Fully-Dynamic Verifiable Zero-Knowledge Order Queries for Network Data

Esha Ghosh*[1], Michael T. Goodrich †[2], Olga Ohrimenko‡[3] and Roberto Tamassia§[1]

[1]Dept. Computer Science, Brown University
[2]Dept. Computer Science, U. California, Irvine
[3]Microsoft Research

Abstract

We show how to provide privacy-preserving (zero-knowledge) answers to order queries on network data that is organized in lists, trees, and partially-ordered sets of bounded dimension. Our methods are efficient and dynamic, in that they allow for updates in the ordering information while also providing for quick and verifiable answers to queries that reveal no information besides the answers to the queries themselves.

1 Introduction

Maintaining an ordered list of elements in a trustworthy and privacy-preserving manner has a number of applications in network information management.

- *Firewall policies* are often expressed as an ordered list of rule-action pairs [16, 32], $((r_1, a_1), (r_2, a_2), \ldots, (r_n, a_n))$, where if a network packet, p, matches two rules, r_i and r_j, with $i < j$, then the action a_i should be applied, rather than the action a_j. The contents and ordering of such firewall policy lists are potentially sensitive from a security perspective, so it is desirable that external rule-comparison queries to such a list are answered on a "need to know" basis, without revealing other rules in the list or even the number, n, of rules in the list.
- In *collaborative filtering* and *reputation management* systems, one maintains an ordered preference list for a set of items (e.g., products or people), based on popularity or feedback scores. Due to the potential for feedback extortion [31], answers to queries on such lists should be limited to reporting the preference order between two items without revealing relative orderings between other items.
- In *distributed grid computing*, such as folding@home and distributed.net, incentives are provided to the top-k most productive participants. Due to the prevalence of cheating [30], however, the incentive service should ideally prove to a participant that she is the kth most productive without revealing the ranking or relative ordering of the other participants.

In addition to security and privacy issues, the above applications raise interesting algorithmic challenges, in that ordered lists tend to change over time (e.g., see [32]). Thus, we would like to have efficient ways to dynamically maintain ordered lists so as to securely and privately answer

*esha_ghosh@brown.edu
†goodrich@uci.edu
‡oohrim@microsoft.com
§roberto_tamassia@brown.edu

relative-order queries. Moreover, we are interested in solutions that are themselves implemented in a networked, cloud-computing setting, where a data owner outsources query processing to a cloud server, who then answers queries for a set of distributed clients.

Besides simple linear order queries, as outlined above, there are also richer order queries that one can perform on trees and even partially-ordered sets of bounded dimension, which likewise have security and privacy concerns in networking and network information management applications.

- *XML* is a common format for distributing and managing information in networked environments, but, because it is human-readable, it has security and privacy concerns [38]. Thus, we would like to be able to perform verifiable queries on the tree structure of an XML document so that the answer reveals no more information than can be inferred from the answer itself.
- In *wireless networking* applications, access control can be defined by geo-spatial location, where access policies are defined in terms of rectangular regions [3]. Since rectangle inclusion is a poset of bounded dimension, and access control involves sensitive policies, this work motivates the need for secure, verifiable, private methods for querying partial orders of bounded dimension.
- In distributed computing settings, proposed general formulations for *key management and access control* are based on directed acyclic graphs (DAGs) that represent posets of bounded dimension [2].
- Finally, a broad class of *firewall policies* can be expressed in terms of DAGs representing posets of bounded dimension [53]. Moreover, for several policies, the DAG is planar and thus its associated poset has dimension 3.

These applications have security and privacy concerns that need to be addressed in a dynamic environment where the tree (or DAG) evolves over time.

Contributions: Our contributions can be summarized as follows.

- Motivated by networking and cloud computing applications, we introduce a formal model of a dynamic privacy-preserving authenticated data structures (DPPADS) in a three party model where the owner outsources his data structure to a server who answers queries issued by a set of distributed clients. The owner can at any point update the data structure. The server answers queries in such a way that the clients (1) can verify the correctness of the answers but (2) do not learn anything about the data structure besides what can be inferred from the query answers (Section 4).
- We give an efficient and provably secure construction of a DPPADS for a list that supports order queries and updates. This construction is based on standard cryptographic assumptions and has optimal performance in all cost measures, except for a logarithmic overhead on the query time (Section 5).
- We define a space-efficient variation of the above model and give an efficient construction for it (Section 6).
- We present an efficient extension of DPPADS to trees and posets of bounded dimension (Section 7).

In summary, our work provides efficient, secure and privacy-preserving mechanisms for networking applications that rely on querying order information stored in lists, trees, or posets of bounded dimension.

2 Related Work

Adding privacy to range [47], order [28] and dictionary queries on *static* data structures, has received considerable attention [19, 21, 40, 43, 36, 29, 50]. In parallel, a rich body of literature has been developed on digital signature schemes, where it is possible to generate a signature on a subdocument of a static parent document without signer's secret key [17, 34, 44, 45, 52, 59]. This work was extended to support privacy of the document, where a derived signature for a subdocument reveals no extra information about the parent document [1, 4, 55, 22]. Support for the desired notion of privacy (also called leakage-freeness) on signatures of static structural data was addressed in [38, 15, 48, 39, 20, 51].

Recent work [49] on dynamic updates for signatures on a set supports privacy-preserving verification of only positive membership (i.e., a proof is returned only when the queried elements are members of the given set). It extends formal definitions of security and gives methods satisfying them for two update operation: addition of new elements and merge of two sets. In comparison, we consider operations on lists, trees and support delete and replace operations in the three party model. Their notion of privacy is based on an indistinguishability game as opposed to simulability property in our definition. Our definition is not tailored to a specific data structure, while the definition in [49] cannot be easily extended to support richer data structures and queries. The authors leave the question of efficient construction for more complex data structures as an open problem, which we answer positively in this work.

The model of *zero knowledge set* (ZKS) was introduced by Micali *et al.* [43]. ZKS lets a prover commit to a finite set S in such a way that, later on, she can 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. Hence, the size of S remains secret as well. The notion of Updatable Zero-Knowledge Set (U-ZKS) (or, more generically, Updatable Zero-Knowledge Elementary DataBase, U-ZK-EDB)was first proposed in [41]. This work gives two definitions of updates: transparent and opaque. The transparent definition explicitly reveals that an update has occurred and the verifier can determine whether previously queried elements were updated. Constructions satisfying transparent updates are given in [41] and [18], where [18] has a better performance and uses updatable vector commitments. The author of [41] leaves an efficient construction that supports opaque updates as an important open problem. As we will see, our zero-knowledge definition supports opaque updates in the three party model, which is also satisfied by our constructions.

