Verifiable Order Queries and Order Statistics 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

Given a list $\mathcal{L}$ with n elements, an order query on $\mathcal{L}$ asks whether a given element $x \in \mathcal{L}$ precedes or follows another element $y \in \mathcal{L}$. More generally, given a set of m elements from $\mathcal{L}$, an order query asks for the set ordered according to the positions of the elements in $\mathcal{L}$. We introduce two formal models for answering order queries on a list in a verifiable manner and in zero-knowledge. We also present efficient constructions for these models.

Our first model, called *zero-knowledge list* (ZKL), generalizes membership queries on a set to order queries on a list in zero-knowledge. We present a construction of ZKL based on zero-knowledge sets and a homomorphic integer commitment scheme.

Our second model, *privacy-preserving authenticated list* (PPAL), extends authenticated data structures by adding a zero-knowledge privacy requirement. In this model, a list is outsourced by a trusted owner to an untrusted cloud server, which answers order queries issued by clients. The server also returns a proof of the answer, which is verified by the client using a digest of the list obtained from the owner. PPAL supports the security properties of data integrity against a malicious server and privacy protection against a malicious client. Though PPAL can be implemented using our ZKL construction, this construction is not as efficient as desired in cloud applications. To this end, we present an efficient PPAL construction based on blinded bilinear accumulators and bilinear maps, which is provably secure and zero-knowledge (e.g., hiding even the size of the list). Our PPAL construction uses proofs of $O(m)$ size and allows the client to verify a proof in $O(m)$ time. The owner executes the setup in $O(n)$ time and space. The server uses $O(n)$ space to store the list and related authentication information, and takes $O(\min(m \log n, n))$ time to answer a query and generate a proof. Both our ZKL and PPAL constructions have one round of communication and are secure in the random oracle model.

Finally, we show that our ZKL and PPAL frameworks can be extended to support fundamental statistical queries (including maximum, minimum, median, threshold and top-t elements) efficiently and in zero-knowledge.

Keywords: order queries, order statistics, zero-knowledge, integrity, authenticated data structures bilinear accumulators, leakage-free redactable signatures, cloud security, cloud-privacy

*The full version of this work is available on `http://eprint.iacr.org/2014/632` and an early version (May 2014) appeared in `http://arxiv.org/abs/1405.0962`.

[†]esha_ghosh@brown.edu

[‡]oohrim@microsoft.com

[§]rt@cs.brown.edu

1 Introduction

Releasing verifiable partial information while maintaining privacy is a requirement in many practical scenarios where the data being dealt with is sensitive. A basic case is releasing a subset of a set and proving its authenticity in a privacy-preserving way (referred to as zero-knowledge property) [MRK03, CHL+05, CFM08, LY10]. However, in many other cases, the information is stored in data structures to support richer type of queries. In this paper, we consider *order queries* on two or more elements of a list, where the answer to the query returns the elements rearranged according to their order in the list. Order queries lie at the heart of applications where the order between queried elements is revealed and proved but the rank of the queried elements in the list and information about other elements in the list should be protected. We give three practical examples below.

Consider an auction with a single winner (e.g., online ad auction for a single ad spot) where every participant submits her secret bid to the auction organizer. After the top bidder is announced by the organizer, a participant wishes to verify that her bid was inferior. The organizer would then provide a proof without revealing the amount of the top bid, the rank of the participant's bid, or any information about other bids.

Lenders often require an individual or a couple to prove eligibility for a loan by providing a current bank statement and a pay stub. Such documents contain a lot of sensitive information beyond what the lender is looking for: whether the bank account balance and salary are above given thresholds. A desirable alternative would be to provide a proof from the bank and employer that these thresholds are met without revealing exact figures and even hiding who of the two spouses earns more.

Consider yet another scenario 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 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 server would need to release the relevant information in such a way that the querying division can verify that the data came from the legitimate source but not learn anything beyond the query result.

The above examples can be generalized using order queries on an ordered set, aka list, that return the order of the queried elements as well as a proof of this order but without revealing anything more than the answer itself. We address this problem by introducing two different models: *zero knowledge lists* (ZKL) and *privacy-preserving authenticated lists* (PPAL).

ZKL considers two party model and extends zero knowledge sets [MRK03, CHL+05] to lists. In ZKL a prover commits to a list and a verifier queries the prover to learn the order of a subset of list elements. The verifier should be able to verify the answer but learn no information about the rest of the list, e.g., the size of the list, the order of other elements of the list or the rank of the queried element(s). Here both the prover and the verifier can act as malicious adversaries. While the prover may want to give answers inconsistent with the initial list he committed to, the verifier may try to learn information beyond the query answer or arbitrarily deviate from the protocol.

PPAL considers three parties: the owner of the list, the server who answers list queries on behalf of the owner, and the client who queries the server. The privacy guarantee of PPAL is the same as in ZKL. For authenticity, PPAL assumes that the owner is trusted while the server and the client could be malicious. This trust model allows for a much more efficient construction than ZKL, as we will see later in the paper. PPAL has direct applications to outsourced services where the server is modeling the cloud service that the owner uses to interact with her clients.

We note that PPAL can be viewed as a privacy-preserving extension of authenticated data structures (ADS) (see, e.g., [Mer80, Mer89, Tam03, MTGS01]), which also operate in a three party model: the server stores the owner's data and proves to the client the answer to a query. However, privacy properties have not

been studied in this model and as a consequence, known ADS constructions leak information about the rest of the data through their proofs of authenticity. For example, the classic Merkle hash tree [Mer80, Mer89] on a set of n elements proves membership of an element via a proof of 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.

In this paper, we define the security properties for ZKL and PPAL and provide efficient constructions for them. The privacy property against the verifier in ZKL and the client in PPAL is *zero knowledge*. That is, the answers and the proofs are indistinguishable from those that are generated by a simulator that knows nothing except the previous and current queries and answers and, hence, cannot possibly leak any information beyond that. While we show that PPAL can be implemented using our ZKL construction, we also provide a direct PPAL construction that is considerably more efficient thanks to the trust that clients put in the list owner. Let n be the size of the list and m be the size of the query, i.e., the number of list elements whose order is sought. Our PPAL construction uses proofs of $O(m)$ size and allows the client to verify a proof in $O(m)$ time. The owner executes the setup in $O(n)$ time and space. The server uses $O(n)$ space to store the list and related authentication information, and takes $O(\min(m \log n, n))$ time to answer a query and generate a proof. In contrast, in the ZKL construction, the time and storage requirements have an overhead that linearly depends on the security parameter. Note that ZKL also supports (non-)membership queries. The client in PPAL and the verifier in ZKL require only one round of communication for each query. Our ZKL construction is based on zero knowledge sets and homomorphic integer commitments. Our PPAL construction uses a novel technique of blinding of accumulators along with bilinear aggregate signatures. Both constructions are secure in the random oracle model.

2 Problem Statement, Models, Related Work, and Contributions

In this section, we state our problem, outline our models, review related work, and summarize our contributions. Formal definitions, constructions, and security proofs are in the rest of the paper.

2.1 Problem Statement and Models

We start by defining order queries and its applications, introduce our adversarial model and security properties, and discuss our efficiency goals.

2.1.1 Query

Let $\mathcal{L}$ be a linearly ordered list of distinct elements. An *order query* on $\mathcal{L}$ is defined as follows: given a pair of elements (x, y) of $\mathcal{L}$, return the pair with its elements rearranged according to their order in $\mathcal{L}$ and a proof of this order. For example, if y precedes x in $\mathcal{L}$, then the pair (y, x) is returned as an answer. For generality, we define a *batch order query*: given a list of query elements δ, each from $\mathcal{L}$, return the permutation of δ according to the ordering of the elements in $\mathcal{L}$ and a proof of the order. Both models we introduce, PPAL and ZKL, support this query. ZKL, in addition to order queries, supports provable membership and non-membership queries. 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. Beside providing authenticity, the proofs are required not to leak any information beyond the answer.

