for MT-Aggr, RSA-Aggr, and RSA-Alg, and (2) computing membership proofs for RSA constructions.

Computing audit proofs for all checkpoint ranges. Due to how checkpoints are chosen, the only audit proofs the server needs to provide are those corresponding to complete subtrees in the superimposed tree over epochs (see Figure 2). Of which, there are only $\mathcal{O}(N \log N)$ such ranges compared to the $\mathcal{O}(N^2)$ possible ranges that exist without our checkpoint structure (where N is the number of epochs). The server will precompute and store audit proofs for all checkpoint ranges,

allowing immediate responses to Audit queries.

As new epochs are published, the server computes audit proofs for any newly completed checkpoint ranges (i.e., completed subtrees). The cost to produce an audit proof is linear in the length of the range. Intuitively, the completion of a large range, and the expensive audit proof computation incurred, is a relatively rare event. By a classic amortization argument [Ove83], the amortized work for computing all audit proofs is $\mathcal{O}(\log N)$ for each new published epoch.

For our SNARK aggregation constructions, the server computes and stores the invariant proof for each adjacent epoch digest pair as a SNARK. Whenever a checkpoint range is completed, the server will run the [BMM+19] aggregation protocol on the SNARKs in the range. For RSA-Alg, the server simply computes a new algebraic invariant proof including all the key updates in the range.

This novel combination of SNARK aggregation with amortization also admits a new approach to IVC, which may be of independent interest. For a computation of depth N, our approach is verifier-efficient with proofs and verifier time of size $\mathcal{O}(\log^2 N)$. It is also amortized-prover-efficient where the prover does amortized $\mathcal{O}(|C| + \log N)$ work for each step of computation, where $|C|$ is the size of the computation circuit, but requires $\mathcal{O}(N)$ storage long term to store the individual SNARKs for each step.

Computing RSA (non-)membership proofs. One solution to alleviate the server workload and client latency in waiting for these proofs is the use of *promises*. A *promise* [MBB+15] is a signed statement by the server of a claimed lookup value and the promise to compute a corresponding (non-)membership proof for a specific epoch (by a certain time). A promise allows a client to act on their query without waiting for the full proof; the client can later query the full proof by the promised time and provide evidence of a broken promise if appropriate.

Delaying computation of (non-)membership proofs is also of benefit to the server, as it allows use of existing techniques [BBF19, TXN20] for computing a batch of M (non-)membership proofs together in time $O(N + M \log M)$ time, an improvement over computing each proof individually at the time of request ($O(N \cdot M)$) where N is the total number of past updates. Even with batching, there is a dependence on N: for a long-running system, the total number of updates grows large ($> 2^{30}$), leading to batched computations potentially taking months to complete (i.e., fulfill a promise).

To partially address this issue, we introduce two new techniques for membership proof computation of RSA accumulators to allow low latency promise fulfillment for *some* keys. The first technique is a key caching approach to support a small cache of "hot" keys for which membership proofs may be computed at low latency and progressively larger caches of "colder" keys with longer latency proof computation. Keys can be assigned to caches based on any policy. Some reasonable policies might be by the frequency at which they are queried or by how critical low latency monitoring is to the key's use. If, for example, key lookups follow a power law distribution (Zipf's law), caching may significantly improve median case promise fulfillment.

The second technique is an amortized checkpointing approach to divide up the N updates over the system's history into smaller, more manageable ranges allowing dynamic caches that support key eviction and loading policies. Computing non-membership proofs remains expensive. The full details and proofs of security are deferred to Appendix F.

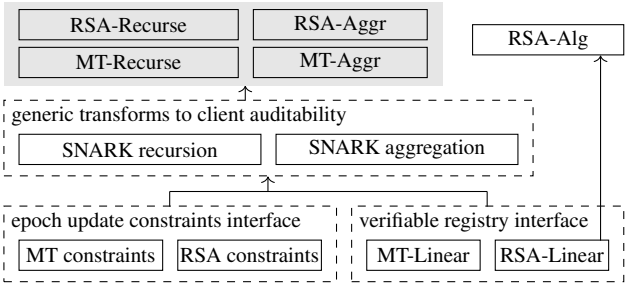

Figure 3: Implementation overview. Implementation includes generic interfaces for transforming a verifiable registry that supports epoch audit proofs for consecutive digests to a registry supporting efficient range audits using SNARKs.

7 Implementation

We implement our proposed constructions in Rust. Our implementation consists of a number of parts (see Figure 3). We define a generic registry interface for a registry that provides update proofs for consecutive epochs, and implement MT-Linear (cf. [MBB+15, MKL+20]) and RSA-Linear (Section 4.3) — were a client to audit these, it would require linear work. An additional interface is defined for generating SNARK constraints of the verification of an epoch update of a registry; we implement these constraints for MT-Linear and the RSA-Linear. We further implement our two transforms for efficient client auditability via SNARKs (recursion and aggregation) to generically construct a client-auditable registry given a registry that implements the above interfaces. This leads to our constructions MT-Recurse, MT-Aggr, RSA-Recurse, and RSA-Aggr. Our final construction, RSA-Alg, which does not rely on SNARKs, is implemented directly from RSA-Linear. The constructions refer to the client-auditable verifiable registry which includes their namesake AVD as well as the checkpointing and amortization optimizations given in Sections 5 and 6. In total, our implementation consists of $\approx$ 11000 lines of code.

The constraints and generic SNARK transforms are implemented within the `arkworks` ecosystem for SNARKs. The RSA constraints make use of optimizations for multiprecision arithmetic [KPS18] and hashing to primes [OWWB20]. The generic SNARK transforms make use of the SNARK

implementations from `arkworks`. The SNARK aggregation transform is specific to [Gro16], while the SNARK recursion transform is implemented generically for any SNARK that supports recursion; we instantiate and evaluate the recursion constructions on [Gro16]. To target 128 bits of security, the aggregation transform is implemented over the BLS12-381 pairing-friendly curve and the recursion transform over MNT4-753 and MNT6-753 pairing-friendly cycle of curves; the RSA constructions use an RSA group of 2048 bits. We set the height of the Merkle tree in the MT constructions to 32, which with our open addressing optimization (see Appendix B) can support 2^{30} keys, and instantiate the hash function using both Pedersen and Poseidon algebraic hash functions [GKK+19].

8 Evaluation

We wish to answer the following questions:

- *Client auditing costs:* What are the bandwidth and computation costs for a client to audit a range of epochs? How does it compare to previous solutions that require auditing every epoch in the range?
- *Server update costs:* What are the computation costs for the server to incorporate key updates and publish a new epoch digest? At what latency can new digests be published; supporting what key update throughput?
- *Lookup costs:* What are the bandwidth and computation costs for lookups?

Experimental setup. We benchmark our constructions using an Amazon EC2 `r5.16xlarge` instance with 32 CPU cores and 512 GB memory. Client computation is evaluated single-threaded, and network costs of gathering client input are not evaluated; our experiments simulate client input, generating random requests of the appropriate size.

Some of our constructions grow in update cost over the history of the registry due to increasing amortized costs of aggregation. We evaluate and present costs for epoch updates as follows. First, it is infeasible to exactly measure the costs of a valid epoch update at large epochs, as constructing a valid state and audit proof at a large epoch is not possible without having run valid updates for all prior epochs (our constructions share many properties with verifiable delay functions [BBBF18]). Instead, we simulate the cost of an epoch update by running it over a dummy state, constructing SNARKs over invalid statements of appropriate size. This approach properly evaluates the cost of constructing an audit proof, but the resulting proof will not meaningfully verify.

Second, we present the amortized costs of aggregation for epoch 2^k by averaging the aggregation costs incurred between the 2^{k-1} updates from epoch $2^{k-1}+1$ to 2^k. While these aggregation costs occur in spikes over the range, our systems are not delayed by the need to complete an expensive aggregation; the aggregations can be computed in the background and audits can still be fulfilled, albeit with slightly longer proofs (using unmerged ranges). Therefore, we believe reporting the amortized costs in this manner leads to a fair evaluation.

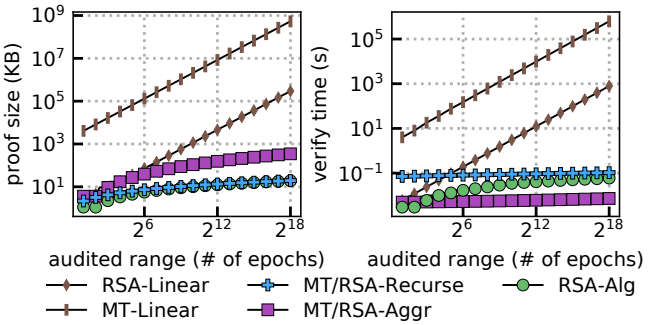

Figure 4: Client auditing costs. The size (left) and verification time (right) of audit proofs for varying epoch range lengths.

8.1 Client Auditing

We contrast the auditing costs in terms of proof size and verification time for different lengths of audit ranges; the results are shown in Figure 4. The goal of our client-auditable constructions is to outperform the linear scaling of MT-Linear and RSA-Linear. All of our constructions scale logarithmically in the length of the audit range for both proof size and verification time. The SNARK recursion and aggregation verification times appear almost constant as their logarithmic costs are negligible: verification of log-number Merkle paths and a log-sized aggregation proof, respectively.

Our constructions are 1-10× more efficient at epoch ranges of length 32 and 100-1000× more efficient at epoch ranges of length 1000 when compared to RSA-Linear which has compact, constant size update proofs. When compared to MT-Linear with 1000 key updates per epoch, the efficiency improvements increase by another 100×. If we consider an epoch publishing time of 20 minutes, auditing at epoch ranges of length 32 and 1000 correspond to a client auditing around twice a day or once every two weeks, respectively.

8.2 Server Epoch Updates

Building efficiently auditable proofs for epoch ranges adds significant computational costs to the server. We investigate what levels of key update throughput are achievable and at what latency. To anchor our evaluation, we set a target of ≈ 60 key updates per second based on current statistics from the certificate transparency ecosystem [Clo20].

First, we evaluate how epoch update latency is affected over time: some of our constructions incur increased cost as the amortized cost of reaggregation increases. Figure 5 (right) plots this (non-)increase. We find that the amortized costs of reaggregation are negligible. These costs grow logarithmically so we expect them to remain manageable even for much larger epochs.

Second, we evaluate the epoch latency for an update while varying the number of key updates included in the epoch:

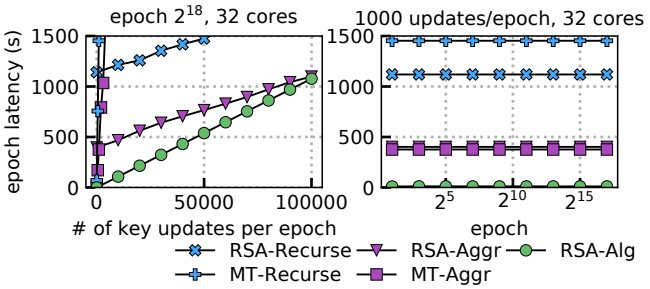

Figure 5: Server key update costs. (Left) The epoch update latency for epochs of varying the number of key updates. (Right) The epoch update latency varying the history of the registry. The key update throughput is computed as the number of key updates per epoch divided by the epoch latency.

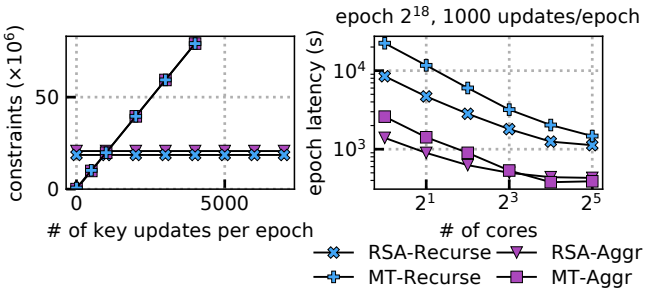

Figure 6: (Left) The number of constraints in the SNARK circuit for varying number of key updates. (Right) The epoch latency (dominated by the SNARK proving time) for different levels of hardware parallelism.

update throughput is computed as number of key updates divided by latency. We fix the epoch at 2^{18} which corresponds to approximately a year of service if publishing epochs every minute, however as already demonstrated, costs are not expected to increase significantly for larger epochs. Figure 5 (left) depicts these results. The MT-Recurse, MT-Aggr, and RSA-Alg constructions are more or less directly proportional to the number of key updates in the epoch, supporting throughputs of 0.6, 2.6, and 92 key updates per second, respectively; throughput is constant regardless of the number of key updates batched in each epoch. Constructions MT-Recurse and MT-Aggr scale in this manner because each additional key update adds a Merkle path verification to the SNARK circuit which makes up the dominant cost. The RSA-Alg construction scales with the dominant cost of amortized reaggregation of Wesolowski proofs which is linearly dependent on the total number of key updates. These constructions support low latency epochs without sacrificing achievable throughput.