Verifiable databases with efficient Updates (VDB), formalized in [6] is a primitive where a client with limited resources wants to store a large database on a server so that she can later retrieve a database record, and update a record by assigning a new value to it efficiently. The security of the scheme guarantees that the server should not be able to tamper with any record of the database without being detected by the client. Another related line of work on *dynamic searchable symmetric encryption* (D-SSE) [35, 46] allows a client to encrypt its data and outsource in such a way that this data can still be searched and data can be added and deleted securely. VDB and D-SSE do not allow for public verifiability except for the VC based construction for VDB proposed in [18].

We compare privacy properties and the asymptotic complexity of our constructions with the existing static constructions ([52, 34, 20, 15, 51, 48, 37, 28]) and updatable construction [49] in Table 1. In Table 2, we compare our constructions with [49] based on the dynamic operations. We show that we are the only construction that supports fully dynamic zero-knowledge updates (inserts and deletes) and zero-knowledge queries (order and positive membership) with near optimal proof size and complexities for all three parties. In particular, the time and space complexities for setup and verification and space complexity of query are optimal.

Table 1: Comparison of the efficiency of our construction with existing static and dynamic constructions that support privacy-preserving queries in the three party model. All the time and space complexities are asymptotic. Notation: n is the list size, m is the query size, k is the security parameter. W.l.o.g. we assume list elements are k bit long. Following the standard convention, we omit a multiplicative factor of $O(k)$ for element size in every cell. 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) [**non-standard**]; Division Intractible Hash Function (DIHF); n-Bilinear Diffie Hellman Inversion Assumption (nBDHI).

	[52]	[34]	[20]	[15]	[51]	[48]	[37]	[49]	[28]	DPPAL/ SE-DPPAL
Zero-knowledge Query				✓	✓	✓		✓	✓	✓
Setup time	$n \log n$	n	n	n^2	n^2	n	n	n	n	n
Storage Space	n	n	n	n^2	n^2	n	n^2	n	n	n
Order Query time	m	$n \log n$	n	mn	m	n	n		$\min(m \log n, n)$	$\min(m \log n, n)$
(Positive) Member Query time	m	$n \log n$	n	mn	m	n	n	m	$\min(m \log n, n)$	$\min(m \log n, n)$
Order Verification time	$m \log n \log m$	$m \log n$	n^2	m^2	m^2	m	m		m	m
(Positive) Member Verification time	$m \log n \log m$	$m \log n$	n^2	m^2	m^2	m	m	m	m	m
Proof size	m	$m \log n$	n	m^2	m^2	m	n	m	m	m
Assumption	RSA	RSA	SRSA, Division	EUCMA	ROM, nEAE	**AnAHF**	ROM RSA	DIHF SRSA	ROM, nBDHI	ROM nBDHI

Table 2: Comparison of the efficiency of the dynamic operations of our construction with an existing updatable construction that supports privacy-preserving queries in the three party model. All the time and space complexities are asymptotic. Notation: n is the list size, L is the number of insertions/deletions in a batch, M is the number of distinct elements that have been queried since the last update (insertion/deletion), k is the security parameter. W.l.o.g. we assume list elements are k bit long. Following the standard convention, we omit a multiplicative factor of $O(k)$ for element size in every cell. Assumptions: Strong RSA Assumption (SRSA); Random Oracle Model (ROM); Division Intractible Hash Function (DIHF); n-Bilinear Diffie Hellman Inversion Assumption (nBDHI).

	[49]	DPPAL	SE-DPPAL
Zero-knowledge Update		✓	✓
Transparent Update	✓		
Owner's state size	n	n	1
Storage Space	n	n	n
Insertion time	L	$L+M$	$L\log n+M$
Deletion Time		$L+M$	$L\log n+M$
Assumption	DIHF SRSA	ROM, nBDHI	ROM, nBDHI

3 Preliminaries

3.1 Terminology and Cryptographic Primitives

Let $k \in \mathbb{N}$ be a security parameter. A function $\nu : \mathbb{N} \to \mathbb{R}$ is called negligible if it approaches zero faster than the reciprocal of any polynomial, i.e., $\forall c \in \mathbb{N}$, $\exists k_c \in \mathbb{Z}$, s.t. $\nu(k) \leq k^{-c}$ for all $k \geq k_c$. Then the success probability of the adversary is "too small to matter" if it is negligible in k. We consider an adversary $\mathcal{A}$ which is a probabilistic polynomial time (PPT) Turing Machine running in time polynomial in the security parameter of the scheme, i.e., $\mathsf{poly(k)}$. Let p be a large k-bit prime and $n = \mathsf{poly(k)}$. G and G_1 are multiplicative groups of prime order p. A bilinear map $e : G \times G \to G_1$ is a map with the following properties:

1. $\forall u, v \in G$ and $\forall a, b \in \mathbb{Z}$, $e(u^a, v^b) = e(u, v)^{ab}$;
2. Non-degeneracy: $e(g, g) \neq 1$ where g is a generator of G.

Bilinear Aggregate Signature Scheme [13]: 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.

Definition 3.1 (P-Bilinear Diffie Hellman Inversion assumption [12]) *Let s be a random element of $\mathbb{Z}_p^*$ and P be a positive integer. Then, for every PPT adversary $\mathcal{A}$ there exists a negligible function $\nu(.)$ such that:*
$Pr[s \xleftarrow{\$} \mathbb{Z}_p^*; y \leftarrow \mathcal{A}(\langle g, g^s, g^{s^2}, \ldots, g^{s^P} \rangle) : y = e(g, g)^{\frac{1}{s}}] \leq \nu(k)$.

3.2 Order Maintenance Problem

The *order maintenance problem* is the problem of maintaining a non-empty list $\mathcal{L}$ of records under a sequence of the following three types of operations:

- Insert(x, y): Insert record y after record x in the list. The record y must not already be in the list.
- Delete(x): Delete record x from the list.
- Order(x, y): Return true if x is before y in the list, otherwise return false.

Any such data structure is called *order data structure*(OD). OD has been a widely studied problem in data structure literature [23, 54, 25] and has several interesting applications. *Online list labeling* problem [33, 26, 24, 10, 7, 5] (also known as *file maintenance problem* [56, 57, 58]) is a special case of order maintenance problem. In online list labeling, mapping from a dynamic set of n elements is to be maintained to the integers in the universe $U = [1, N)$ such that the order of the elements respect the order of U. The integers, that the elements are mapped to, are called *tags*. The requirement of the mapping is to match the order of the tags with the order of the corresponding elements. The problem, which was introduced by Itai, Konheim and Rodeh [33] has several interesting applications including cache-oblivious data structures [14, 9, 8] and distributed resource allocation [27].

We will use the order data structure construction presented in [7] and will denote it as OD. We will briefly describe OD and summarize its performance in the subsequent sections.