2.1.2 Adversarial model and security properties

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

ZKL The ZKL model considers two parties: prover and 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 $\mathcal{L}$ and the prover responds with a proof. Both the prover and the verifier can be malicious:

- The prover may try to give answers which are inconsistent with the initial commitment.

- The verifier may try to learn from the proofs additional information about $\mathcal{L}$ beyond what he has inferred from the answers. E.g., if the verifier has performed two order queries with answers $x < y$ and $x < z$, he may want to find out whether $y < z$ or $z < y$. Here, $x < y$ denotes that x appears before y in $\mathcal{L}$.

The security properties of ZKL, *completeness, soundness* and *zero-knowledge*, guarantee security against malicious prover and verifier. 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 oracle access to $\mathcal{L}$, can simulate proofs for membership and order queries that are indistinguishable from real proofs. We discuss the security properties of ZKL in more detail in Section 4.2.

PPAL The PPAL model considers, instead, three parties: owner, server and client. The owner generates list $\mathcal{L}$ and outsources it to the server. The owner also sends (possibly different) digest information with respect to $\mathcal{L}$ to the server and the client. Given an order query from the client, the server, using the server digest, builds and returns to the client the answer and its proof, which is verified by the client using the client digest. Both the server and the client can be malicious:

- The server may try to forge proofs for incorrect answers to (order) queries, e.g., prove an incorrect ordering of a pair of elements of $\mathcal{L}$.
- The client, similar to the verifier in ZKL, may try to learn from the proofs additional information about list $\mathcal{L}$ beyond what he has inferred from the answers.

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, possibly malicious, behavior from the client where he tries to ask ill-formed queries or queries violating the access control policy. However, we assume that the server enforces client's legitimate behavior by refusing to answer illegal queries. Hence, the security model for PPAL is defined as follows.

The properties of PPAL, *Completeness*, *Soundness* and *Zero-Knowledge*, guarantee security against malicious server and client. They are close to the ones of ZKL except for soundness. For PPAL it enforces that the client does not accept proofs forged by the server for incorrect answers w.r.t. owner's list. PPAL's owner and server together can be thought of as a single party in ZKL, the prover. Hence, ZKL soundness protects against the prover who tries to give answers inconsistent with her own initial commitment. In the PPAL model, the owner and the server are separate parties where the owner is trusted and soundness protects against a malicious server only. We discuss the security properties of PPAL in more detail in Section 5.2.

To understand the strength of the zero-knowledge property, let us illustrate to what extent the proofs are non-revealing. This property guarantees that a client, who adaptively queries a static list, does not learn anything about ranks of the queried elements, the distance between them or even the size of $\mathcal{L}$. The client is not able to infer any relative order information that is not inferable by the rule of transitivity from the previously queried orders. It is worth noting that in the context of leakage-free redactable signature schemes, privacy property has been defined using game-based definitions in *transparency* [BBD+10, SPB+12] and *privacy* [CLX09, KAB12]. However, our definition of simulatability of the query responses, or the zero-knowledge property, is a simpler and more intuitive way to capture the property of leakage-freeness.

2.1.3 Efficiency

We characterize the ideal efficiency goals of our models as follows, where $\mathcal{L}$ is a list of n items and m is the query size:

Storage space The space for storing list $\mathcal{L}$ and the auxiliary information for generating proofs should be $O(n)$, irrespective of the number of queries answered.

Setup time The setup to preprocess list $\mathcal{L}$ should take $O(n)$ time.

Proof size The proof of the answer to a query should have $O(m)$ size.
Query time Processing a query to generate the answer and its proof should take $O(m)$ time.
Verification time Verifying the proof of an answer should take $O(m)$ time.

2.1.4 Applications of order queries to order statistics

Our PPAL order queries can be used as a building block to answer efficiently and in zero knowledge (i.e., the returned proofs should be simulatable) many interesting statistical queries about a list $\mathcal{L}$ with n elements. Let a *pair order proof* denote the proof of the order of two elements from $\mathcal{L}$. Then a PPAL client can send the server a subset $\mathcal{S}$ of m list elements and request the server to return the *maximum*, *minimum*, or the *median* element of $\mathcal{S}$ w.r.t. the order of the elements in the list. This can be done by providing m pair order proofs. Order queries also can be extended to return the *top* t elements of $\mathcal{S}$ by means of $t(m-t)$ pair order proofs, or only $m-1$ pair order proofs if the order between the top t elements can be revealed, where $t < m$. Finally, given an element a in $\mathcal{L}$, the server can return the elements of $\mathcal{S}$ that are above (or below) the *threshold* value a by means of m pair order proofs. It is important to note that neither of these queries reveal anything more than the answer itself. Moreover, the size of the proof returned for each query is proportional to the query size and is optimal for the threshold query where the proof size is proportional to the answer size. We note that these statistical queries are also supported by ZKL and are defined formally in Section 8.

2.2 Related Work

We discuss related literature in three sections. First, we discuss work on data structures that answer queries in zero knowledge. Our ZKL is the first extension of this work to lists and order queries. We then mention signature schemes that can be used to instantiate outsourced data structures that require privacy and integrity to be maintained. However, such instantiations are not efficient since they are based on different models of usage and underlying data. Finally, we outline leakage-free redactable signature schemes for ordered lists and other structured data. These signature schemes are are not as efficient as our construction and their definitions are game-based as opposed to our intuitive zero-knowledge definition.

Zero Knowledge Data Structures Zero-knowledge dictionary and range queries have received considerable attention in literature [MRK03, CHL$^+$05, CFM08, LY10, ORS04]. Our proposed ZKL model is the first generalization of this line of work that supports order queries.

The model of *zero knowledge set* (ZKS) (more generally, *zero knowledge elementary database*) was introduced by Micali *et al.* [MRK03] where a prover commits to a finite set S in such a way that, later on, she 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 by proving contradictory statements about an element. Chase *et al.* [CHL$^+$05] abstracted the above solution and described the properties a commitment scheme should possess in order to allow a similar construction. This work introduced a new commitment scheme, called *mercurial commitment*, which was later generalized to q-trapdoor mercurial commitments in [CFM08] and further 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. A closely related notion of *vector commitments* was proposed in [CF13], where the authors show that a (concise) q-trapdoor mercurial commitment can be obtained from a vector commitment and a trapdoor mercurial commitment.

The 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 N possible elements. and the proof size for (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] where the proof size for (non-)membership for an individual element is $O(1)$ but the set size is not private.

Ostrovsky *et al.* [ORS04] consider a prover who commits to a multidimensional dataset and later pro-

vides answers range queries that are provably consistent with the commitment. They also consider adding privacy to this protocol. However their construction uses interaction (which can be avoided in the random oracle model) and requires the prover to keep a counter of past queries. Also, their use of NP-reductions and probabilistically checkable proofs makes their generic construction expensive. The authors also provide a simpler construction based on an explicit-hash Merkle Tree. However, this construction that does not hide the size of the database since the proof size is $O(\log N)$ where N is the upper bound on the size of the dataset.

We note that a recent work on DNSSEC zone enumeration by Goldberg *et al.* [GNP+14] uses a model related to our PPAL model and is independently developed. The framework supports only set (non-)membership queries and answers them in f-zero knowledge. This property ensures that the information leaked to the verifier is in terms of a function f on the set, e.g., f is the set size in [GNP+14]. The authors also propose a weaker definition: selective membership security.

