Verifiable Member and Order Queries on a List in Zero-Knowledge

Esha Ghosh*[1], Olga Ohrimenko†[2] and Roberto Tamassia‡[1]

[1]Department of Computer Science, Brown University
[2]Microsoft Research

Abstract

We introduce a formal model for order queries on lists in zero knowledge in the traditional authenticated data structure model. We call this model Privacy-Preserving Authenticated List (PPAL). In this model, the queries are performed on the list stored in the (untrusted) cloud where data integrity and privacy have to be maintained. To realize an efficient authenticated data structure, we first adapt consistent data query model. To this end we introduce a formal model called Zero-Knowledge List (ZKL) scheme which generalizes consistent membership queries in zero-knowledge to consistent membership and order queries on a totally ordered set in zero knowledge. We present a construction of ZKL based on zero-knowledge set and homomorphic integer commitment scheme. Then we discuss why this construction is not as efficient as desired in cloud applications and present an efficient construction of PPAL based on bilinear accumulators and bilinear maps which is provably secure and zero-knowledge.

1 Introduction

Authentically releasing partial information while maintaining privacy is a well known requirement in many practical scenarios where the data being dealt with is sensitive.

Consider, for example, the following medical case study presented in [BB12]. Each patient has a personal health record (PHR) that contains the medication and vaccination history of the patient. Entries are made against the dates when medicines are taken and vaccinations are performed. Thus, the PHR is a chronologically sorted document signed by the medical provider and given to the patient. Now the patient might need to authorize the release of a subset of the PHR with only the relevant information to be sent to third parties on an as needed basis, without the medical provider's involvement. For example, let us say the patient wants to join a summer camp that requires the vaccination record of the patient and the order in which the vaccinations were taken. the patient wants to release the relevant information in a way such that the camp can verify that the data came from a legitimate medical provider, but the camp cannot learn anything beyond the authorized subset of relevant information, i.e., the vaccinations and their chronological order, but not the exact date when they were taken.

Consider another example where there are multiple regional sales divisions of a company distributed across three neighboring states. A monthly sales report contains the number of products sold by each of the divisions, arranged in non-decreasing order. Each monthly sales report is signed by the authority and stored on a cloud server. By the company's access control policy, each sales division is allowed to learn how it did in comparison to the other units, but not anything else. That

*esha_ghosh@brown.edu
†oohrim@microsoft.com
‡rt@cs.brown.edu

is, a division cannot learn the sales numbers of other divisions or their relative performance beyond what it can infer by the comparisons with itself. Thus, the cloud would need to release the relevant information in such a way that the querying division can verify the data came from the legitimate source but not learn anything beyond the query result.

The above examples motivate the following model: a trusted *owner* generates an ordered set of elements. Let us call this ordered set a *list*. The owner outsources the list to a (possibly untrusted) party, let us call it *server*. There is another party involved who issues order queries on the list, let us call this party *client*. The client only interacts with the server. So the server has to release information in a way such that the client can verify the authenticity of the data it receives, i.e., that it is truly generated by the trusted owner. But the client should not be able to learn anything beyond the answers to its queries.

This above model specifies an *authenticated data structure* with an additional privacy requirement. An *authenticated data structure* [Tam03] is a structured collection of data (e.g., a list, tree, graph, or map) along with a set of query operations defined on it. Three parties are involved in an authenticated data structure (ADS) scheme, namely, the data owner, the server and the client/user. ADS framework allows the data owner to outsource data processing tasks to an untrusted server without loss of data integrity for clients. This is achieved as follows. The (trusted) data owner produces authentication information about the dataset (ordered list in our case) and a short digest signature and sends a copy of the dataset along with the authentication information to the (untrusted) server and the digest signature to the client. The server responds to the (legitimate) client queries about the dataset by returning the query answer and a compact proof of the answer. The client uses the digest signature (obtained from the owner), the query answer and the proof obtained from the server to verify the integrity of the answer.

Classic hash-based authenticated data structures were designed without taking into account privacy goals and provide proofs that leak information about the dataset beyond the query answer. For example, in a hash tree [Mer80, Mer89] for a set of n elements, the proof of the membership of an element in the set has size $\log n$, thus leaking information about the size of the set. Also, if the elements are stored at the leaves in sorted order, the proof of *membership* of an element reveals its rank. Similar information leaks occur in other hash-based authenticated data structures for dictionaries and maps, such as authenticated skips lists [MTGS01]. As another example, consider an approach for supporting *non-membership* proofs using an authenticated data structure that supports membership proofs. This method involves storing intervals of consecutive elements (x_i, x_{i+1}) and returning as a proof of non-membership of a query element x the interval (x_i, x_{i+1}) such that $x_i < x < x_{i+1}$. Hence, the proof trivially reveals two elements of the set.

We define a *privacy-preserving authenticated data structure* as an ADS with an additional privacy property that ensures that the proof returned by the server to the client does not reveal any information about the dataset beyond what can be learned from the current and previous answers to queries to the dataset. In this paper we study one such data structure, a *privacy-preserving authenticated list* (PPAL), for which we consider order queries.

A privacy-preserving authenticated list (PPAL) allows an owner to outsource to the server data with different access control policies imposed on it. Hence, when the owner outsources it to the server, clients can access only parts of it from the server and verify that it is indeed owner's authentic data but should not be allowed to learn about the data they do not have permission to access. Hence, privacy policy should be also imposed on the proofs of authenticity of the data that the clients learn (the property not supported by classical ADS). PPAL has several interesting applications as we have already seen in the motivating examples. We also envision a PPAL list to be an important building block for designing efficient hierarchical privacy-preserving data structures e.g., ordered trees that store XML data.

In this paper, we present an efficient solution for privacy-preserving authenticated lists that supports queries on the relative order of two (or more) elements of the list. This framework guarantees integrity of the order queries through a compact proof returned to the client. The proof does not reveal the actual ranks of the elements nor any order information between elements other that what can be inferred from the current and previous answers by the rule of transitivity.

We first present a generic approach to this problem in the traditional consistent query model [MRK03, CHL+05, ORS04, CFM08, LY10] where there are two parties involved: the prover, $\mathsf{Prover} = (\mathsf{P}_1, \mathsf{P}_2)$ and the verifier, Verifier. P_1 takes an ordered list as input and produces a short commitment which is made public. Then Verifier generates membership and order queries on the list and P_2 responds with the answers and the proofs. Once Prover commits to a list, it cannot give answers inconsistent with the commitment that will pass the verification test by Verifier. We formalize this framework as *Zero Knowledge List (ZKL)* and give a construction in Section 3.

It is easy to see this model can be interpreted in the PPAL framework as follows. We can make the owner run P_1, the server run P_2 and the client run Verifier. In fact in the ZKL model we get stronger security guarantee as once the owner commits to a list, even a malicious one, it cannot give inconsistent answers later. Moreover the ZKL framework supports both membership and order queries in Zero Knowledge. However, as we discuss in Section 3.5, the construction is not very practical for cloud computing scenario where the client can be a mobile device.

The ZKL model indeed gives stronger security guarantees, but in the privacy-preserving authenticated list (PPAL) model, the owner is in fact a *trusted* party. We discuss the PPAL framework in Section 4 and design an efficient PPAL scheme, exploiting the fact that the owner is an honest party. In our scheme, for a source list of size n and a query of size m, neither the server nor the owner runs in time more than $O(n)$ or requires storage more that $O(n)$. The running time for the server can be brought down to $O(m \log n)$ with $O(n)$ preprocessing time and the client requires time and space proportional to $O(m)$. We give the construction and discuss its efficiency in Section 5.

1.1 Problem Statement

In this section we define order queries, introduce our adversarial model and security properties, and discuss our efficiency goals.

1.1.1 Query

Let $\mathcal{L}$ be a linearly ordered list of non-repeated elements. An *order query* on a list $\mathcal{L}$ of distinct elements is defined as follows: given a pair of query elements (x, y) of $\mathcal{L}$, the server returns the pair with its elements rearranged according to their order in $\mathcal{L}$ together with proofs of membership of x and y and a proof of the returned order. For example, if y precedes x in $\mathcal{L}$, then the pair (y, x) is returned as an answer.

For generality, the data structure also supports *batch order query*: Given a list of query elements δ, the server returns a permutation of δ according to the ordering of the elements in $\mathcal{L}$, together with a proof of the membership of the elements and of their ordering in $\mathcal{L}$.

The above model captures the query model of a privacy-preserving authenticated list. In comparison, zero knowledge list structure supports the same queries as well as non-membership queries. Hence, as a response to a (non-)membership element query the prover returns a boolean value indicating if the element is in the list and a corresponding proof of (non-)membership.

1.1.2 Adversarial model and security properties

In this section we present adversarial models and security properties of PPAL and ZKL.

Following the authenticated data structure model, list $\mathcal{L}$ plus authentication information about it is created by a data owner and given to a server, who answers queries on $\mathcal{L}$ issued by a client, who

verifies the answers and proofs returned by the server using the public key of the data owner. We assume the data owner is trusted by the client, who has the public key of the data owner. However, both the client and server can act as adversaries, as follows:

- The server is malicious and may try to forge proofs for incorrect answers to (ordering) queries. For example, the server may try to prove an incorrect ordering of a pair of elements of $\mathcal{L}$.
- The client tries to learn from the proofs additional information about list $\mathcal{L}$ beyond what it has inferred from the answers. For example, if the client has performed two ordering queries with answers $x < y$ and $x < z$, it may want to find out whether $y < z$ or $z < y$.

Note that in typical cloud database applications, the client is allowed to have only a restricted view of the data structure and the server enforces an access control policy that prevents the client from getting answers to unauthorized queries. This motivates the curious behavior by the client. The client may behave maliciously and try to ask ill-formed queries or queries violating the access control policy. But the server may just refuse to answer when the client asks illegal queries. So the client's legitimate behavior can be enforced by the server.

We wish to construct a privacy-preserving authenticated data structure for list $\mathcal{L}$, i.e., a data structure with the following security properties:

Completeness ensures that honestly generated proofs are always accepted by the client.

Soundness mandates that proofs forged by the server for incorrect answers to queries do not pass the verification performed by the client.

Zero Knowledge means that each proof received by the client to a query reveals and verifies the answer and nothing else. In other words, for any element $x_i \in \{0,1\}^*$, the simulator, given oracle access to $\mathcal{L}$, should be able to simulate proofs for order queries that are indistinguishable from real proofs.

To understand the strength of the zero-knowledge property, let us illustrate to what extent the proofs are non-revealing. One of the guarantees of this property is that receiving a response to a query δ does not reveal where in $\mathcal{L}$ queried elements of δ are. In other words, no information about $\mathcal{L}$, other than what is queried for in δ is revealed. It is worth noting that in the context of leakage-free redactable signature schemes, this property has been referred to as *transparency* [BBD$^+$10, SPB$^+$12] and *privacy* [CLX09, KAB12]. Moreover, zero knowledge also provides security for the size of the list $\mathcal{L}$ from the client.

Since we let the client ask multiple queries on a static list adaptively, in principle, it is possible that even though the individual query responses and proofs do not leak any extraneous information about the source list, when the responses and proofs are collected together, the client is able to infer some structural information about the source list, which it had not explicitly queried for. Hence, we need to ensure that the scheme is immune against any potential leakage of any structural information that has not been explicitly asked for by the client. More concretely, in a linearly ordered list $\mathcal{L}$, the client should not be able to infer any relative order that is not inferable by the rule of transitivity from the queried orders. This security guarantee also follows from the zero-knowledge property.

The adversarial model in ZKL is different from that of PPAL. The ZKL model considers only two parties: the prover and the verifier. The prover initially computes a commitment to a list $\mathcal{L}$ and makes this commitment public (i.e., the verifier also receives it). Later the verifier asks membership and order queries on the list and the prover responds accordingly. In ZKL both the prover and the verifier can be malicious as follows:

- The prover may try to give answers which are inconsistent with the initial commitment.
- The adversarial behavior of the verifier is the same as that of the client in the PPAL model.

The security properties of ZKL (Completeness, Soundness, Zero-Knowledge) guarantee security against malicious prover and verifier. They are close to the ones of PPAL except for Soundness which captures that the prover can try to create a forgery on a list of his choice. We discuss the

security properties of ZKL in more detail in Section 3.2.

1.1.3 Efficiency

We characterize the efficiency of a privacy-preserving authenticated data structure for a list, $\mathcal{L}$, of n items by means of the following space and time complexity measures:

- *Server storage:* Space at the server for storing list $\mathcal{L}$ and the authentication information and for processing queries. Ideally, the server storage is $O(n)$, irrespective of the number of queries answered.
- *Proof size:* Size of the proof returned by the server to the client. Ideally, the proof has size proportional to the answer size.
- *Setup time:* Work performed by the data owner to create the authentication information that is sent to the server. Ideally, this should be $O(n)$.
- *Query time:* Work performed by the server to answer a query and produce its proof. Ideally, this work is proportional to the answer size.
- *Verification time:* Work performed by the client to verify the answer to a query using the proof provided by the server and the public key of the data owner. Ideally, this work is proportional to the answer size.

1.2 Related Work

We discuss related literature in three sections. First, we discuss work on data structures that answer queries in zero knowledge. This work is the closest to our work on zero knowledge lists. We then discuss signature schemes that can be interpreted in the privacy-preserving authenticated data structure model. Finally, we highlight the body of literature regarding leakage-free redactable signature schemes for ordered lists in detail. The latter is the closest to the problem of privacy preserving authenticated lists that we are addressing in this manuscript.

Zero Knowledge Data Structures Buldas *et al.* [BLL02] showed how to prove answers to dictionary queries using an authenticated search tree-based construction, but did not consider privacy. For a set of size n, the construction produces a proof of (non)membership of an element in the set that has size $O(k \log n)$. Similar to other work on authenticated data structures [Mer80, Mer89, MTGS01], the proof reveals information about the underlying set, e.g., its size and the location of the queried entry w.r.t. other entries.

The model of a *zero knowledge set* (more generally, *zero knowledge elementary database*) was first introduced by Micali *et al.* [MRK03]. This is a secure data structure which allows a prover to commit to a *finite set* S in such a way that, later on, it will be able to efficiently (and non-interactively) prove statements of the form $x \in S$ or $x \notin S$ without leaking any information about S beyond what has been queried for, not even the size of S. The security properties guarantee that the prover should not be able to cheat and prove contradictory statements about an element. Later, Chase *et al.* [CHL$^+$05] abstracted away Micali *et al.*'s solution and described the exact properties a commitment scheme should possess in order to allow a similar construction. This work introduced a new commitment scheme, called *mercurial commitment*. A generalization of mercurial commitments allowing for committing to an ordered sequence of messages (q-trapdoor mercurial commitment) was proposed in [CFM08] and later improved in [LY10]. A q-trapdoor mercurial commitment allows a committer to commit to an ordered sequence of message and later open messages with respect to specific positions.

Th above zero knowledge set constructions [MRK03, CHL$^+$05, CFM08, LY10] use an implicit ordered q-way hash tree ($q \geq 2$) built on the universe of all possible elements. The size of this tree is exponential in the security parameter. However, only a portion of the tree of size polynomial in the security parameter is explicitly stored in the data structure. Let N denote the universe size. Then

the proof size for membership and non-membership for an individual element is $O(\log_q N)$. Kate *et al.* [KZG10] suggested a weaker primitive called *nearly-zero knowledge set* based on *polynomial commitment* [KZG10]. In their construction the proof size for membership and non-membership for every individual element is $O(1)$, but the set size is not private.

A related notion of *vector commitments* was introduced by [CF13] where they show that a (concise) q-trapdoor mercurial commitment can be obtained from a vector commitment and a trapdoor mercurial commitment. A vector commitment scheme allows a committer to commit to an ordered sequence of values $(x_1, \ldots, x_n)$ in such a way that the committer can later open the commitment at specific positions (e.g., prove that x_i is the i-th committed message).