Order Data structure OD(n) Let $U = [1, N]$ be the tag universe size and n be the number of elements in the dynamic set to be mapped to tags from U, where N is a function of n and is set to be a power of two. Then we consider a complete binary tree on the tags of U, where each leaf represents a tag form the universe. Note that, this binary tree is *implicit*, it is never explicitly constructed, but it is useful for the description and analysis.

At any state of the algorithm, n of the leaves are occupied, i.e., the tags used to label list elements. Each internal node corresponds to a (possible empty) sublist of the list, namely, the elements that have the tags corresponding to the leaves below that node. The *density* of a node is the fraction of its descendant leaves that are occupied. Then *overflow threshold* for the density of a node is defined as follows. Let α be a constant between 1 and 2. For a range of size 2^0 (leaf), the overflow threshold τ_0 is set to 1. Otherwise, for a range of size 2^i, $\tau_i = \frac{\tau_{i-1}}{\alpha} = \alpha^{-i}$. A range is in *overflow* if its density is above its overflow threshold.

Now we are ready to describe the algorithm.

insertafter(x, y)**:** To insert an element y after x, do the following:

- Examine the enclosing tag ranges of x.
- Calculate the density of a tag range by traversing the elements within the tag range.
- Relabel the smallest enclosing tag range that is not overflowing.
- Return the relabelled tags and the tag of y.

delete(x): Delete element x from the list and mark the corresponding tag as unoccupied.

tag(x): Returns the tag of element x.

order(x, y): Returns true if tag$(x) <$ tag(y), and false, otherwise.

N is set to $(2n)^{\frac{1}{1-\log\alpha}}$ so that, the algorithm can proceed as long as the number of elements in the list is between $n/2$ and $2n$. Hence, the algorithm needs $\frac{\log n}{1-\log\alpha}$ bits to represent a tag. If at any point, the number of elements fall below $n/2$ or exceed $2n$, the data structure is rebuilt for the new value of n. The rebuild introduces a constant amortized overhead.

The amortized cost of insertion in this algorithm is $O(\log n)$. By adding one level of indirection, the update time can be improved to $O(1)$, amortized, while still keeping the query time $O(1)$

worst case. Briefly, the technique proceeds as follows: the list is represented as a list of $n/\log(n+1)$ sublists, with each sublist of size $\log(n+1)$. Within each sublist, elements are assigned monotonically increasing tags. When the algorithm inserts into a sublist, if the length of the sublist, say l, becomes at least $2\log(n+1)$, then the sublist is split into $\lfloor l/\log(n+1) \rfloor$ sublists, each of size at least $\log(n+1)$. Then each sublist is inserted into the original list of sublists. For details, please refer to [25, 7].

Theorem 3.1 *[7] The data structure* $\mathsf{OD}(n)$

- *uses* $O(\log n)$ *bits per tag and needs to keep track of* $N = O(n)$ *tags at most and hence uses* $O(n \log n)$ *bits for tags;*
- *performs in* $O(1)$ *amortized insertion and deletion time and* $O(1)$ *worst case query time;*
- *requires* $O(n)$ *space.*

4 Dynamic Privacy Preserving Authenticated Data Structure (DP-PADS)

An Abstract Data Type (ADT) is a data structure (DS) $\mathcal{D}$ with two types of operations defined on it: immutable operations $Q()$ and mutable operations $U()$. $Q(\mathcal{D}, \delta)$ takes as input a query δ on the elements of $\mathcal{D}$ and returns the answer and it does not alter $\mathcal{D}$. $U(\mathcal{D}, u)$ takes as input an update request u (e.g., insert or delete), changes $\mathcal{D}$ accordingly, and outputs the modified data structure, $\mathcal{D}'$.

We present a three party model where a trusted owner generates an instantiation of an ADT, denoted as $(\mathcal{D}, Q, U)$, and outsources it to an untrusted server along with some auxiliary information. The owner also publicly releases a short digest of $\mathcal{D}$. The curious (potentially malicious) client(s) issues queries on the elements of $\mathcal{D}$ and gets answers and proofs from the server, where the proofs are zero-knowledge, i.e., they reveal nothing beyond the query answer. The client can use the proofs and the digest to verify query answers. Additionally, the owner can insert, delete or update elements in $\mathcal{D}$ and update the public digest and the auxiliary information that the server holds. We also require the updates to be zero-knowledge, i.e., an updated digest should be indistinguishable from a new digest generated for the unchanged $\mathcal{D}$.

4.1 Model

DPPADS is a tuple of six probabilistic polynomial time algorithms ($\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{UpdateOwner}, \mathsf{UpdateServer}, \mathsf{Query}, \mathsf{Verify}$). We first describe how these algorithms are used between the three parties of our model and then give their API.

The owner uses KeyGen to generate the necessary keys. He then runs Setup to prepare $\mathcal{D}_0$ for outsourcing it to the server and to compute digests for the client and the server. The owner can update his data structure and make corresponding changes to digests using $\mathsf{UpdateOwner}$. Since the data structure and the digest of the server need to be updated on the server as well, the owner generates an update string that is enough for the server to make the update herself using $\mathsf{UpdateServer}$. The client can query the data structure by sending queries to the server. For a query δ, the server runs Query and generates answer. Using her digest, she also prepares a proof of the answer. The client then uses Verify to verify the query answer against proof and the digest he has received from the owner after the last update.

$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^k)$ where 1^k is the security parameter. KeyGen outputs a secret key (for the owner) and the corresponding public key pk.

$(\mathsf{state}_O, \mathsf{digest}_C^0, \mathsf{digest}_S^0) \leftarrow \mathsf{Setup}(\mathsf{sk}, \mathsf{pk}, \mathcal{D}_0)$ where $\mathcal{D}_0$ is the initial data structure. Setup outputs the internal state information for the owner state_O, digests digest_C^0 and digest_S^0 for the client and the server, respectively.

$(\mathsf{state}_O, \mathsf{digest}_C^{t+1}, \mathsf{Upd}_{t+1}, \mathcal{D}_{t+1}, u_t) \leftarrow \mathsf{UpdateOwner}(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^t, \mathsf{digest}_S^t, \mathcal{D}_t, u_t, \mathsf{SID}_t)$ where u_t is an update operation to be performed on $\mathcal{D}_t$. SID_t is set to the output of a function f on the queries invoked since the last update (Setup for the 0^{th} update). UpdateOwner returns the updated internal state information state_O, the updated public/client digest digest_C^{t+1}, update string Upd_{t+1} that is used to update digest_S^t and the updated $\mathcal{D}_{t+1} := U(\mathcal{D}_t, u_t)$.

$(\mathsf{digest}_S^{t+1}, \mathcal{D}_{t+1}) \leftarrow \mathsf{UpdateServer}(\mathsf{digest}_S^t, \mathsf{Upd}_{t+1}, \mathcal{D}_t, u_t)$ where Upd_{t+1} is used to update digest_S^t to digest_S^{t+1} and u_t is used to update $\mathcal{D}_t$ to $\mathcal{D}_{t+1}$.