On the other hand, the RSA-Recurse and RSA-Aggr constructions have a large constant dominant cost of verification of the Wesolowski proof within a SNARK circuit that is independent of the number of key updates, leading to a large minimum achievable latency. However, increasing the key updates per epoch adds comparatively less work than the other constructions, leading to RSA-Recurse and RSA-Aggr outscaling update throughput of other solutions for high epoch latency. Figure 5 shows that RSA-Aggr reaches the desired throughput of 60 key updates per second at 40,000 key updates per epoch with a latency of 12 minutes, and surpasses the throughput of RSA-Alg at 100,000 key updates per epoch with a latency of 20 minutes, but will cap out at ≈ 150 key updates per second due to the costs of RSA exponentiation.

When comparing the two generic SNARK approaches to IVC, we find apart from slightly larger client proof sizes, our new aggregation approach is strictly better than recursion: providing $> 2.5\times$ speedup in this setting where circuit proving costs dominate aggregation costs.

Improving throughput via parallelism. The dominant cost for our constructions relying on SNARKs is the SNARK proving time, and it has been shown that the SNARK proving work is highly parallelizable [WZC+18]. Figure 6 (left) shows the number of constraints to be proved in the SNARK circuit for different numbers of key updates batched per epoch. The RSA circuit is of constant size, just under 20M constraints. The MT circuit grows linearly with the number of key updates, $\approx 20,000$ constraints per key update. We demonstrate the parallelism of the workload by measuring epoch update latency using different numbers of physical cores, shown in Figure 6 (right). For the circuit sizes evaluated, doubling the number of processors halves the epoch latency up until between 16 and 32 processors where the marginal benefits of adding more processors decreases. Larger circuit sizes, e.g. by adding more key updates to the MT constructions, will continue to benefit from increased processors [WZC+18].

Construction RSA-Alg does not compute a circuit SNARK. Instead, the dominant cost consists of reaggregating append-only proofs for subranges by proving new Wesolowski proofs. While proving a single Wesolowski proof is mostly a sequential task, at any one time there will be approximately $\log N$ (for N total epochs) such Wesolowski proofs being proved in the background, one for each subrange length that is being merged. These tasks can be easily parallelized given $\log N$ processors. A similar strategy to parallelize the aggregation work for MT-Aggr and RSA-Aggr does not lead to overall throughput gain, since the aggregation work is a negligible share of the total work for these constructions.

Improving throughput via sharding. A second way to increase throughput is by sharding the key space and running separate instances of a verifiable registry. If perfectly sharded, i.e., key updates are evenly distributed across shards, then the throughput of the system is expected to increase proportionally to the number of shards (assuming the total computing resources are also increased proportionally). However, client auditing costs will increase proportionally: clients must audit each shard assuming keys are distributed randomly across shards. If we can guarantee that each client will only be interacting with a small number of shards, then the throughput gains of sharding may come with little increase in client cost.

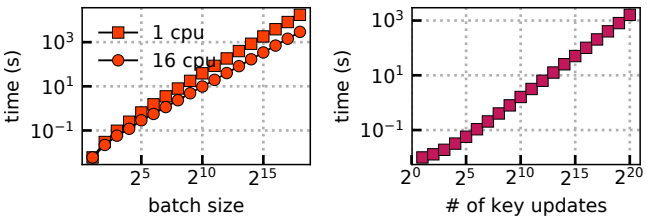

Figure 7: (Left) Batch computation of RSA membership proofs for varying levels of hardware parallelism. (Right) Update computation of an individual RSA membership proof over a range of key updates.

8.3 Key Lookups

The RSA-based constructions achieve the highest key update throughput, but also incur large costs for computing membership proofs for key lookups. Figure 7 (left) shows the time to compute membership proofs for a batch of keys. We present an approach using promises, membership proof checkpointing, and key caches to manage these costs. As a concrete example, consider a registry with 2^{30} unique keys divided into 16 caches of size $2^{15}, 2^{15}, 2^{16}, 2^{17}, \ldots, 2^{29}$. We find through batch computation, given a single thread dedicated to each cache, the membership proof promises for the smallest caches of size 2^{15} can be fulfilled approximately every 20 minutes, while promises for the largest cache of size 2^{29}, we extrapolate will be fulfilled on a schedule of around 3 months.

Keys in caches with long schedules for updating membership proofs can be updated outside of their schedule individually at higher cost (without the savings from batch computation). Figure 7 (right) shows the cost of updating an individual key's membership proof over a set of key updates. At 60 updates per second, the time to update a key whose last membership proof was 1 week ago is approximately 10 hours, and we extrapolate the cost to update a valid proof from 3 months ago to be on the order of one week.

Lastly, lookup proofs are small. The base lookup proof size is 1 KB and 0.8 KB for the MT and RSA constructions, respectively. Using some of our optimizations to minimize server costs, these increase but remain reasonable. When using the membership proof checkpointing optimization for RSA constructions, the lookup proof size grows logarithmically with the total number of epochs ($0.8 \times \log \frac{N}{C}$ KB for N total epochs and C checkpoint length). For the Merkle tree constructions, the open addressing optimization increases the size of the proof proportionally by the maximum nonce ω (16 KB for $\omega = 16$).

8.4 Summary

RSA-Aggr and RSA-Alg meet the throughput requirements for real verifiable registry workloads on our single machine experiments. In contrast, for MT-Aggr to reach similar throughput levels, it would require scaling up the computing and memory resources by $> 20\times$ [WZC+18]. On the other hand, the RSA constructions rely heavily on promises for responding to lookup queries. If fast detection of equivocation for large numbers of keys is security critical, then the RSA constructions will require a large amount of resources for membership proof computation, tipping the scale in favor of MT-Aggr.

9 Related Work

Most previous proposals for verifiable registries (or transparency logs) are constructed via Merkle trees and rely on trusting powerful global auditors that do work linear in the number of key updates to the registry [BCK+14, KHP+13, Lau14, Rya14, CDGM19, MBB+15, MKL+20]. We described a representative system, CONIKS [MBB+15], earlier; Mog [MKL+20] extends CONIKS to efficiently support the append-only invariant using chronological Merkle trees [CW09] at each leaf of the CONIKS lexicographic tree.

Merkle^2 [HHK+21] reduces the work required by global auditors per epoch to be logarithmic in the number of key updates instead of linear; auditors verify a Merkle extension proof. However, Merkle^2 fundamentally relies on a stronger assumption called *signature chains* in which updates must be signed by an authorization key not controlled by the server. We design for a recovery model (similar to CONIKS [MBB+15]) in which users can recover after losing all key material. If this stronger key update policy is desired, Merkle^2 can be adapted using our checkpointing approach to construct an extremely efficient client-auditable registry; the Merkle extension proofs provide succinct append-only proofs for arbitrary ranges of epochs.

There are a few proposals using non-Merkle-based commitments, still in the trusted auditor model. AAD [TBP+19] and Aardvark [LGG+20] use bilinear pairing-based accumulators. AAD admits logarithmic-sized verification work for the auditor per epoch, but is concretely expensive. Recently, RSA accumulators have been proposed to construct a verifiable registry with constant-sized verification work per epoch [TXN20]. However, the proposal is not efficient, requiring values to be committed bit-by-bit; our RSA constructions realize this approach efficiently using new techniques for RSA key-value commitment [AR20].

With regards to efficient client auditability, Chen et al. [CCDW20] formalize the general approach of incremental verification for ledger systems (IVLS). Our constructions using the IVC via SNARK recursion transform can be thought of as falling under this general framework. We achieve improved general performance using our SNARK aggregation approach, as well as improved performance in the specific case of registries using our RSA-based approaches.

Google Key Transparency proposed an approach for client auditability called "meet-in-the-middle" (MitM) [Goo20] that resembles our shared checkpoints approach. MitM is less efficient as it requires clients to verify checkpoints on a per-key basis, while our constructions provide security of all keys by verifying a single sequence of checkpoints for the registry.

Privacy of registry contents (specifically *selective dis-*

closure) has also been considered in prior work. Techniques to keep lookup keys private using verifiable random functions and lookup values private using commitments [MBB+15, EMBB17] can be adapted directly to all of our constructions. Our RSA constructions offer additional privacy benefits including keeping the total directory size and update patterns private [CDGM19].

Acknowledgments

Nirvan Tyagi was supported by a Facebook Graduate Research Fellowship, and part of his work took place while a visiting student at the University of Washington. Ben Fisch was funded by NSF, DARPA, and a grant from ONR. Joseph Bonneau was supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the United States Government or DARPA. Stefano Tessaro was supported by NSF grants CNS-1930117 (CAREER), CNS1926324, CNS-2026774, a Sloan Research Fellowship, and a JP Morgan Faculty Award.

References

[AM18] Mustafa Al-Bassam and Sarah Meiklejohn. Contour: A practical system for binary transparency. 11025:94–110, 2018.

[AR20] Shashank Agrawal and Srinivasan Raghuraman. KVaC: Key-value commitments for blockchains and beyond. In *ASIACRYPT (3)*, volume 12493 of *Lecture Notes in Computer Science*, pages 839–869. Springer, 2020.

[ASB+17] Ruba Abu-Salma, M. Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. Obstacles to the adoption of secure communication tools. In *IEEE Symposium on Security and Privacy*, pages 137–153. IEEE Computer Society, 2017.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 757–788. Springer, 2018.

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *CRYPTO (1)*, volume 11692 of *Lecture Notes in Computer Science*, pages 561–586. Springer, 2019.

[BBG+20] Josh Blum, Simon Booth, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Jack O'Connor, et al. E2e encryption for zoom meetings, 2020.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO (3)*, volume 11694 of *Lecture Notes in Computer Science*, pages 701–732. Springer, 2019.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*, pages 111–120. ACM, 2013.

[BCK+14] David A. Basin, Cas J. F. Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: attack resilient public-key infrastructure. In *CCS*, pages 382–393. ACM, 2014.

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In *TCC (2)*, volume 12551 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2020.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO (2)*, volume 8617 of *Lecture Notes in Computer Science*, pages 276–294. Springer, 2014.

[Ben87] Josh Daniel Cohen Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, USA, 1987.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from DARK compilers. In *EUROCRYPT (1)*, volume 12105 of *Lecture Notes in Computer Science*, pages 677–706. Springer, 2020.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. *IACR Cryptol. ePrint Arch.*, 2019:1021, 2019.

[BMM+19] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Noah Vesely. Proofs for inner pairing products and applications. *IACR Cryptology ePrint Archive*, 2019/1177, 2019.

[Bon16] Joseph Bonneau. Ethiks: Using ethereum to audit a CONIKS key transparency log. In *Financial Cryptography Workshops*, volume 9604 of *Lecture Notes in Computer Science*, pages 95–105. Springer, 2016.

[CBM15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE Symposium on Security and Privacy*, pages 321–338. IEEE Computer Society, 2015.

[CCDW20] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P. Ward. Reducing participation costs via incremental verification for ledger systems. *IACR Cryptol. ePrint Arch.*, 2020:1522, 2020.

[CDGM19] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *CCS*, pages 1639–1656. ACM, 2019.

[CGJ+17] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *CCS*, pages 719–728. ACM, 2017.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. In *EUROCRYPT (1)*, volume 12105 of *Lecture Notes in Computer Science*, pages 738–768. Springer, 2020.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2002.

[Clo20] Cloudflare. Merkle town, 2020.

[CvO13] Jeremy Clark and Paul C. van Oorschot. Sok: SSL and HTTPS: revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symposium on Security and Privacy*, pages 511–525. IEEE Computer Society, 2013.

[CW09] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334. USENIX Association, 2009.

[DSB+16] Sergej Dechand, Dominik Schürmann, Karoline Busse, Yasemin Acar, Sascha Fahl, and Matthew Smith. An empirical study of textual key-fingerprint representations. In *USENIX Security Symposium*, pages 193–208. USENIX Association, 2016.

[EMBB17] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. Certificate transparency with privacy. *Proc. Priv. Enhancing Technol.*, 2017(4):329–344, 2017.

[EPS15] Chris Evans, Chris Palmer, and Ryan Sleevi. Public key pinning extension for HTTP. *RFC*, 7469:1–28, 2015.

[FDP+14] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek, and Matthew Smith. Hey, NSA: stay away from my market! future proofing app markets against powerful attackers. In *CCS*, pages 1143–1155. ACM, 2014.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[GKK+19] Lorenzo Grassi, Daniel Kales, Dmitry Khovratovich, Arnab Roy, Christian Rechberger, and Markus Schofnegger. Starkad and poseidon: New hash functions for zero knowledge proof systems. *IACR Cryptol. ePrint Arch.*, 2019:458, 2019.

[Goo20] Google Key Transparency. Key transparency 2.0, 2020.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, 2019:953, 2019.