Signature Schemes A three party model where the owner digitally signs a data document and outsources it to the server and the server discloses to the client only part of the signed document along with a legitimately derived signature on it (without the owner's involvement), can be instantiated with a collection of signature schemes, namely, *content extraction, quotable, arithmetic, redactable, homomorphic, sanitizable and transitive signatures* [SBZ01, JMSW02, MHI06, MR02, Yi06, CH12]. Additionally, if the signatures reveal no information about the parent document, then this approach can be used to add privacy to outsourced data structure queries. However the generic instantiation, with signature schemes that do not specifically address structured data, is inefficient for most practical purposes.

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 an object x' from a signature on another parent object 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 privacy definition was recently refined by [ALP12].

The authors propose a computationally expensive scheme based on the RSA accumulator. 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. Computing a m-symbol quote from a n-symbol message requires time $O(n(n-m))$ and the verification of a m-symbol quote takes time $O(m^2)$. Predicates for specific data structures are not considered. Also, 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.

Chase et al. [CKLM13] give a definition and construction of a malleable signature scheme, where 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 without access to the secret key. Their definition of simulation context hiding requires transformed signatures to be indistinguishable from freshly simulated signatures on the transformed messages. This definition is stronger than that of [ABC+12] as it allows for adversarially-generated keys and signatures.

A motivating example proposed in [ABC+12] deals with the impossibility of linking a quote to its source document. Context-hiding definition in [CKLM13] also requires unlinkability. However, in our PPAL model, it is important for the client to verify membership, i.e., given a quote from a document and a proof of the quote, the client should be able to verify that the quote is indeed in the document. Also note that the owner is a trusted party in our setting of privacy preserving authenticated lists and therefore the stronger notion of simulation context hiding is not relevant in this framework. So a PPAL cannot be directly instantiated using a malleable signature scheme and PPAL and malleable signature scheme have different applications.

Leakage-Free Signature Schemes for Structural Data 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 leakage-freeness property ensures that the redacted document and its signature do not reveal anything about the content or position of the removed parts. In this section, we discuss the leakage-free redactable signature schemes present in the literature that specifically looks at structural data and ordered lists in particular. We will see that PPAL outperforms known LRSS constructions. Another, significant difference in our definition of privacy. We require simulatability of the query responses, or the zero-knowledge property, as opposed to the game based definitions in the LRSS literature [BBD$^+$10, SPB$^+$12]. Our definition is much more intuitive and simple in capturing the leakage-freeness property.

Kundu and Bertino [KB08] introduced the idea of structural signatures for ordered trees (subsuming ordered lists) that support public redaction of subtrees by third-parties. This work was later extended to undirected graphs and DAGs [KB13]. The notion was later formalized as *LRSS* for ordered trees in [BBD$^+$10] and subsequently several attacks on [KB08] were also proposed in [BBD$^+$10, PSPDM12]. The basic idea of the LRSS scheme presented in [BBD$^+$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 a 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. Kundu *et al.* [KAB12], presented a construction that uses quadratic space at the server and is not leakage-free. We discuss the attack in Section 5.

Chang *et al.* [CLX09] presented a leakage-free redactable signature scheme for a string (which can be viewed as a list) that hides the location of the redacted or deleted portions of the string at the expense of quadratic verification cost.

2.3 Contributions and Organization of the Paper

Our contributions are novel models and efficient constructions. They are summarized below.

- After reviewing preliminary concepts and cryptographic primitives, in Section 3, we introduce the zero-knowledge list (ZKL) model. We describe our ZKL construction, its security and efficiency in Section 4.
- In Section 5, we introduce the privacy-preserving authenticated list (PPAL) model. An efficient PPAL construction based on bilinear maps, its performance and security properties are given in Section 6 and the proof of security is given in Section 7. Finally, in Section 8, we extend order queries to support many interesting statistical queries on a list in zero-knowledge.

In Table 1, we compare our ZKL and PPAL construction with previous work in terms of performance and assumptions. We specifically 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 a three party model of PPAL (see Section 6 for details). Our PPAL construction outperforms all previous work that is based on widely accepted assumptions [BBD$^+$10, SPB$^+$12].

3 Preliminaries

3.1 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., $\mathcal{L}(x_i) = \mathsf{true}$ if $x_i \in \mathcal{L}$ and $\mathcal{L}(x_i) = \mathsf{false}$ if $x_i \notin \mathcal{L}$. For all x_i such that $\mathcal{L}(x_i) = \mathsf{true}$, $\mathsf{rank}(\mathcal{L}, x_i)$ denotes the rank of element x_i in the list, $\mathcal{L}$.

3.2 Cryptographic Primitives

We now describe a signature scheme that is used in our construction and cryptographic assumptions that underlie the security of our method. In particular, our zero knowledge list construction relies on homomorphic integer commitments (Section 3.2.1), zero knowledge protocol to prove a number is non-negative (Section 3.2.2) and zero knowledge sets (Section 3.2.3), while the construction for privacy preserving lists relies on bilinear aggregate signatures and n-Bilinear Diffie Hellman Inversion assumption (Section 3.3).

3.2.1 Homomorphic Integer Commitment Scheme

We use a homomorphic integer commitment scheme $\mathsf{HomIntCom}$ that is statistically hiding and computationally binding [Bou00, DF02]. The latter implies the existence of a trapdoor and, hence, can be used to "equivocate" a commitment (i.e., open the commitment to any message using the trapdoor). We denote a commitment to x as $C(x;r)$ where r is the randomness used for the commitment. For simplicity, we sometimes drop r from the notation and use $C(x)$ to denote the commitment to x. Homomorphic integer 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.

3.2.2 Proving an integer is non-negative in zero-knowledge

We use the following (interactive) protocol between a prover and a verifier: the prover sends a commitment c to an integer $x \geq 0$ to the verifier and proves in zero-knowledge that the committed integer is non-negative, without opening c. We denote this protocol as $\mathsf{P} \leftrightarrow \mathsf{V}(x, r : c = C(x;r) \wedge x \geq 0)$ (Figure 2). In our construction, we will use the commitment scheme $\mathsf{HomIntCom}$ described in Figure 1 and use the algorithm IntCom to compute c.

3.2.3 Zero Knowledge Set scheme

Let D be a set of of key value pairs. If (x, v) is a key, value pair of D, 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) [MRK03] consists of three probabilistic polynomial time algorithms,

	[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
Storage 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	ROM, nEAE	AnAHF	ROM, RSA	ROM, FC, SRSA	ROM, nBDHI

Table 1: Comparison of our ZKL and PPAL constructions with previous work. ZKL is a construction based on Zero-Knowledge lists from Section 4.3 and PPAL is a direct PPAL construction from Section 6 . All the time and space complexities are asymptotic. Notation: n is the list size, m is the query size, and N is the size of the universe from which list elements are taken. Assumptions: Strong RSA Assumption (SRSA); Existential Unforgeability under Chosen Message Attack (EUCMA) of the underlying signature scheme; Random Oracle Model (ROM); n-Element Aggregate Extraction Assumption (nEAE); Associative non-abelian hash function (AnAHF); Factoring a composite (FC); n-Bilinear Diffie Hellman Inversion Assumption(nBDHI).

Figure 1: Homomorphic Integer Commitment Model.

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

$\mathsf{PK}_C \leftarrow \mathsf{IntComSetup}(1^k)$**:** 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 serves as the commitment value for m and r is the opening value.

$x \leftarrow \mathsf{IntComOpen}(\mathsf{PK}_C, c, r)$**:** IntComOpen takes as input the public key, a commitment c and the corresponding opening information r and returns the committed integer x.