$(\mathsf{answer}, \mathsf{proof}) \leftarrow \mathsf{Query}(\mathsf{digest}_S^t, \mathcal{D}_t, \delta)$ where δ is a query on elements of $\mathcal{D}_t$, answer is the query answer, and proof is the proof of the answer.

$b \leftarrow \mathsf{Verify}(\mathsf{pk}, \mathsf{digest}_C^t, \delta, \mathsf{answer}, \mathsf{proof})$ where input arguments are as defined above. The output bit b is set to accept if $\mathsf{answer} = Q(\mathcal{D}_t, \delta)$, and reject, otherwise.

We leave function f undefined and to be specified by a particular instantiation. Once selected it remains fixed for the instantiation. This definition gives the flexibility to define the UpdateOwner algorithm as *session dependent* or *session independent*. If UpdateOwner uses information from the queries since the last update (we call it last session) then it is *session dependent*. For example, given queries $q_i, \ldots, q_j$, $i \leq j$, f could be implemented as an identity function or return the cardinality of its input, $j - i + 1$. If f's output is independent of the queries, then UpdateOwner is *session independent*. Since the function is public, anybody, who has access to the (authentic) queries since the last update, can compute it.

Our model also supports the execution of a batch of updates as a single operation, which may be used to optimize overall performance (Section 5.3).

4.2 Security Properties

A DPPADS has three security properties: completeness, soundness and zero-knowledge.

Completeness dictates that if all three parties are honest, then for an instantiation of any ADT, the client will always accept an answer to his query from the server. Here honest behavior implies that whenever the owner updates the data structure and its public digest, the server updates the DS and her digest accordingly and replies client's queries faithfully w.r.t. the latest DS and digest.

Definition 4.1 (Completeness) *For an ADT $(\mathcal{D}_0, Q, U)$, any sequence of updates $u_0, u_1, \ldots, u_L$ on the data structure $\mathcal{D}_0$, and for all queries δ on $\mathcal{D}_L$:*

$$\Pr[(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^k); (\mathsf{state}_O, \mathsf{digest}_C^0, \mathsf{digest}_S^0) \leftarrow \mathsf{Setup}(\mathsf{sk}, \mathsf{pk}, \mathcal{D}_0);$$

$$\{(\mathsf{state}_O, \mathsf{digest}_C^{t+1}, \mathsf{Upd}_{t+1}, \mathcal{D}_{t+1}, u_t) \leftarrow \mathsf{UpdateOwner}(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^t, \mathsf{digest}_S^t, \mathcal{D}_t, u_t, \mathsf{SID}_t);$$
$$(\mathsf{digest}_S^{t+1}, \mathcal{D}_{t+1}) \leftarrow \mathsf{UpdateServer}(\mathsf{digest}_S^t, \mathsf{Upd}_{t+1}, \mathcal{D}_t, u_t);\}_{0 \leq t \leq L}$$

$$(\mathsf{answer}, \mathsf{proof}) \leftarrow \mathsf{Query}(\mathsf{digest}_S^L, \mathcal{D}_L, \delta) :$$
$$\mathsf{Verify}(\mathsf{pk}, \mathsf{digest}_C^L, \delta, \mathsf{answer}, \mathsf{proof}) = \mathsf{accept} \wedge \mathsf{answer} = Q(\mathcal{D}_L, \delta)] = 1.$$

Soundness protects the client against a malicious server. This property ensures that if the server forges the answer to a client's query, then the client will accept the answer with at most negligible probability. The definition considers adversarial server that picks the data structure and adaptively requests updates. After seeing all the replies from the owner, she can pick any point of time (w.r.t. updates) to create a forgery.

The game captures the adversarial behavior of the server. Since, given the server digest, the server can compute answers to queries herself, it is superfluous to give her explicit access to Query algorithm. Therefore, we set input of f to empty and SID to $\perp$, as a consequence, in algorithm UpdateOwner.

Since, given the server digest, the server can compute answers to queries herself, it is superfluous to give Adv explicit access to Query algorithm. Therefore, we set input of f to empty and SID to $\perp$, as a consequence, in algorithm UpdateOwner.

Definition 4.2 (Soundness) *For all PPT adversaries,* Adv *and for all possible valid queries δ on the data structure $\mathcal{D}_j$ of an ADT, there exists a negligible function $\nu(.)$ such that, the probability of winning the following game is negligible:*

Setup Adv *receives* pk *where* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^k)$. *Given* pk, Adv *picks an ADT of its choice,* $(\mathcal{D}_0, Q, U)$ *and receives the server digest* digest_S^0 *for* $\mathcal{D}_0$, *where* $(\mathsf{state}_O, \mathsf{digest}_C^0, \mathsf{digest}_S^0) \leftarrow \mathsf{Setup}(\mathsf{sk}, \mathsf{pk}, \mathcal{D}_0)$.

Query Adv *requests a series of updates* $u_1, u_2, \ldots, u_L$, *where* $L = \mathsf{poly}(\mathsf{k})$, *of its choice. For every update request* Adv *receives an update string. Let* $\mathcal{D}_{i+1}$ *denote the state of the data structure after the* (i)*th update and* Upd_{i+1} *be the corresponding update string received by the adversary, i.e.,* $(\mathsf{state}_O, \mathsf{digest}_C^{i+1}, \mathsf{Upd}_{i+1}, \mathcal{D}_{i+1}, u_i) \leftarrow \mathsf{UpdateOwner}(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^i, \mathsf{digest}_S^i, \mathcal{D}_i, u_i, \mathsf{SID}_i)$ *where* $\mathsf{SID}_i = \perp$.

Response *Finally,* Adv *outputs* $(\mathcal{D}_j, \delta, \mathsf{answer}, \mathsf{proof})$, $0 \leq j \leq L$, *and wins the game if the following holds:*

$$\mathsf{answer} \neq Q(\mathcal{D}_j, \delta) \wedge \mathsf{Verify}(\mathsf{pk}, \mathsf{digest}_C^j, \delta, \mathsf{answer}, \mathsf{proof}) = \mathsf{accept}.$$

Zero-knowledge captures privacy guarantees about the data structure against a curious (malicious) client. Recall that the client receives a proof for every query answer. Periodically he also receives an updated digest, due to the owner making changes to the DS. Informally, (1) the proofs should reveal nothing beyond the query answer, and (2) an updated digest should reveal nothing about update operations performed on the DS. This security property guarantees that the client does not learn which elements were updated, unless he queries for an updated element (deleted or replaced), before and after the update.