Ostrovsky *et al.* [ORS04] generalized the idea of membership queries to support membership and orthogonal range queries on a multidimensional dataset. [ORS04] describe constructions for consistent database queries, which allow the prover to commit to a database, and then provide query answers that are provably consistent with the commitment. They also consider the problem of adding privacy to such protocols. However their construction requires interaction (which can be avoided in the random oracle model) and requires the prover to keep a counter for the questions asked so far. The use of NP-reductions and probabilistically checkable proofs makes their generic construction expensive. The authors of [ORS04] also provide a simpler protocol based on *explicit-hash Merkle Tree*. However, this construction does not hide the size of the database as the proof size is $O(\lceil \log n \rceil)$ where n is the upper bound on the size of the database.

Signature Schemes A collection of signature schemes, namely *content extraction signature* [SBZ01], *redactable signature* [JMSW02] and *digital document sanitizing scheme* [MHI06] can be viewed in a three-party model where the owner digitally signs a data document and the server discloses to the client only part of the signed document along with a legitimately derived signature on it. The server derives the signature without the owner's involvement and the client verifies the authenticity of the document it receives from the server by running the verification algorithm of the underlying scheme. A related concept is that of *transitive signature scheme*, where given the signatures of two edges (a, b) and (b, c) of a graph, it is possible to compute the signature for the edge (or path) (a, c) without the signer's secret key [MR02, Yi06, CH12]. However, these signature schemes are not designed to preserve privacy of the signed object, which may include the content and/or the structure in which the content is stored.

Ahn *et al.* [ABC$^+$12] present a unified framework for computing on authenticated data via the notion of slightly homomorphic or P-homomorphic signatures, which was later improved by [Wan12]. This broad class of P-homomorphic signatures includes *quotable, arithmetic, redactable, homomorphic, sanitizable and transitive signatures*. This framework allows a third party to derive a signature on the object x' from a signature on x as long as $P(x, x') = 1$ for some predicate P that captures the *authenticatable relationship* between x and x'. A derived signature reveals no extra information about the parent x, referred to as *strong context hiding*. This work does not consider predicates of a specific data structure.

The authors propose a general RSA-accumulator based scheme that is expensive in terms of computation. In particular, the cost of signing depends on the predicate P and the size of the message space and is $O(n^2)$ for a n-symbol message space. This privacy definition was recently refined by [ALP12]. This line of work cannot be directly used for privacy preserving data structures where efficiency is an important requirement and quadratic overhead may be prohibitive depending on the application.

[CKLM13] gives definition and construction of malleable signature scheme. A signature scheme is defined to be malleable if, given a signature σ on a message x, it is possible to efficiently derive a signature σ' on a message x' such that $x' = T(x)$ for an *allowable* transformation T. Their

definition of context hiding requires unlinkability and allows for adversarially-generated keys and signatures. This definition is stronger than that of [ABC$^+$12] as it allows for adversarially-generated keys and signatures. Unlinkability implies the following: a quoted (or derived) signature should be indistinguishable from a fresh signature.

A motivating example proposed in [ABC$^+$12] deals with the impossibility of linking a quote to its source document. However, in the framework of privacy preserving authenticated data structures, it is important for the client to verify membership, i.e., given a quote from a document and a signature on the quote, the client should be able to verify that the quote is indeed in the document. Context-hiding definition in [CKLM13] also requires unlinkability.

Leakage-Free Signature Schemes for Ordered Lists A *leakage-free redactable signature scheme (LRSS)* allows a third party to remove parts of a signed document without invalidating its signature. This action, called *redaction*, does not require the signer's involvement. As a result, the verifier only sees the remaining redacted document and is able to verify that it is valid and authentic. Moreover, the redacted document and its signature do not reveal anything about the content or position of the removed parts. This problem can be easily interpreted in the privacy-preserving authenticated data structure model, where the signer is the owner, the third party is the server and the verifier is the client.

Kundu and Bertino [KB08] were first to introduce the idea of structural signatures for ordered trees (subsuming ordered lists) which support public redaction of subtrees (by third-parties) while retaining the integrity of the remaining parts. This was later extended to DAGs and graphs [KB13]. The notion was later formalized as *LRSS* for ordered trees in [BBD$^+$10] and subsequently several attacks on [KB08] were also proposed in [BBD$^+$10, PSPDM12].

The authors of [CLX09] presented a leakage-free redactable signature scheme for strings (which can be viewed as an ordered list) that hides the location of the redacted or deleted portions of the list at the expense of quadratic verification cost.

The basic idea of the LRSS scheme presented in [BBD$^+$10] is to sign *all possible ordered pairs* of elements of an ordered list. So both the computation cost and the storage space are quadratic in the number of elements of the list. Building on the work of [BBD$^+$10], [SPB$^+$12] proposed an LRSS for lists that has quadratic time and space complexity. Poehls *et al.* [PSPDM12] presented a LRSS scheme for a list that has linear time and space complexity but assumes an associative non-abelian hash function, whose existence has not been formally proved. The authors of [KAB12], presented a construction that uses quadratic space at the server and is not leakage-free. We discuss the attack in Section 4.

1.3 Contributions and Organization of the Paper

The main contributions of this work are as follows:

- After reviewing preliminary concepts and the cryptographic primitives we use in this paper, in Section 2, we introduce the Zero-Knowledge List (ZKL) model, present a construction, prove its security and analyze its efficiency in Section 3.
- In Section 4, we introduce a formal model for a privacy-preserving authenticated list that supports order queries on its elements.
- In Section 5, we present a construction of the above data structure based on bilinear maps and we analyze its performance.
- Formal proofs for the security properties of our construction are given in Section 6.

In Table 1 we compare our constructions of a privacy-preserving authenticated list with previous work in terms of performance, and assumptions. We also indicate which constructions satisfy the zero-knowledge property. We include a construction based on our new primitive, ZKL, and our direct construction of PPAL. We note that ZKL model is a two party model but can be adapted to

	[SBZ01]	[JMSW02]	[CLX09]	[BBD+10]	[SPB+12]	[PSPDM12]	[KAB12]	This paper ZKL	This paper PPAL
Zero-knowledge				✓	✓	✓		✓	✓
Setup time	$n \log n$	n	n	n^2	n^2	n	n	$n \log N$	n
Server Space	n	n	n	n^2	n^2	n	n^2	$n \log N$	n
Query time	m	$n \log n$	n	mn	m	n	n	$m \log N$	$\min(m \log n, n)$
Verification time	$m \log n \log m$	$m \log n$	n^2	m^2	m^2	m	m	$m \log N$	m
Proof size	m	$m \log n$	n	m^2	m^2	m	n	$m \log N$	m
Assumption	RSA	RSA	SRSA, Division	EUCMA	ROH, nEAE	AnAHF	ROH, RSA	ROH, FC, SRSA	ROH,nBDHI

Table 1: Comparison of our constructions of a privacy-preserving authenticated list with previous work. ZKL is a construction based on Zero-Knowledge lists from Section 3.3 and PPAL is a direct PPAL construction from Section 5 . All the time and space complexities are asymptotic. Notation: n is the number of elements of the list, m is the number of elements in the query, and N is the number of all possible l-bit strings from where list elements can be drawn from. Acronyms for the assumptions: Strong RSA Assumption (SRSA); Existential Unforgeability under Chosen Message Attack (EUCMA) of the underlying signature scheme; Random Oracle Hypothesis (ROH); n-Element Aggregate Extraction Assumption (nEAE); Associative non-abelian hash function (AnAHF); Factoring a composite (FC); n-Bilinear Diffie Hellman Inversion Assumption(nBDHI).

a three party model of PPAL (see Section 5 for details). Our PPAL construction outperforms all previous work that is based on widely accepted assumptions [BBD+10, SPB+12].

2 Preliminaries

2.1 Data Type

We consider a *linearly ordered list* $\mathcal{L}$ as a data structure that the owner wishes to store with the server. A list is an ordered set of elements $\mathcal{L} = \{x_1, x_2, \ldots, x_n\}$, where each $x_i \in \{0,1\}^*$, $\forall x_1, x_2 \in \mathcal{L}, x_1 \neq x_2$ and either $x_1 < x_2$ or $x_2 < x_1$. Hence, $<$ is a strict order on elements of $\mathcal{L}$ that is irreflexive, asymmetric and transitive.

We denote the set of elements of the list $\mathcal{L}$ as $\mathsf{Elements}(\mathcal{L})$. A sublist of $\mathcal{L}$, δ, is defined as: $\delta = \{x \mid x \in \mathsf{Elements}(\mathcal{L})\}$. Note that the order of elements in δ may not follow the order of $\mathcal{L}$. We denote with $\pi_{\mathcal{L}}(\delta)$ the permutation of the elements of δ under the order of $\mathcal{L}$.

$\mathcal{L}(x_i)$ denotes the membership of element x_i in $\mathcal{L}$, i.e., $\mathcal{L}(x_i) \neq \perp$ if $x_i \in \mathcal{L}$ and $\mathcal{L}(x_i) = \perp$ if $x_i \notin \mathcal{L}$. We interpret $\mathcal{L}(x_i)$ as a boolean value, i.e., $\mathcal{L}(x_i) \neq \perp$ is equivalent to $\mathcal{L}(x_i) = \mathsf{true}$ and $\mathcal{L}(x_i) = \perp$ is equivalent to $\mathcal{L}(x_i) = \mathsf{false}$. For all x_i such that $\mathcal{L}(x_i) \neq \perp$, $\mathsf{rank}(\mathcal{L}, x_i)$ denotes the rank of element x_i in the list, $\mathcal{L}$.

2.2 Cryptographic Primitives

We now describe a signature scheme that is used in our construction and cryptographic assumptions that underly the security of our method. In particular, our zero knowledge list construction relies on homomorphic integer commitments (Section 2.2.1), zero knowledge protocol to prove a number is non-negative (Section 2.2.2) and zero knowledge sets (Section 2.2.3), while the construction for privacy preserving lists relies on bilinear aggregate signatures and n-Bilinear Diffie Hellman Inversion assumption (Section 2.3).

2.2.1 Homomorphic Integer Commitment Scheme

We use a homomorphic integer commitment scheme $\mathsf{HomIntCom}$ that is statistically hiding and computationally binding [Bou00, DF02]. The later implies the existence of a trapdoor and, hence, can be used to "equivocate" a commitment, that is open the original message of the commitment to another message. The above commitment scheme is defined in terms of three algorithms $\mathsf{HomIntCom} = \{\mathsf{IntComSetup}, \mathsf{IntCom}, \mathsf{IntComOpen}\}$ and the corresponding trapdoor commitment (we call it a

simulator) as: $\mathsf{SimHomIntCom} = \{\mathsf{SimIntComSetup}, \mathsf{SimIntCom}, \mathsf{SimIntComOpen}\}$. We describe these algorithms in Figure 1. The *homomorphism* of $\mathsf{HomIntCom}$ is defined as $\mathsf{IntCom}(x + y) = \mathsf{IntCom}(x) \times \mathsf{IntCom}(y)$. For specific constructions of $\mathsf{HomIntCom}$ see Figure 8 in Appendix.

Figure 1: Homomorphic Integer Commitment Protocols.

$\mathsf{HomIntCom} = (\mathsf{IntComSetup}, \mathsf{IntCom}, \mathsf{IntComOpen})$

- $\mathsf{PK}_C \leftarrow \mathsf{IntComSetup}(1^k)$**:** $\mathsf{IntComSetup}$ is a randomized algorithm that takes as input the security parameter and generates a public key PK_C
- $(c, r) \leftarrow \mathsf{IntCom}(\mathsf{PK}_C, x)$**:** IntCom is a randomized algorithm that takes as input the public key, an integer x and generates a commitment,opening pair (c, r) with additive homomorphic properties. c servers as the commitment value for m and r is the opening value.
- $x \leftarrow \mathsf{IntComOpen}(\mathsf{PK}_C, c, r)$**:** $\mathsf{IntComOpen}$ takes as input the public key, a commitment c and the corresponding opening information r and returns the committed integer x.

$\mathsf{SimHomIntCom} = (\mathsf{SimIntComSetup}, \mathsf{SimIntCom}, \mathsf{SimIntComOpen})$

- $(\mathsf{PK}_C, \mathsf{TK}_C) \leftarrow \mathsf{SimIntComSetup}(1^k)$**:** $\mathsf{SimIntComSetup}$ takes as input the security parameter and returns a public key PK_C and a trapdoor TK_C.
- $(c, r) \leftarrow \mathsf{SimIntCom}(\mathsf{PK}_C, x)$**:** $\mathsf{SimIntCom}$ takes PK_C and an integer x and returns a commitment c and the opening information r.
- $x' \leftarrow \mathsf{SimIntComOpen}(\mathsf{PK}_C, \mathsf{TK}_C, c, r)$**:** $\mathsf{SimIntComOpen}$ takes as input $\mathsf{PK}_C, \mathsf{TK}_C$ and a commitment c and the corresponding opening information r and returns an arbitrary integer x', which might not be equal to x; c being the commitment to integer x.

2.2.2 Proving an integer is positive in zero-knowledge

We use following protocol between a prover and a verifier: the verifier holds prover's commitment c to an integer x and wishes to verify if this integer is positive, $x > 0$, without opening c. We denote this protocol as $\mathsf{Protocol}(x, r : c = C(x; r) \wedge x > 0)$ (Figure 2). In our construction, we will use the commitment scheme $\mathsf{HomIntCom}$ described in Figure 1 and use IntCom to compute c.

Figure 2: Protocol to prove non-negativity of an integer

$\mathsf{Protocol}(x, r : c = C(x; r) \wedge x \geq 0)$: We use this notation to concisely represent an (interactive) protocol between two parties P and V: P sends a commitment c to a non-negative value x to V and proves, without opening c, that $x \geq 0$. The symbol $c = C(x; r)$ denotes c is the commitment to x and r is the corresponding opening information. Note that r is not sent to V.

As a concrete construction we extend the protocol of [Lip03] which allows one to prove that $x \geq 0$ to supply a prove that $x - 1 \geq 0$. This proves $x > 0$. The protocol of [Lip03] is a Σ protocol, which is *honest verifier zero knowledge* and can be made *non-interactive general zero knowledge* in the Random Oracle model using Fiat-Shamir heuristic [FS86]. For details of the protocol refer to Figure 10.

Figure 3: Zero Knowledge Set (ZKS) model

$\mathsf{ZKS} = (\mathsf{ZKSSetup}, \mathsf{ZKSProver} = (\mathsf{ZKSP}_1, \mathsf{ZKSP}_2), \mathsf{ZKSVerifier})$

$\mathsf{PK}_D \leftarrow \mathsf{ZKSSetup}(1^k)$**:** The $\mathsf{ZKSSetup}$ algorithm takes the security parameter as input and produces a public key PK_D for the scheme. The prover and the verifier both take as input the string PK_D that can be a random string (in which case, the protocol is in the common random string model) or have a specific structure (in which case the protocol is in the trusted parameters model).

$(\mathsf{com}, \mathsf{state}) \leftarrow \mathsf{ZKSP}_1(1^k, \mathsf{PK}_D, D)$**:** ZKSP_1 takes the security parameter, the public key, PK_D and the set D and produces a short digest commitment com for D.

$(D(x), \mathsf{proof}_x) \leftarrow \mathsf{ZKSP}_2(\mathsf{PK}_D, \mathsf{state}, x)$**:** ZKSP_2 takes a query x and produces the corresponding value, $v = D(x)$ and the corresponding proof of membership/non-membership, proof_x.

$b \leftarrow \mathsf{ZKSVerifier}(1^k, \mathsf{PK}_D, \mathsf{com}, x, D(x), \mathsf{proof}_x)$**:** $\mathsf{Verifier}$ takes the security parameter, $\mathsf{PK}, \mathsf{com}$ and a query x and an answer $D(x)$ and a proof proof_x and returns a bit b, where $b = \mathsf{ACCEPT}/\mathsf{REJECT}$.