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

$(\mathsf{PK}_C, \mathsf{TK}_C) \leftarrow \mathsf{SimIntComSetup}(1^k)$**:** SimIntComSetup takes as input the security parameter and returns a public key PK_C and a trapdoor TK_C.

$(c, r) \leftarrow \mathsf{SimIntCom}(\mathsf{PK}_C, x)$**:** SimIntCom takes PK_C and an integer x and returns a commitment c and the opening information r.

$x' \leftarrow \mathsf{SimIntComOpen}(\mathsf{PK}_C, \mathsf{TK}_C, c, r)$**:** 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.

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

$\mathsf{P} \leftrightarrow \mathsf{V}(x, r : c = C(x; r) \wedge x \geq 0)$: 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.

$\mathsf{ZKS} = (\mathsf{ZKSSetup}, \mathsf{ZKSProver} = (\mathsf{ZKSP}_1, \mathsf{ZKSP}_2), \mathsf{ZKSVerifier})$, 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 in Appendix A.3.

3.2.4 Bilinear Aggregate Signature Scheme

We use bilinear aggregate signature scheme developed by Boneh *et al.* [BGLS03] for our PPAL scheme. Given signatures $\sigma_1, \ldots, \sigma_n$ on *distinct* messages $M_1, \ldots, M_n$ from n distinct users $u_1, \ldots, u_n$, it is possible to aggregate these signatures into a single short signature σ such that it (and the n messages) convince the verifier that the n users indeed signed the n original messages (i.e., user i signed message M_i). We use the special case where a single user signs n *distinct* messages $M_1, \ldots, M_n$. 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 to construct it.

Bilinear Aggregate Signature Construction A bilinear aggregate signature scheme is a 5 tuple of algorithm *Key Generation, Signing, Verification, Aggregation*, and *Aggregate Verification*. We discuss the construction for the case of a single user signing n *distinct* messages $M_1, M_2, \ldots, M_n$ in Figure 4. The description of the generic case of n different users can be found at [BGLS03]. The following notation is used

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 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.

$b \leftarrow \mathsf{ZKSVerifier}(1^k, \mathsf{PK}_D, \mathsf{com}, x, D(x), \mathsf{proof}_x)$**:** 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 $b =$ ACCEPT/REJECT.

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 $e : G \times G \to G_1$
- $H : \{0,1\}^* \to G$ is a full domain hash function viewed as a random oracle that can be instantiated with a cryptographic hash function.

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 $e(\sigma, g) = e(\prod_{i=1}^{k} H(M_i), x)$.

Bilinear Aggregate Signature Security The formal model of security is called the aggregate chosen-key security model. The security of aggregate signature schemes is expressed via the following 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].

3.3 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 $e : G \times G \to G_1$ 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 $\mathcal{A}$ 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 3.1 (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: $Pr[s \xleftarrow{\$} \mathbb{Z}_p^*; y \leftarrow \mathcal{A}(\langle g, g^s, g^{s^2}, \ldots, g^{s^n}\rangle) : y = e(g,g)^{\frac{1}{s}}] \leq \nu(k)$.*

4 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 non-interactively and in zero-knowledge, (proofs reveal nothing beyond the query answer, not even the size of the list) queries of the following form:

- Is $y_i \in \mathcal{L}$ or $y_i \notin \mathcal{L}$, i.e., $\mathcal{L}(y_i) = \mathsf{true}$ or $\mathcal{L}(y_i) = \mathsf{false}$?
- 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]. There are two parties: the *prover* and the *verifier*. The *prover* initially commits to a list of elements and makes the commitment (a short digest) public. We now formally describe the model and the security properties.

4.1 Model

A Zero Knowledge List scheme (ZKL) consists of three probabilistic polynomial time algorithms: $(\mathsf{Setup}, \mathsf{Prover} = (\mathsf{P}_1, \mathsf{P}_2), \mathsf{Verifier})$. 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 (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. proof_O is set 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 $\tilde{\delta}=\{z_i \mid i\in[1,m]\wedge\mathcal{L}(z_i)=\mathsf{true}\}$. 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.

$b\leftarrow\mathsf{Verifier}(1^k,\mathsf{PK},\mathsf{com},\delta,\mathsf{flag},\mathsf{member},\mathsf{proof}_M,\mathsf{order},\mathsf{proof}_O)$ Verifier takes the security parameter, the public key PK, the commitment com and a query (δ,flag) and member, proof_M, order, 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. 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.

4.2 Security Properties

ZKL has three security properties. The first property, completeness, mandates that honestly generated proofs always satisfy the verification test.

Definition 4.1 (Completeness) *For every list $\mathcal{L}$, every query δ and every* flag*,*

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

The second property, soundness, guarantees that the prover should not be able to come up with a query, and corresponding inconsistent (with the initial commitment) answers and convincing proofs.

Definition 4.2 (Soundness) *For every PPT malicious prover algorithm,* Adv*, for every query δ and for every* flag *there exists a negligible function $\nu(.)$ such that:*

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

Finally, zero-knowledge property ensures 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.

Definition 4.3 (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:*

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

where 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 $\mathcal{L}$.*

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 $\mathbb{H}: \{0,1\}^* \to \{0,1\}^l$, if the elements of the list $\mathcal{L}$ are larger that l bits. In particular, given an input list $\mathcal{L}$ the prover P_1 creates a set D where for every element $y_j \in \mathcal{L}$ it adds a (key,value) pair $(\mathbb{H}(y_j), C(j))$. $\mathbb{H}(y_j)$ is a hash of y_j and $C(j)$ is a homomorphic integer commitment of $\mathsf{rank}(\mathcal{L}, y_j)$ (assuming $\mathsf{rank}(\mathcal{L}, y_j) = j$, wlog). P_1 sets up a zero knowledge set on D using ZKSP_1 from Figure 3. The output of ZKSP_1 is a commitment to D, com, that P_1 sends to the verifier.

P_2 operates as follows. 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 corresponding to this 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 order proofs for order queries.

For a given order query $(\delta, 1)$, for every adjacent pair of elements in the returned order, order, P_2 gives a proof of order. Recall that order contains the member elements of δ, arranged according to their order in the list $\mathcal{L}$. P_2 proves the order between two elements y_i and y_j as follows. Let $\mathsf{rank}(\mathcal{L}, y_i) = i, \mathsf{rank}(\mathcal{L}, y_j) = j$, and $C(i)$, $C(j)$ be the corresponding commitments and, wlog, let $i < j$. As noted above, $C(i)$ and $C(j)$ are already returned by P_2 as part of the membership proof. Additionally, P_2 returns a commitment to 1, $C(1)$, and its opening information ρ.

The verification of the query answer proceeds as follows. Verifier computes $C(j-i-1) := C(j)/(C(i)C(1))$ using the homomorphic property of the integer commitment scheme. P_2 uses the zero knowledge protocol $\mathsf{P} \leftrightarrow \mathsf{V}(x, r : c = C(x;r) \wedge x \geq 0)$ to convince Verifier that $C(j-i-1)$ is a commitment to value ≥ 0. Note that we use the non-interactive general zero-knowledge version of the protocol as discussed in Section 3.2.2. Hence, the query phase proceeds in a single round.

It is important to understand why we require Verifier to verify that $j-i-1 \geq 0$ and not $j-i \geq 0$. By the soundness of the protocol $\mathsf{P} \leftrightarrow \mathsf{V}(x, r : c = C(x;r) \wedge x \geq 0)$, the probability that a cheating prover Adv will be able to convince 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 Adv 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.