The definition of the zero-knowledge property captures the adversarial client's (Adv) view in two games. In the Real game, Adv interacts with the honest owner and the honest server (jointly called challenger), whereas in the Ideal game, it interacts with a simulator, who mimics the behavior of the challenger with oracle access to the source list, i.e., it is allowed to query the list only with client's queries and does not know anything else about the list or the updates.

Adv picks $\mathcal{D}_0$ and adaptively asks for queries and updates on it. Its goal is to determine if it is talking to the real challenger or to the simulator, with non-negligible advantage over a random guess. If Adv fails to distinguish the two, then the simulator, knowing *only* query answers and the fact that some update has occurred, can simulate proofs and update the digest. Hence, the proof units and the updated digests reveal no information (beyond the query answeror that an update has occurred).

Our zero-knowledge definition is close to the opaque update definition in [41] for Updatable Zero-Knowledge Sets where an updated client digest is indistinguishable from a fresh digest, and old proofs are not valid after an update.

We note that here SID need not be used explicitly in the definition, since the challenger and the simulator know all the queries and can compute f themselves.

Definition 4.3 (Zero-Knowledge) *Let* $\mathsf{Real}_{\mathcal{E},\mathsf{Adv}}$ *and* $\mathsf{Ideal}_{\mathcal{E},\mathsf{Adv},\mathsf{Sim}}$ *be defined as follows where, wlog the adversary is assumed to ask only for valid queries and valid update requests.*[1]

Game $\mathsf{Real}_{\mathcal{E},\mathsf{Adv}}(1^k)$:

Setup *The challenger runs* $\mathsf{KeyGen}(1^k)$ *to generate* sk, pk *and sends* pk *to* Adv_1. *Given* pk, Adv_1 *picks an ADT* $(\mathcal{D}_0, Q, U)$ *of its choice and receives* digest_C^0 *corresponding to* $\mathcal{D}_0$ *from the real challenger* $\mathcal{C}$ *who runs* $\mathsf{Setup}(\mathsf{sk}, \mathsf{pk}, \mathcal{D}_0)$ *to generate them.* Adv_1 *saves its state information in* state_A.

Query Adv_2 *has access to* state_A *and requests a series of queries* $\{q_1, q_2, \ldots, q_M\}$, *for* $M = \mathsf{poly}(\mathsf{k})$.

If q_i is an update request: $\mathcal{C}$ *runs* UpdateOwner *algorithm. Let* $\mathcal{D}_t$ *be the most recent data structure and* digest_C^t *be the public digest on it generated by the* UpdateOwner *algorithm.*

$\mathcal{C}$ *returns* digest_C^t *to* Adv_2.

If q_i is a query: $\mathcal{C}$ *runs* Query *algorithm for the query with the most recent data structure and the corresponding digest as its parameter.*

$\mathcal{C}$ *returns* answer *and* proof *to* Adv_2.

Response Adv_2 *outputs a bit* b.

Game $\mathsf{Ideal}_{\mathcal{E},\mathsf{Adv},\mathsf{Sim}}(1^k)$:

Setup *Initially* Sim_1 *generates a public key, i.e.,* $(\mathsf{pk}, \mathsf{state}_S) \leftarrow \mathsf{Sim}_1(1^k)$ *and sends it to* Adv_1. *Given* pk, Adv_1 *picks an ADT* $(\mathcal{D}_0, Q, U)$ *of its choice and receives* digest_C^0 *from the simulator,* Sim_1, *i.e.,* $(\mathsf{digest}_C^0, \mathsf{state}_S) \leftarrow \mathsf{Sim}_1(\mathsf{state}_S)$. Adv_1 *saves its state information in* state_A. *Let* $\mathcal{F}(q)$ *be a function that takes as input a request* q. *If* q *is a query it returns* q, *otherwise it returns* $\perp$.

Query Adv_2, *who has access to* state_A, *requests a series of queries* $\{q_1, q_2, \ldots, q_M\}$, *for* $M = \mathsf{poly}(\mathsf{k})$.

Sim_2 *is given oracle access to the most recent data structure and is allowed to query the data structure oracle only for queries that are queried by* Adv. *Let* $\mathcal{D}_{t-1}$ *denote the state of the data structure at the time of* q_i. *The simulator runs* $(\mathsf{state}_S, a) \leftarrow \mathsf{Sim}_2^{\mathcal{D}_{t-1}}(1^k, \mathsf{state}_S, \mathcal{F}(q_i))$ *and returns answer* a *to* Adv_2 *where:*

If q_i is an update request: $a = \mathsf{digest}_C^t$, *the updated digest.*

If q_i is a query: $a = (\mathsf{answer}, \mathsf{proof})$ *corresponding to the query* q_i.

Response Adv_2 *outputs a bit* b.

A DPPADS $\mathcal{E}$ *is zero-knowledge if there exists a PPT algorithm* $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ *s.t. for all malicious stateful adversaries* $\mathsf{Adv} = (\mathsf{Adv}_1, \mathsf{Adv}_2)$ *there exists a negligible function* $\nu(.)$ *s.t.*

$$|\Pr[\mathsf{Real}_{\mathcal{E},\mathsf{Adv}}(1^k) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{E},\mathsf{Adv},\mathsf{Sim}}(1^k) = 1]| \leq \nu(k).$$

[1] This is not a limiting constraint, as we can easily force this behavior by checking if a query/update is valid in the Real game; and set $\mathcal{F}$ to return an additional Boolean value indicating if the request is valid or not in the Ideal game.

5 Dynamic Privacy-Preserving Authenticated List

We have presented formal definitions of an abstract data structure that supports privacy and integrity in a three party model. In this section we instantiate it with a list (an ordered set of distinct elements) and propose an efficient construction. We refer to it as *dynamic privacy preserving authenticated list* (DPPAL).

Let $\mathcal{L}$ denote a list and $\mathsf{Elements}(\mathcal{L})$ denote the unordered set corresponding to $\mathcal{L}$. We define order queries on the elements of a list as δ. The query answer, answer is the elements of δ rearranged according to their order in $\mathcal{L}$, i.e., $\mathsf{answer} = \pi_{\mathcal{L}}(\delta)$. An update operation on a list can be one of the following:

$\mathsf{linsertafter}(x, y)$: Insert element x after element $y \in \mathcal{L}$. Since no duplication is allowed, x should be a distinct element not in the list, i.e., $x \notin \mathcal{L}$.

$\mathsf{ldelete}(x)$: Delete element x from $\mathcal{L}$.

$\mathsf{lreplace}(x', x)$: Replace element $x' \in \mathcal{L}$ in the list with element $x \notin \mathcal{L}$.