2.2.3 Zero Knowledge Set scheme

Let D be a set of of key value pairs. If (x, v) is a key, value pair of D, i.e, $(x, v) \in D$, then we write $D(x) = v$ to denote v is the value corresponding to the key x. For the keys that are not present in D, $x \notin D$, we write $D(x) = \perp$. A Zero Knowledge Set scheme (ZKS) consists of three probabilistic polynomial time algorithms - $\mathsf{ZKS} = (\mathsf{ZKSSetup}, \mathsf{ZKSProver} = (\mathsf{ZKSP}_1, \mathsf{ZKSP}_2), \mathsf{ZKSVerifier})$ and queries are of the form "is key x in D?". We describe the algorithms in Figure 3.

For our construction of zero knowledge lists we pick a ZKS construction of [CHL+05] that is based on mercurial commitments and describe it in more details in Figure 11.

2.2.4 Bilinear Aggregate Signature Scheme

We use bilinear aggregate signature scheme developed by Boneh *et al.* [BGLS03] for our privacy preserving authenticated data structure scheme. Given n signatures on n *distinct* messages $M_1, M_2, \ldots, M_n$ from n distinct users $u_1, u_2, \ldots, u_n$, it is possible to aggregate all these signatures into a single short signature such that the single signature (and the n messages) will convince the verifier that the n users indeed signed the n original messages (i.e., user i signed message M_i for $i = 1, \ldots, n$). Here we describe the scheme for the case of a single user signing n *distinct* messages $M_1, M_2, \ldots, M_n$. The decryption of the generic case of n different users can be found at [BGLS03]. The following notation is used in the scheme:

- G, G_1 are multiplicative cyclic groups of prime order p
- g is a generator of G
- e is computable bilinear nondegenerate map $e : G \times G \rightarrow G_1$
- $H : \{0,1\}^* \rightarrow G$ is a full domain hash function viewed as a random oracle that can be instantiated with a cryptographic hash function.

Formally, a bilinear aggregate signature scheme is a 5 tuple of algorithm *Key Generation, Signing, Verification, Aggregation*, and *Aggregate Verification*. We discuss the construction in Figure 4.

Security Informally, the security requirement of an aggregate signature scheme guarantees that the aggregate signature σ is valid if and only if the aggregator used all σ_i's, for $1 \leq i \leq k$, to

Figure 4: Bilinear Aggregate Signature Scheme

Key Generation: The secret key v is a random element of $\mathbb{Z}_p$ and the public key x is set to g^v.

Signing: The user signs the hash of each *distinct message* $M_i \in \{0,1\}^*$ via $\sigma_i \leftarrow H(M_i)^v$.

Verification: Given the user's public key x, a message M_i and its signature σ_i, accept if $e(\sigma_i, g) = e(H(M_i), x)$ holds.

Aggregation: This is a public algorithm which does not need the user's secret key to aggregate the individual signatures. Let σ_i be the signature on a distinct message $M_i \in \{0,1\}^*$ by the user, according to the Signing algorithm ($i = 1, \ldots, n$). The aggregate signature σ for a subset of k signatures, where $k \leq n$, is produced via $\sigma \leftarrow \prod_{i=1}^{k} \sigma_i$.

Aggregate Verification: Given the aggregate signature σ, k original messages $M_1, M_2, \ldots, M_k$ and the public key x:

1. ensure that all messages M_i are distinct, and reject otherwise.
2. accept if $e(\sigma, g) = e(\prod_{i=1}^{k} H(M_i), x)$.

construct it. The formal model of security is called the aggregate chosen-key security model. The security of aggregate signature schemes is expressed via a game where an adversary is challenged to forge an aggregate signature:

Setup: The adversary $\mathcal{A}$ is provided with a public key PK of the aggregate signature scheme.

Query: $\mathcal{A}$ adaptively requests signatures on messages of his choice.

Response: Finally, $\mathcal{A}$ outputs k distinct messages $M_1, M_2, \ldots, M_k$ and an aggregate signature σ. $\mathcal{A}$ wins if the aggregate signature σ is a valid aggregate signature on messages $M_1, M_2, \ldots, M_k$ under PK, and σ is nontrivial, i.e., $\mathcal{A}$ did not request a signature on $M_1, M_2, \ldots, M_k$ under PK. A formal definition and a corresponding security proof of the scheme can be found in [BGLS03].

2.3 Hardness assumption

Let p be a large k-bit prime where $k \in \mathbb{N}$ is a security parameter. Let $n \in \mathbb{N}$ be polynomial in k, $n = \mathsf{poly}(\mathsf{k})$. Let $e : G \times G \rightarrow G_1$ be a bilinear map where G and G_1 are groups of prime order p and g be a random generator of G. We denote a probabilistic polynomial time (PPT) adversary $\mathcal{A}$, or sometimes $\mathcal{B}$, as an adversary who is running in time $\mathsf{poly}(\mathsf{k})$. We use $\mathcal{A}^{\mathsf{alg}(\mathsf{input},\ldots)}$ to show that an adversary $\mathcal{A}$ has an oracle access to an instantiation of an algorithm alg with first argument set to input and $\ldots$ denoting that $\mathcal{A}$ can give arbitrary input for the rest of the arguments.

Definition 2.1 (n-Bilinear Diffie Hellman Inversion (n-BDHI) [BB04]) *Let s be a random element of $\mathbb{Z}_p^*$ and n be a positive integer. Then, for every PPT adversary $\mathcal{A}$ there exists a negligible function $\nu(.)$ such that: $Pr[s \xleftarrow{\$} \mathbb{Z}_p^*, y \leftarrow G_1 : \mathcal{A}(\langle g, g^s, g^{s^2}, \ldots, g^{s^n} \rangle) : y = e(g,g)^{\frac{1}{s}}] \leq \nu(k)$.*

3 Zero Knowledge List (ZKL)

We generalize the idea of consistent set membership queries [MRK03, CHL+05] to support membership and order queries in *zero knowledge* on a list with *no repeated elements*. More specifically, given a totally ordered list of unique elements $\mathcal{L} = \{y_1, y_2, \ldots, y_n\}$, we want to support in zero knowledge queries of the following form:

- Is $y_i \in \mathcal{L}$ or $y_i \notin \mathcal{L}$?
- For two elements $y_i, y_j \in \mathcal{L}$, what is their relative order, i.e., $y_i < y_j$ or $y_j < y_i$ in $\mathcal{L}$?

We adopt the same adversarial model as in [MRK03, ORS04, CHL+05]. Thus, we require that proofs reveal nothing beyond the query answer, not even the size of the list. There are two parties: the *prover* and the *verifier*. The *prover* initially commits to a list of values and makes the commitment (a short digest) public. Informally, the security properties can be stated as follows. Completeness mandates that honestly generated proofs always satisfy the verification test. Soundness states that the prover should not be able to come up with a query, and corresponding inconsistent (with the initial commitment) answers and convincing proofs. Finally, zero-knowledge means that each proof reveals the answer and nothing else. In other words, there must exist a simulator, that given only an oracle access to $\mathcal{L}$, can simulate proofs for membership and order queries that are indistinguishable from real proofs. Next, we formally describe the model and the security properties.

3.1 Model

A Zero Knowledge List scheme (ZKL) consists of three probabilistic polynomial time algorithms - $(\mathsf{Setup}, \mathsf{Prover} = (\mathsf{P}_1, \mathsf{P}_2), \mathsf{Verifier})$ and the queries are of the form (δ, flag) where $\delta = \{z_1, \ldots, z_m\}$, $z_i \in \{0,1\}^*$, is a collection of elements, $\mathsf{flag} = 0$ denotes a membership/non-membership query and $\mathsf{flag} = 1$ denotes an order query. In the following sections, we will use state to represent a variable that saves the current state of the algorithm (when it finishes execution).

$\mathsf{PK} \leftarrow \mathsf{Setup}(1^k)$

The Setup algorithm takes the security parameter as input and produces a public key PK for the scheme. The prover and the verifier both take as input the string PK that can be a random string (in which case, the protocol is in the common random string model) or have a specific structure (in which case the protocol is in the trusted parameters model).

$(\mathsf{com}, \mathsf{state}) \leftarrow \mathsf{P}_1(1^k, \mathsf{PK}, \mathcal{L})$

P_1 takes the security parameter, the public key PK and the list $\mathcal{L}$, and produces a short digest commitment com for the list.

$(\mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O) \leftarrow \mathsf{P}_2(\mathsf{PK}, \mathsf{state}, \delta, \mathsf{flag})$ where $\delta = \{z_1, \ldots, z_m\}$ and flag denotes the type of query. P_2 produces the membership information of the queried elements, $\mathsf{member} = \{\mathcal{L}(z_1), \ldots, \mathcal{L}(z_m)\}$ and the proof of membership (and non-membership), proof_M. Then depending on flag:

- $\mathsf{flag} = 0$**:** P_2 sets order and proof_O to $\perp$ and returns $(\mathsf{member}, \mathsf{proof}_M, \perp, \perp)$.
- $\mathsf{flag} = 1$**:** Let $\tilde{\delta} = \{z_i \mid i \in [1,m] \wedge \mathcal{L}(z_i) \neq \perp\}$. P_2 produces the correct list order among the elements of $\tilde{\delta}$, $\mathsf{order} = \pi_{\mathcal{L}}(\tilde{\delta})$ and the proof of the order, proof_O.

$b \leftarrow \mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O)$

$\mathsf{Verifier}$ takes the security parameter, the public key PK, the commitment com and a query (δ, flag) and $\mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O$ and returns a bit b, where $b = \mathsf{ACCEPT}/\mathsf{REJECT}$.

Example Let us illustrate the above functionality with a small example. Let $\mathcal{L} = \{A, B, C\}$ and $(\delta, \mathsf{flag}) = (\{B, D, A\}, 1)$ be the query. Then given this query P_2 returns $\mathsf{member} = \{\mathcal{L}(B), \mathcal{L}(D), \mathcal{L}(A)\} = \{\mathsf{true}, \mathsf{false}, \mathsf{true}\}$, the corresponding proofs of membership and non-membership in proof_M, $\mathsf{order} = \{A, B\}$ and the corresponding proof of order between A and B in proof_O.

3.2 Security Properties

Definition 3.1 (Completeness) *For every list $\mathcal{L}$, every sublist δ and every* flag*,*

$$\Pr[\mathsf{PK} \leftarrow \mathsf{Setup}(1^k); (\mathsf{com}, \mathsf{state}) \leftarrow \mathsf{P}_1(1^k, \mathsf{PK}, \mathcal{L}); (\mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O) \leftarrow \mathsf{P}_2(\mathsf{PK}, \mathsf{state}, \delta, \mathsf{flag}) : \mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O) = \mathsf{ACCEPT}] = 1$$

Definition 3.2 (Soundness) *For every PPT malicious prover algorithm,* Prover'*, for every sublist δ and for every* flag *there exists a negligible function $\nu(.)$ such that:*

$$\Pr[\mathsf{PK} \leftarrow \mathsf{Setup}(1^k); (\mathsf{com}, \mathsf{member}^1, \mathsf{proof}^1_M, \mathsf{order}^1, \mathsf{proof}^1_O, \mathsf{member}^2, \mathsf{proof}^2_M, \mathsf{order}^2, \mathsf{proof}^2_O) \leftarrow \mathsf{Prover}'(1^k, \mathsf{PK}) : \mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}^1, \mathsf{proof}^1_M, \mathsf{order}^1, \mathsf{proof}^1_O) = \mathsf{ACCEPT} \wedge \mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}^2, \mathsf{proof}^2_M, \mathsf{order}^2, \mathsf{proof}^2_O) = \mathsf{ACCEPT} \wedge ((\mathsf{member}^1 \neq \mathsf{member}^2) \vee (\mathsf{order}^1 \neq \mathsf{order}^2))] \leq \nu(k)$$

Definition 3.3 (Zero-Knowledge) *There exists a PPT simulator* $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Sim}_3)$ *such that for every PPT malicious verifier* $\mathsf{Adv} = (\mathsf{Adv}_1, \mathsf{Adv}_2)$*, there exists a negligible function $\nu(.)$ such that:*

$$|\Pr[\mathsf{PK} \leftarrow \mathsf{Setup}(1^k); (\mathcal{L}, \mathsf{state}_A) \leftarrow \mathsf{Adv}_1(1^k, \mathsf{PK}); (\mathsf{com}, \mathsf{state}_P) \leftarrow \mathsf{P}_1(1^k, \mathsf{PK}, \mathcal{L}) : \mathsf{Adv}_2^{\mathsf{P}_2(\mathsf{PK}, \mathsf{state}_P, \cdot)}(\mathsf{com}, \mathsf{state}_A) = 1] -$$
$$\Pr[(\mathsf{PK}, \mathsf{state}_S) \leftarrow \mathsf{Sim}_1(1^k); (\mathcal{L}, \mathsf{state}_A) \leftarrow \mathsf{Adv}_1(1^k, \mathsf{PK}); (\mathsf{com}, \mathsf{state}_S) \leftarrow \mathsf{Sim}_2(1^k, \mathsf{state}_S) : \mathsf{Adv}_2^{\mathsf{Sim}_3^{\mathcal{L}}(1^k, \mathsf{state}_S)}(\mathsf{com}, \mathsf{state}_A) = 1]| \leq \nu(k)$$

Here Sim_3 *has oracle access to $\mathcal{L}$, that is, given a query (δ, flag),* Sim_3 *can query the list $\mathcal{L}$ to learn only the membership/non-membership of elements in δ and, if* $\mathsf{flag} = 1$*, learn the list order of the elements of δ in $\mathcal{L}$.*

3.3 Zero Knowledge List (ZKL) Construction

Intuition The construction uses zero knowledge set scheme, homomorphic integer commitment scheme, zero-knowledge protocol to prove non-negativity of an integer and a collision resistant hash function $\mathbb{H} : \{0,1\}^* \rightarrow \{0,1\}^l$, if the elements of the list $\mathcal{L}$ are larger that l bits. In particular, given an input list $\mathcal{L}$ the prover creates a set D where for every element $y_j \in \mathcal{L}$ it adds a (key,value) pair $(\mathbb{H}(y_j), C(j))$ where $\mathbb{H}(y_j)$ is a hash of y_j and $C(j)$ is a homomorphic integer commitment of $\mathsf{rank}(\mathcal{L}, y_j)$ (assuming $\mathsf{rank}(\mathcal{L}, y_j) = j$ without loss of generality). The prover then sets up a zero knowledge set on D using ZKSP_1 from zero knowledge set construction in Figure 3. The output of ZKSP_1 is a commitment to D, com, that the prover sends to the verifier.

Membership and non-membership queries of the form $(\delta, 0)$ are replied in the same fashion as in zero knowledge set, by invoking ZKSP_2 on the hash of every element of sublist δ. Recall that as a response to a membership query for a key, ZKSP_2 returns the value against that key. In our case, the queried key is $\mathbb{H}(y_j)$ and the value returned by ZKSP_2, $D(\mathbb{H}(y_j))$ is the commitment $C(j)$ where j is the rank of element y_j in the list $\mathcal{L}$, if $y_j \in \mathcal{L}$. If $y_j \notin \mathcal{L}$, the value returned is $\perp$. Hence,

the verifier receives the commitments to ranks for queried member elements. These commitments are never opened but are used as part of a proof for order queries.

For a given order query $(\delta, 1)$, for every adjacent pair of elements in the returned order, order, the prover gives a proof of order. Recall that order contains the member elements of δ, arranged according to their order in the list, $\mathcal{L}$. To prove the order between two elements y_i, y_j, the prover does the following. Let $\mathsf{rank}(\mathcal{L}, y_i) = i, \mathsf{rank}(\mathcal{L}, y_j) = j$, and $C(i)$, $C(j)$ the corresponding commitments and wlog $i < j$. As noted above, $C(i), C(j)$ are already returned by the prover as a part of membership proof. Additionally, the prover augments the membership proof with a commitment to 1, $C(1)$, and its opening information ρ.