4.3 ZKL Construction

Let $\mathsf{HomIntCom} = (\mathsf{IntComSetup}, \mathsf{IntCom}, \mathsf{IntComOpen})$ be the homomorphic integer commitment scheme defined in Section 3.2.1 and $\mathsf{ZKS} = (\mathsf{ZKSSetup}, \mathsf{ZKSProver} = (\mathsf{ZKSP}_1, \mathsf{ZKSP}_2), \mathsf{ZKSVerifier})$ be a ZKS scheme defined in Section 3.2.3. We denote the output of the prover during the non-interactive statistical zero knowledge protocol $\mathsf{P} \leftrightarrow \mathsf{V}(x, r : c = C(x;r) \wedge x \geq 0)$ 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 $\mathcal{L} = \{y_1, \ldots, y_n\}$.

4.4 ZKL 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 ho-

Figure 5: Zero Knowledge List (ZKL) Construction

$\mathsf{PK} \leftarrow \mathsf{Setup}(1^k)$**:** 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 $\mathsf{PK} = (\mathsf{PK}_C, \mathsf{PK}_D)$.

$(\mathsf{com}, \mathsf{state}) \leftarrow \mathsf{P_1}(1^k, \mathsf{PK}, \mathcal{L})$**:** 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 $(\mathsf{com}, \mathsf{state})$.

$(\mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O) \leftarrow \mathsf{P_2}(\mathsf{PK}, \mathsf{state}, \delta, \mathsf{flag})$ 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 $\mathsf{proof}_M := \{\Delta_x \mid x \in S\}$. Note that $\mathcal{L}(z_j) = \mathsf{true}$ when $D(\mathbb{H}(z_j)) \neq \perp$ and $\mathcal{L}(z_j) = \mathsf{false}$ when $D(\mathbb{H}(z_j)) = \perp$.

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

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

Let $\tilde{\delta} = \{z_j \mid \forall j \in [1, m] \wedge \mathcal{L}(z_j) = \mathsf{true}\}$ 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 $(\mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O)$.

$b \leftarrow \mathsf{Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, \mathsf{flag}, \mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O)$ **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 $\mathsf{proof}_M := \{\Delta_x = (D(x), \mathsf{proof}_x) \mid x \in S\}$. Recall that $\mathcal{L}(\mathbb{H}^{-1}(x)) = \mathsf{true}$ when $D(x) \neq \perp$ and $\mathcal{L}(\mathbb{H}^{-1}(x)) = \mathsf{false}$ when $D(x) = \perp$.
- 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 $\mathsf{proof}_{\mathsf{rank}(\mathcal{L}, j+1) - \mathsf{rank}(\mathcal{L}, j) - 1 \geq 0}$ and verification steps of $\mathsf{P} \leftrightarrow \mathsf{V}(x, r : c = C(x; r) \wedge x \geq 0)$ where $x = \mathsf{rank}(\mathcal{L}, j+1) - \mathsf{rank}(\mathcal{L}, j) - 1$.

If all the verifications pass, only then return ACCEPT.

momorphic integer commitment of [DF02] and the non-interactive general zero-knowledge version of the Σ-protocol to prove non-negativity in [Lip03] in the random oracle model. Each of these constructions is described in 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 A.3.

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 $\mathsf{P} \leftrightarrow \mathsf{V}(x,r : c = C(x;r) \wedge x \geq 0)$ which takes $O(\log^2 n) = O(\log^2(\mathsf{poly(k)}))$ expected time, since $n = \mathsf{poly(k)}$. Hence, computing a membership proof for a single element takes time $O(k)$ and an order proof for two elements takes time $O(k + \log^2(\mathsf{poly(k)})) = O(k)$. More generally, the prover's time for a query on sublist δ is $O(mk + m\log^2(\mathsf{poly(k)})) = O(mk)$, 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 the verifications steps of $\mathsf{P} \leftrightarrow \mathsf{V}(u,r : c = C(u;r) \wedge u \geq 0)$. Therefore, the asymptotic run time of the verification is $O(mk)$.

4.5 ZKL Security Proofs

Proof of Completeness Completeness of the ZKL construction in Section 4.3 directly follows from the Completeness of Zero Knowledge Set and Completeness of the protocol $\mathsf{P} \leftrightarrow \mathsf{V}(x,r : c = C(x;r) \wedge x \geq 0)$. ∎

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

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

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

Then, Definition 4.2 can be rewritten as

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

Now, by the Soundness property of the ZKS in Section 3.2.3, $\Pr[\mathbb{E}_1]$ 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 3.2.1, Adv cannot equivocate $C(i-j)$ or $C(j-i)$ (which is computed by

Verifier in the protocol). (Note that by the soundness property of ZKS, the probability that Adv 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_M^1 and proof_M^2, respectively, is negligible w.r.t. the same commitment, com.) Then according to the protocol, it must be the case that Adv could convince 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 $\mathsf{P} \leftrightarrow \mathsf{V}(x, r : c = C(x;r) \wedge x \geq 0)$, 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 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 4.3), 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 (statistical) hiding property of the integer commitment scheme, the commitments are statistically closely 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 $\mathsf{P} \leftrightarrow \mathsf{V}(u, r : c = C(u;r) \wedge u \geq 0)$ to compute $\mathsf{proof}_{u \geq 0}$. Since the protocol $\mathsf{P} \leftrightarrow \mathsf{V}(u, r : c = C(u;r) \wedge u \geq 0)$ 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 SimHomIntCom we use the construction of [DF02] and for completeness define a simulator in Figure 9. We summarize the properties and efficiency of our ZKL construction in Theorem 4.1.

Theorem 4.1 *The zero-knowledge list (ZKL) construction of Figure 5 satisfies the security properties of completeness (Definition 4.1), soundness (Definition 4.2) and zero-knowledge (Definition 4.3) in the random oracle model. 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 commitment phase in $O(n \log N)$ time and space.*
- *In the query phase, the prover computes the proof of the answer in $O(m \log N)$ time.*
- *The verifier verifies the proof in $O(m \log N)$ time and space.*

5 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 as we discuss in Section 5.3. 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.

5.1 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 on the elements of the list and verifies corresponding answers. We note that this model assumes that the query is on the member elements of the list, i.e., for any query, δ, $\mathsf{Elements}(\delta) \subseteq \mathsf{Elements}(\mathcal{L})$. In other words, this model does not support proofs of non-membership, similar to other data structures that support only positive membership proofs, e.g., [KB08, CLX09, CKS09, BBD+10, PSPDM12, KAB12, CF13].

$(\mathsf{digest}_C, \mathsf{digest}_S) \leftarrow \mathsf{Setup}(1^k, \mathcal{L})$ 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 server digest 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. Wlog, we assume $|\delta| > 1$. In the trivial case of $|\delta| = 1$, the server returns an empty proof, i.e., $(\mathsf{order} = \delta, \mathsf{proof} = \perp)$.

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

5.2 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.

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

Definition 5.1 (Completeness) *For all lists $\mathcal{L}$ and all sublists δ of $\mathcal{L}$*

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

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 list $\mathcal{L}$, even a malicious server will not be able to convince the client of an incorrect order of elements belonging to the list $\mathcal{L}$. This property ensures integrity of the scheme.

Definition 5.2 (Soundness) *For all PPT malicious query algorithms* Adv*, for all lists $\mathcal{L}$ and all query sublists δ of $\mathcal{L}$, there exists a negligible function $\nu(.)$ such that:*

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

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 a honest owner and honest server who know $\mathcal{L}$ and answer w.r.t. $\mathcal{L}$, or to the simulator that only has oracle access to the list $\mathcal{L}$.