5.1 Static Construction

The work of [28] proposes a Privacy-Preserving Authenticated List (PPAL). This construction is static but can efficiently answer positive membership and order queries on a list. At a high level, the construction of PPAL works as follows: every element of the static list is associated with a member witness that encodes the rank of the element (using a component of the bilinear accumulator public key) "blinded" with randomness. Every pair of element and its member witness is signed by the owner and the signatures are aggregated using bilinear aggregate signature scheme (see Section 3.1) to generate the public list digest. The client and the server receive the list digest, while the server also receives the signatures, member witnesses and the randomness used for blinding. Given a query from the client on a sublist of the source list, the server returns this sublist ordered as it is in the list with a corresponding proof of membership and order. The server proves membership of every element in the query using the homomorphic nature of bilinear aggregate signature, that is, without owner's involvement. The server then uses the randomness and the bilinear accumulator public key to compute the order witness. The order witness encodes the distance between two elements, i.e., the difference between element ranks, without revealing anything about it.

Although this construction is very efficient for static lists, in practice, data structures are dynamic. A trivial way of making [28] handle, for example, insertion would require regeneration of member witnesses and signatures following the new element, since the ranks of these elements change. Hence an insertion would take time $O(n)$ and grow proportionally to the list size. Deletion of an element would also require $O(n)$ time. In comparison, our dynamic construction below achieves $O(1)$ amortized time for every update operation.

5.2 Dynamic Construction

We use the order labeling data structure $\mathsf{OD}(n)$ from Section 3.2 to maintain the underlying list $\mathcal{L}$. $\mathsf{OD}(n)$ lets us use tags for the elements (instead of their ranks) to maintain order, thus enabling efficient updates. Our construction consists of instantiating algorithms of DPPADS: Setup, UpdateOwner, UpdateServer, Query and Verify. We describe each algorithm in this section and give pseudo-code of KeyGen in Algorithm 1 Setup in Algorithm 2, UpdateOwner in Algorithm 4, UpdateServer in Algorithm 6, Query in Algorithm 7, and Verify in Algorithm 8. Note that build in Algorithm 3 and refresh in Algorithm 5 are subroutines that are called within Setup and UpdateOwner algorithms. Algorithms 1–8 use the following notation. $\mathcal{H} : \{0,1\}^* \to G$: full domain hash function

(instantiated with a cryptographic hash function); all arithmetic operations are performed using mod p. System parameters are $(p, G, G_1, e, g, \mathcal{H})$, where p, G, G_1, e, g are defined in Section 3. $\mathcal{L}_0$ is the input list of size $n = \mathsf{poly}(\mathsf{k})$, where x_i's are distinct. $\mathsf{OD}(n)$ is used to generate the tags for the list elements and supports insertafter, delete, and tag operations.

KeyGen and Setup Phase The owner executes KeyGen (Algorithm 1) and Setup (Algorithm 2) to prepare the keys and digests before outsourcing his list $\mathcal{L}_0$ to the server. The owner randomly picks $s, v \in \mathbb{Z}_p^*$ and $\omega \xleftarrow{\$} \{0,1\}^*$ as part of his secret key. He then inserts the elements of $\mathcal{L}_0$ in an empty order data structure $\mathsf{O} := \mathsf{OD}(n)$ respecting their order in $\mathcal{L}_0$ and generating tag for each element. Hence, the order induced by the tags of the elements is the list order. For every element $x_i \in \mathcal{L}_0$, the owner generates fresh randomness to blind $\mathsf{tag}(x_i)$: $r_i \xleftarrow{\$} \mathbb{Z}_p^*$; he computes member witness as $t_{x_i \in \mathcal{L}_0} \leftarrow g^c$, where g is a generator of G and $c = s^{\mathsf{tag}(x_i)} r_i$, and a signature $\sigma_{x_i} \leftarrow \mathcal{H}(t_{x_i \in \mathcal{L}_t} || x_i)^v$, where $\mathcal{H}$ is a full domain hash function. Using the property of bilinear aggregate signatures (see Section 3), he computes a list signature $\sigma_{\mathcal{L}_0}$, blinded by value $\mathsf{salt} = (\mathcal{H}(\omega))^v$. The owner sends $\sigma_{\mathcal{L}_0}$ to the client as client digest digest_C^0 and $\mathcal{L}_0$ to the server. He also sends the server a digest digest_S^0 which contains $\mathsf{tag}(x_i)$, r_i, $t_{x_i \in \mathcal{L}_0}$ and σ_{x_i} for every element $x_i \in \mathcal{L}_0$, and $g^{s^j}, \forall j \in [0, n]$. The owner saves $\mathcal{L}_0, \mathsf{O}, \mathsf{digest}_S^0$ in his state variable state_O. We note that the randomness r_i allows the owner to protect the tag of the element when it is encoded in the member witness. Similarly, salt is used to hide the size of the list from the client when he happens to query all the elements in the list (i.e., if salt was not used).

Algorithm 1 $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathbf{KeyGen}(1^k)$, where 1^k is the security parameter.

1: Generate the secret key for the owner $\mathsf{sk} = \langle s \xleftarrow{\$} \mathbb{Z}_p^*, v \xleftarrow{\$} \mathbb{Z}_p^*, \omega \xleftarrow{\$} \{0,1\}^* \rangle$ % *ω is the nonce used to for a particular list* and the public key $\mathsf{pk} = g^v$.
2: **return** $(\mathsf{sk}, \mathsf{pk})$

Update Phase UpdateOwner (Algorithm 4) lets the owner perform update u_t on his outsourced data structure and propagate the update in the digests. The owner uses O to efficiently compute the new tag of an element and update the tags of the elements affected by the update. Let $\mathcal{Y}$ be a set of elements that were updated due to linsertafter (insertafter in O, where $\mathcal{Y}$ is of amortized size $O(1)$). Since the member-witness and a signature of every element depends on its tag, UpdateOwner needs to update the authentication units corresponding to elements in $\mathcal{Y}$ and a new element x_{new}, in case u_t was linsertafter or lreplace. This step is equivalent to the steps in Setup for generating authentication units but for elements in $\mathcal{Y}$ and element x_{new}. Finally, the owner updates the list digest signature $\sigma_{\mathcal{L}_t}$ as follows: (1) replace signatures on the elements that have changed (i.e., elements in $\mathcal{Y}$) in case of linsertafter; (2) add a signature for x_{new} in case of linsertafter and lreplace; (3) remove the signature of the old element in case of ldelete or lreplace.

Algorithm 2 $(\mathsf{state}_O, \mathsf{digest}_C^0, \mathsf{digest}_S^0) \leftarrow \textbf{Setup}(\mathsf{sk}, \mathsf{pk}, \mathcal{L}_0)$, where $\mathcal{L}_0 = \{x_1, \ldots, x_n\}$ and x_i's are distinct, for $n = \mathsf{poly}(\mathsf{k})$. sk, pk are the keys generated by KeyGen. Setup is executed by the owner to prepare the digests before outsourcing his list $\mathcal{L}_0$ to the server.