[HHK+21] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle^2: A low-latency transparency log system. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2021.

[KHP+13] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil D. Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In *WWW*, pages 679–690. International World Wide Web Conferences Steering Committee / ACM, 2013.

[KPS18] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 944–961. IEEE Computer Society, 2018.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.

[Lau14] Ben Laurie. Certificate transparency. *Commun. ACM*, 57(10):40–46, 2014.

[LGG+20] Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, and Nickolai Zeldovich. Aardvark: A concurrent authenticated dictionary with short proofs. *IACR Cryptol. ePrint Arch.*, 2020:975, 2020.

[LKMS04] Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis E. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, pages 121–136. USENIX Association, 2004.

[LLK13] Ben Laurie, Adam Langley, and Emilia Käsper. Certificate transparency. *RFC*, 6962:1–27, 2013.

[MB02] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *USENIX Security Symposium*, pages 297–312. USENIX, 2002.

[MBB+15] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: bringing key transparency to end users. In *USENIX Security Symposium*, pages 383–398. USENIX Association, 2015.

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

[MKL+20] Sarah Meiklejohn, Pavel Kalinnikov, Cindy S. Lin, Martin Hutchinson, Gary Belvin, Mariana Raykova, and Al Cutter. Think global, act local: Gossip and client audits in verifiable data structures. *CoRR*, abs/2011.04551, 2020.

[NKJ+17] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds. In *USENIX Security Symposium*, pages 1271–1287. USENIX Association, 2017.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983.

[OWWB20] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *USENIX Security Symposium*, pages 2075–2092. USENIX Association, 2020.

[Rya14] Mark Dermot Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*. The Internet Society, 2014.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *CRYPTO (3)*, volume 12172 of *Lecture Notes in Computer Science*, pages 704–737. Springer, 2020.

[STV+16] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In *IEEE Symposium on Security and Privacy*, pages 526–545. IEEE Computer Society, 2016.

[TBB+17] Joshua Tan, Lujo Bauer, Joseph Bonneau, Lorrie Faith Cranor, Jeremy Thomas, and Blase Ur. Can unicorns help users compare crypto key fingerprints? In *CHI*, pages 3787–3798. ACM, 2017.

[TBP+19] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *CCS*, pages 1299–1316. ACM, 2019.

[TD17] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via bitcoin. In *IEEE Symposium on Security and Privacy*, pages 393–409. IEEE Computer Society, 2017.

[TXN20] Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated dictionaries with cross-incremental proof (dis)aggregation. *IACR Cryptol. ePrint Arch.*, 2020:1239, 2020.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[VWO+17] Elham Vaziripour, Justin Wu, Mark O'Neill, Jordan Whitehead, Scott Heidbrink, Kent E. Seamons, and Daniel Zappala. Is that you, alice? A usability study of the authentication ceremony of secure messaging applications. In *SOUPS*, pages 29–47. USENIX Association, 2017.

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT (3)*, volume 11478 of *Lecture Notes in Computer Science*, pages 379–407. Springer, 2019.

[WZC+18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *USENIX Security Symposium*, pages 675–692. USENIX Association, 2018.

[YRC15] Jiangshan Yu, Mark Ryan, and Cas Cremers. How to detect unauthorised usage of a key. *IACR Cryptol. ePrint Arch.*, 2015:486, 2015.

A Merkle Tree Preliminaries

A Merkle tree is a binary tree that stores the accumulated values in the labels of its leaves. The internal nodes of the tree are given a label equal to the hash of the concatenation of its children's labels. The hash label of the root of the tree is the digest. The binding property is ensured given collision-resistance of the hash function. When using a Merkle tree as an authenticated key-value dictionary, the key is defined by the path from the root to the leaf; i.e., left children encode 0 and right children encode 1.

A membership proof for a value consists of the labels of all the sibling nodes along the path from the leaf to the root. Verification is run by using the claimed value and sibling nodes to compute the labels along the path and comparing the final label to the root digest. Proving an update to a leaf can be done similarly. Given a membership proof for a leaf's old value, the claimed updated value is used to compute new labels along the path using the same siblings and comparing the final label with the new root digest; this additionally verifies no other leaves were modified. These proofs both are of size $\mathcal{O}(\log N)$ and incur $\mathcal{O}(\log N)$ verification time for a balanced tree of size N.

A *sparse Merkle tree* allows for initializing a complete Merkle tree over a large key space of size $N = 2^h$ efficiently (in $\mathcal{O}(\log N)$ time). All leaf labels are implicitly initialized to some canonical null value. A canonical null label for internal nodes at a given height i are also computed as the hash of the concatenation of two null labels of height $i-1$. As values are added, non-null internal nodes are stored explicitly. This way, storage of a sparse Merkle tree is of $\mathcal{O}(n \log N)$ instead of $\mathcal{O}(N)$ where n is the number of non-null values.

B Open Addressing Optimization for Merkle Tree Update Circuit Representation

We present an optimization to reduce the length of the Merkle paths in the circuit from 256 required for collision-resistance to a height determined by the number of expected keys in the registry. The tree height is reduced to one in which collisions may occur and collisions are handled by remapping the colliding key to a different index using open addressing, a technique used in hash tables. This produces a more efficient circuit representation than other approaches used for path compression, e.g., Merkle Patricia tries [MBB+15, CDGM19].

More specifically, to find the index for a key, the key is hashed along with a counter nonce ω initialized to 0, $\mathsf{H}(k \,\|\, \omega)$. If the index is already populated, ω is incremented and a new index is computed until the first open index is found (up to some max increment ω_{max}). Since it is possible to find collisions for an index, each leaf now additionally encodes the key k when it is initially populated (for version-only, leafs encode $k \,\|\, v \,\|\, u$); constraints are added to ensure future updates to the leaf do not change the encoded key. This approach allows the tree height to be set based on the expected max capacity of the registry. For example, if the registry is not expected to exceed 2^{30} keys, a tree height set to 32 with $\omega_{\mathsf{max}} = 16$ leads to a failure probability of less than $1/2^{64}$. Any reduction in Merkle path length is significant as it leads to an equally proportional decrease in proving time (in this example, $4\times$).

The tradeoff to using open addressing for a faster epoch update proving time is that lookup proofs increase in size and verification time. A lookup proof for a key inserted at nonce ω includes Merkle paths for all indices derived from nonces 0 to ω, and in the case of a non-membership proof, will include all ω_{max} Merkle paths. Still, these proofs are relatively small and hashes are fast to compute, so this tradeoff is largely beneficial.

C Proof of Shared Checkpoint Epoch

Theorem 1. *For any two ranges (ℓ_1, r_1) and (ℓ_2, r_2) that are overlapping, i.e., $\ell_1 \leq \ell_2 < r_1 \leq r_2$, the two subtree partitions induced by each range share a common boundary.*

Proof. First consider the binary tree imposed over all epochs. In this binary tree, define node x as the root of the smallest subtree that contains ℓ_2 and r_1. We will show that their exists a shared boundary in the subtree partitions induced by (ℓ_1, r_1) and (ℓ_2, r_2) somewhere within this subtree rooted at x.

Consider the following two exhaustive cases (depicted in Figure 8):

Case 1: ℓ_2 is the leftmost leaf of x's subtree.

In this case, we will show that ℓ_2 itself is a shared boundary between the two subtree partitions.

If r_2 is not in x's subtree, then x's subtree (or a supertree of x's subtree if x is a left child) is included in the subtree partition of (ℓ_2, r_2). Otherwise, if r_2 is in x's subtree, r_2 would necessarily be in x's right subtree (since $r_2 \geq r_1$) and thus x's left subtree is included in the subtree partition. For the other range, since r_1 is included in x's right subtree and $\ell_1 < \ell_2$ is not in x's left subtree, then x's left subtree is included in the subtree partition of (ℓ_1, r_1).

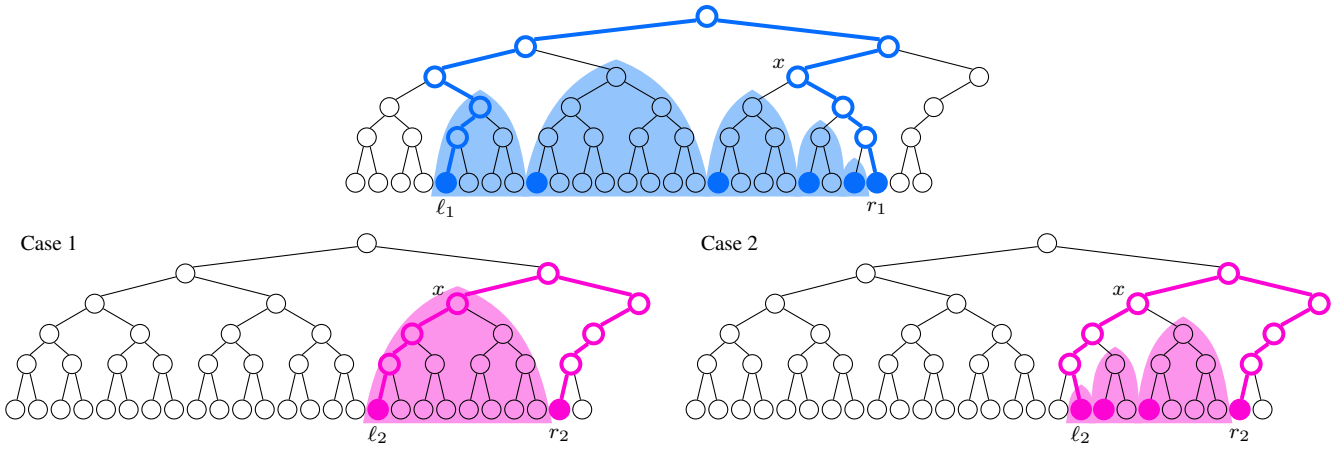

Figure 8: Cases for proof of shared checkpoint epoch. Case 1 (left) has ℓ_2 as leftmost leaf of x's subtree. Case 2 (right) does not.

Case 2: ℓ_2 is not the leftmost leaf of x's subtree.

Call y the leftmost leaf of x's right subtree. We argue that y is a shared boundary between the two subtree partitions.

First, we argue that if the right endpoint of a range is in a subtree T *and* the left endpoint is not, then the leftmost leaf of T is a boundary in the range's subtree partition. Consider two cases. First, if the right endpoint is in T's right subtree, then T's left subtree is a part of the partition and the leftmost leaf of T is a boundary. Second, if the right endpoint is in T's left subtree, then we use a recursive argument to claim that the leftmost leaf of T's left subtree (i.e., the leftmost leaf of T) is a boundary. The base case of this argument is that the right endpoint is itself the leftmost leaf of T, in which case, it is a boundary.

Now with this argument, first consider r_1 which is in x's right subtree (ℓ_1 is not), then y is a boundary of (ℓ_1, r_1). Next consider, (ℓ_2, r_2). If r_2 is in x's right subtree (ℓ_2 is not), then y is a boundary by the same argument as above. Else, r_2 is not in x's subtree, so since ℓ_2 is not the leftmost leaf of x's subtree, then x's right subtree is included in the subtree partition of (ℓ_2, r_2) and y is a boundary. □

D SNARK Aggregation

Here we present protocols for aggregating N Groth16 SNARKs [Gro16]. These protocols are simplified, more optimized versions of the related aggregation protocols presented in [BMM+19], since here we focus on the case where the SNARKs are over the same relation and setup.

Bilinear pairing groups. We will make use of the following notation for bilinear pairing groups. (1) Groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are cyclic groups of prime order p. (2) Group element g is a generator of $\mathbb{G}_1$, h is a generator of $\mathbb{G}_2$. (3) Pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a computable map with the following properties: *Bilinearity*: $\forall\ u \in \mathbb{G}_1, v \in \mathbb{G}_2$, and $a, b \in \mathbb{Z}$, $e(u^a, v^b) = e(u, v)^{ab}$, and *Non-degeneracy*: $e(g, h) \neq 1$. We assume an efficient setup algorithm that on input security parameter λ, generates a bilinear group, $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g, h, e) \leftarrow \mathcal{G}(1^\lambda)$, where $|p| = \lambda$.