Definition 5.3 (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 The scheme of [KAB12] generates a n' bit secure name, where $n' \geq n$, for each element of the list $\mathcal{L}$ of size n. The secure name of element x has one distinct bit that encodes the pairwise order between x and every other element in $\mathcal{L}$. To prove the order between two elements, the server sends their secure names to the client, who can easily identify and compute the bit that encodes the order between the corresponding elements. Hence non-transitive orders can be easily inferred as follows. Let $\{A < B < C\}$ be the list order. The client issues order queries for (A,B) and (A,C), which returns the secure names for A,B,C. Now the client can find out the order between B and C, that was not queried for. Hence, the scheme is not zero-knowledge.

5.3 Construction of PPAL via ZKL

We show how a PPAL can be instantiated via a ZKL in Theorem 5.1 and then discuss that the resulting construction does not yield the desired efficiency.

Theorem 5.1 *Given a ZKL scheme* $\mathsf{ZKL} = (\mathsf{Setup}, \mathsf{Prover} = (\mathsf{P}_1, \mathsf{P}_2), \mathsf{Verifier})$*, which supports queries of the form* (δ, flag) *on a list $\mathcal{L}$, we can instantiate a PPAL scheme for the list $\mathcal{L}$,* $\mathsf{PPAL} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Verify})$*, which supports queries of the form δ, where δ is a sublist of $\mathcal{L}$, as follows:*

$\mathsf{PPAL.Setup}(1^k, \mathcal{L})$ *can be instantiated as follows:*

1. *Invoke* $\mathsf{ZKL.Setup}(1^k)$. *Let* $\mathsf{ZKL.Setup}(1^k) = \mathsf{PK}$
2. *Invoke* $\mathsf{ZKL.P}_1(1^k, \mathsf{PK}, \mathcal{L})$. *Let* $\mathsf{ZKL.P}_1(1^k, \mathsf{PK}, \mathcal{L}) = (\mathsf{com}, \mathsf{state})$.
3. *Set* $\mathsf{digest}_C = (\mathsf{PK}, \mathsf{com}), \mathsf{digest}_S = (\mathsf{PK}, \mathsf{com}, \mathsf{state})$ *and output* $(\mathsf{digest}_C, \mathsf{digest}_S)$.

$\mathsf{PPAL.Query}(\mathsf{digest}_S, \mathcal{L}, \delta)$ *can be instantiated as follows:*

1. *Invoke* $\mathsf{ZKL.P}_2(\mathsf{PK}, \mathsf{state}, \delta, 1)$. *Let* $\mathsf{ZKL.P}_2(\mathsf{PK}, \mathsf{state}, \delta, 1) = (\mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O)$
2. *Output* $(\mathsf{order}, \mathsf{proof} = (\mathsf{proof}_M, \mathsf{proof}_O))$

$\mathsf{PPAL.Verify}(\mathsf{digest}_C, \delta, \mathsf{order}, \mathsf{proof}_M, \mathsf{proof}_O)$ *can be instantiated as follows:*

1. *Set* $\mathsf{member} = \{1, 1, \ldots, 1\}$ *such that* $|\mathsf{member}| = |\delta| = |\mathsf{order}|$ *(recall that in a PPAL scheme, the query* δ*, is a sublist of* $\mathcal{L}$*).*
2. *Invoke* $\mathsf{ZKL.Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, 1, \mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O)$ *(recall that* $\mathsf{digest}_C = (\mathsf{PK}, \mathsf{com})$*).*
3. *Output bit b where* $\mathsf{ZKL.Verifier}(1^k, \mathsf{PK}, \mathsf{com}, \delta, 1, \mathsf{member}, \mathsf{proof}_M, \mathsf{order}, \mathsf{proof}_O) = b$

Theorem 5.2 *A PPAL scheme instantiated using a ZKL scheme,* $\mathsf{ZKL} = (\mathsf{Setup}, \mathsf{Prover} = (\mathsf{P}_1, \mathsf{P}_2), \mathsf{Verifier})$ *has the following performance:*

- *The owner's runtime and space are proportional to the runtime and space of* $\mathsf{ZKL.Setup}$ *and* $\mathsf{ZKL.P}_1$*, respectively.*
- *The server's runtime and space are proportional to the runtime and space of* $\mathsf{ZKL.P}_2$.
- *The client's runtime and space are proportional to the runtime and space of* $\mathsf{ZKL.Verifier}$.

The correctness of Theorems 5.1 and 5.2 follow from the definition of PPAL and ZKL models.

We estimate the asymptotic efficiency of a PPAL construction based on the construction of ZKL presented inSection 4.3. From the discussion of efficiency of the ZKL construction in Theorem 4.1, the time and space complexity of each party in PPAL adaptation of ZKL readily follows.

Owner The owner runs in time $O(kn)$ and $O(kn)$ space, assuming each element of the list is k-bits.

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.

As we see, this generic construction is not very efficient due to the multiplicative factor $O(k)$ and heavy cryptographic primitives. In Section 6, we present a direct PPAL construction which 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.

6 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 blinded component of the bilinear accumulator public key. This allows us to encode the rank of the element in the member witness and then "blind" rank information with randomness. Every pair of (element, member witness) is signed by the owner and the signatures are aggregated using bilinear aggregate signature schemepresented 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 owner also sends the list digest signature and the public key of the bilinear aggregate signature scheme to the client. The advantage of using an aggregate signature is for 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 invocation to aggregate signature verification.

The owner also sends to the server 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 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. 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 $t_{x_i < x_j}$. 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 compute the digest signature for the sublist, σ_{order}, and its membership verification unit $\lambda_{\mathcal{L}'}$ which are part of the Σ_{order} component of the proof. To compute the Ω_{order} component of the proof, the server uses corresponding blinding values in $\Omega_{\mathcal{L}}$ and elements g^{s^d} where d's correspond to distances between ranks of queried elements.

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}. Hence, the client uses bilinear map for two purposes: first as part of member verification and then to verify the 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_{order} 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 $\mathcal{L} = \{x_1, \ldots, x_n\}$. A balanced binary tree is built over n leaves, where the ith leave corresponds to x_i and stores ψ_i. Each internal node of the tree stores the product of the values stored at 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 $\mod p$. $\mathcal{L}$ is the input list of size $n = \mathsf{poly}(\mathsf{k})$, where x_i's are distinct and $\mathsf{rank}(\mathcal{L}, x_i) = i$. System parameters are $(p, G, G_1, e, g, \mathcal{H})$.

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

$\sigma_{\mathcal{L}}$ 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: $\sigma_{\mathcal{L}} \leftarrow \mathsf{salt} \times \prod_{1 \le i \le n} \sigma_i$.

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

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

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

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

$\Sigma_{\mathsf{order}} = (\sigma_{\mathsf{order}}, T, \lambda_{\mathcal{L}'})$ 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 $d = |i' - i''|$.

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

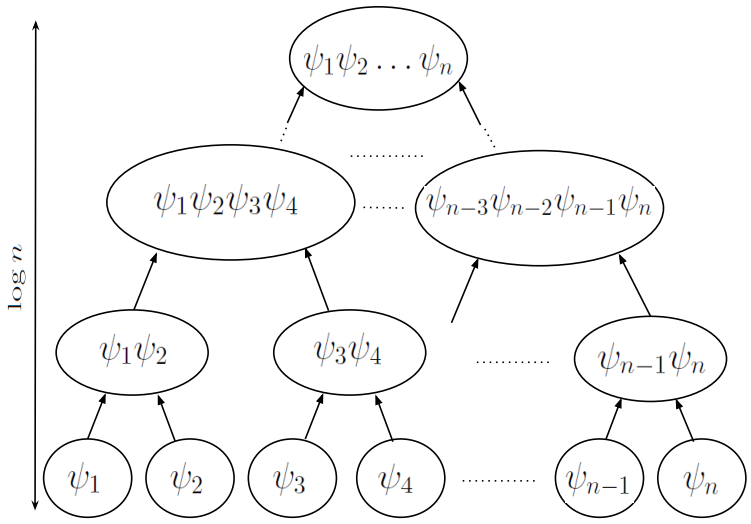

Figure 6: Range tree showing the precomputed products where $\psi_i = \mathcal{H}(t_{x_i \in \mathcal{L}} || x_i)$. Precomputed products allow to speed up the computation time of Query algorithm in Figure 7 when $m \ll n$.