1: Set the internal state variable $\mathsf{state}_O := \langle \mathcal{L}_0, \perp, \perp, \perp \rangle$.
2: $\mathsf{salt} \leftarrow (\mathcal{H}(\omega))^v$ where ω is a nonce from sk.
   % salt *is treated as a list identifier that protects against mix-and-match attacks and from revealing that the queried elements represent the complete list.*
3: % *Generate auxiliary data structure and authenticated information.*
   $(\sigma_{\mathcal{L}_0}, \mathsf{O}, \Sigma_{\mathcal{L}_0}, \Omega_{\mathcal{L}_0}) \leftarrow \mathsf{build}(\mathsf{sk}, \mathsf{state}_O, \mathcal{L}_0)$
4: $\mathsf{state}_O := \langle \mathcal{L}_0, \mathsf{O}, \forall x_i \in \mathcal{L}_0 : (t_{x_i \in \mathcal{L}_0}, \sigma_{x_i}, r_i) \rangle$
5: $\mathsf{digest}_C^0 := \sigma_{\mathcal{L}_0}$
6: $\mathsf{digest}_S^0 := \langle \mathsf{pk}, \sigma_{\mathcal{L}_0}, \langle g, g^s, g^{s^2}, \ldots, g^{s^n} \rangle, \Sigma_{\mathcal{L}_0}, \Omega_{\mathcal{L}_0} \rangle$
7: **return** $(\mathsf{state}_O, \mathsf{digest}_C^0, \mathsf{digest}_S^0)$

Algorithm 3 $(\sigma_{\mathcal{L}_t}, \mathsf{O}, \Sigma_{\mathcal{L}_t}, \Omega_{\mathcal{L}_t}) \leftarrow \mathsf{build}(\mathsf{sk}, \mathsf{state}_O, \mathcal{L}_t)$ where sk contains v and ω and $\mathcal{L}_t = \{x_1, \ldots, x_{n'}\}$.
build is run by the owner to generate tags, member-witnesses and individual signatures for the elements in $\mathcal{L}_t$ and to compute the list digest signature $\sigma_{\mathcal{L}_t}$.

1: % *Build the order labeling data structure* O *to generate* $\mathsf{tag}(x_i)$ $\forall x_i \in \mathcal{L}_t$.
   $\mathsf{O} := \mathsf{OD}(n')$ where $|\mathcal{L}_t| = n'$
2: **For every** $i < i \leq n'$:
3: $\quad \mathsf{O}.\mathsf{insertafter}(x_{i-1}, x_i)$.
4: **For every** $x_i \in \mathcal{L}_t$:
5: $\quad$ Pick $r_i \xleftarrow{\$} \mathbb{Z}_p^*$; compute member witness as $t_{x_i \in \mathcal{L}_0} \leftarrow g^c$, where $c = s^{\mathsf{tag}(x_i)} r_i$; compute signature $\sigma_{x_i} \leftarrow \mathcal{H}(t_{x_i \in \mathcal{L}_t} || x_i)^v$.
6: Compute list digest signature $\sigma_{\mathcal{L}_t} \leftarrow \mathsf{salt} \times \prod_{x_i \in \mathcal{L}_t} \sigma_{x_i}$, where $\mathsf{salt} = (\mathcal{H}(\omega))^v$.
7: $\Sigma_{\mathcal{L}_t} := \langle \forall x_i \in \mathcal{L}_t : (t_{x_i \in \mathcal{L}_t}, \sigma_{x_i}), \mathcal{H}(\omega) \rangle$
8: $\Omega_{\mathcal{L}_t} := \langle \forall x_i \in \mathcal{L}_t : (r_i, \mathsf{tag}(x_i)) \rangle$
9: **return** $(\sigma_{\mathcal{L}_t}, \mathsf{O}, \Sigma_{\mathcal{L}_t}, \Omega_{\mathcal{L}_t})$

Algorithm 4 $(\mathsf{state}_O, \mathsf{digest}_C^{t+1}, \mathsf{Upd}_{t+1}, \mathcal{L}_{t+1}, u_t) \leftarrow$ **UpdateOwner**$(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^t, \mathsf{digest}_S^t, \mathcal{L}_t, u_t, \mathsf{SID}_t)$, where sk and state_O are as defined before; digest_C^t and digest_S^t are the client and the server digests corresponding to $\mathcal{L}_t$, respectively; $\mathcal{L}_t$ is the list after $(t-1)$th update, u_t is the update request (either $\mathsf{linsertafter}, \mathsf{ldelete}$ or $\mathsf{lreplace}$); and SID_t contains all the elements that were accessed by queries since update operation u_{t-1}.
UpdateOwner is executed by the owner to update the outsourced list $\mathcal{L}_t$ with operation u_t and to refresh any elements that were accessed by the client since the last update.