Then the verifier computes $C(j - i - 1) := C(j)/(C(i)C(1))$ using the homomorphic property of the integer commitment scheme. The prover and the verifier then engage in $\mathsf{Protocol}(x, r : c = C(x; r) \wedge x \geq 0)$ to convince the verifier that $C(j - i - 1)$ is a commitment to value $x = j - i - 1 \geq 0$. Note that we use the non-interactive zero-knowledge version of the protocol as discussed in Section 2.2.2.

It is important to understand why we require $\mathsf{Verifier}$ to verify that $j - i - 1 \geq 0$ and not $j - i \geq 0$. By the soundness of the protocol $\mathsf{Protocol}(x, r : c = C(x; r) \wedge x \geq 0)$, the probability that a cheating prover Prover' will be able to convince $\mathsf{Verifier}$ about the non-negativity of a negative integer is negligibly small. However, since 0 is non-negative, a cheating prover can do the following: instead of the rank of an element store the same arbitrary non-negative integer for every element in the list. Then, $C(j - i)$ and $C(i - j)$ are commitments to 0 and Prover' can always succeed in proving an arbitrary order. To avoid this attack, we require the prove to hold for $C(j - i - 1)$. An honest prover can always prove the non-negativity of $C(j - i - 1)$ as $|j - i| \geq 1$ for any rank i, j of the list.

Also, we note that the commitments to ranks can be replaced by commitments to a strictly monotonic sequence as long as there is a 1:1 correspondence with the rank sequence. In this case, the distance between two elements will also be positive and, hence, the above protocol still holds.

Construction Let $\mathsf{HomIntCom} = (\mathsf{IntComSetup}, \mathsf{IntCom}, \mathsf{IntComOpen})$ be the homomorphic integer commitment scheme defined in Section 2.2.1 and $\mathsf{ZKS} = (\mathsf{ZKSSetup}, \mathsf{ZKSProver} = (\mathsf{ZKSP}_1, \mathsf{ZKSP}_2), \mathsf{ZKSVerifier})$ be a ZKS scheme defined in Section 2.2.3. We denote the output of the prover during the non-interactive statistical zero knowledge protocol $\mathsf{Protocol}(x, r : c = C(x; r) \wedge x \geq 0)$ as $\mathsf{proof}_{x \geq 0}$. The construction also uses a hash function, $\mathbb{H} : \{0,1\}^* \to \{0,1\}^l$. In Figure 5 we describe in detail our ZKL construction on an input list $\mathcal{L} = \{y_1, \ldots, y_n\}$.

3.4 Security Proofs

Proof of Completeness Completeness of the ZKL construction in Section 3.3 directly follows from the Completeness of Zero Knowledge Set and Completeness of the protocol $\mathsf{Protocol}(x, r : c = C(x; r) \wedge x \geq 0)$. ∎

Proof of Soundness: To simplify the notation, first let us denote using $\mathbb{E}_1$ and $\mathbb{E}_2$ the following two events:

$$\mathbb{E}_1 = [\mathsf{PK} \leftarrow \mathsf{Setup}(1^k);$$
$$(\mathsf{com}, \mathsf{member}^1, \mathsf{proof}^1_M, \mathsf{order}^1, \mathsf{proof}^1_O, \mathsf{member}^2, \mathsf{proof}^2_M, \mathsf{order}^2, \mathsf{proof}^2_O) \leftarrow \mathsf{Prover}'(1^k, \mathsf{PK}) :$$
$$\mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}^1, \mathsf{proof}^1_M, \mathsf{order}^1, \mathsf{proof}^1_O) = \mathsf{ACCEPT} \wedge$$
$$\mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}^2, \mathsf{proof}^2_M, \mathsf{order}^2, \mathsf{proof}^2_O) = \mathsf{ACCEPT} \wedge$$
$$(\mathsf{member}^1 \neq \mathsf{member}^2)]$$

Figure 5: Zero Knowledge List (ZKL) Construction

$\mathsf{PK} \leftarrow \textbf{Setup}(1^k)$**:** The Setup algorithm takes the security parameter as input and runs $\mathsf{PK}_C \leftarrow \mathsf{IntComSetup}(1^k), \mathsf{PK}_D \leftarrow \mathsf{ZKSSetup}(1^k)$ and outputs $\mathsf{PK} = (\mathsf{PK}_C, \mathsf{PK}_D)$.

$(\mathsf{com}, \mathsf{state}) \leftarrow \mathbf{P_1}(1^k, \mathsf{PK}, \mathcal{L})$**:** Wlog, let $\mathsf{rank}(\mathcal{L}, y_j) = j$ and $C(j)$ denote an integer commitment to j under public key PK_C, i.e., $(C(j), r_j) = \mathsf{IntCom}(\mathsf{PK}_C, j)$. Then, P_1 proceeds as follows:

- For every $y_j \in \mathcal{L}$, compute $\mathbb{H}(y_j)$ and $C(j)$.
- Set $D := \{(\mathbb{H}(y_j), C(j)) \mid \forall y_j \in \mathcal{L}\}$.
- Run $(\mathsf{com}, \mathsf{state}) \leftarrow \mathsf{ZKSP}_1(1^k, \mathsf{PK}_D, D)$ and output $(\mathsf{com}, \mathsf{state})$.

$(\mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O) \leftarrow \mathbf{P_2}(\mathsf{PK}, \mathsf{state}, \delta, \mathsf{flag})$ **where** $\delta = \{z_1, \ldots, z_m\}$**:** Let $S := \{\mathbb{H}(z_1), \ldots, \mathbb{H}(z_m)\}$. For all $x \in S$ do the following:

- Run $(D(x), \mathsf{proof}_x) \leftarrow \mathsf{ZKSP}_2(\mathsf{PK}_D, \mathsf{state}, x)$.
- Set $\Delta_x := (D(x), \mathsf{proof}_x)$.

Set $\mathsf{member} := \{\mathcal{L}(z_j) \mid \forall z_j \in \delta\}$ and $\mathsf{proof}_M := \{\Delta_x \mid x \in S\}$.

If $\mathsf{flag} = 0$ return $(\mathsf{member}, \mathsf{proof}_M, \perp, \perp)$.

If $\mathsf{flag} = 1$ do the following:

Let $\tilde{\delta} = \{z_j \mid \forall j \in [1, m] \wedge \mathcal{L}(z_j) \neq \perp\}$ and $\pi_{\mathcal{L}}(\tilde{\delta}) = \{w_1, \ldots, w_{m'}\}$ where $m' \leq m$.

- For all $1 \leq j < m'$, compute $\Delta_{w_j < w_{j+1}} = \mathsf{proof}_{\mathsf{rank}(\mathcal{L}, w_{j+1}) - \mathsf{rank}(\mathcal{L}, w_j) - 1 \geq 0}$.
- Compute $(C(1), \rho) = \mathsf{IntCom}(\mathsf{PK}_C, 1)$.

Set $\mathsf{order} := \pi_{\mathcal{L}}(\tilde{\delta})$ and $\mathsf{proof}_O = (\{\Delta_{w_j < w_{j+1}} \mid (w_j, w_{j+1}) \in \tilde{\delta}\}, C(1), \rho)$ and return $(\mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O)$.

$b \leftarrow \textbf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O)$ **where** $\delta = \{z_1, \ldots, z_m\}$**:** The Verifier algorithm does the following:

- Compute $S = \{\mathbb{H}(z_1), \ldots, \mathbb{H}(z_m)\}$.
- Parse proof_M as $\mathsf{proof}_M := \{\Delta_x = (D(x), \mathsf{proof}_x) \mid x \in S\}$.
- For all $x \in S$, run $b \leftarrow \mathsf{ZKSVerifier}(1^k, \mathsf{PK}_D, x, D(x), \mathsf{proof}_x)$.

If $\mathsf{flag} = 0$ and $b = \mathsf{ACCEPT}$ for all $x \in S$, output ACCEPT.

If $\mathsf{flag} = 1$, perform the following additional verification steps:

- Let $\mathsf{order} = \{w_1, \ldots, w_{m'}\}$.
- Parse proof_O as $(\{\Delta_{w_j < w_{j+1}} \mid (w_j, w_{j+1}) \in \mathsf{order}\}, C(1), \rho)$.
- Verify that $\mathsf{IntComOpen}(\mathsf{PK}_C, C(1), \rho)$ is 1.
- Compute $D(\mathbb{H}(w_{j+1}))/(D(\mathbb{H}(w_j)) \times C(1)) = C(\mathsf{rank}(\mathcal{L}, w_{j+1}) - \mathsf{rank}(\mathcal{L}, w_j) - 1)$
- Verify that $\mathsf{rank}(\mathcal{L}, j+1) - \mathsf{rank}(\mathcal{L}, j) > 0$ using $\mathsf{proof}_{\mathsf{rank}(\mathcal{L}, j+1) - \mathsf{rank}(\mathcal{L}, j) - 1 \geq 0}$ using $\mathsf{Protocol}(x, r : c = C(x; r) \wedge x \geq 0)$ where $x = \mathsf{rank}(\mathcal{L}, j+1) - \mathsf{rank}(\mathcal{L}, j) - 1$.

If all the verifications pass, only then return ACCEPT.

$$\mathbb{E}_2 = [\mathsf{PK} \leftarrow \mathsf{Setup}(1^k);$$
$$(\mathsf{com}, \mathsf{member}^1, \mathsf{proof}^1_M, \mathsf{order}^1, \mathsf{proof}^1_O, \mathsf{member}^2, \mathsf{proof}^2_M, \mathsf{order}^2, \mathsf{proof}^2_O) \leftarrow \mathsf{Prover}'(1^k, \mathsf{PK}) :$$
$$\mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}^1, \mathsf{proof}^1_M, \mathsf{order}^1, \mathsf{proof}^1_O) = \mathsf{ACCEPT} \wedge$$
$$\mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}^2, \mathsf{proof}^2_M, \mathsf{order}^2, \mathsf{proof}^2_O) = \mathsf{ACCEPT} \wedge$$
$$(\mathsf{order}^1 \neq \mathsf{order}^2)]$$

Then, Definition 3.2 can be rewritten as

$$\Pr[\mathsf{PK} \leftarrow \mathsf{Setup}(1^k);$$
$$(\mathsf{com}, \mathsf{member}^1, \mathsf{proof}^1_M, \mathsf{order}^1, \mathsf{proof}^1_O, \mathsf{member}^2, \mathsf{proof}^2_M, \mathsf{order}^2, \mathsf{proof}^2_O) \leftarrow \mathsf{Prover}'(1^k, \mathsf{PK}) :$$
$$\mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}^1, \mathsf{proof}^1_M, \mathsf{order}^1, \mathsf{proof}^1_O) = \mathsf{ACCEPT} \wedge$$
$$\mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}^2, \mathsf{proof}^2_M, \mathsf{order}^2, \mathsf{proof}^2_O) = \mathsf{ACCEPT} \wedge$$
$$((\mathsf{member}^1 \neq \mathsf{member}^2) \vee (\mathsf{order}^1 \neq \mathsf{order}^2))] = \Pr[\mathbb{E}_1 \vee \mathbb{E}_2] \leq \Pr[\mathbb{E}_1] + \Pr[\mathbb{E}_2]$$

Now, by the Soundness property of the ZKS in Section 2.2.3, $\Pr[\mathbb{E}_1]$ is negligible in k. Let $\Pr[\mathbb{E}_1] = \nu_1(k)$.

Let us consider the event $\mathbb{E}_2$. If the malicious prover is successful in outputting two contradictory orders for a collection of elements, then there must exist at least one inversion pair, i.e., a pair of elements $(x_i, x_j) \in \delta$ such that $x_i < x_j$ in order^1 and $x_j < x_i$ in order^2. Let $C(i)$ and $C(j)$ be the commitments used as values to prove the membership of x_i and x_j, correspondingly. Then by the binding property of the integer commitment scheme of Section 2.2.1, Prover' cannot equivocate $C(i-j)$ or $C(j-i)$ (which is computed by $\mathsf{Verifier}$ in the protocol). (Note that by the soundness property of ZKS, the probability that Prover' can return two commitments $C(i)$ and $C(i')$, $C(i) \neq C(i')$, where $C(i)$ and $C(i')$ are returned to prove membership of x_i in proof^1_M and proof^2_M, respectively, is negligible w.r.t. the same commitment, com.) Then according to the protocol, it must be the case that Prover' could convince $\mathsf{Verifier}$ that both $C(i-j)$ and $C(j-i)$ are commitments to positive integers where i, j are two integers. However, due to the soundness of the protocol $\mathsf{Protocol}(x, r : c = C(x; r) \wedge x \geq 0)$, the probability is negligible in k. Let $\Pr[\mathbb{E}_2] = \nu_2(k)$.

Therefore we have, $\Pr[\mathbb{E}_1 \vee \mathbb{E}_2] \leq \nu_1(k) + \nu_2(k) \leq \nu(k)$, for some negligible function $\nu(.)$ Hence the soundness error of the ZKL construction must be negligible in k. ∎

Proof of Zero-Knowledge: Let $\mathsf{SimHomIntCom} = (\mathsf{SimIntComSetup}, \mathsf{SimIntCom}, \mathsf{SimIntComOpen})$ be the simulator of $\mathsf{HomIntCom}$ defined in Figure 1. Let $\mathsf{SimZKS} = (\mathsf{SimZKSSetup}, \mathsf{SimZKSProver} = (\mathsf{SimZKSP}_1, \mathsf{SimZKSP}_2), \mathsf{SimZKSVerifier})$ be the simulator for the ZKS in Figure 3.

Now let us define $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Sim}_3)$, a simulator for ZKL (Definition 3.3), that has access to the system parameter $\mathbb{H}$.

- $\mathsf{Sim}_1(1^k)$ runs $(\mathsf{PK}_D, \mathsf{TK}_D) \leftarrow \mathsf{SimZKSSetup}(1^k)$ and $(\mathsf{PK}_C, \mathsf{TK}_C) \leftarrow \mathsf{SimIntComSetup}(1^k)$. $\mathsf{Sim}_1(1^k)$ outputs $\{\mathsf{PK} = (\mathsf{PK}_D, \mathsf{PK}_C), \mathsf{TK} = (\mathsf{TK}_D, \mathsf{TK}_C)\}$.
- Sim_2 runs $\mathsf{SimZKSP}_1$ to generate commitment com.
- In response to membership queries ($\mathsf{flag} = 0$), Sim_3 does the following:
  - Sim_3 maintains a table of queried elements as tuples $\langle x_i, v_i, r_i \rangle$ where x_i is the queried element and v_i is the value that Sim_3 has sent when x_i was queried. We explain how r_i is computed next.
  - For a queried element y, Sim_3 checks the table. If y is not in the table and, hence, has not been queried before, Sim_3 makes an oracle access to $\mathcal{L}$ on y. If $y \in \mathcal{L}$, Sim_3 computes a fresh commitment to 0, $(C(0), r) := \mathsf{SimIntCom}(0)$, and stores $\langle y, C(0), r \rangle$. If $y \notin \mathcal{L}$, then Sim_3 stores $\langle y, \perp, \perp \rangle$.
  - Sim_3 responds to membership queries by invoking $\mathsf{SimZKSP}_2$ on $\mathbb{H}(y)$ and returning the same output.
- For order queries ($\mathsf{flag} = 1$), Sim_3 additionally does the following. Let δ be the queried sublist. Sim_3 makes an oracle access to $\mathcal{L}$ to get the correct list order of the elements of δ that are present in $\mathcal{L}$. Let $\mathsf{order} = \{y_1, \ldots, y_m\}$ be the returned order.
- Sim_3 computes $(C(1), \rho) = \mathsf{SimIntCom}(\mathsf{PK}_C, 1)$.