Inner product arguments. We will make use of the inner product arguments TIPP and MIPP_k for the following relations; we refer the reader to [BMM+19] for details on their construction:

$$\mathcal{R}_{\mathsf{TIPP}} = \left\{ \left(\begin{array}{r} g^\beta \in \mathbb{G}_1,\ h^\alpha \in \mathbb{G}_2,\ T, U, Z \in \mathbb{G}_T,\ \gamma \in \mathbb{Z}_p\ ; \\ [w]_i = \left[g^{\alpha^{2i}}\right]_{i=0}^{m-1}, [A_i]_{i=0}^{m-1} \in \mathbb{G}_1^m, [v]_i = \left[h^{\beta^{2i}}\right]_{i=0}^{m-1}, [B_i]_{i=0}^{m-1} \in \mathbb{G}_2^m, [r]_i = [\gamma^{2i}]_{i=0}^{m-1} \in \mathbb{Z}_p^m \\ T = \prod_{i=0}^{m-1} e(A_i, v_i)\ \wedge\ U = \prod_{i=0}^{m-1} e(w_i, B_i)\ \wedge\ Z = \prod_{i=0}^{m-1} e(A_i^{r_i}, B_i) \end{array} \right) : \right\},$$

$$\mathcal{R}_{\mathsf{MIPP}\text{-}k} = \left\{ \left(\begin{array}{r} g^\beta \in \mathbb{G}_1,\ T \in \mathbb{G}_T,\ Z \in \mathbb{G}_1,\ \gamma \in \mathbb{Z}_p\ ; \\ [A_i]_{i=0}^{m-1} \in \mathbb{G}_1^m, [v]_i = \left[h^{\beta^{2i}}\right]_{i=0}^{m-1}, [b]_i = [\gamma^i]_{i=0}^{m-1} \in \mathbb{Z}_p^m \\ T = \prod_{i=0}^{m-1} e(A_i, v_i)\ \wedge\ Z = \prod_{i=0}^{m-1} A_i^{b_i} \end{array} \right) : \right\}.$$

KZG polynomial commitments. The KZG polynomial commitment scheme [KZG10] commits to polynomials of some max degree n. For polynomial $f(X) = \sum_{i=0}^{n-1} a_i X^i$ where coefficient vector $a = [a_i]_{i=0}^{n-1}$, the commitment is computed with an trapdoor commitment key $ck = [g^{\alpha^i}]_{i=0}^{n-1}$ as $\mathsf{KZG.Commit}(ck, a) = \prod_i (ck_i)^{a_i}$.

To prove that $y = f(x)$ at a point x, KZG uses the polynomial remainder theorem which says $f(x) = y \Leftrightarrow \exists q(X) : f(X) - y = q(X)(X - x)$. The proof is just a KZG commitment to the quotient polynomial $q(X)$ where if $q(X)$ has coefficients $b = [b_i]_i$, then $\mathsf{KZG.Open}(ck, a, x) = \prod_i (ck_i)^{b_i}$. The verifier key consists of $vk = h^\alpha$, and the verifier runs $\mathsf{KZG.Ver}(h^\alpha, C, W, x, y)$ for commitment C and opening W and checks that $e(CW^x/g^y, h) = e(W, h^\alpha)$.

Groth16 SNARK. We recall some relevant notation for the structure and verification of Groth16 [Gro16] SNARKS. The verifier's verification key is as follows:

$$vk_{\mathsf{G16}} = \left(h^\gamma, h^\delta, e(g^\alpha, h^\beta), \left[S_j = g^{(\beta \cdot u_j(\tau) + \alpha \cdot v_j(\tau) - w_j(\tau))/\gamma} \right]_{j=0}^{\ell-1} \right),$$

where $\alpha, \beta, \gamma, \delta, \tau \in \mathbb{Z}_p$ are secret values and $[u_j(X), v_j(X), w_j(X)]_{j=0}^{\ell-1}$ are public polynomials that define a circuit relation with a statement of ℓ elements of $\mathbb{Z}_p$. A proof consists of three group elements, $\pi = (A, B, C) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1$ and is verified with a statement $[x_i]_{i=0}^{\ell-1}$ by checking the following pairing product equation:

$$e(A, B) \stackrel{?}{=} e(g^\alpha, h^\beta) \cdot e\left(\prod_{j=0}^{\ell-1} S_j^{x_j}, h^\gamma \right) \cdot e(C, h^\delta).$$

Aggregation. [BMM+19] describe how to aggregate the verification of a vector of proofs $[\pi_i = (A_i, B_i, C_i)]_{i=0}^{n-1}$ for statements $\left[[x_{i,j}]_{i=0}^{n-1}\right]_{j=0}^{\ell-1}$ into a single pairing product equation by combining them with a random linear combination. More specifically, the verifier samples a random $r \leftarrow\!\!\$\, \mathbb{Z}_p$ and then checks:

$$\prod_{i=0}^{n-1} e\Big((A_i)^{r^i}, B_i \Big) \stackrel{?}{=} e\big(g^\alpha, h^\beta\big)^{\sum_{i=0}^{n-1} r^i} \cdot e\left(\prod_{j=0}^{\ell-1} S_j^{\sum_{i=0}^{n-1} x_{i,j} \cdot r^i}, h^\gamma \right) \cdot e\left(\prod_{i=0}^{n-1} (C_i)^{r^i}, h^\delta \right).$$

We present the details of an aggregation proof that proves that this check succeeds in Figure 9; it is for the following relation:

$$\mathcal{R}_{\mathsf{G16\text{-}Aggr}} = \left\{ \left(vk_{\mathsf{G16}},\ \left[[x_{i,j}]_{j=0}^{\ell-1} \right]_{i=0}^{n-1};\quad [\pi_i]_{i=0}^{n-1} \right) :\ \bigwedge_{i=0}^{n-1} \mathsf{G16.Ver}(vk, [x_{i,j}]_{j=0}^{\ell-1}, \pi_i) \right\}.$$

Aggregation with sequential statements. Verification of the general aggregation protocol from Figure 9 is $\mathcal{O}(\ell \cdot n)$ time since the verifier must compute all of the "aggregate" Z_j values from the n statements $[x_i]_i$. Here we present a modified aggregation protocol that allows for $\mathcal{O}(\ell + \log n)$ verification time for statements that follow a specific "sequential" structure.

The sequential structure that we require is that each statement x_i is made up of two parts $x_i = (a_i, b_i)$. The first part is shared as the second part of the previous statement, $x_{i-1} = (a_{i-1}, b_{i-1})$ where $b_{i-1} = a_i$, and the second part is shared as the first part of the following statement, $x_{i+1} = (a_{i+1}, b_{i+1})$ where $a_{i+1} = b_i$. In other words, there exists a sequence of values $[a_i]_{i=0}^{n}$ such that the statements are of the form $[x_i = (a_i, a_{i+1})]_{i=0}^{n-1}$. Here, each $a_i = [a_{i,j}]_{j=0}^{\ell-1}$ is a vector of ℓ field elements, and the SNARKs are over statements of 2ℓ elements.

In this case, we can use the structure of the statements to efficiently prove knowledge of accepting intermediate statements without requiring the verifier to themselves check linear statements. We provide an aggregation protocol for the following relation with details given in Figure 10:

$$\mathcal{R}_{\mathsf{G16\text{-}Aggr\text{-}Seq}} = \left\{ \left(vk_{\mathsf{G16}},\ [a_{0,j}]_{j=0}^{\ell-1},\ [a_{n,j}]_{j=0}^{\ell-1};\ \left[[a_{i,j}]_{j=0}^{\ell-1} \right]_{i=1}^{n-1}, [\pi_i]_{i=0}^{n-1} \right) :\ \bigwedge_{i=0}^{n-1} \mathsf{G16.Ver}(vk_{\mathsf{G16}}, ([x_i = ([a_{i,j}], [a_{i+1,j}])]_{j=0}^{\ell-1}, \pi_i) \right\}.$$

E Version-only Proofs and Batch Updates for RSA Key-Value Commitment

Groups of unknown order. We assume the existence of a randomized polynomial time sampling algorithm $\mathsf{GGen}(\lambda)$ that takes

$\underline{\mathsf{G16\text{-}Aggr.Setup}(n)}$

1. Generate commitment keys: $\alpha, \beta \leftarrow\!\!\$\, \mathbb{Z}_p$, $w \leftarrow \left[g^{\alpha^{2i}}\right]_{i=0}^{n-1}$, $v \leftarrow \left[h^{\beta^{2i}}\right]_{i=0}^{n-1}$.
2. Generate shared verification key and proving key for TIPP and MIPP_k:
   $\left(vk = (g^\beta, h^\alpha), pk = (vk, \left[g^{\alpha^i}\right]_{i=0}^{2n-2}, \left[h^{\beta^i}\right]_{i=0}^{2n-2})\right) \leftarrow \mathsf{TIPP.Setup}(n, (\alpha, \beta))$.
3. Return (vk, pk). (Notice w, v included in pk)

$\underline{\mathsf{G16\text{-}Aggr.Aggregate}(pk, [(x_i, \pi_i = (A_i, B_i, C_i))]_{i=0}^{n-1})}$

1. Compute $(\pi, r) \leftarrow \mathsf{AggregateHelper}(pk, [x_i]_{i=0}^{n-1}, [\pi_i]_{i=0}^{n-1})$, and return π.

$\underline{\mathsf{G16\text{-}Aggr.Ver}(vk, vk_{\mathsf{G16}}, \left[x_i = [x_{i,j}]_{j=0}^{\ell-1}\right]_{i=0}^{n-1}, \pi)}$

1. Parse $\left(h^\gamma, h^\delta, e(g^\alpha, h^\beta), [S_j]_{j=0}^{\ell-1}\right) \leftarrow vk_{\mathsf{G16}}$ and $((C_A, C_B, C_C), (Z_{AB}, Z_C), (\pi_{AB}, \pi_C)) \leftarrow \pi$.
2. Compute $r \leftarrow \mathsf{H}([x_i]_{i=0}^{n-1}, C_A, C_B, C_C)$ and $[Z_j]_j \leftarrow \left[S_j^{\sum_{i=0}^{n-1} x_{i,j} \cdot r^i}\right]_{j=0}^{\ell-1}$.
3. Return $\mathsf{VerifyHelper}(vk, vk_{\mathsf{G16}}, \pi, [Z_j]_{j=0}^{\ell-1}, r)$.

$\underline{\mathsf{AggregateHelper}(pk, x, [\pi_i = (A_i, B_i, C_i)]_{i=0}^{n-1})}$

1. Parse w, v and g^β, h^α from pk.
2. Commit to proof elements: $C_A = \prod_{i=0}^{n-1} e(A_i, v_i)$, $C_B = \prod_{i=0}^{n-1} e(w_i, B_i)$, $C_C = \prod_{i=0}^{n-1} e(C_i, v_i)$.
3. Compute challenge $r \leftarrow \mathsf{H}([x_i]_{i=0}^{n-1}, C_A, C_B, C_C)$.
4. Compute inner products $Z_{AB} \leftarrow \prod_{i=0}^{n-1} e((A_i)^{r^i}, B_i)$, $Z_C \leftarrow \prod_{i=0}^{n-1} (C_i)^{r^i}$.
5. Prove using TIPP and MIPP_k correct computation of inner products with respect to commitments:
   $\pi_{AB} \leftarrow \mathsf{TIPP.Prove}\big(pk, (g^\beta, h^\alpha, C_A, C_B, Z_{AB}, r), (w, [A_i]_i, v, [B_i]_i, [r^i]_i)\big)$,
   $\pi_C \leftarrow \mathsf{MIPP}_k\mathsf{.Prove}\big(pk, (g^\beta, C_C, Z_C, r), (v, [C_i]_i, [r^i]_i)\big)$.
6. Return $\pi \leftarrow ((C_A, C_B, C_C), (Z_{AB}, Z_C), (\pi_{AB}, \pi_C))$.

$\underline{\mathsf{VerifyHelper}(vk, vk_{\mathsf{G16}}, \pi, [Z_j]_{j=0}^{\ell-1}, r)}$

1. Parse $\left(h^\gamma, h^\delta, e(g^\alpha, h^\beta), [S_j]_{j=0}^{\ell-1}\right) \leftarrow vk_{\mathsf{G16}}$ and $((C_A, C_B, C_C), (Z_{AB}, Z_C), (\pi_{AB}, \pi_C)) \leftarrow \pi$.
2. Check inner product proofs:
   $\mathsf{TIPP.Ver}\big(vk, (g^\beta, h^\alpha, C_A, C_B, Z_{AB}, r), \pi_{AB}\big) \stackrel{?}{=} 1$,
   $\mathsf{MIPP}_k\mathsf{.Ver}\big(pk, (g^\beta, C_C, Z_C, r), \pi_C\big) \stackrel{?}{=} 1$.
3. Check aggregate pairing product equation:
   $Z_{AB} \stackrel{?}{=} e(g^\alpha, h^\beta)^{\frac{r^n - 1}{r - 1}} \cdot e(\prod_{j=0}^{\ell-1} Z_j, h^\gamma) \cdot e(Z_C, h^\delta)$.
4. Return 1 if above checks pass otherwise 0.

Figure 9: Aggregation of Groth16 [Gro16] SNARKs.