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 $O(\log n)$ 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 witness for each element, along with signature for each element. Hence, the algorithm runs in time $O(n)$ and requires $O(n)$ space.

Server Computing $\lambda_{\mathcal{L}'}$ takes time $O(n-m)$, as it touches $|\mathcal{L} \setminus \delta|$ elements, and computing σ_{order} takes time $O(m)$. Hence, the overall runtime of computing $\lambda_{\mathcal{L}'}$ and σ_{order} is $O(n)$. The server can precompute and store some products, as mentioned in the preprocessing step, 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.

Batch ordering query The client can learn the total order among m different elements of the list using a basic order 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.

7 Security of the PPAL Construction

In this section we prove that the construction presented in Section 6 is secure according to the definitions of completeness, soundness and zero knowledge in Section 5.

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

$$\begin{aligned} e(\sigma_{\mathsf{order}}, g) =& e(\textstyle\prod_{y_j \in \mathsf{order}} \sigma_{\mathsf{rank}(\mathcal{L}, y_j)}, g) = e(\textstyle\prod_{y_j \in \mathsf{order}} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i)^v, g) = \\ & e(\textstyle\prod_{y_j \in \mathsf{order}} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i), g^v) = e(\textstyle\prod_{y_j \in \delta} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i), g^v) = e(\xi, g^v). \end{aligned}$$

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.

$$\begin{aligned} e(\sigma_{\mathsf{order}}, g) \times e(\lambda_{\mathcal{L}'}, g^v) &= e(\textstyle\prod_{y_j \in \mathsf{order}} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i)^v, g) \times e(\mathcal{H}(\omega) \times \textstyle\prod_{x \in \mathcal{L}'} \mathcal{H}(t_{x_{\mathsf{rank}(\mathcal{L}, x)} \in \mathcal{L}} || x), g^v) \\ &= e(\textstyle\prod_{y_j \in \mathsf{order}} \mathcal{H}(t_{y_j \in \mathcal{L}} || y_i), g^v) \times e(\mathcal{H}(\omega) \times \textstyle\prod_{x \in \mathcal{L}'} \mathcal{H}(t_{x_{\mathsf{rank}(\mathcal{L}, x)} \in \mathcal{L}} || x), g^v) \\ = e(\mathcal{H}(\omega) \times \textstyle\prod_{x \in \mathcal{L}} \mathcal{H}(t_{x_{\mathsf{rank}(\mathcal{L}, x)} \in \mathcal{L}} || x), g^v) &= e(\mathcal{H}(\omega)^v \times \textstyle\prod_{x \in \mathcal{L}} \mathcal{H}(t_{x_{\mathsf{rank}(\mathcal{L}, x)} \in \mathcal{L}} || x)^v, g) = e(\sigma_{\mathcal{L}}, g). \end{aligned}$$

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 $r'' = \Omega_{\mathcal{L}}[i'']$.

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

7.2 Proof of Soundness

Soundness follows by reduction from the n-Bilinear Diffie Hellman assumption (see Definition 3.1 for details). To the contrary of the Soundness Definition 5.2, assume that given a list $\mathcal{L}$, the malicious server, Adv produces a forged order order on a non-trivial sublist $\delta = \{x_1, x_2, \ldots, x_m\}$, where $m \geq 2$, such that $\mathsf{order} \neq \pi_{\mathcal{L}}(\delta)$ and corresponding order proof is accepted by the client, i.e., by algorithm Verify in Figure 7. By the security of the signature scheme [BGLS03], $x_1, x_2, \ldots, x_m \in \mathcal{L}$. Since $|\delta| > 1$, there exists at least one inversion pair (x_i, x_j) in order where $i, j \in [1, m]$. Let us assume, wlog, $x_i < x_j$ 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 Adv 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 Adv and using its forged witness, $t_{x_j < x_i}$, 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 7.1 *If n-Bilinear Diffie Hellman assumption holds, then* PPAL *scheme satisfies Soundness in Definition 5.2.*

Proof We show that if there exists a malicious Adv 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 $v \xleftarrow{\$} \mathbb{Z}_p^*$ a list $\mathcal{L}$ such that $|\mathcal{L}| = n$.
   Pick $\Omega_{\mathcal{L}} = \{r_i \xleftarrow{\$} \mathbb{Z}_p^*\}_{\forall i \in [1,n]}$ 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 $\omega \xleftarrow{\$} \{0,1\}^*$ 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 Adv outputs a forged order order for some non-trivial 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 Adv has successfully generated a valid proof $t_{x_j < x_i} = \left(g^{s^{(u-v)}}\right)^{r_u r_v^{-1}}$.
3. Now $\mathcal{A}$ outputs $e(t_{x_j < x_i}, (g^{s^{v-u-1}})^{r_u{}^{-1} r_v}) = e(g,g)^{\frac{1}{s}}$.

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

7.3 Proof of Zero-Knowledge

We define Zero Knowledge Simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ from Definition 5.3 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 in 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 order, 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 7.2.

Theorem 7.2 *The privacy-preserving authenticated list (PPAL) construction of Figure 7 satisfies the security properties of completeness (Definition 5.1), soundness (Definition 5.2) and zero-knowledge (Defini-*

tion 5.3) in the random oracle model and under the n-BDHI assumption (Definition 3.1). Also, the construction has the following performance, where n denotes the list size and m denotes the query size.

- *The owner and the 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.*

8 Applications of Zero Knowledge Order Queries

In this section we present queries that can be answered in zero knowledge using zero knowledge queries defined in this paper. Let $\mathcal{L}$ be a list of n elements and order queries are defined w.r.t. $\mathcal{L}$. Let $\mathcal{S} = \{x_1, \ldots, x_m\}$ be a subset of $\mathcal{L}$. We note that elements in $\mathcal{S}$ are not necessarily ordered according to their order in $\mathcal{L}$. We use the notation $\mathsf{proof}_{x<y}$ to denote the (simulatable) proof that x precedes y in $\mathcal{L}$. We note that order proofs returned by ZKL and PPAL are simulatable. The queries below can be answered by both ZKL and PPAL. However, as a specific instantiation, we use our PPAL construction in Figure 7, since it returns $\mathsf{proof}_{x<y}$ of constant size, i.e., $|\mathsf{proof}_{x<y}| = O(1)$. The ZKL construction in Figure 5, on the other hand, returns $|\mathsf{proof}_{x<y}| = O(k)$, where k is the security parameter of the scheme.

Maximum, Minimum: We define the queries $\mathsf{Max}(\mathcal{S})$ and $\mathsf{Min}(\mathcal{S})$ with respect to $\mathcal{L}$ as follows: let $\tilde{\mathcal{S}}$ be the set $\mathcal{S}$ with the elements rearranged according to the total order induced by the list, i.e., $\tilde{\mathcal{S}} = \pi_{\mathcal{L}}(\mathcal{S})$. $\mathsf{Max}(\mathcal{S})$ and $\mathsf{Min}(\mathcal{S})$ return the highest ranked and the lowest ranked element of $\tilde{\mathcal{S}}$, respectively, along with the corresponding proofs. Formally, $(x, \mathsf{proof}_{max}) \leftarrow \mathsf{Max}(\mathcal{S})$ where $\mathsf{proof}_{max} = \{\mathsf{proof}_{y<x} \mid y \in \mathcal{S} \wedge y \neq x\}$ and $(x, \mathsf{proof}_{min}) \leftarrow \mathsf{Min}(\mathcal{S})$ where $\mathsf{proof}_{min} = \{\mathsf{proof}_{x<y} \mid y \in \mathcal{S} \wedge y \neq x\}$. We note that the proof size, $|\mathsf{proof}_{max}|$ (or $|\mathsf{proof}_{min}|$) is $O(|\mathcal{S}|)$.