- Let $\{\langle y_1, v_1, r_1\rangle, \ldots, \langle y_m, v_m, r_m\rangle\}$ be the entries of Sim_3's table that correspond to elements in δ. Then for every pair (y_j, y_{j+1}), Sim_3 equivocates $(v_{j+1}/(v_j \times C(1)))$ using TK_C to a commitment to any arbitrary positive integer u. In other words, Sim_3 equivocates the commitment $C(\mathsf{rank}(\mathcal{L}, y_{j+1}) - \mathsf{rank}(\mathcal{L}, y_j) - 1)$ to a commitment to an arbitrary positive integer u. Finally, Sim_3 computes $\mathsf{proof}_{u\geq 0}$ to prove the order between (y_j, y_{j+1}).

Sim_3 achieves the following. For every newly queried element that is in the list, Sim_3 generates and stores a fresh commitment to 0, and sends it to the verifier. Hence, Sim_3 sets $\mathsf{rank} = 0$ to all queried elements. By the hiding property of the integer commitment scheme, the commitments are identically distributed to the commitments computed by the real prover, P_1. Now, with the help of TK_C, Sim_3 can equivocate a commitment to any value it wants. Hence, whenever he needs to prove order $y_i < y_j$, Sim_3 equivocates the commitment to $\mathsf{rank}(\mathcal{L}, y_{j+1}) - \mathsf{rank}(\mathcal{L}, y_j) - 1$ to any arbitrary positive integer u and invokes the protocol $\mathsf{Protocol}(u, r : c = C(u; r) \wedge u \geq 0)$ to compute $\mathsf{proof}_{u>0}$.

Since the protocol $\mathsf{Protocol}(u, r : c = C(u; r) \wedge u \geq 0)$ is Zero Knowledge (Statistical), $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Sim}_3)$ simulates our ZKL scheme. ∎

We note that the constructions with which we instantiate ZKL have the simulators assumed above. In particular, for SimZKS we use the simulator of the ZKS construction of [CHL+05]. For $\mathsf{SimHomIntCom}$ we use the construction of [DF02] and for completeness define a simulator in Figure 9.

3.5 Efficiency

The efficiency of our ZKL construction depends on the efficiency of the underlying constructions that we use. We consider the the ZKS construction used in [CHL+05] based on Mercurial Commitments, the homomorphic integer commitment of [DF02] and a protocol for non-negative proof of a commitment from [Lip03]. Each of these constructions is described in more detail in Appendix. Mercurial commitment was later generalized by [CFM08, LY10] but the basic ZKS construction remains the same.

Recall that k is the security parameter of the scheme, l is the size of the output of the hash function $\mathbb{H}$, n is the number of elements in the list $\mathcal{L}$ and m is the number of elements in query δ. Similarly to [CHL+05] we assume that $l = k$. For every element in $\mathcal{L}$, P_1 hashes the element and computes a commitment to its rank, taking time $O(1)$. It then computes n height-k paths to compute the commitment com to a list, $\mathcal{L}$, takes time $O(kn)$, where $|\mathcal{L}| = n$. For further details please see Appendix C.

Membership (non-membership) proof of a single element consists of $O(k)$ mercurial decommitments. Using [LY10], we can have each mercurial decommitment constant size, i.e, $O(1)$. The order proof between two elements requires membership proofs for both elements and $\mathsf{proof}_{u-1\geq 0}$ where u is the absolute difference between the rank of the corresponding elements. $\mathsf{proof}_{u-1\geq 0}$ is computed using $\mathsf{Protocol}(x, r : c = C(x; r) \wedge x \geq 0)$ which takes $O(1)$ time. Hence, computing a membership proof for a single element or an order proof for two elements takes time $O(k)$. More generally, the prover's time for a query on sublist δ is $O(mk)$, where $m = |\delta|$.

The verifier needs to verify $O(k)$ mercurial decommitments for every element in the query δ and verify order between every adjacent pair of elements in δ using $\mathsf{Protocol}(u, r : c = C(u; r) \wedge u \geq 0)$. Therefore, the asymptotic run time of the verification is $O(mk)$.

We summarize the properties and efficiency of our ZKL construction in Theorem 3.1.

Theorem 3.1 *The zero-knowledge list (ZKL) construction of Figure 5 satisfies the security properties of completeness (Definition 3.1), soundness (Definition 3.2) and zero-knowledge (Definition 3.3). The construction has the following performance, where n is the list size, m is the query*

size, each element of the list is a k-bit[1] *string and N is the number of all possible k-bit strings.*

- *The prover executes the setup phase in $O(n \log N)$ time and space.*
- *In the query phase, the prover computes the proof of the answer to a query in $O(m \log N)$ time.*
- *The verifier verifies the proof in $O(m \log N)$ time and space.*

4 Privacy Preserving Authenticated List (PPAL)

In the previous section we presented a model and a construction for a new primitive called zero knowledge lists. As we noticed earlier, ZKL model gives the desired functionality to verify order queries on lists. However, the corresponding construction does not provide the efficiency one may desire in cloud computing setting where the verifier (client) has limited memory resources. In this section we address this setting and define a model for privacy preserving authenticated lists, PPAL, that is executed between three parties. This model, arguably, fits cloud scenario better and as we will see our construction is also more efficient. In particular, the size of a single proof in PPAL is $O(1)$ vs. $O(k)$ in ZKL.

4.1 Model

PPAL is a tuple of three probabilistic polynomial time algorithms $(\mathsf{Setup}, \mathsf{Query}, \mathsf{Verify})$ executed between the owner of the data list $\mathcal{L}$, the server who stores $\mathcal{L}$ and answers queries from the client and the client who issues queries and verifies corresponding answers.

$(\mathsf{digest}_C, \mathsf{digest}_S) \leftarrow \mathsf{Setup}(1^k, \mathcal{L})$

This algorithm takes the security parameter and the source list $\mathcal{L}$ as input and produces two digests digest_C and digest_S for the list. This algorithm is run by the owner. digest_C is sent to the client and digest_S is sent to the server.

$(\mathsf{order}, \mathsf{proof}) \leftarrow \mathsf{Query}(\mathsf{digest}_S, \mathcal{L}, \delta)$

This algorithm takes the key generated by the owner, digest_S, the source list, $\mathcal{L}$ and a queried sublist, δ, as input, where a sublist of a list $\mathcal{L}$ is defined as: $\mathsf{Elements}(\delta) \subseteq \mathsf{Elements}(\mathcal{L})$. The algorithm produces the list order of the elements of $\mathcal{L}$, $\mathsf{order} = \pi_{\mathcal{L}}(\delta)$, and a proof, proof, of the answer. This algorithm is run by the server.

$b \leftarrow \mathsf{Verify}(\mathsf{digest}_C, \delta, \mathsf{order}, \mathsf{proof})$

This algorithm takes digest_C, a queried sublist δ, order and proof and returns a bit b, where $b = \mathsf{ACCEPT}$ iff $\mathsf{Elements}(\delta) \subseteq \mathsf{Elements}(\mathcal{L})$ and $\mathsf{order} = \pi_{\mathcal{L}}(\delta)$. Otherwise, $b = \mathsf{REJECT}$. This algorithm is run by the client.

4.2 Security Properties

A PPAL has three important security properties. The first property is *Completeness*. This property ensures that for any list $\mathcal{L}$ and for any sublist δ of $\mathcal{L}$, if the $\mathsf{digest}_C, \mathsf{digest}_S, \mathsf{order}, \mathsf{proof}$ are generated honestly, i.e., the owner and the server honestly execute the protocol, then the client will be always convinced about the correct list order of δ.

Definition 4.1 (Completeness) *For all lists $\mathcal{L}$ and all sublists δ*

$$\Pr[(\mathsf{digest}_C, \mathsf{digest}_S) \leftarrow \mathsf{Setup}(1^k, \mathcal{L}); (\mathsf{order}, \mathsf{proof}) \leftarrow \mathsf{Query}(\mathsf{digest}_S, \mathcal{L}, \delta) : \mathsf{Verify}(\mathsf{digest}_C, \delta, \mathsf{order}, \mathsf{proof}) = \mathsf{ACCEPT}] = 1$$

[1] If not, we can use a hash function to reduce every element to a k-bit string, as shown in the construction

The second security property is *Soundness*. This property ensures that once an honest owner generates a pair $(\mathsf{digest}_C, \mathsf{digest}_S)$ for a given list $\mathcal{L}$, even a malicious server will not be able to convince the client of incorrect order of elements belonging to the list $\mathcal{L}$. This property ensures integrity of the scheme.

Definition 4.2 (Soundness) *For all PPT malicious Query algorithms* Query'*, for all lists* $\mathcal{L}$ *and all query sublists* δ*, there exists a negligible function* $\nu(.)$ *such that:*

$$\begin{aligned}\Pr[(\mathsf{digest}_C, \mathsf{digest}_S) \leftarrow \mathsf{Setup}(1^k, \mathcal{L}); (\mathsf{order}_1, \mathsf{proof}_1, \mathsf{order}_2, \mathsf{proof}_2) \leftarrow \mathsf{Query}'(\mathsf{digest}_S, \mathcal{L}) : \\ \mathsf{Verify}(\mathsf{digest}_C, \delta, \mathsf{order}_1, \mathsf{proof}_1) = \mathsf{ACCEPT} \wedge \\ \mathsf{Verify}(\mathsf{digest}_C, \delta, \mathsf{order}_2, \mathsf{proof}_2) = \mathsf{ACCEPT} \wedge \\ (\mathsf{order}_1 \neq \mathsf{order}_2)] \leq \nu(k)\end{aligned}$$

The last property is *Zero-Knowledge*. This property captures that even a malicious client cannot learn anything about the list (and its size) beyond what the client has queried for. Informally, this property involves showing that there exists a simulator such that even for adversarially chosen list $\mathcal{L}$, no adversarial client (verifier) can tell if it is talking to an honest owner and server pair who are committed to $\mathcal{L}$ or to the simulator who only has oracle access to the list $\mathcal{L}$.

Definition 4.3 (Zero-Knowledge) *There exists a PPT simulator* $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ *such that for all PPT malicious verifiers* $\mathsf{Adv} = (\mathsf{Adv}_1, \mathsf{Adv}_2)$*, there exists a negligible function* $\nu(.)$ *such that:*

$$\begin{aligned}|\Pr[(\mathcal{L}, \mathsf{state}_A) \leftarrow \mathsf{Adv}_1(1^k); (\mathsf{digest}_C, \mathsf{digest}_S) \leftarrow \mathsf{Setup}(1^k, \mathcal{L}) : \\ \mathsf{Adv}_2^{\mathsf{Query}(\mathsf{digest}_S, \mathcal{L}, .)}(\mathsf{digest}_C, \mathsf{state}_A) = 1] - \\ \Pr[(\mathcal{L}, \mathsf{state}_A) \leftarrow \mathsf{Adv}_1(1^k); (\mathsf{digest}_C, \mathsf{state}_S) \leftarrow \mathsf{Sim}_1(1^k) : \\ \mathsf{Adv}_2^{\mathsf{Sim}_2^{\mathcal{L}}(1^k, \mathsf{state}_S)}(\mathsf{digest}_C, \mathsf{state}_A) = 1]| \leq \nu(k)\end{aligned}$$

Here Sim_2 has oracle access to $\mathcal{L}$, that is given a sublist δ of $\mathcal{L}$, Sim_2 can query the list $\mathcal{L}$ to learn only the correct list order of the sublist δ and cannot look at $\mathcal{L}$.

Attack on [KAB12]'s scheme We observer that the scheme presented in [KAB12] does not satisfy the zero knowledge property of PPAL for the following reason. The scheme of [KAB12] generates a n' bit secure name, where $n' \geq n$, for each element of the list of size n. A high level idea of the scheme is as follows. The secure name of an element has dedicated bits, where each bit corresponds to the pairwise order between this element and every other element in the list. To prove the order between any two elements, the verifier needs to know secure names for both of them. Then, given any two secure names, the verifier can easily compute the required bit. Two order queries $A < B$ and $A < C$, as per the scheme of [KAB12], reveal to the client the secure names of all three elements A, B and C. Hence, given the secure names of B and C, the client can easily compute the bit which preserves the order information between B and C and infer the order between them. Therefore, it is impossible to write a simulator Sim for an adversarially generated list such that the view of the adversary is indistinguishable as in Definition 4.3.

5 PPAL Construction

We present an implementation of a privacy preserving authenticated list in Figure 7. We provide the intuition of our method followed by a more detailed description.

Intuition Every element of the list is associated with a member witness where a member witness is a randomized bilinear accumulator. This allows us to encode the rank of the element (i.e., accumulate it) inside of the member witness and then "blind" this rank information with randomness. Every pair of (element, member witness) is signed by the owner and the signatures are aggregated using bilinear aggregate signature scheme presented in Figure 4, to compute the list digest signature. Signatures and digest are sent to the server, who can use them to prove authenticity when answering client queries. The advantage of using an aggregate signature is for the the server to be able to compute a valid digest signature for any sublist of the source list by exploiting the homomorphic nature of aggregate signatures, that is without owner's involvement. Moreover, the client can verify the individual signatures in a single shot using aggregate signature verification.

The owner also sends linear (in the list size) number of random elements used in the encoding of member witnesses. These random elements allow the server to compute the order witnesses on queried elements, without the owner's involvement. The order witness encodes the distance between two elements, i.e., the difference between element ranks, without revealing anything about it. Together with randomized accumulators as member witnesses, the client can later use bilinear map to verify the order of the elements.

Construction Our construction for PPAL is presented in Figure 7. It is based on bilinear accumulators and bilinear aggregate signature introduced in [BGLS03] and described here in Section 2.2.4. We denote *member witness* for $x_i \in \mathcal{L}$ as $t_{x_i \in \mathcal{L}}$. For two elements $x_i, x_j \in \mathcal{L}$, such that $x_i < x_j$ in $\mathcal{L}$, $t_{x_i < x_j}$ is an *order witness* for the order between x_i and x_j.

The construction works as follows. In the Setup phase, the owner generates secret key (v, s) and public key g^v, where v is used for signatures. The owner picks a distinct random element r_i from the group $\mathbb{Z}_p^*$ for each element x_i in the list $\mathcal{L}$, $i \in [1, n]$. The element r_i is used to compute the member witness $t_{x_i \in \mathcal{L}}$. Later in the protocol, together with r_j, it is also used by the server to compute the order witness $t_{x_i < x_j}$ for x_i and $x_j \in \mathcal{L}$ where $x_i < x_j$ in $\mathcal{L}$. The owner also computes individual signatures, σ_i's, for each element and aggregates them into a digest signature $\sigma_{\mathcal{L}}$ for the list. It returns the signatures and member witnesses for every element of $\mathcal{L}$ in $\Sigma_{\mathcal{L}}$ and the set of random numbers picked for each index to be used in order witnesses in $\Omega_{\mathcal{L}}$. The owner sends $\mathsf{digest}_C = (g^v, \sigma_{\mathcal{L}})$ to the client and $\mathsf{digest}_S = (g^v, \sigma_{\mathcal{L}}\ , \langle g, g^s, g^{s^2}, \ldots, g^{s^n} \rangle, \Sigma_{\mathcal{L}}, \Omega_{\mathcal{L}})$ and $\mathcal{L}$ to the server.

Given a query δ, the server returns a response list order that contains elements of δ in the order they appear in $\mathcal{L}$. The server uses information in $\Sigma_{\mathcal{L}}$ to build Σ_{order} from member witnesses of elements in δ, and compute the digest signature σ_δ for δ and its membership verification unit $\lambda_{\mathcal{L}'}$ where $\mathcal{L}' = \mathcal{L} \setminus \delta$. The server uses information in $\Omega_{\mathcal{L}}$ to compute Ω_{order}. The client first checks that all the returned elements are indeed signed by the owner using Σ_{order} and then verifies the order of the returned elements using Ω_{order}.

Preprocessing at the Server For a query δ on the list $\mathcal{L}$ of length m and n, respectively, the Query algorithm in Figure 7 takes $O(m)$ time to compute σ_δ and $O(n - m)$ to compute $\lambda_{\mathcal{L}'}$. The server can precompute and store some products to reduce the overall running time of this algorithm to $O(m \log n)$ when $m \ll n$. The precomputation proceeds as follows.