$\mathsf{G16\text{-}Aggr\text{-}Seq.Setup}(n)$

1. Generate $(vk_{\mathsf{G16\text{-}Aggr}}, pk_{\mathsf{G16\text{-}Aggr}}) \leftarrow \mathsf{G16\text{-}Aggr.Setup}(n)$ and $(vk_{\mathsf{KZG}}, ck_{\mathsf{KZG}}) \leftarrow \mathsf{KZG.Setup}(n)$.
2. Return $(vk = (vk_{\mathsf{G16\text{-}Aggr}},\ vk_{\mathsf{KZG}}), pk = (pk_{\mathsf{G16\text{-}Aggr}}, ck_{\mathsf{KZG}}))$.

$\mathsf{G16\text{-}Aggr\text{-}Seq.Aggregate}(pk, \left[[a_{i,j}]_{j=0}^{\ell-1}\right]_{i=0}^{n}, [(\pi_i = (A_i, B_i, C_i))]_{i=0}^{n-1})$

1. Compute $(\pi_{\mathsf{Aggr}}, r) \leftarrow \mathsf{AggregateHelper}(pk_{\mathsf{G16\text{-}Aggr}}, (a_0, a_n), [\pi_i]_{i=0}^{n-1})$.
2. Commit to statements: $\left[C_{S,j}\right]_j \leftarrow \left[\mathsf{KZG.Commit}(ck_{\mathsf{KZG}}, [a_{i,j}]_{i=0}^{n-1})\right]_{j=0}^{\ell-1}$.
3. Compute statement scalar inner products (exponents of S_j): $[z_j]_j \leftarrow \left[\sum_{i=0}^{n-1} a_{i,j} \cdot r^i\right]_{j=0}^{\ell-1}$.
4. Prove correct computation of inner product by opening KZG commitment:
   $[W_j]_j \leftarrow \left[\mathsf{KZG.Open}(ck_{\mathsf{KZG}}, [a_{i,j}]_{i=0}^{n-1}, r)\right]_{j=0}^{\ell-1}$.
5. Return $\pi \leftarrow (\pi_{\mathsf{Aggr}}, \left[(C_{S,j}, W_j, z_j)\right]_{j=0}^{\ell-1})$.

$\mathsf{G16\text{-}Aggr\text{-}Seq.Ver}(vk, vk_{\mathsf{G16}}, [a_{0,j}]_{j=0}^{\ell-1}, [a_{n,j}]_{j=0}^{\ell-1}, \pi)$

1. Parse $\left(h^\gamma, h^\delta, e(g^\alpha, h^\beta), [S_j]_{j=0}^{\ell-1}\right) \leftarrow vk_{\mathsf{G16}}$, $(\pi_{\mathsf{Aggr}}, \left[(C_{S,j}, W_j, z_j)\right]_{j=0}^{\ell-1}) \leftarrow \pi$, and
   $((C_A, C_B, C_C), (Z_{AB}, Z_C), (\pi_{AB}, \pi_C)) \leftarrow \pi_{\mathsf{Aggr}}$.
2. Compute challenge $r \leftarrow \mathsf{H}([x_i]_{i=0}^{n-1}, C_A, C_B, C_C)$.
3. Compute statement elements taking advantage of sequential property:
   $[Z_j]_{j=0}^{\ell-1} \leftarrow \left[S_j^{z_j}\right]_{j=0}^{\ell-1}$, $[Z_j]_{j=\ell}^{2\ell-1} \leftarrow \left[S_j^{\frac{z_j - a_{0,j}}{r} + a_{n,j} \cdot r^{n-1}}\right]_{j=0}^{\ell-1}$.
4. Check $\mathsf{VerifyHelper}(vk_{\mathsf{G16\text{-}Aggr}}, vk_{\mathsf{G16}}, \pi_{\mathsf{Aggr}}, [Z_j]_{j=0}^{2\ell-1}, r) \stackrel{?}{=} 1$.
5. Check KZG proofs: $\left[\mathsf{KZG.Ver}(vk_{\mathsf{KZG}}, C_{S,j}, W_j, r, z_j)\right]_{j=0}^{\ell-1} \stackrel{?}{=} [1]_j$.
6. Return 1 if above checks pass otherwise 0.

Figure 10: Aggregation of Groth16 [Gro16] SNARKs with sequential statements.

$\underline{\mathsf{KVaC.Setup}(\lambda)}$
$(a, b, \mathbb{G}) \leftarrow\!\!\$\ \mathsf{GGen}(\lambda)$
$g \leftarrow\!\!\$\ \mathbb{G}$
Return $(a, b, \mathbb{G}, g)$

$\underline{\mathsf{KVaC.Init}()}$
Return $(1, g)$

$\underline{\mathsf{KVaC.Commit}([(k, v, u)]_i)}$
$[z]_i \leftarrow [\mathsf{H}(k)]_i$
$C_1 \leftarrow g^{\sum_j \left(v_j z_j^{u_j - 1} \prod_{i \neq j} z_i^{u_i} \right)}$
$C_2 \leftarrow g^{\prod_i z_i^{u_i}}$
Return (C_1, C_2)

$\underline{\mathsf{KVaC.ProveMem}([(k, v, u)]_i, m)}$
$[z]_i \leftarrow [\mathsf{H}(k)]_i$
$\pi_1 \leftarrow g^{\sum_{j \neq m} \left(v_j z_j^{u_j - 1} \prod_{i \neq j, m} z_i^{u_i} \right)}$
$\pi_2 \leftarrow g^{\prod_{i \neq m} z_i^{u_i}}$
$(a, b) \leftarrow \mathsf{EEA}(\prod_{i \neq m} z_i^{u_i}, z_m)$
$\pi \leftarrow ((\pi_1, \pi_2), (g^b, a), u_m)$
Return π

$\underline{\mathsf{KVaC.VerifyMem}(C, (k, v), \pi)}$
$z \leftarrow \mathsf{H}(k)$
$((\pi_1, \pi_2), (B, a), u) \leftarrow \pi$
$(C_1, C_2) \leftarrow C$
Return $\bigwedge \begin{pmatrix} (\pi_1)^{z^u} (\pi_2)^{v \cdot z^{u-1}} = C_1 \\ (\pi_2)^{z^u} = C_2 \\ (\pi_2)^a B^z = g \end{pmatrix}$

$\underline{\mathsf{KVaC.Update}(C, (k, \delta))}$
$z \leftarrow \mathsf{H}(k)$
$(C_1, C_2) \leftarrow C$
$C' \leftarrow (C_1^z C_2^\delta, C_2^z)$
Return C'

$\underline{\mathsf{KVaC.UpdateMemProof}((k, \pi), (k_\delta, \delta))}$
$z \leftarrow \mathsf{H}(k)$
$((\pi_1, \pi_2), (B, a), u) \leftarrow \pi$
If $k = k_\delta$ then
  $\pi' \leftarrow ((\pi_1, \pi_2), (B, a), u + 1)$
Else
  $z_\delta \leftarrow \mathsf{H}(k_\delta)$
  $(s, t) \leftarrow \mathsf{EEA}(z, z_\delta)$
  $q \leftarrow \lfloor \frac{at}{z} \rfloor$; $r \leftarrow at \mod z$
  $a' \leftarrow r$; $B' \leftarrow \pi_2^{at + qZ} B$
  $\pi' \leftarrow ((\pi_1^{z_\delta} \pi_2^\delta, \pi_2^{z_\delta}), (B', a'), u)$
Return π'

Figure 11: KVaC construction from [AR20].

as input the security parameter λ and generates a group of unknown order consisting of two integers a, b along with a description of the group $\mathbb{G}$. The group $\mathbb{G}$ is of unknown order in the range $[a, b]$ where a, b, and $a - b$ are all exponential in λ.

The RSA quotient group $\mathbb{Z}_N^\times \setminus \{\pm 1\}$ where N is an RSA modulus is believed to have no element of known order other than the identity. The group generation algorithm here may require trusted setup to generate the group modulus N.

Adaptive root assumption. The adaptive root assumption tasks an adversary with computing the random root of a non-trivial group element. We define the advantage of an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ against the adaptive root assumption as follows:

$$\mathbf{Adv}^{\text{adap-root}}_{\mathsf{GGen}, \mathcal{A}}(\lambda) = \Pr\left[u^\ell = w \neq 1 \ : \begin{array}{l} (a, b, \mathbb{G}) \leftarrow\!\!\$\ \mathsf{GGen}(\lambda); \\ (w, st) \leftarrow\!\!\$\ \mathcal{A}_0(a, b, \mathbb{G}); \\ \ell \leftarrow\!\!\$\ \mathsf{Primes}(\lambda); \\ u \leftarrow\!\!\$\ \mathcal{A}_1(st, \ell) \end{array} \right].$$

Strong RSA assumption. The strong RSA assumption tasks an adversary with computing a chosen non-trivial root of a random group element. We define the advantage of an adversary $\mathcal{A}$ against the strong RSA assumption as follows:

$$\mathbf{Adv}^{\text{strong-rsa}}_{\mathsf{GGen}, \mathcal{A}}(\lambda) = \Pr\left[\begin{array}{l} u^\ell = w \\ \ell \in \mathsf{Primes}(\lambda) \setminus \{2\} \end{array} \ : \begin{array}{l} (a, b, \mathbb{G}) \leftarrow\!\!\$\ \mathsf{GGen}(\lambda); \\ w \leftarrow\!\!\$\ \mathbb{G}; \\ (u, \ell) \leftarrow\!\!\$\ \mathcal{A}(a, b, \mathbb{G}, w) \end{array} \right].$$

Extended Euclidean algorithm. Given two integers x, y such that the $\gcd(x, y) = 1$, then $(a, b) \leftarrow \mathsf{EEA}(x, y)$ returns the Bézout coefficients (a, b) where $ax + by = 1$. The coefficients are such that $a \leq y$ and $b \leq x$. The algorithm runs in time $\mathcal{O}(\max(|x|, |y|))$.

E.1 RSA Key-Value Commitment

We make use of the key-value commitment KVaC from [AR20]; the construction pseudocode is given in Figure 11. The hash function H maps keys to primes of size 2^λ that are larger than the group order upper bound b. The space of values that can be committed to is the set of positive integers bounded above by b. [AR20] prove KVaC secure with respect to a weak key binding property in which the commitment must have been produced correctly, rather than adversarially. This is not sufficient for the verifiable registry setting; in the next section we show how to augment KVaC with update proofs to protect against adversarially generated commitments.

$\underline{\mathsf{KVaC.BatchUpdate}(C,[(k,\delta)]_i))}$
$[z]_i \leftarrow [\mathsf{H}(k)]_i$
$(C_1,C_2) \leftarrow C$
$Z \leftarrow \prod_i z_i$
$\Delta \leftarrow \sum_j \left(\delta_j \prod_{i\neq j} z_i\right)$
$C' \leftarrow (C_1^Z C_2^\Delta, C_2^Z)$
Return C'

$\underline{\mathsf{KVaC.UpdateMemProof}((k,\pi),(Z,\Delta))}$
$z \leftarrow \mathsf{H}(k)$
$((\pi_1,\pi_2),(B,a),u) \leftarrow \pi$
$(s,t) \leftarrow \mathsf{EEA}(z,Z)$
$q \leftarrow \lfloor \frac{at}{z} \rfloor$; $r \leftarrow at \mod z$
$a' \leftarrow r$; $B' \leftarrow \pi_2^{at+qZ} B$
$\pi' \leftarrow ((\pi_1^Z \pi_2^\Delta, \pi_2^Z),(B',a'),u)$
Return π'

$\underline{\mathsf{KVaC.ProveUpdate}(C,C',(Z,\Delta))}$
$(C_1,C_2) \leftarrow C$; $(C_1',C_2') \leftarrow C'$
$\pi \leftarrow \mathsf{BBF.Prove}((Z,\Delta),(C_1,C_2,C_1',C_2'))$
Return π

$\underline{\mathsf{KVaC.VerUpdate}(C,C',\pi)}$
$(C_1,C_2) \leftarrow C$; $(C_1',C_2') \leftarrow C'$
Return $\mathsf{BBF.Ver}((C_1,C_2,C_1',C_2'),\pi)$

$\mathcal{R}_{\mathsf{KVaC}} = \left\{((X_1,X_2,Y_1,Y_2);(\alpha,\beta)) : Y_1 = X_1^\alpha X_2^\beta \wedge Y_2 = X_2^\alpha\right\}$

$\underline{\mathsf{BBF.Prove}((\alpha,\beta),(X_1,X_2,Y_1,Y_2))}$
$s_a \leftarrow g^\alpha$; $s_b \leftarrow g^\beta$
$\ell \leftarrow \mathsf{H}_{\mathsf{Primes}}(X_1 \,\|\, X_2 \,\|\, Y_1 \,\|\, Y_2 \,\|\, s_a \,\|\, s_b)$
$q_a \leftarrow \lfloor \alpha/\ell \rfloor$; $r_a \leftarrow \alpha \mod \ell$
$q_b \leftarrow \lfloor \beta/\ell \rfloor$; $r_b \leftarrow \beta \mod \ell$
$W_a \leftarrow g^{q_a}$; $W_b \leftarrow g^{q_b}$
$W_1 \leftarrow X_1^{q_a} X_2^{q_b}$; $W_2 \leftarrow X_2^{q_a}$
$\pi \leftarrow (W_a,W_b,W_1,W_2,r_a,r_b,\ell)$
Return π

$\underline{\mathsf{BBF.Ver}((X_1,X_2,Y_1,Y_2),\pi)}$
$\pi \leftarrow (W_a,W_b,W_1,W_2,r_a,r_b,\ell)$
$s_a \leftarrow W_a^\ell g^{r_a}$; $s_b \leftarrow W_b^\ell g^{r_b}$
Return $\bigwedge \begin{pmatrix} \ell = \mathsf{H}_{\mathsf{Primes}}(X_1 \,\|\, X_2 \,\|\, Y_1 \,\|\, Y_2 \,\|\, s_a \,\|\, s_b) \\ Y_1 = W_1^\ell X_1^{r_a} X_2^{r_b} \\ Y_2 = W_2^\ell X_2^{r_a} \end{pmatrix}$

Figure 12: Extension for KVaC to batch many key updates together (left). Extension to prove that key updates satisfy a versioned invariant (center) using the generalized proof of linear homomorphism from [BBF19], shown for the particular update homomorphism relevant to KVaC (right).