Median: The query $\mathsf{Median}(\mathcal{S})$ returns the median element along with a proof of the answer, which proves that the returned element precedes $\lfloor |\mathcal{S}|/2 \rfloor$ elements and succeeds $\lceil |\mathcal{S}|/2 \rceil$ elements in the list, and reveals nothing more. Formally, $(x, \mathsf{proof}_{median}) \leftarrow \mathsf{Median}(\mathcal{S})$, where x is the median element of $\mathcal{S}$ and $\mathsf{proof}_{median} = \{\mathsf{left}, \mathsf{right}\}$ with $\mathsf{left} = \{\mathsf{proof}_{y<x} \mid y \in \mathcal{S} \wedge y \neq x\}$, $\mathsf{right} = \{\mathsf{proof}_{x<y} \mid y \in \mathcal{S} \wedge y \neq x\}$, $|\mathsf{left}| = \lfloor |\mathcal{S}|/2 \rfloor$, and $|\mathsf{right}| = \lceil |\mathcal{S}|/2 \rceil$. We note that the proof size, $|\mathsf{proof}_{median}| = O(|\mathcal{S}|)$.

Top t: The query $\mathsf{Top}(t, \mathcal{S})$ returns the top t elements along with a proof of the answer, where $1 \leq t \leq |\mathcal{S}| - 1$. Formally, $(\mathcal{S}_t, \mathsf{proof}_t) \leftarrow \mathsf{Top}(t, \mathcal{S})$ where $\mathcal{S}_t$ is the *unordered* set of the top t elements of $\tilde{\mathcal{S}}$, where $\tilde{\mathcal{S}} = \pi_{\mathcal{L}}(\mathcal{S})$. Let $\mathcal{S}' = \mathcal{S} \setminus \mathcal{S}_t$. Then $\mathsf{proof}_t = \{\mathsf{proof}_{x<y} \mid x \in \mathcal{S}' \wedge y \in \mathcal{S}_t\}$. The size of proof_t is $O(t \cdot |\mathcal{S}'|) = O(|\mathcal{S}|)$, since $1 \leq t \leq |\mathcal{S}| - 1$. We note that $\mathsf{Top}(t, \mathcal{S})$ can be also implemented to return the *ordered* set of the top t elements, $\tilde{\mathcal{S}}_t$, instead of $\mathcal{S}_t$ where $\tilde{\mathcal{S}}_t = \pi_{\mathcal{L}}(\mathcal{S}_t)$. Then $\mathsf{proof}_t = \{\mathsf{proof}_{x<y} \mid \forall \text{ adjacent pairs } (x,y) \in \tilde{\mathcal{S}}_t\} \cup \{\mathsf{proof}_{x<y} \mid x \in \mathcal{S}' \text{ and } y \text{ is the last element in } \tilde{\mathcal{S}}_t\}$. In this case also, the size of proof_t is $O(|\mathcal{S}|)$. Note that in the *ordered set* case, if $t = |\mathcal{S}|$, then $\mathsf{Top}(t, \mathcal{S})$ reduces to an order query on $\mathcal{S}$ and the proof size is $O(|\mathcal{S}|)$.

Threshold proof: Let a be a public element of $\mathcal{L}$. We define the query $\mathsf{Threshold}(\mathcal{S}, a)$ to return a boolean vector $\mathbb{B} = \{b_1, \ldots, b_m\}$ along with a proof that indicates whether elements are below or above the threshold element a. $\mathbb{B}$ is defined as follows: $b_i = 1$ if $a < x_i$ and $b_i = 0$, otherwise. Formally, $(\mathbb{B}, \mathsf{proof}_{threshold}) \leftarrow \mathsf{Threshold}(\mathcal{S}, a)$, where $\mathsf{proof}_{threshold} = \{\mathsf{proof}_{x_i<a} \mid x_i \in \mathcal{S} \wedge b_i = 0\} \cup \{\mathsf{proof}_{a<x_i} \mid x_i \in \mathcal{S} \wedge b_i = 1\}$. The size of $\mathsf{proof}_{threshold}$ is $O(|\mathcal{S}|)$.

9 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, Markulf Kohlweiss, Anna Lysyanskaya, 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.

[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.

[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.

[CKS09] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In StanisÅĆaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography âĂŞ PKC 2009*, volume 5443 of *Lecture Notes in Computer Science*, pages 481–500. Springer Berlin Heidelberg, 2009.

[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.

[GNP+14] Sharon Goldberg, Moni Naor, Dimitrios Papadopoulos, Leonid Reyzin, Sachin Vasant, and Asaf Ziv. NSEC5: Provably preventing DNSSEC zone enumeration. Cryptology ePrint Archive, Report 2014/582, 2014.

[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.

[RS86] Michael O. Rabin and Jeffery O. Shallit. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics*, 39(S1):S239–S256, 1986.

[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 Preliminaries

A.1 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. The construction of Homomorphic Integer Commitment Scheme [DF02] is given

Figure 8: Homomorphic Integer Commitment Scheme [DF02].

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

$\mathsf{PK}_C \leftarrow \mathsf{IntComSetup}(1^k)$**:** The 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 $h \xleftarrow{\$} \mathcal{G}$ (by group assumption, $ord(h)$ is $F(k)$-rough with overwhelming probability) and a random secret key $s \xleftarrow{\$} \mathbb{Z}_{ord(\mathcal{G})}$ and sets $g := h^s$. 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, $r \xleftarrow{\$} \mathbb{Z}_{2^{B+k}}$, 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 $b := 1$.

in Figure 8. This 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 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 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 $(x', (r + sx - sx') \mod ord(\mathcal{G}), b = 1)$ to the verifier.

A.2 Proving an Integer is Non-negative [Lip03]

We present the Σ protocol [Lip03] implementing the above functionality 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].

At high level, the protocol is 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 A.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 and the prover can efficiently compute the Lagrange representation of a given integer x in expected time $O(\log^2 x)$ [RS86], the *Prover* runs in time $O(\log^2 x)$ and the *Verifier* runs in asymptotic constant time, i.e., $O(1)$ in the protocol in Figure 10.

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

Step 1: The *Prover* P 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* V 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 $g^{m_{2i}} h^{r_{4i}} {c_{1i}}^{-e} \stackrel{?}{=} c_2$
- $(\prod_i {c_{1i}}^{m_{2i}}) h^{r_5} c^{-e} \stackrel{?}{=} c_3$

A.3 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 denote the randomness used to produce the commitment (hard or soft) of x.

Before describing 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, $C_\varepsilon = \mathsf{nil}$, redefine $C_\varepsilon = \mathsf{SoftComm}(\mathsf{PK}_D, r_\varepsilon)$. This happens only when $D = \phi$. Finally define $C_D = C_\varepsilon = \mathsf{com}$.

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

x ∈ 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 ∉ 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 ≠ ⊥: The verifier ZKSVerifier performs the following:

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

x = ⊥: 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 $1 \leq i < l$
- $\mathsf{VerTease}(\mathsf{PK}_D, C_D, (C_0, C_1), \tau_\varepsilon)$ and $\mathsf{VerTease}(\mathsf{PK}_D, C_x, \perp, \tau_x)$