Let $\psi_i = \mathcal{H}(t_{x_i \in \mathcal{L}} || x_i)$ for every element in $\mathcal{L} = \{x_1, \ldots, x_n\}$. Then the precomputation proceeds by computing a balanced binary tree over n leaves, where ith leave corresponds to x_i and stores ψ_i. Each internal node of the tree stores the product of its children. Therefore the root stores the complete product $\prod_{i=1}^{n} \psi_i$. (See Figure 6 for an illustration of the tree.) Computing each internal node takes time $O(1)$ since at each internal node product of at most two children is computed. Since the tree has $O(n)$ nodes, the precomputation takes time $O(n)$ and requires $O(n)$ storage.

Figure 7: Privacy-Preserving Authenticated List (PPAL) Construction

Notation: $k \in \mathbb{N}$ is the security parameter of the scheme; G, G_1 multiplicative cyclic groups of prime order p where p is large k-bit prime; g: a random generator of G; e: computable bilinear nondegenerate map $e : G \times G \to G_1$; $\mathcal{H} : \{0,1\}^* \to G$: full domain hash function (instantiated with a cryptographic hash function); all arithmetic operations are performed using $\bmod p$. $\mathcal{L}$ is the input list of size $n = \mathsf{poly}(\mathsf{k})$, where x_i are distinct and $\mathsf{rank}(\mathcal{L}, x_i) = i$. System parameters are $(p, G, G_1, e, g, \mathcal{H})$.

$(\mathsf{digest}_C, \mathsf{digest}_S) \leftarrow \textbf{Setup}(1^k, \mathcal{L})$, where

$\mathcal{L}$ is the input list of length n;

$\mathsf{digest}_C = (g^v, \sigma_{\mathcal{L}})$;

$\mathsf{digest}_S = (g^v, \sigma_{\mathcal{L}}\ , \langle g, g^s, g^{s^2}, \ldots, g^{s^n}\rangle, \Sigma_{\mathcal{L}}, \Omega_{\mathcal{L}})$ and

$\langle s \xleftarrow{\$} \mathbb{Z}_p^*, v \xleftarrow{\$} \mathbb{Z}_p^*\rangle$ is the secret key of the owner;

$\Sigma_{\mathcal{L}} = \langle \{t_{x_i \in \mathcal{L}}, \sigma_i\}_{1 \le i \le n}, \mathcal{H}(\omega)\rangle$ is member authentication information and ω is the list nonce;

$\Omega_{\mathcal{L}} = \langle r_1, r_2, \ldots, r_n\rangle, r_i \neq r_j$ for $i \neq j$, is order authentication informations;

$\sigma_{\mathcal{L}}$ is the digest signature of the list $\mathcal{L}$.

These elements are computed as follows:

For every element x_i in $\mathcal{L} = \{x_1, \ldots, x_n\}$: Pick $r_i \xleftarrow{\$} \mathbb{Z}_p^*$. Compute member witness for index i as $t_{x_i \in \mathcal{L}} \leftarrow (g^{s^i})^{r_i}$ and signature for element x_i as $\sigma_i \leftarrow \mathcal{H}(t_{x_i \in \mathcal{L}} || x_i)^v$.

Pick the nonce, $\omega \xleftarrow{\$} \{0,1\}^*$, which should be unique for each list.

Set $\mathsf{salt} \leftarrow (\mathcal{H}(\omega))^v$. salt is treated as a list identifier which protects against mix-and-match attack and also protects from the leakage that the queried result is the complete list.

The list digest signature is computed as: $\sigma_{\mathcal{L}} \leftarrow \mathsf{salt} \times \prod_{1 \le i \le n} \sigma_i$.

$(\mathsf{order}, \mathsf{proof}) \leftarrow \textbf{Query}(\mathsf{digest}_S, \mathcal{L}, \delta)$, where

$\delta = \{z_1, \ldots, z_m\}$ s.t. $z_i \in \mathcal{L}$, $\forall i \in [1, m]$, is the queried sublist;

$\mathsf{order} = \pi_{\mathcal{L}}(\delta) = \{y_1, y_2, \ldots, y_m\}$;

$\mathsf{proof} = (\Sigma_{\mathsf{order}}, \Omega_{\mathsf{order}})$:

$\Sigma_{\mathsf{order}} = (\sigma_{\mathsf{order}}, T, \lambda_{\mathcal{L}'})$ where $\mathcal{L}' = \mathcal{L} \setminus \delta$;

$T = \{t_{y_1 \in \mathcal{L}}, \ldots, t_{y_m \in \mathcal{L}}\}$;

$\Omega_{\mathsf{order}} = \{t_{y_1 < y_2}, t_{y_2 < y_3}, \ldots, t_{y_{m-1} < y_m}\}$.

These elements are computed as follows:

The digest signature for the sublist: $\sigma_{\mathsf{order}} \leftarrow \prod_{y_j \in \mathsf{order}} \sigma_{\mathsf{rank}(\mathcal{L}, y_j)}$.

The member verification unit: $\lambda_{\mathcal{L}'} \leftarrow \mathcal{H}(\omega) \times \prod_{x \in \mathcal{L}'} \mathcal{H}(t_{x_{\mathsf{rank}(\mathcal{L}, x)} \in \mathcal{L}} || x)$.

For every $j \in [1, m-1]$: Let $i' = \mathsf{rank}(\mathcal{L}, y_j)$ and $i'' = \mathsf{rank}(\mathcal{L}, y_{j+1})$, and $r' = \Omega_{\mathcal{L}}[i']^{-1}$ and $r'' = \Omega_{\mathcal{L}}[i'']$. Compute $t_{y_j < y_{j+1}} \leftarrow (g^{s^d})^{r'r''}$ where $d = |i' - i''|$.

$b \leftarrow \textbf{Verify}(\mathsf{digest}_C, \delta, \mathsf{order}, \mathsf{proof})$ where $\mathsf{digest}_C, \delta, \mathsf{order}, \mathsf{proof}$ are defined as above.

The algorithm checks the following:

- Compute $\xi \leftarrow \prod_{y_j \in \delta} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_j)$ and check if $e(\sigma_{\mathsf{order}}, g) \stackrel{?}{=} e(\xi, g^v)$
- Check if $e(\sigma_{\mathcal{L}}, g) \stackrel{?}{=} e(\sigma_{\mathsf{order}}, g) \times e(\lambda_{\mathcal{L}'}, g^v)$
- For every $j \in [1, m-1]$, $e(t_{y_j \in \mathcal{L}}, t_{y_j < y_{j+1}}) \stackrel{?}{=} e(t_{y_{j+1} \in \mathcal{L}}, g)$

Return ACCEPT iff all the equalities of the three steps verify, REJECT otherwise.

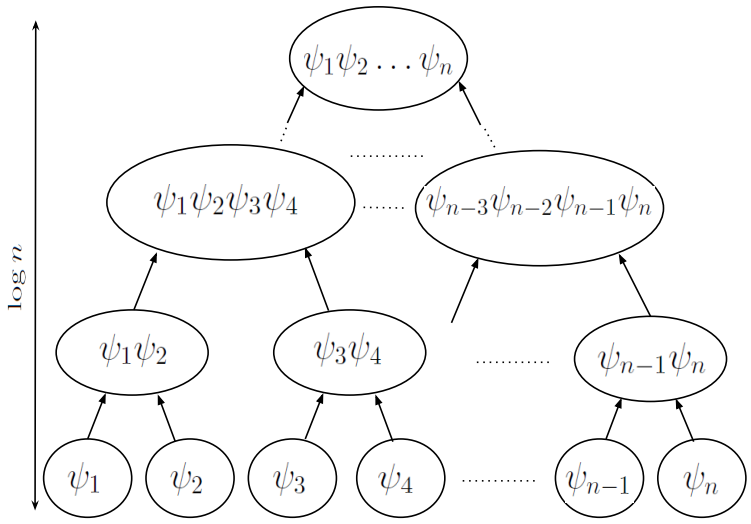

Figure 6: Range tree showing the precomputed products where $\psi_i = \mathcal{H}(t_{x_i \in \mathcal{L}} || x_i)$. Precomputed products allow to speed up the computation time of Query algorithm in Figure 7 when $m \ll n$.

Now, computing $\lambda_{\mathcal{L}'}$ will require computing the product of $m+1$ partial products, i.e., the intervals between elements in the query. Since each partial product can be computed using at most $O(\log n)$ of the precomputed products (as the height of the tree is $O(\log n)$), the total time required to compute the product of $m+1$ partial products is $O((m+1)\log n) = O(m \log n)$. Hence, the precomputation is useful whenever $m \ll n$. Otherwise, when $m = O(n)$, the server can run the Query as mentioned in the scheme in Figure 7 in time $O(n)$.

Efficiency We measure the time and space complexity of our scheme in terms of n, the length of the list $\mathcal{L}$, and m, the length of the queried sublist δ. We use $|\mathcal{L}|$ notation to denote the length of a list $\mathcal{L}$. Recall that $\mathsf{Elements}(\delta) \subseteq \mathsf{Elements}(\mathcal{L})$. We discuss and summarize the time and space complexity for each party as follows:

Owner The Setup algorithm computes member and order witnesses for each element, along with signatures for each element. Hence, the algorithm runs in time $O(n)$ and requires $O(n)$ space.

Server Computing $\lambda_{\mathcal{L}'}$ that takes time $O(n-m)$, as it touches $|\mathcal{L} \setminus \delta|$ elements and computing σ_δ takes time $O(m)$. Hence, the overall runtime of computing $\lambda_{\mathcal{L}'}$ and σ_δ is $O(n)$. The server can precompute and store some products of the signatures, as mentioned above, to reduce the overall running time to $O(\min\{m \log n, n\})$. In addition the server calculates $m-1$ order witnesses each taking constant time, hence, $O(m)$ in total. So the overall run time for the server is $O(\min\{m \log n, n\})$. The server needs to store the list itself, digest_S and the precomputed products. Since each of these objects is of size $O(n)$, the space requirement at the server is $O(n)$.

Client Verify computes a hash for each element in the query δ, and then checks the first two equalities using bilinear map. This requires $O(m)$ computation. In the last step Verify checks $O(m)$ bilinear map equalities which takes time $O(m)$. Hence the overall verification time of the client is $O(m)$. During the query phase, the client requires $O(m)$ space to store its query

and its response with the proof for verification. The client also needs to store digest_C which requires $O(1)$ space.

Efficiency of PPAL via ZKL We noted in the introduction that we can adapt zero knowledge lists to implement a PPAL scheme. Recall that we can do this by making the owner run P_1, the server run P_2 and the client run Verifier of ZKL (see Section 3.1 for the description of ZKL algorithms). Here we estimate the efficiency of a PPAL construction based on the construction of ZKL presented in Figure 5 and compare it with the PPAL construction presented in this section.

From the discussion of efficiency of the ZKL construction in Section 3.5, the time and space complexity of each party in PPAL adaptation of ZKL readily follows below.

Owner The owner runs in time $O(kn)$ and $O(kn)$ space, where k is the security parameter.

Server To answer a query of size m, the server runs in time $O(km)$. The space requirement at the server is $O(kn)$ since he has to store the $O(kn)$ commitments produced by the owner.

Client The verification time of the client is $O(km)$. During the query phase, the client requires $O(km)$ space to store its query and its response with the proof for verification.

Hence, the PPAL construction presented in Figure 7 is a factor of $O(k)$ more efficient in space and computation requirements as compared to an adaptation of the ZKL construction from Figure 5 in PPAL model.

Batch ordering query The client can learn the total order among m different elements of the list using a basic ordering query on two elements. This requires $O(m^2)$ individual order queries, where each verification takes one multiplication in group G and six bilinear map computations. Since our construction supports a query of multiple elements, the client can optimize the process and ask a singe batch ordering query for m elements. In this case, the verification will require only m multiplications in the group G and $2m + 2$ bilinear map computations.

6 Security of the PPAL Construction

In this section we prove that the construction presented in Section 5 is PPAL construction according to definitions of completeness, soundness and zero knowledge in Section 4.

6.1 Proof of Completeness

If all the parties are honest, all the equations in Verify evaluate to true. This is easy to see just by expanding the equations as follows:

Equation $e(\sigma_{\mathsf{order}}, g) \stackrel{?}{=} e(\xi, g^v)$ **:** Let $\mathsf{order} = \{y_1, \ldots, y_m\} = \pi_{\mathcal{L}}(\delta)$

$$e(\sigma_{\mathsf{order}}, g) = e(\prod\nolimits_{y_j \in \mathsf{order}} \sigma_{\mathsf{rank}(\mathcal{L}, y_j)}, g) = e(\prod\nolimits_{y_j \in \mathsf{order}} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i)^v, g) =$$
$$e(\prod\nolimits_{y_j \in \mathsf{order}} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i), g^v) = e(\prod\nolimits_{y_j \in \delta} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i), g^v) = e(\xi, g^v).$$

Equation $e(\sigma_{\mathcal{L}}, g) \stackrel{?}{=} e(\sigma_{\mathsf{order}}, g) \times e(\lambda_{\mathcal{L}'}, g^v)$**:** Let $\mathsf{order} = \{y_1, \ldots, y_m\} = \pi_{\mathcal{L}}(\delta)$ and $\mathcal{L}' = \mathcal{L} \setminus \delta$. We start with the right hand side.

$$e(\sigma_{\mathsf{order}}, g) \times e(\lambda_{\mathcal{L}'}, g^v) = e(\prod\nolimits_{y_j \in \mathsf{order}} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i)^v, g) \times e(\mathcal{H}(\omega) \times \prod\nolimits_{x \in \mathcal{L}'} \mathcal{H}(t_{x_{\mathsf{rank}(\mathcal{L}, x)} \in \mathcal{L}} || x), g^v)$$
$$= e(\prod\nolimits_{y_j \in \mathsf{order}} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i), g^v) \times e(\mathcal{H}(\omega) \times \prod\nolimits_{x \in \mathcal{L}'} \mathcal{H}(t_{x_{\mathsf{rank}(\mathcal{L}, x)} \in \mathcal{L}} || x), g^v)$$
$$= e(\mathcal{H}(\omega) \times \prod\nolimits_{x \in \mathcal{L}} \mathcal{H}(t_{x_{\mathsf{rank}(\mathcal{L}, x)} \in \mathcal{L}} || x), g^v) = e(\mathcal{H}(\omega)^v \times \prod\nolimits_{x \in \mathcal{L}} \mathcal{H}(t_{x_{\mathsf{rank}(\mathcal{L}, x)} \in \mathcal{L}} || x)^v, g) = e(\sigma_{\mathcal{L}}, g).$$

Equation $e(t_{y_j\in\mathcal{L}},t_{y_j<y_{j+1}}) \stackrel{?}{=} e(t_{y_{j+1}\in\mathcal{L}},g)$**:** Let $i' = \mathsf{rank}(\mathcal{L},y_j)$ and $i'' = \mathsf{rank}(\mathcal{L},y_{j+1})$ and $r' = \Omega_{\mathcal{L}}[i']^{-1}$ and $r'' = \Omega_{\mathcal{L}}[i'']$.

$$\begin{aligned} e(t_{y_j\in\mathcal{L}},t_{y_j<y_{j+1}}) &= e(g^{s^{i'}(r')^{-1}},g^{s^{i''-i'}r''r'}) = e(g,g)^{s^{i''-i'+i'}r''r'(r')^{-1}} \\ &= e(g,g)^{s^{i''}r''} = e(g^{s^{i''}r''},g) = e(t_{y_{j+1}\in\mathcal{L}},g). \end{aligned}$$

6.2 Proof of Soundness

Soundness follows by reduction to the n-Bilinear Diffie Hellman assumption (see Definition 2.1 for details). To the contrary of the Soundness Definition 4.2, assume that given a list $\mathcal{L}$, the malicious server, Query' produces two different orders $\mathsf{order}_1 \neq \mathsf{order}_2$ for some sublist δ such that corresponding order proofs are accepted by the client, i.e., by algorithm Verify in Figure 7. Let $\delta = \{x_1, x_2, \ldots, x_m\}$. Since $\mathsf{order}_1 \neq \mathsf{order}_2$, then there exists at least one inversion pair (x_i, x_j) such that $x_i < x_j$ in order_1 and $x_j < x_i$ in order_2, where $i, j \in [1, m]$. Moreover, it must be the case that either $x_i < x_j$ or $x_j < x_i$ is the correct order in $\mathcal{L}$, since both $x_i, x_j \in \mathcal{L}$. (Note that, due to the security of bilinear aggregate signature scheme, it must be the case that all the elements of δ are indeed elements of $\mathcal{L}$, i.e, $x_1, x_2, \ldots, x_m \in \mathcal{L}$ (except with negligible probability).)