$\underline{\text{Game } \mathrm{G}^{\text{version}}_{\Pi,\mathcal{A}}(\lambda)}$
$pp \leftarrow\!\$\, \Pi.\mathsf{Setup}(\lambda)$; $C_\varnothing \leftarrow\!\$\, \Pi.\mathsf{Init}^{pp}()$
$(k,(v_A,u_A,\pi_A,[(C_A,\lambda_A)]_{i=1}^n),(v_B,u_B,\pi_B,[(C_B,\lambda_B)]_{i=1}^m) \leftarrow\!\$\, \mathcal{A}(pp,C_\varnothing)$
Return $\bigwedge \begin{pmatrix} C_{A,0} \leftarrow C_\varnothing \;;\; \bigwedge_{i=1}^n \Pi.\mathsf{VerUpdate}(C_{A,i-1},C_{A,i},\lambda_{A,i}) \\ C_{B,0} \leftarrow C_{A,n} \;;\; \bigwedge_{i=1}^m \Pi.\mathsf{VerUpdate}(C_{B,i-1},C_{B,i},\lambda_{B,i}) \\ \Pi.\mathsf{VerifyMem}(C_{A,n},(k,v_A,u_A),\pi_A) \\ \Pi.\mathsf{VerifyMem}(C_{B,m},(k,v_B,u_B),\pi_B) \\ \bigvee \begin{pmatrix} u_A > u_B \\ v_A \neq v_B \wedge u_A = u_B \end{pmatrix} \end{pmatrix}$

Figure 13: Security game for versioned invariant preservation through updates.

E.2 Versioned Update Proofs

Figure 12 shows our protocol for proving updates preserve a versioned invariant. We use the generalized proof of linear homomorphism [BBF19] to prove that the commitment is updated only by a particular homomorphism that we show guarantees a versioned invariant. The proof of knowledge from [BBF19] is sound with respect to the adaptive root assumption. We also show (in Figure 12) how to batch many key-value updates together such that the batched update follows the same homomorphic form as a single update. Individual membership proofs can be updated with respect to batched changes.

Here we prove that the update proofs from Figure 12 enforce the versioned invariant is preserved. We say a key-value commitment is *versioned* if it is not possible to provide two membership proofs for the same key that break the versioned invariant. That is, (1) the key's version number does not decrease, and (2) two different values for a key cannot be shown for the same version number. The versioned property is defined by the security game in Figure 13. The advantage of an adversary is defined as $\mathbf{Adv}^{\text{version}}_{\Pi,\mathcal{A}}(\lambda) = \Pr[\mathrm{G}^{\text{version}}_{\Pi,\mathcal{A}}(\lambda) = 1]$.

Theorem 2. *For any adversary $\mathcal{A}$ against the versioned property of* KVaC*, we give adversaries $\mathcal{B}$ and $\mathcal{C}$ such that*

$$\mathbf{Adv}^{\textit{version}}_{\mathsf{KVaC},\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{\textit{strong-rsa}}_{\mathsf{GGen},\mathcal{B}}(\lambda) + \mathbf{Adv}^{\textit{sound}}_{\mathsf{BBF},\mathcal{C},\mathcal{X}}(\lambda),$$

where GGen *is the group generation algorithm for the RSA quotient group used in* KVaC *and $\mathcal{X}$ is the knowledge extractor for* BBF *[BBF19].*

Proof. First, observe the update structure of KVaC. Using the extractor $\mathcal{X}$ for BBF, we extract the series of updates $[\alpha_A,\beta_A]_i^n$

corresponding to update proofs $[\lambda_A]_i^n$. If the extractor fails, we build adversary $\mathcal{C}$ against the soundness of BBF. Else, we can rewrite $C_A = C_{A,n}$ as follows:

$$C_A = \left(g^{\sum_i^n (\beta_{A,i} \prod_{j \neq i} \alpha_{A,j})}, g^{\prod_i^n \alpha_{A,i}}\right)$$

This can be considered as a single update from $C_\varnothing = (1, g)$ where

$$\alpha_A = \prod_i^n \alpha_{A,i} \qquad \beta_A = \sum_i^n \left(\beta_{A,i} \prod_{j \neq i} \alpha_{A,j}\right)$$

Similarly, $C_B = C_{B,m}$ can be considered as a single update from C_A:

$$C_B = \left(C_{A,1}^{\alpha_B} C_{A,2}^{\beta_B}, C_{A,2}^{\alpha_B}\right) \qquad \alpha_B = \prod_i^m \alpha_{B,i} \qquad \beta_B = \sum_i^m \left(\beta_{B,i} \prod_{j \neq i} \alpha_{B,j}\right)$$

This means that we can simplify the proof of the versioned property to consider just a single update from $C_\varnothing$ to C_A of (α_A, β_A) and a single update from C_A to C_B of (α_B, β_B).

First, we will prove some useful lemmas.

Lemma 1. *[Shamir's trick] For any integer modulo N, given integers $u, v \in \mathbb{Z}_N^\times$ and $x, y \in \mathbb{Z}$, such that $u^x = v^y \mod N$ and $\gcd(x, y) = 1$, it is efficient to compute $w \in \mathbb{Z}_N^\times$ where $w^a = v \mod N$.*

Proof. Since $\gcd(x, y) = 1$, we can compute the Bézout coefficients $(a, b) \leftarrow \mathsf{EEA}(x, y)$ where $ax + by = 1$. Let $w = u^b v^a \mod n$, then

$$w^x = u^{bx} v^{ax} = (u^x)^b v^{ax} = (v^y)^b v^{ax} = v \pmod N.$$

□

Lemma 2. *[Non-trivial root of unity] For RSA quotient group $\mathbb{G}$ with elements of unknown order bounded above by b, given integers $u, v \in \mathbb{G}$ and prime $z > b$, if $u^z = v^z$, then $u = v$.*

Proof. Let $\alpha = u/v \in \mathbb{G}$. Then $\alpha^z = 1$. Since z is prime, if $\alpha \neq 1$, then z must be the order of α in $\mathbb{G}$. However, $z > b$, an upper bound on the order of elements in $\mathbb{G}$, which is not possible, so $\alpha = 1$ and $u = v$. □

Lemma 3. *[Coprime] For RSA quotient group $\mathbb{G}$, given integers $u, w \in \mathbb{G}$, random integer $v \in \mathbb{G}$, integers $a, b, c \in \mathbb{Z}$, and prime z, then if $u^{z^c} = v^a$ and $u^b w^z = v$, then $z^c \mid a$ and if let $d = a/z^c \in \mathbb{Z}$, then $u = v^d$ and $\gcd(z, d) = 1$.*

Proof. First, we prove that d exists, i.e., that $z^c \mid a$. Consider $(u^{z^{c-1}})^z = v^a$. If $z \nmid a$, then $\gcd(z, a) = 1$ and by Lemma 1, we can compute $x^z = v$ which wins the strong RSA security game. Therefore $z \mid a$ and $u^{z^{c-1}} = g^{a/z}$ by Lemma 2. We can repeat this argument for $(u^{z^{c-i}})^z = v^{a/z^{i-1}}$ for $i \in [2, c]$, ultimately arriving at $z^c \mid a$ and $u = v^{a/z^c} = v^d$.

Next, we show that $z \nmid d$. Consider $u^b w^z = v$ rewritten as $v^{bd-1} = w^{-z}$. If $z \mid d$, then $\gcd(bd - 1, -z) = 1$, and by Lemma 1, we can compute $x^z = v$ which again wins the strong RSA security game. Therefore, $z \nmid d$ meaning $\gcd(z, d) = 1$. □

The proof proceeds by considering each of the two winning conditions and showing that, in each case, a winning adversary can break strong RSA.

(1) $u_A > u_B$

(2) $v_A \neq v_B \wedge u_A = u_B$

Case 1: $u_A > u_B$

From the verification equations of π_A, we have that:

$$\pi_{A,2}^{z^{u_A}} = C_{A,2} = g^{\alpha_A}, \qquad \pi_{A,2}^{\pi_{A,4}} \pi_{A,3}^z = g.$$

Thus, by Lemma 3, we know that $\pi_{A,2} = g^{\alpha_A / z^{u_A}}$. Similarly, from the verification equations of π_B, we have that:

$$\pi_{B,2}^{z^{u_B}} = C_{B,2} = g^{\alpha_A \alpha_B}, \qquad \pi_{B,2}^{\pi_{B,4}} \pi_{B,3}^z = g.$$

Again, by Lemma 3, we have that $\pi_{B,2} = g^{\alpha_A \alpha_B / z^{u_B}}$ and $\gcd(\alpha_A \alpha_B / z^{u_B}, z) = 1$. Since $u_A > u_B$, we can construct group element u as follows:

$$u = \pi_{A,2}^{\alpha_B \cdot z^{u_A - u_B - 1}} \qquad \text{and then,} \qquad u^z = (\pi_{A,2}^{\alpha_B \cdot z^{u_A - u_B - 1}})^z = ((g^{\alpha_A / z^{u_A}})^{\alpha_B \cdot z^{u_A - u_B - 1}})^z = g^{\alpha_A \alpha_B / z^{u_B}}.$$

Since $\gcd(\alpha_A \alpha_B / z^{u_B}, z) = 1$, we can compute w from Lemma 1, where $w^z = g$ which wins the strong RSA security game.

Case 2: $v_A \neq v_B \wedge u_A = u_B$

Let $u = u_A = u_B$. By the verification equation of π_B, we have:

$$C_{B,1} = \pi_{B,1}^{z^u} \pi_{B,2}^{v_B z^{u-1}}$$

We also have, from the update proof and verification equations of π_A, that:

$$\begin{aligned} C_{B,1} &= C_{A,1}^{\alpha_B} C_{A,2}^{\beta_B} \\ &= \left(\pi_{A,1}^{\alpha_B z^u} \pi_{A,2}^{\alpha_B v_A z^{u-1}}\right)\left(\pi_{A,2}^{\beta_B z^u}\right) \end{aligned}$$

We also can derive the following relation:

$$\begin{aligned} \pi_{A,2}^{z^u} &= C_{A,2} && \text{(by verification of } \pi_A\text{)} \\ \pi_{A,2}^{\alpha_B z^u} &= C_{A,2}^{\alpha_B} = C_{B,2} \\ \pi_{B,2}^{z^u} &= C_{B,2} && \text{(by verification of } \pi_B\text{)} \\ \pi_{A,2}^{\alpha_B} &= \pi_{B,2} && \text{(by repeated application of Lemma 2)} \end{aligned}$$

Putting this together we have as follows:

$$\begin{aligned} \pi_{A,1}^{\alpha_B z^u} \pi_{A,2}^{\alpha_B v_A z^{u-1}} \pi_{A,2}^{\beta_B z^u} &= \pi_{B,1}^{z^u} \pi_{B,2}^{v_B z^{u-1}} && \text{(by equality to } C_{B,1}\text{)} \\ \frac{\pi_{A,1}^{\alpha_B z^u} \pi_{A,2}^{\beta_B z^u}}{\pi_{B,1}^{z^u}} &= \frac{\pi_{B,2}^{v_B z^{u-1}}}{\pi_{A,2}^{\alpha_B v_A z^{u-1}}} \\ \frac{\pi_{A,1}^{\alpha_B z^u} \pi_{A,3}^{\beta_B z^u}}{\pi_{B,1}^{z^u}} &= \pi_{B,2}^{(v_B - v_A) z^{u-1}} && \text{(by relation between } \pi_{B,2} \text{ and } \pi_{A,2}\text{)} \\ \left(\left(\frac{\pi_{A,1}^{\alpha_B} \pi_{A,2}^{\beta_B}}{\pi_{B,1}}\right)^z\right)^{z^{u-1}} &= \left(\pi_{B,2}^{v_B - v_A}\right)^{z^{u-1}} \\ \left(\frac{\pi_{A,1}^{\alpha_B} \pi_{A,2}^{\beta_B}}{\pi_{B,1}}\right)^z &= \pi_{B,2}^{v_B - v_A} && \text{(by repeated application of Lemma 2)} \end{aligned}$$

Thus, we have found a z^{th} root of a non-trivial element. By Lemma 3, we have that $\pi_{B,2} = g^{\alpha_A \alpha_B / z^u}$ where $\gcd(\alpha_A \alpha_B / z^u, z) = 1$. This gives us

$$\left(\frac{\pi_{A,1}^{\alpha_B} \pi_{A,2}^{\beta_B}}{\pi_{B,1}}\right)^z = g^{\frac{(v_B - v_A)\alpha_A \alpha_B}{z^u}}.$$

Since z is prime and the domain of values is chosen to be smaller than all z, we also have that $\gcd(v_A - v_B, z) = 1$, and therefore by Lemma 1, we can compute w where $w^z = g$ winning the strong RSA security game.