Without loss of generality, let us assume $x_i < x_j$ is the order in $\mathcal{L}$ and $\mathsf{rank}(\mathcal{L}, x_i) = u < v = \mathsf{rank}(\mathcal{L}, x_j)$. This implies $x_j < x_i$ is the forged order for which Query' has successfully generated a valid proof, i.e., $e(t_{x_j\in\mathcal{L}}, t_{x_j<x_i}) = e(t_{x_i\in\mathcal{L}}, g)$ has verified since Verify accepted the corresponding proof. We show that by invoking Q' and using its output, $t_{x_j<x_i}$, we construct a PPT adversary $\mathcal{A}$ that successfully solves the *n-BDHI Problem* [BB04] thereby contradicting n-Bilinear Diffie Hellman assumption. The formal reduction follows:

Theorem 6.1 *If n-Bilinear Diffie Hellman assumption holds, then* PPAL *scheme satisfies Soundness in Definition 4.2.*

Proof We show that if there exists a malicious Query' as discussed above, then we construct a PPT adversary $\mathcal{A}$ that successfully solves the *n-BDHI Problem* [BB04]. Algorithm $\mathcal{A}$ is given the public parameters (p, G, G_T, e, g) and $\mathcal{T} = \langle g, g^s, g^{s^2}, \ldots, g^{s^n}\rangle$, where $n = \mathsf{poly}(\mathsf{k})$. $\mathcal{A}$ runs as follows:

1. Pick $v \stackrel{\$}{\leftarrow} \mathbb{Z}_p^*$ a list $\mathcal{L}$ such that $|\mathcal{L}| = n$.
   Pick $\Omega_{\mathcal{L}} = \{r_i \stackrel{\$}{\leftarrow} \mathbb{Z}_p^*\}_{\forall i\in[1,n]}$ and compute $t_{x_i\in\mathcal{L}} \leftarrow (g^{s_i})^{r_i}\ \forall i \in [1, n]$.
   Compute $\sigma_i \leftarrow \mathcal{H}(t_{x_i\in\mathcal{L}}||x_i)^v$, $\forall x_i \in \mathcal{L}$.
   Pick the nonce, $\omega \stackrel{\$}{\leftarrow} \{0,1\}^*$ and compute $\mathsf{salt} \leftarrow (\mathcal{H}(\omega))^v$.
   The list digest signature is computed as: $\sigma_{\mathcal{L}} \leftarrow \mathsf{salt} \times \prod_{1\le i\le n}\sigma_i$.
   Set $\mathsf{digest}_S = \{g^v, \sigma_{\mathcal{L}}, \mathcal{T}, \Sigma_{\mathcal{L}}, \Omega_{\mathcal{L}}\}$ where $\Sigma_{\mathcal{L}} = \langle\{t_{x_i\in\mathcal{L}}, \sigma_i\}_{1\le i\le n}, \mathcal{H}(\omega)\rangle$.
2. Finally Query' outputs two contradicting orders $\mathsf{order}_1 \neq \mathsf{order}_2$ for some sublist, $\delta = \{x_1, x_2, \ldots, x_m\}$. As discussed above, let (x_i, x_j) be an inversion pair such that $x_i < x_j$ is the order in $\mathcal{L}$ and $\mathsf{rank}(\mathcal{L}, x_i) = u < v = \mathsf{rank}(\mathcal{L}, x_i)$.
   This implies $x_j < x_i$ is the forged order for which Query' has successfully generated a valid proof $t_{x_j<x_i} = (g^{s^{(u-v)}})^{r_2 r_1^{-1}}$.
3. Now $\mathcal{A}$ outputs $e(t_{x_j<x_i}, (g^{s^{v-u-1}})^{r_2{}^{-1}r_1}) = e(g,g)^{\frac{1}{s}}$.

$\mathcal{A}$ inherits success probability of Query', therefore if Query' succeeds with non-negligible advantage, so does $\mathcal{A}$. Hence, a contradiction. ∎

6.3 Proof of Zero-Knowledge

We define Zero Knowledge Simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ from Definition 4.3 as follows. Sim has access to the system parameters $(p, G, G_1, e, g, \mathcal{H})$ and executes the following steps:

- Sim_1 picks a random element $v \xleftarrow{\$} \mathbb{Z}_p^*$ and a random element $g_1 \xleftarrow{\$} G$ and publishes as $\mathsf{digest}_C = (g^v, g_1^v)$ and keeps v as the secret key.
- Sim_2 maintains a table of the elements already queried of tuples $\langle x_i, r_i \rangle$ where x_i is the element already queried and r_i is the corresponding random element picked from $\mathbb{Z}_p^*$ by Sim_2.
  For a query on sublist $\delta = \{x_1, x_2, \ldots, x_m\}$, Sim_2 makes an oracle access to list $\mathcal{L}$ to get the list order of the elements. Let us call it $\mathsf{order} = \pi_{\mathcal{L}}(\delta) = \{y_1, y_2, \ldots, y_m\}$.
  - For every $i \in [1, m]$ Sim_2 checks if y_i is in the table. If it is, Sim_2 uses the corresponding random element from the table. Otherwise, Sim_2 picks a random element $r_i \xleftarrow{\$} \mathbb{Z}_p^*$ and adds $\langle y_i, r_i \rangle$ to the table.
  - Sim_2 sets the member authentication unit as $t_{y_i \in \mathcal{L}} := g^{r_i}$ and computes $\sigma_i \leftarrow \mathcal{H}(t_{y_i \in \mathcal{L}} || y_i)^v$.
  - Sim_2 sets $\sigma_{\mathsf{order}} := \prod_{y_i \in \mathsf{order}} \sigma_i$ and $\lambda_{\mathcal{L}'} := \frac{g_1}{\prod_{y_i \in \mathsf{order}} \mathcal{H}(t_{y_i \in \mathcal{L}} || y_i)}$.
  - For every pair of elements y_i, y_{i+1} in order, Sim_2 computes $t_{y_i < y_{i+1}} \leftarrow g^{r_{i+1}/r_i}$.
  - Finally, Sim_2 returns $\mathsf{order}, \mathsf{proof}$, where $\mathsf{proof} = (\Sigma_{\mathsf{order}}, \Omega_{\mathsf{order}})$, $\Sigma_{\mathsf{order}} = (\sigma_{\mathsf{order}}, T, \lambda_{\mathcal{L}'})$, $T = \{t_{y_1 \in \mathcal{L}}, \ldots, t_{y_m \in \mathcal{L}}\}$ and $\Omega_{\mathsf{order}} = \{t_{y_1 < y_2}, t_{y_2 < y_3}, \ldots, t_{y_{m-1} < y_m}\}$.

The simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ produces outputs that are identically distributed to the distribution outputs of the true Setup and Query algorithms. In both cases v is picked randomly. Let $x, y, z \in \mathbb{Z}_p^*$ where x is a fixed element and $z = xy$. Then z is identically distributed to y in $\mathbb{Z}_p^*$. In other words, if y is picked with probability γ, then so is z. The same argument holds for elements in G and G_1. Therefore all the units of Σ_{order} and Ω_{order} are distributed identically in both cases. Thus our PPAL scheme is simulatable and the Zero-Knowledge is perfect. ∎

We summarize the properties and efficiency of our PPAL construction in Theorem 6.2.

Theorem 6.2 *The privacy-preserving authenticated list (PPAL) construction of Figure 7 satisfies the security properties of completeness (Definition 4.1), soundness (Definition 4.2) and zero-knowledge (Definition 4.3). Also, the construction has the following performance, where n denotes the list size and m denotes the query size.*

- *The owner and server use $O(n)$ space.*
- *The owner performs the setup phase in $O(n)$ time.*
- *The server performs the preprocessing phase in $O(n)$ time.*
- *The server computes the answer to a query and its proof in $O(\min\{m \log n, n\})$ time.*
- *The client verifies the proof in $O(m)$ time and space.*

7 Acknowledgment

This research was supported in part by the National Science Foundation under grant CNS–1228485. Olga Ohrimenko worked on this project in part while at Brown University. We are grateful to Melissa Chase, Anna Lysyanskaya, Markulf Kohlweiss, and Claire Mathieu for useful discussions and for their feedback on early drafts of this work. We would also like to thank Ashish Kundu for introducing us to his work on structural signatures and Jia Xu for sharing a paper through personal communication.

References

[ABC+12] Jae Hyun Ahn, Dan Boneh, Jan Camenisch, Susan Hohenberger, Abhi Shelat, and Brent Waters. Computing on authenticated data. In *Proc. of TCC*, number 7194 in LNCS, 2012.

[ALP12] Nuttapong Attrapadung, Benoît Libert, and Thomas Peters. Computing on authenticated data: New privacy definitions and constructions. In *ASIACRYPT*, pages 367–385, 2012.

[BB04] Dan Boneh and Xavier Boyen. Efficient selective-ID secure identity based encryption without random oracles. In *Proceedings of Eurocrypt 2004, volume 3027 of LNCS*, pages 223–238. Springer-Verlag, 2004.

[BB12] Jordan Brown and Douglas M. Blough. Verifiable and redactable medical documents. *AMIA Annu Symp Proc*, pages 1148 – 1157, 2012.

[BBD+10] Christina Brzuska, Heike Busch, Özgür Dagdelen, Marc Fischlin, Martin Franz, Stefan Katzenbeisser, Mark Manulis, Cristina Onete, Andreas Peter, Bertram Poettering, and Dominique Schröder. Redactable signatures for tree-structured data: Definitions and constructions. In *ACNS*, pages 87–104, 2010.

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Advances in cryptology - EUROCRYPT 2003*, pages 416–432. Springer, 2003.

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

[Bou00] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. In *EUROCRYPT*, pages 431–444, 2000.

[CF13] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public Key Cryptography*, pages 55–72, 2013.

[CFM08] Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-knowledge sets with short proofs. In *Proceedings of the Theory and Applications of Cryptographic Techniques 27th Annual International Conference on Advances in Cryptology*, EUROCRYPT'08, pages 433–450, Berlin, Heidelberg, 2008. Springer-Verlag.

[CH12] Philippe Camacho and Alejandro Hevia. Short transitive signatures for directed trees. In *CT-RSA*, pages 35–50, 2012.

[CHL+05] Melissa Chase, Alexander Healy, Anna Lysyanskaya, Tal Malkin, and Leonid Reyzin. Mercurial commitments with applications to zero-knowledge sets. In *EUROCRYPT*, pages 422–439, 2005.

[CKLM13] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Malleable signatures: Complex unary transformations and delegatable anonymous credentials. *IACR Cryptology ePrint Archive*, 2013:179, 2013.

[CLX09] Ee-Chien Chang, Chee Liang Lim, and Jia Xu. Short redactable signatures using random trees. In *Proc. RSA Conf. — Cryptographer's Track (CT-RSA)*, LNCS, pages 133–147, Berlin, Heidelberg, 2009. Springer.

[DF02] Ivan Damgård and Eiichiro Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *ASIACRYPT*, pages 125–142, 2002.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[JMSW02] Robert Johnson, David Molnar, Dawn Xiaodong Song, and David Wagner. Homomorphic signature schemes. In *Proc. RSA Conf. — Cryptographer's Track (CT-RSA)*, LNCS, pages 244–262, London, UK, UK, 2002. Springer.

[KAB12] Ashish Kundu, Mikhail J. Atallah, and Elisa Bertino. Leakage-free redactable signatures. In *Proc. ACM Conf. on Data and Application Security and Privacy (CODASPY)*, pages 307–316, 2012.

[KB08] Ashish Kundu and Elisa Bertino. Structural signatures for tree data structures. *PVLDB*, 1(1):138–150, 2008.

[KB13] Ashish Kundu and Elisa Bertino. Privacy-preserving authentication of trees and graphs. *Int. J. Inf. Sec.*, 12(6):467–494, 2013.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, pages 177–194, 2010.

[Lip03] Helger Lipmaa. On diophantine complexity and statistical zero-knowledge arguments. In *ASIACRYPT*, pages 398–415, 2003.

[LY10] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *Proceedings of the 7th International Conference on Theory of Cryptography*, TCC'10, pages 499–517, Berlin, Heidelberg, 2010. Springer-Verlag.

[Mer80] Ralph C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[Mer89] Ralph C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.

[MHI06] Kunihiko Miyazaki, Goichiro Hanaoka, and Hideki Imai. Digitally signed document sanitizing scheme based on bilinear maps. In *Proc. ACM Symp. on Information, Computer and Communications Security*, (ASIACCS), pages 343–354, New York, NY, USA, 2006. ACM.

[MR02] Silvio Micali and Ronald L. Rivest. Transitive signature schemes. In *CT-RSA*, pages 236–243, 2002.

[MRK03] Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *FOCS*, pages 80–91, 2003.

[MTGS01] Roberto Tamassia Michael T. Goodrich and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. *DARPA Information Survivability Conference and Exposition II*, pages 68 – 82, 2001.

[ORS04] Rafail Ostrovsky, Charles Rackoff, and Adam Smith. Efficient consistency proofs for generalized queries on a committed database. In *Proc. Int. Colloquium on Automata, Languages and Programming (ICALP)*, volume 3142 of *LNCS*, pages 1041–1053. Springer, 2004.

[PSPDM12] Henrich C. Poehls, Kai Samelin, Joachim Posegga, and Hermann De Meer. Length-hiding redactable signatures from one-way accumulators in $O(n)$. Technical Report MIP-1201, Faculty of Computer Science and Mathematics (FIM), University of Passau, 2012.

[SBZ01] Ron Steinfeld, Laurence Bull, and Yuliang Zheng. Content extraction signatures. In *Int. Conf. on Information Security and Cryptology (ICISC)*, volume 2288 of *LNCS*, pages 285–304. Springer, 2001.

[SPB+12] Kai Samelin, Henrich C. Poehls, Arne Bilzhause, Joachim Posegga, and Hermann De Meer. Redactable signatures for independent removal of structure and content. In *Proc. Int. Conf. on Information Security Practice and Experience (ISPEC)*, volume 7232 of *LNCS*. Springer, 2012.

[Tam03] Roberto Tamassia. Authenticated data structures. In *Proc. European Symp. on Algorithms (ESA)*, volume 2832 of *LNCS*, pages 2–5. Springer, 2003.

[Wan12] Zhiwei Wang. Improvement on Ahn et al.'s RSA P-homomorphic signature scheme. In *SecureComm*, pages 19–28, 2012.

[Yi06] Xun Yi. Directed transitive signature scheme. In *Proc. RSA Conf. — Cryptographer's Track (CT-RSA)*, LNCS, pages 129–144, Berlin, Heidelberg, 2006. Springer-Verlag.

Appendix

A Homomorphic Integer Commitment Scheme [DF02] and its Simulator

We write the commitment scheme of [DF02], in the trusted parameter model, i.e., the public key is generated by a trusted third party. However, in the original paper [DF02], the prover and the verifier interactively set up the public parameters.

Figure 8: Homomorphic Integer Commitment Scheme [DF02].

$\mathsf{HomIntCom} = (\mathsf{IntComSetup}, \mathsf{IntCom}, \mathsf{IntComOpen})$

$\mathsf{PK}_C \leftarrow \mathsf{IntComSetup}(1^k)$**:** The $\mathsf{IntComSetup}$ algorithm, takes the security parameter as input and generates the description of a finite Abelian group $\mathcal{G}$, $desc(\mathcal{G})$, and a large integer $F(k)$ such that it is feasible to factor numbers that are smaller than $F(k)$. A number having only prime factors at most $F(k)$ are called $F(k)$-smooth and a number having prime factors larger then $F(k)$ are called $F(k)$-rough. The algorithm then chooses a random element $h \stackrel{\$}{\leftarrow} \mathcal{G}$ (by group assumption, $ord(h)$ is $F(k)$-rough with overwhelming probability) and a random secret key $s \stackrel{\$}{\leftarrow} \mathbb{Z}_{ord(\mathcal{G})}$ and sets $g := h^s$. $\mathsf{IntComSetup}$ outputs $(desc(\mathcal{G}), F(k), g, h)$ as the public key of the commitment scheme, PK_C.

$(c, r) \leftarrow \mathsf{IntCom}(\mathsf{PK}_C, x)$**:** To commit to an integer x, the algorithm IntCom chooses a random r, $r \stackrel{\$}{\leftarrow} \mathbb{Z}_{2^{B+k}}$, and computes $c = g^x h^r$ (where B is a reasonably close upper bound on the order of the group $\mathcal{G}$, i.e., $2^B > ord(\mathcal{G})$, and given $desc(\mathcal{G})$, B can be computed efficiently). IntCom outputs (c, r).

$x \leftarrow \mathsf{IntComOpen}(\mathsf{PK}_C, c, r)$**:** To open a commitment c, the committer must send the opening information (x, r, b) to the verifier such that $c = g^x h^r b$ and $b^2 = 1$. An honest committer can always set $b := 1$.

The above commitment scheme is *homomorphic* as

$$\mathsf{IntCom}(\mathsf{PK}_C, x + y) = \mathsf{IntCom}(\mathsf{PK}_C, x) \times \mathsf{IntCom}(\mathsf{PK}_C, y).$$

In Figure 9 we present a simulator for $\mathsf{HomIntCom}$. We note that the distribution of outputs from the simulator algorithms is identical to the distribution of outputs from a true prover (committer): in both cases $desc(\mathcal{G}), F(k), g, h$ and commitments are generated identically.

Efficiency Assuming group exponentiation take constant time, both IntCom and $\mathsf{IntComOpen}$ run in asymptotic time $O(1)$.

Figure 9: Simulator for HomIntCom.

SimHomIntCom = (SimIntComSetup, SimIntCom, SimIntComOpen)

- $(\mathsf{PK}_C, \mathsf{TK}_C) \leftarrow \mathsf{SimIntComSetup}(1^k)$**:** SimIntComSetup works exactly as the IntComSetup except that it saves s and the order of the group $\mathcal{G}$, $ord(\mathcal{G})$. SimICSetup sets $\mathsf{TK}_C = (ord(\mathcal{G}), s)$ and outputs $(\mathsf{PK}_C = (desc(\mathcal{G}), F(k), g, h), \mathsf{TK}_C)$.
- $(c, r) \leftarrow \mathsf{SimIntCom}(\mathsf{PK}_C, x)$**:** SimIntCom behaves exactly as IntCom and outputs (c, r) where $c = g^x h^r$, $r \xleftarrow{\$} \mathbb{Z}_{2^{B+k}}$ and B is as defined in Figure 8.
- $x' \leftarrow \mathsf{SimIntComOpen}(\mathsf{PK}_C, \mathsf{TK}_C, c, r)$**:** To open a commitment c, originally committed to some integer x, to any arbitrary integer $x' \neq x$, send $(x', (r + sx - sx') \bmod ord(\mathcal{G}), b = 1)$ to the verifier.

B Proving an Integer is Non-negative [Lip03]

We present the Σ protocol presented in [Lip03] in Figure 10. This protocol is honest-verifier zero knowledge with 3 rounds of interaction and can be converted to non-interactive general zero knowledge in the Random Oracle model using Fiat-Shamir heuristic [FS86].

The protocol is essentially based on two facts: a negative number cannot be a sum of squares and every non-negative integer is a sum of four squared integers. The representation of a non-negative integer as the sum of four squares is called the Lagrange representation of a non-negative integer. [Lip03] presents an efficient probabilistic time algorithm to compute the Lagrange representation of a non-negative integer.

Theorem B.1 *[Lip03] An integer x can be represented as $x = \omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2$, with integer ω_i, $i \in [1, 4]$, iff $x \geq 0$. Moreover, if $x \geq 0$, then the corresponding representation $(\omega_1, \omega_2, \omega_3, \omega_4)$ can be computed efficiently.*

Efficiency The algorithm to compute Lagrange's representation of a non-negative integer is probabilistic polynomial time [Lip03]. Assuming group exponentiation is done in constant time, both the *Prover* and the *Verifier* in the protocol in Figure 10 run in asymptotic constant time, i.e., $O(1)$.

Figure 10: Proving non-negativity of an integer [Lip03]: $\mathsf{Protocol}(x, r : c = C(x; r) \wedge x \geq 0)$

Step 1: The *Prover* commits to an integer $x \in \{-M, M\}$ as $c := \mathsf{IntCom}(\mathsf{PK}_C, x) = g^x h^\rho$ where $\rho \in \mathbb{Z}_{2^{B+k}}$ and sends it to the *Verifier*. Now the *Prover* computes the following:

- represent x as $x = \omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2$
- for $i \in [1, 4]$: pick $r_{1i} \xleftarrow{\$} \mathbb{Z}_{2^{B+2k}}$ such that $\sum_i r_{1i} = \rho$
- for $i \in [1, 4]$: pick $r_{2i} \xleftarrow{\$} \mathbb{Z}_{2^{B+2k} F(k)}$ and $r_3 \xleftarrow{\$} \mathbb{Z}_{2^{B+2k} F(k)\sqrt{M}}$
- for $i \in [1, 4]$: pick $m_{1i} \xleftarrow{\$} \mathbb{Z}_{2^k F(k)\sqrt{M}}$
- for $i \in [1, 4]$: compute $c_{1i} \leftarrow g^{\omega_1} h^{r_{1i}}$
- compute $c_2 \leftarrow g^{\sum_i m_{1i}} h^{\sum_i r_{1i}}$
- compute $c_3 \leftarrow (\prod_i {c_{1i}}^{m_{1i}}) h^{r_3}$

The *Prover* sends $(c_{11}, c_{12}, c_{13}, c_{14}, c_2, c_3)$ to the *Verifier*.

Step 2: The *Verifier* generates $e \xleftarrow{\$} \mathbb{Z}_{F(k)}$ and sends it to the *Prover*.

Step 3: The *Prover* computes the following:

- for $i \in [1, 4]$: compute $m_{2i} \leftarrow m_{1i} + e\omega_i$
- for $i \in [1, 4]$: compute $r_{4i} \leftarrow r_{2i} + e r_{1i}$
- compute $r_5 \leftarrow r_3 + e \sum_i (1 - \omega_i) r_{1i}$

The *Prover* sends $(m_{21}, m_{22}, m_{23}, m_{24}, r_{41}, r_{42}, r_{43}, r_{44}, r_5)$ to the *Verifier*.

Step 4: The *Verifier* checks the following:

- for $i \in [1, 4]$: check $g^{m_{2i}} h^{r_{4i}} {c_{1i}}^{-e} \stackrel{?}{=} c_2$
- $(\prod_i {c_{1i}}^{m_{2i}}) h^{r_5} c^{-e} \stackrel{?}{=} c_3$

C Zero Knowledge Set (ZKS) Construction [CHL$^+$05]

Here we give the construction of ZKS based on mercurial commitments and collision-resistant hash functions. For the details, please refer to Section 3 of [CHL$^+$05].

For a finite database D, the prover views each key x as an integer numbering of a leaf of a height-l binary tree and places a commitment to the information $v = D(x)$ into leaf number x. To generate the commitment C_D to the database D, the prover Prover_D generates an incomplete binary tree of commitments, resembling a Merkle tree as follows. Let $\mathsf{Merc} = \{\mathsf{MercSetup}, \mathsf{HardComm}, \mathsf{SoftComm}, \mathsf{Tease}, \mathsf{VerTease}, \mathsf{MercOpen}, \mathsf{VerOpen}\}$ be a Mercurial Commitment scheme and PK_D be the public key of the mercurial commitment scheme, i.e., $\mathsf{PK}_D \leftarrow \mathsf{MercSetup}(1^k)$. Let r_x denotes the randomness used to produce the commitment (hard or soft) of x.

Before getting into the details of the ZKS construction using mercurial commitments in Figure 11, let us give an informal description of mercurial commitments. Mercurial commitments slightly relax the binding property of commitments. Partial opening, which is called *teasing*, is not truly binding: it is possible for the committer to come up with a commitment that can be teased to any value of its choice. True opening, on the other hand, is binding in the traditional sense: it is infeasible for the committer to come up with a commitment that it can open to two different values. If the committer can open a commitment at all, then it can be teased to only one value. Thus, the committer must choose, at the time of commitment, whether to *soft-commit*, so as to be able to tease to multiple values but not open at all, or to *hard-commit*, so as to be able to tease and to open to only one particular value. It is important to note that hard-commitments and soft-commitments are computationally indistinguishable.

Efficiency Let us assume that the elements are hashed to k bit strings, so that $l = k$. Let us also assume (as in [CHL$^+$05]) that the collision resistant hash is built into the mercurial commitment scheme, allowing to form k-bit commitments to pairs of k-bit strings. Therefore, computing the commitment com takes time $O(ln) = O(kn)$, where $|D| = n$.

The proofs of membership and non-membership consists of $O(k)$ mercurial decommitments each and the verifier needs to verify $O(l) = O(k)$ mercurial decommitments to accept the proof's validity.

A constant time speed-up can be achieved using the q-Trapdoor Mercurial Commitment (q-TMC) scheme and collision resistant hash function as building blocks. q-TMC was introduced by [CFM08] and later improved by [LY10]. The construction is similar to [CHL$^+$05], except a q-ary tree of height h is used ($q > 2$) instead of a binary tree and each leaf is expressed in q-ary encoding. Using q-TMC as a building block achieves significant improvement in ZKS implementation [CFM08, LY10] though the improvement is not asymptotic.

Figure 11: Zero Knowledge Set (ZKS) construction from Mercurial Commitments [CHL+05].

$\mathsf{ZKS} = (\mathsf{ZKSSetup}, \mathsf{ZKSProver} = (\mathsf{ZKSP}_1, \mathsf{ZKSP}_2), \mathsf{ZKSVerifier})$

$\mathsf{PK}_D \leftarrow \mathsf{ZKSSetup}(1^k)$**:** Run $\mathsf{PK}_D \leftarrow \mathsf{MercSetup}(1^k)$ and output PK_D.

$(\mathsf{com}, \mathsf{state}) \leftarrow \mathsf{ZKSP}_1(1^k, \mathsf{PK}_D, D)$**:** ZKSP_1 runs as follows:

- For each x such that $D(x) = v \neq \perp$, produce $C_x = \mathsf{HardComm}(\mathsf{PK}_D, v, r_x)$.
- For each x such that $D(x) = \perp$ but $D(x') \neq \perp$, where x' denotes x with the last bit flipped, produce $C_x = \mathsf{SoftComm}(\mathsf{PK}_D, r_x)$.
- Define $C_x = \mathsf{nil}$ for all other x and build the tree in bottom up fashion. For each level i from $l-1$ upto 0, and for each string σ of length i, define the commitment C_σ as follows:
  1. For all σ such that $C_{\sigma 0} \neq \mathsf{nil} \wedge C_{\sigma 1} \neq \mathsf{nil}$, let $C_\sigma = \mathsf{HardComm}(\mathsf{PK}_D, (C_{\sigma 0}, C_{\sigma 1}), r_\sigma)$.
  2. For all σ such that $C_{\sigma'}$ have been defined in Step 1 (where σ' denotes σ with the last bit flipped) but C_σ has not, define $C_\sigma = \mathsf{SoftComm}(\mathsf{PK}_D, r_\sigma)$.
  3. For all other σ, define $C_\sigma = \mathsf{nil}$.
- If the value of the root, $C_\epsilon = \mathsf{nil}$, redefine $C_\epsilon = \mathsf{SoftComm}(\mathsf{PK}_D, r_\epsilon)$. This happens only when $D = \phi$. Finally define $C_D = C_\epsilon = \mathsf{com}$.

$(D(x), \Pi_x) \leftarrow \mathsf{ZKSP}_2(\mathsf{PK}_D, \mathsf{state}, x)$**:** For a query x, ZKSP_2 runs as follows:

$x \in D$, i.e., $D(x) = v \neq \perp$: Let $(x|i)$ denote the first i bits of the string x and $(x|i)'$ be $(x|i)$ with the last bit flipped. Let $\mathsf{proof}_x = \mathsf{MercOpen}(\mathsf{PK}_D, D(x), r_x, C_x)$ and $\mathsf{proof}_{(x|i)} = \mathsf{MercOpen}(PK_D, (C_{(x|i0)}, C_{(x|i1)}), r_{(x|i)}, C_{(x|i)})$ for all $0 \leq i < l$, where $C_{(x|i)}$ is a commitment to its two children $C_{(x|i0)}$ and $C_{(x|i1)}$.
Return $(D(x), \Pi_x = (\{C_{(x|i)}, C_{(x|i)'}\}_{i\in[1,l]}, \{\mathsf{proof}_{(x|i)}\}_{i\in[0,l]}))$.

$x \notin D$, i.e., $D(x) = \perp$: If $C_x = \mathsf{nil}$, let h be the largest value such that $C_{(x|h)} \neq \mathsf{nil}$, let $C_x = \mathsf{HardComm}(\mathsf{PK}_D, \perp, r_x)$ and build a path from x to $C_{(x|h)}$ as follows: define $C_{(x|i)} = \mathsf{HardComm}(\mathsf{PK}_D, (C_{(x|i0)}, C_{(x|i1)}), r_{(x|i)}), C_{(x|i)'} = \mathsf{SoftComm}(\mathsf{PK}_D, r_{(x|i)'})$ for all $i \in [l-1, h+1]$. Let $\tau_x = \mathsf{Tease}(\mathsf{PK}_D, D(x), r_x, C_x)$ and $\tau_{(x|i)} = \mathsf{Tease}(PK_D, (C_{(x|i0)}, C_{(x|i1)}), r_{(x|i)}, C_{(x|i)})$ for all $0 \leq i < l$.
Return $(\perp, \Pi_x = (\{C_{(x|i)}, C_{(x|i)'}\}_{i\in[1,l]}, \{\tau_{(x|i)}\}_{i\in[0,l]}))$

$b \leftarrow \mathsf{ZKSVerifier}(1^k, \mathsf{PK}_D, \mathsf{com}, x, D(x), \Pi_x)$**:**

$x \neq \perp$: The verifier $\mathsf{ZKSVerifier}$ performs the following:

- $\mathsf{VerOpen}(\mathsf{PK}_D, C_{(x|i)}, (C_{(x|i0)}, C_{(x|i1)}), \mathsf{proof}_x)$ for all $1 \leq i < l$
- $\mathsf{VerOpen}(\mathsf{PK}_D, C_D, (C_0, C_1), \mathsf{proof}_\epsilon)$ and $\mathsf{VerOpen}(\mathsf{PK}_D, C_x, D(x), \mathsf{proof}_x)$.

$x = \perp$: The verifier $\mathsf{Verifier}_D$ performs the following:

- $\mathsf{VerTease}(\mathsf{PK}_D, C_{(x|i)}, (C_{(x|i0)}, C_{(x|i1)}), \tau_x)$ for all $1 \leq i < l$
- $\mathsf{VerTease}(\mathsf{PK}_D, C_D, (C_0, C_1), \tau_\epsilon)$ and $\mathsf{VerTease}(\mathsf{PK}_D, C_x, \perp, \tau_x)$