□

$\underline{\mathsf{KVaC.BatchProveMem}(\mathcal{K}=[(k,v,u)]_i^n, \mathcal{K}'=[(k,v,u)]_i^m)}$

$Z \leftarrow \prod_{i\in\mathcal{K}\setminus\mathcal{K}'} \mathsf{H}(k_i)^{u_i}$; $Z' \leftarrow \prod_{i\in\mathcal{K}'} \mathsf{H}(k_i)^{u_i}$

$\Delta \leftarrow \sum_{i\in\mathcal{K}\setminus\mathcal{K}'} \left(v_i \cdot \mathsf{H}(k_i)^{u_i-1} \prod_{j\neq i\in\mathcal{K}\setminus\mathcal{K}'} \mathsf{H}(k_j)^{u_j}\right)$

$h' \leftarrow g^{\Delta}$; $g' \leftarrow g^{Z}$

$(a,b) \leftarrow \mathsf{EEA}(Z,Z')$; $B \leftarrow g^b$

Return $\mathsf{BatchRecurse}(h',g',B,a,\mathcal{K}')$

$\underline{\mathsf{BatchRecurse}(h,g,B,a,[(z,v,u)]_i^n)}$

If $n=1$ then return $[(h,g,B,a)]$

$n' \leftarrow n/2$

$Z_L \leftarrow \prod_{i=1}^{n'} z_i^{u_i}$; $\Delta_L \leftarrow \sum_{i=1}^{n'} \left(v_i z_i^{u_i-1} \prod_{j\neq i} z_j^{u_j}\right)$

$Z_R \leftarrow \prod_{i=n'}^{n} z_i^{u_i}$; $\Delta_R \leftarrow \sum_{i=n'}^{n} \left(v_i z_i^{u_i-1} \prod_{j\neq i} z_j^{u_j}\right)$

$h_L \leftarrow h^{Z_L} g^{\Delta_L}$; $g_L \leftarrow g^{Z_L}$

$h_R \leftarrow h^{Z_R} g^{\Delta_R}$; $g_R \leftarrow g^{Z_R}$

$(s,t) \leftarrow \mathsf{EEA}(Z_L,Z_R)$

$q_L \leftarrow \lfloor \frac{at}{Z_L} \rfloor$; $r_L \leftarrow at \mod Z_L$

$q_R \leftarrow \lfloor \frac{as}{Z_R} \rfloor$; $r_R \leftarrow as \mod Z_R$

$a_L \leftarrow r_L$; $B_L \leftarrow g_R^{q_L} g^{as} B^{Z_R}$

$a_R \leftarrow r_R$; $B_R \leftarrow g_L^{q_R} g^{at} B^{Z_L}$

$W_L \leftarrow \mathsf{BatchRecurse}(h_R,g_R,B_L,a_L,[(z_i,v_i,u_i)]_{i=1}^{n'})$

$W_R \leftarrow \mathsf{BatchRecurse}(h_L,g_L,B_R,a_R,[(z_i,v_i,u_i)]_{i=n'}^{n})$

Return $W_L \,\|\, W_R$

$\underline{\mathsf{KVaC.CommitWithCkpt}(C=(C_1,C_2),[(k,v',u')]_i)}$

$[z]_i \leftarrow [\mathsf{H}(k)]_i$

$C_1' \leftarrow C_1^{\prod_i z_i^{u_i'}} C_2^{\sum_j \left(v_j' z_j^{u_j'-1} \prod_{i\neq j} z_i^{u_i'}\right)}$

$C_2' \leftarrow C_2^{\prod_i z_i^{u_i'}}$

Return (C_1',C_2')

$\underline{\mathsf{KVaC.ProveMemWithCkpt}([C=(C_1,C_2),(k,v',u')]_i,m)}$

$[z]_i \leftarrow [\mathsf{H}(k)]_i$

$\pi_1 \leftarrow C_1^{\prod_{i\neq m} z_i^{u_i'}} C_2^{\sum_{j\neq m} \left(v_j' z_j^{u_j'-1} \prod_{i\neq j,m} z_i^{u_i'}\right)}$

$\pi_2 \leftarrow C_2^{\prod_{i\neq m} z_i^{u_i'}}$

$(a,b) \leftarrow \mathsf{EEA}(\prod_{i\neq m} z_i^{u_i'}, z_m)$

$\pi \leftarrow ((\pi_1,\pi_2),(C_2^b,a),u_m')$

Return π

$\underline{\mathsf{KVaC.VerifyMemWithCkpt}(C,C',(k,v'),\pi)}$

$(C_1,C_2) \leftarrow C$; $(C_1',C_2') \leftarrow C'$; $z \leftarrow \mathsf{H}(k)$

$((\pi_1,\pi_2),(B,a),u') \leftarrow \pi$

Return $\bigwedge \begin{pmatrix} (\pi_1)^{z^{u'}} (\pi_2)^{v' \cdot z^{u'-1}} = C_1' \\ (\pi_2)^{z^{u'}} = C_2' \\ (\pi_2)^a B^z = C_2 \end{pmatrix}$

Figure 14: (Left) Algorithms for computing a membership proofs for a batch of keys. (Right) Modified commit, prove membership, and verify membership algorithms for proving membership with respect to key updates since a checkpoint commitment.

F RSA Membership Proof Computation

This section provides more detail for the key caching and membership proof checkpointing optimizations described in Section 4.3 for RSA membership proof computation.

F.1 Batch Computation and Key Caching

Figure 14 (left) provides pseudocode for the $O(N+m\log m)$ algorithm to batch compute membership proofs for a subset of m keys over N total updates in the key-value commitment. We consider a set $\mathcal{K}$ of unique keys in the commitment and a subset $\mathcal{K}' \subseteq \mathcal{K}$ of size m for which to compute membership proofs. The $O(m\log m)$ work takes place in the BatchRecurse protocol which adapts existing techniques from [BBF19, TXN20] for computing batch membership and non-membership proofs for RSA accumulators. The $O(N)$ work is in computing the initial values (h,g,B,a) passed to BatchRecurse representing the state of all keys in $\mathcal{K}\setminus\mathcal{K}'$; we will refer to these as *helper values* for membership proof computation.

Our approach is to maintain the helper values (h,g,B,a) for a cache $\mathcal{K}'$ as a set of ℓ updates $\mathcal{U}=[(k,\delta)]_i^\ell$ are applied to the commitment. If the helper value is maintained to reflect the updated set $\mathcal{K}$ then membership proofs for the keys in $\mathcal{K}'$ can be recomputed with only $O(m\log m)$ work.

$\underline{\mathsf{UpdateHelper}((h,g,B,a),\mathcal{K}'=[(k,v,u)]_i^n,\mathcal{U}=[(k,\delta)]_i^\ell)}$

$Z \leftarrow \prod_{i\in\mathcal{U}} \mathsf{H}(k_i)^{u_i}$; $Z' \leftarrow \prod_{i\in\mathcal{K}'} \mathsf{H}(k_i)^{u_i}$

$\Delta \leftarrow \sum_{i\in\mathcal{U}} \left(\delta_i \prod_{j\neq i\in\mathcal{U}} \mathsf{H}(k_j)\right)$

$h' \leftarrow h^Z g^{\Delta}$; $g' \leftarrow g^Z$

$(s,t) \leftarrow \mathsf{EEA}(Z',Z)$

$q \leftarrow \lfloor \frac{at}{Z'} \rfloor$; $r \leftarrow at \mod Z'$

$a' \leftarrow r$; $B' \leftarrow g^{at+qZ} B$

Return (h',g',B',a')

The UpdateHelper pseudocode above shows how to maintain the helper values by applying updates from $\mathcal{U}$. The update protocol runs in $O(m+\ell)$ time.

Because maintaining a cache requires persistently updating its corresponding helper value, the system cannot efficiently maintain too many caches. One possible distribution of cache sizes if there are K total keys is by sizes of power of 2, leading to $\log K$ caches to maintain. This means that most keys will be placed in caches of large size which will still incur large promise fulfillment times. However, some keys will be placed in smaller caches that can be recomputed often.

F.2 Membership Proof Checkpointing

The approach to key caching requires key membership in a cache to be fixed; adding or removing keys from a cache requires recomputing the helper value incurring an expensive $O(n)$ cost. However dynamic caches supporting key eviction and loading are important for a long-lived system as key lookup patterns evolve. We address this using a similar checkpointing approach as before: past epochs are divided into ranges of size decreasing powers of 2, resulting in a logarithmic total number of ranges where the smallest range size is determined by the cache size. Instead of proving membership of a key across the entire history of n updates, a checkpoint membership proof is provided for the updates that occur within each checkpointed range. The checkpoint membership proofs of all checkpointed ranges together attest to the membership of the key over the full history.

As epochs are added, occasionally ranges get merged together. Recomputing membership proofs for merged large ranges is an expensive task but is amortized over the life of the system. Recomputation is necessary to maintain a logarithmic-sized membership proof, but can be done in the background as it does not affect latency; the membership proofs of the unmerged ranges can be used in the meantime.

We select the checkpointing schedule, i.e., minimum checkpoint range length, for a cache based on its size; larger caches will have proportionally larger minimum checkpoint ranges as computing membership proofs for the cache takes longer. However, keys can be evicted and loaded between caches of different sizes (and schedules) at the time when their most recent checkpoint endpoints coincide. Since both caches will be beginning a new checkpoint range, the helper values are restarted allowing for arbitrary changes in key membership.

To compute checkpoint membership proofs for a subrange of epochs, consider the range starting with epoch digest $d = (X_1, X_2)$ and ending with epoch digest d'. Recall,

$$d' \leftarrow \left(g^{\left(\prod_i \mathsf{H}(k_i)^{u_i}\right)\cdot\left(\sum_i v_i/\mathsf{H}(k_i)\right)}, g^{\prod_i \mathsf{H}(k_i)^{u_i}}\right),$$

for key-values (k_i, v_i) that have been updated u_i times over the registry history. When reformulated:

$$d' \leftarrow \left(X_1^{\prod_i \mathsf{H}(k_i)^{u'_i}} X_2^{\left(\prod_i \mathsf{H}(k_i)^{u'_i}\right)\cdot\left(\sum_i v'_i/\mathsf{H}(k_i)\right)}, X_2^{\prod_i \mathsf{H}(k_i)^{u'_i}}\right),$$

for key-values (k_i, v'_i) updated u'_i times in the range between d and d'. We build checkpoint membership proofs to fit this reformulation and show that a full history membership proof of v for version u consists of the logarithmic sequence of checkpoint membership proofs of (k, v'_j, u'_j) and verifying the sum of values $v = \sum_j v'_j$ and versions $u = \sum_j u'_j$.

Figure 14 (right) provides pseudocode for committing, proving membership, and verifying membership with respect to key updates applied since a prior checkpoint commitment $C = (C_1, C_2)$. By splitting up key updates among checkpointed ranges, we allow for dynamic key caches that can change their key membership at the start of a new checkpoint without incurring expensive costs to recompute a new helper value; the helper value for the new checkpoint starts from scratch (no previous updates in this new range) instead of incurring cost on the order of all previous updates in the system. Computing membership proofs for a batch of keys can also be done, modifying the computation of the helper values in $\mathsf{BatchProveMem}$: $h' \leftarrow C_1^Z C_2^\Delta$ and $g' \leftarrow C_2^Z$.

A membership proof for key k of value and version (v, u) for commitment can be produced as a sequence of w checkpoint membership proofs for k over w contiguous checkpoint ranges for values and version $[(v', u')]_j^w$. Note that some checkpoint ranges may not include an update for k in which case the checkpoint proof is a non-membership proof of k. We will show that if (1) the w checkpoint membership proofs verify, (2) $v = \sum_j^w v'_j$ and $u = \sum_j^w u'_j$, and (3) there exist versioned invariant proofs between the checkpoint commitments, then a membership proof for k of (v, u) for the full commitment can be computed. This in turn implies that the versioned security property from Appendix E hold with respect to checkpoint membership proofs.

Theorem 3. *For any sequence of w contiguous checkpoint proofs and versioned invariant proofs for key k, $\left[(C_j, \pi_j, (v'_j, u'_j), \lambda_j)\right]_{j=1}^w$, such that (where $C_0 \leftarrow \mathsf{KVaC.Init}()$):*

- $\bigwedge_{i=1}^w \mathsf{KVaC.VerUpdate}(C_{j-1}, C_j, \lambda_j)$,
- $\bigwedge_{j=1}^w \mathsf{KVaC.VerifyMemWithCkpt}(C_{j-1}, C_j, (k, v'_j, u'_j), \pi_j)$,

then, given an extractor $\mathcal{X}$ for BBF, we give membership proof π such that $\mathsf{KVaC.VerifyMem}(C_j, (k, \sum_j^w v'_j, \sum_j^w u'_j, \pi) = 1$.

Proof. We will show that any two checkpoint proofs from C_A to C_B to C_C:

- $\mathsf{KVaC.VerUpdate}(C_A, C_B, \lambda_A)$,
- $\mathsf{KVaC.VerUpdate}(C_B, C_C, \lambda_B)$,
- $\mathsf{KVaC.VerifyMemWithCkpt}(C_A, C_B, (k, v'_A, u'_A), \pi_A)$,
- $\mathsf{KVaC.VerifyMemWithCkpt}(C_B, C_C, (k, v'_B, u'_B), \pi_B)$,

can be merged into a single checkpoint proof π from C_A to C_C such that $\mathsf{KVaC.VerifyMemWithCkpt}(C_A, C_C, (k, v'_A + v'_B, u'_A + u'_B), \pi)$. Given this result, the rest of the proof follows by merging the checkpoint proofs in sequence and observing that $\mathsf{KVaC.VerifyMemWithCkpt}$ is equivalent to $\mathsf{KVaC.VerifyMem}$ where the checkpoint commitment is C_0.

We show how to construct the merged proof π for 4 different cases enumerating whether each of the two checkpoint proofs are membership proofs or non-membership proofs. In all cases, we will make use of $z = \mathsf{H}(k)$ and extracted exponents (α_A, β_A) and (α_B, β_B) from λ_A and λ_B, such that $C_B = (C_{A,1}^{\alpha_A} C_{A,2}^{\beta_A}, C_{A,2}^{\alpha_A})$ and $C_C = (C_{B,1}^{\alpha_B} C_{B,2}^{\beta_B}, C_{B,2}^{\alpha_B})$.

Case 1: Both π_A and π_B are non-membership proofs.

We show how to construct a non-membership proof π for the merged ranges. A valid non-membership proof consists of $\pi_A = (B_A, a_A)$ such that $C_{B,2}^{a_A} B_A^z = C_{A,2}$. By soundness of BBF, we have that $C_{B,2} = C_{A,2}^{\alpha_A}$, and so by Lemma 3, we have that $\gcd(z, \alpha_A) = 1$. By a symmetric argument for $\pi_B = (B_B, a_B)$ where $C_{C,2}^{a_B} B_B^z = C_{B,2}$, and $C_{C,2} = C_{B,2}^{\alpha_B}$, we have that $\gcd(z, \alpha_B) = 1$.

Since z is prime, this also gives us that $\gcd(z, \alpha_A \cdot \alpha_B) = 1$. Thus we can compute Bézout coefficients $(a, b) \leftarrow \mathsf{EEA}(\alpha_A \cdot \alpha_B, z)$ such that $a\alpha_A\alpha_B + bz = 1$, and non-membership proof $\pi = (C_{A,2}^b, a)$. We can see this verifies for the merged range:

$$C_{C,2}^a (C_{A,2}^b)^z = (C_{A,2}^{\alpha_A \alpha_B})^a (C_{A,2}^b)^z = C_{A,2}^{a\alpha_A\alpha_B + bz} = C_{A,2}\,.$$

Case 2: π_A is a membership proof and π_B is a non-membership proof.

We construct a membership proof of (v'_A, u'_A) for the merged ranges. Parse membership proof $\pi_A = (\pi_{A,1}, \pi_{A,2}, B_A, a_A)$. From the verification equations of π_A and Lemma 3, we have that $z^{u'_A} \mid \alpha_A$ and $\gcd(z, \frac{\alpha_A}{z^{u'_A}}) = 1$.

We set $\pi_1 = \left((\pi_{A,1})^{\alpha_B} (C_{A,2})^{\frac{\alpha_A \beta_B}{z^{u'_A}}} \right)$ and $\pi_2 = C_{A,2}^{\frac{\alpha_A \alpha_B}{z^{u'_A}}}$. Then we have the following two verification equations satisfied:

$$\begin{aligned}
C_{C,1} &= C_{B,1}^{\alpha_B} C_{B,2}^{\beta_B} \\
&= \left((\pi_{A,1})^{z^{u'_A}} (\pi_{A,2})^{v'_A \cdot z^{u'_A - 1}} \right)^{\alpha_B} \left(C_{A,2}^{\alpha_A} \right)^{\beta_B} \\
&= (\pi_{A,1})^{\alpha_B \cdot z^{u'_A}} (\pi_{A,2})^{\alpha_B \cdot v'_A \cdot z^{u'_A - 1}} (C_{A,2})^{\alpha_A \beta_B} \\
&= \left((\pi_{A,1})^{\alpha_B \cdot z^{u'_A}} (C_{A,2})^{\alpha_A \beta_B} \right) \left((C_{A,2})^{\frac{\alpha_A}{z^{u'_A}}} \right)^{\alpha_B \cdot v'_A \cdot z^{u'_A - 1}} \\
&= \left((\pi_{A,1})^{\alpha_B} (C_{A,2})^{\frac{\alpha_A \beta_B}{z^{u'_A}}} \right)^{z^{u'_A}} \left((C_{A,2})^{\frac{\alpha_A}{z^{u'_A}}} \right)^{\alpha_B \cdot v'_A \cdot z^{u'_A - 1}} \\
&= (\pi_1)^{z^{u'_A}} (\pi_2)^{v'_A \cdot z^{u'_A - 1}} \\
C_{C,2} &= (C_{A,2})^{\alpha_A \alpha_B} = (\pi_2)^{z^{u'_A}}
\end{aligned}$$

To satisfy the last verification equation, first consider the same argument from Case 1 for non-membership proof π_B showing that $\gcd(z, \alpha_B) = 1$. Thus, we have that $\gcd(z, \frac{\alpha_A \alpha_B}{z^{u'_A}}) = 1$, and compute $(a, b) \leftarrow \mathsf{EEA}(\frac{\alpha_A \alpha_B}{z^{u'_A}}, z)$. Finally we set $\pi = (\pi_1, \pi_2, C_{A,2}^b, a)$.

Case 3: π_A is a non-membership proof and π_B is a membership proof.

We construct a membership proof of (v'_B, u'_B) for the merged ranges. Parse membership proof $\pi_B = (\pi_{B,1}, \pi_{B,2}, B_B, a_B)$. From

the verification equations of π_B and Lemma 3, we have that $z^{u'_B} \,|\, \alpha_B$ and $\gcd(z, \frac{\alpha_B}{z^{u'_B}}) = 1$.

We set $\pi_1 = \pi_{B,1}$ and $\pi_2 = \pi_{B,2} = C_{A,2}^{\frac{\alpha_A \alpha_B}{z^{u'_B}}}$. Then we have the following two verification equations satisfied, directly form the verification equations for π_B:

$$C_{C,1} = (\pi_{B,1})^{z^{u'_B}} (\pi_{B,2})^{v'_B \cdot z^{u'_B - 1}} = (\pi_1)^{z^{u'_B}} (\pi_2)^{v'_B \cdot z^{u'_B - 1}}$$
$$C_{C,2} = (\pi_{B,2})^{z^{u'_B}} = (\pi_2)^{z^{u'_B}}$$

To satisfy the last verification equation, first consider the same argument from Case 1 for non-membership proof π_A showing that $\gcd(z, \alpha_A) = 1$. Thus, we have that $\gcd(z, \frac{\alpha_A \alpha_B}{z^{u'_B}}) = 1$, and compute $(a, b) \leftarrow \mathsf{EEA}(\frac{\alpha_A \alpha_B}{z^{u'_B}}, z)$. Finally we set $\pi = (\pi_1, \pi_2, C_{A,2}^b, a)$.

Case 4: Both π_A and π_B are membership proofs.

We construct a membership proof of $(v'_A + v'_B, u'_A + u'_B)$ for the merged ranges. Parse membership proofs $\pi_A = (\pi_{A,1}, \pi_{A,2}, B_A, a_A)$ and $\pi_B = (\pi_{B,1}, \pi_{B,2}, B_B, a_B)$. From the same arguments in Case 2 and Case 3 using Lemma 3, we have that $z^{u'_A} \,|\, \alpha_A$ and $\gcd(z, \frac{\alpha_A}{z^{u'_A}}) = 1$ and $z^{u'_B} \,|\, \alpha_B$ and $\gcd(z, \frac{\alpha_B}{z^{u'_B}}) = 1$.

Next, consider the following line to show that $z^{u'_A} | (\beta_A - \frac{v'_A \alpha_A}{z})$:

$$\begin{aligned}
(\pi_{A,1})^{z^{u'_A}} (\pi_{A,2})^{v'_A \cdot z^{u'_A - 1}} &= C_{B,1} = C_{A,1}^{\alpha_A} C_{A,2}^{\beta_A} \\
(\pi_{A,1})^{z^{u'_A}} &= C_{B,1} = \frac{C_{A,1}^{\alpha_A} C_{A,2}^{\beta_A}}{(\pi_{A,2})^{v'_A \cdot z^{u'_A - 1}}} \\
&= C_{A,1}^{\alpha_A} (C_{A,2})^{\beta_A - \frac{v'_A \alpha_A}{z}} && (\pi_{A,2} = (C_{A,2})^{\frac{\alpha_A}{z^{u'_A}}}) \\
\left(\frac{\pi_{A,1}}{(C_{A,1})^{\frac{\alpha_A}{z^{u'_A}}}} \right)^{z^{u'_A}} &= (C_{A,2})^{\beta_A - \frac{v'_A \alpha_A}{z}}
\end{aligned}$$

Using the same initial proof steps as the proof for Lemma 3, we claim that $z^{u'_A} | (\beta_A - \frac{v'_A \alpha_A}{z})$ else we've found a solution to the strong RSA security game. Symmetrically, we also claim that $z^{u'_B} | (\beta_B - \frac{v'_B \alpha_B}{z})$.

We set $\pi_1 = (C_{A,1})^{\frac{\alpha_A \alpha_B}{z^{u'_A + u'_B}}} (C_{A,2})^{\frac{\left(\alpha_B(\beta_A - v'_A \alpha_A / z)\right) + \left(\alpha_A(\beta_B - v'_B \alpha_B / z)\right)}{z^{u'_A + u'_B}}}$ and $\pi_2 = C_{A,2}^{\frac{\alpha_A \alpha_B}{z^{u'_A + u'_B}}}$. The first two verification equations are satisfied as follows:

$$\begin{aligned}
(\pi_1)^{z^{u'_A + u'_B}} (\pi_2)^{(v'_A + v'_B) \cdot z^{u'_A + u'_B - 1}} &= (C_{A,1})^{\alpha_A \alpha_B} (C_{A,2})^{\left(\alpha_B(\beta_A - v'_A \alpha_A / z)\right) + \left(\alpha_A(\beta_B - v'_B \alpha_B / z)\right)} (C_{A,2})^{\left(\frac{(v'_A + v'_B) \alpha_A \alpha_B}{z}\right)} \\
&= (C_{A,1})^{\alpha_A \alpha_B} (C_{A,2})^{(\alpha_B \beta_A + \alpha_A \beta_B)} \\
&= C_{C,1} \\
(\pi_2)^{z^{u'_A + u'_B}} &= (C_{A,2})^{\alpha_A \alpha_B} = C_{C,2}
\end{aligned}$$

To satisfy the last verification equation, we have that $\gcd(z, \frac{\alpha_A \alpha_B}{z^{u'_A + u'_B}}) = 1$ from $\gcd(z, \frac{\alpha_A}{z^{u'_A}}) = 1$ and $\gcd(z, \frac{\alpha_B}{z^{u'_B}}) = 1$ since z is prime. We compute $(a, b) \leftarrow \mathsf{EEA}(\frac{\alpha_A \alpha_B}{z^{u'_A + u'_B}}, z)$. Finally we set $\pi = (\pi_1, \pi_2, C_{A,2}^b, a)$. □