$\mathcal{L}_{t+1} := U(\mathcal{L}_t, u_t)$ % *Update the list.*
If $n/2 \leq |\mathcal{L}_{t+1}| \leq 2n$, then:
  Initialize $\mathcal{Y} := \{\}$ % *Elements to refresh.*
  Initialize$\sigma_{\mathsf{tmp}} := 1$ % *Accumulates changes to list signature.*
  Initialize$x_{\mathsf{new}} := \perp$ % *New element to add to list.*
  If $u_t = \mathsf{linsertafter}(x, y)$:
    $\mathcal{Y} \leftarrow O.\mathsf{insertafter}(x, y)$ % *Elements whose tags changed after insertion.*
    $x_{\mathsf{new}} \leftarrow x$
  Else if $u_t = \mathsf{lreplace}(x', x)$: % *Replace x' with x, where $x \notin \mathcal{L}_t$.*
    Replace x' with x in O.
    $\sigma_{\mathsf{tmp}} \leftarrow \sigma_{x'}^{-1}$ % *Remove a signature of the old element x'.*
    $x_{\mathsf{new}} \leftarrow x$
  Else if $u_t = \mathsf{ldelete}(z)$ % *Delete z, its signature and auth. info.*
    $O.\mathsf{delete}(z)$
    $\sigma_{\mathsf{tmp}} \leftarrow \sigma_z^{-1}(g^{vr'})$, where $r' \xleftarrow{\$} \mathbb{Z}_p^*$
    $\Sigma_{\mathsf{Upd}}(+) := \langle\rangle$ and $\Sigma_{\mathsf{Upd}}(-) := \langle(t_{z \in \mathcal{L}_t}, \sigma_z)\rangle$ % $\Sigma_{\mathsf{Upd}}(+), \Omega_{\mathsf{Upd}}(+)$ *contain information of elements to be added/replaced*
    $\Omega_{\mathsf{Upd}}(+) := \langle(r', \perp)\rangle$ and $\Omega_{\mathsf{Upd}}(-) := \langle(r_z, \mathsf{tag}(z))\rangle$. % $\Sigma_{\mathsf{Upd}}(-), \Omega_{\mathsf{Upd}}(-)$ *contain information of elements to be deleted*
  If $x_{\mathsf{new}} \neq \perp$ % *Generate auth. info. for new element.*
    $r \xleftarrow{\$} \mathbb{Z}_p^*$
    Generate member witness $t_{x_{\mathsf{new}} \in \mathcal{L}_{t+1}} \leftarrow (g^{s^{\mathsf{tag}(x_{\mathsf{new}})}})^r$.
    Compute signature $\sigma_{x_{\mathsf{new}}} \leftarrow \mathcal{H}(t_{x_{\mathsf{new}} \in \mathcal{L}_{t+1}} || x_{\mathsf{new}})^v$.
    $\sigma_{\mathsf{tmp}} \leftarrow \sigma_{\mathsf{tmp}} \sigma_{x_{\mathsf{new}}}$ % *Add a signature of new element.*
    $\Sigma_{\mathsf{Upd}}(+) := \langle(t_{x_{\mathsf{new}} \in \mathcal{L}_{t+1}}, \sigma_{x_{\mathsf{new}}})\rangle$ and $\Sigma_{\mathsf{Upd}}(-) := \langle\rangle$.
    $\Omega_{\mathsf{Upd}}(+) := \langle(r_{x_{\mathsf{new}}}, \mathsf{tag}(x_{\mathsf{new}}))\rangle$ and $\Omega_{\mathsf{Upd}}(-) := \langle\rangle$.
  $(\sigma_{\mathsf{refresh}}, \forall w \in \mathsf{SID}_t \cup \mathcal{Y} : (r_w, \sigma_w)) \leftarrow \mathsf{refresh}(\mathsf{sk}, \mathsf{state}_O, \mathsf{SID}_t \cup \mathcal{Y})$
  $\Sigma_{\mathsf{Upd}}(+) := \Sigma_{\mathsf{Upd}}(+) \cup \langle \forall w \in \mathsf{SID}_t \cup \mathcal{Y} : (t_{w \in \mathcal{L}_{t+1}}, \sigma_w)\rangle$
  $\Omega_{\mathsf{Upd}}(+) := \Omega_{\mathsf{Upd}}(+) \cup \langle \forall w \in \mathsf{SID}_t \cup \mathcal{Y} : (r_w, \mathsf{tag}(w))\rangle$
  $\sigma_{\mathcal{L}_{t+1}} \leftarrow \sigma_{\mathcal{L}_t} \sigma_{\mathsf{tmp}} \sigma_{\mathsf{refresh}}$ % *Update signature.*
  $\mathsf{Upd}_{t+1} := \langle \sigma_{\mathcal{L}_{t+1}}, \perp, \langle \Sigma_{\mathsf{Upd}}(+), \Sigma_{\mathsf{Upd}}(-)\rangle, \langle \Omega_{\mathsf{Upd}}(+), \Omega_{\mathsf{Upd}}(-)\rangle\rangle$
Else: % *Update u_t significantly changed list size.*
  $(\sigma_{\mathcal{L}_{t+1}}, \mathsf{O}, \Sigma_{\mathcal{L}_{t+1}}, \Omega_{\mathcal{L}_{t+1}}) \leftarrow \mathsf{build}(\mathsf{sk}, \mathsf{state}_O, \mathcal{L}_{t+1})$ % *Regenerate auth. info.*
  $\Sigma_{\mathsf{Upd}}(+) := \langle \forall w \in \mathcal{L}_{t+1} : (t_{w \in \mathcal{L}_{t+1}}, \sigma_w)\rangle$ and $\Sigma_{\mathsf{Upd}}(-) = \langle\rangle$.
  $\Omega_{\mathsf{Upd}}(+) := \langle \forall w \in \mathcal{L}_{t+1} : (r_i, \mathsf{tag}(w_i))\rangle$ and $\Omega_{\mathsf{Upd}}(-) = \langle\rangle$.
  $\mathsf{Upd}_{t+1} := \{\sigma_{\mathcal{L}_{t+1}}, \langle g, g^s, g^{s^2}, \ldots, g^{s^{n'}}\rangle, \langle \Sigma_{\mathsf{Upd}}(+), \Sigma_{\mathsf{Upd}}(-)\rangle, \langle \Omega_{\mathsf{Upd}}(+), \Omega_{\mathsf{Upd}}(-)\rangle\}$.
$\mathsf{digest}_C^{t+1} := \sigma_{\mathcal{L}_{t+1}}$
$\mathsf{state}_O := \langle \mathcal{L}_{t+1}, \mathsf{O}, \forall x_i \in \mathcal{L}_{t+1} : (t_{x_i \in \mathcal{L}_{t+1}}, \sigma_{x_i}, r_i)\rangle$
return $(\mathcal{L}_{t+1}, \mathsf{digest}_C^{t+1}, u_t, \mathsf{Upd}_{t+1}, \mathsf{state}_O)$

Algorithm 5 $(\sigma_{\mathsf{refresh}}, \forall w \in \mathcal{W} : (r_w, \sigma_w)) \leftarrow \mathsf{refresh}(\mathsf{sk}, \mathsf{state}_O, \mathcal{W})$

refresh is run by the owner to regenerate randomness, member-witnesses and signatures for the elements in the set $\mathcal{W}$.

1: Initialize $\sigma_{\mathsf{refresh}} := 1$
2: For every $w \in \mathcal{W}$:
3: $\quad \sigma_{\mathsf{refresh}} \leftarrow \sigma_{\mathsf{refresh}} \sigma_w^{-1}$ % *Remove old signature of* w.
4: $\quad$ Select fresh randomness $r_w \xleftarrow{\$} \mathbb{Z}_p^*$.
5: $\quad$ Regenerate the member witness: $t_{w \in \mathcal{L}_{t+1}} \leftarrow (g^{s^{\mathsf{tag}(w)}})^{r_w}$.
6: $\quad$ Compute new signature $\sigma_w \leftarrow \mathcal{H}(t_{w \in \mathcal{L}_{t+1}} || w)^v$, where v is part of sk.
7: $\quad \sigma_{\mathsf{refresh}} \leftarrow \sigma_{\mathsf{refresh}} \sigma_w$
8: **return** $(\sigma_{\mathsf{refresh}}, \forall w \in \mathcal{W} : (r_w, \sigma_w))$

As described so far, UpdateOwner updates the data and authenticated information. However, it has a viable leakage channel. Recall that an update operation changes authentication units of elements in u_t and $\mathcal{Y}$. Hence, if the client accesses elements in $\mathcal{Y}$, before and after the update, he will notice that its authentication unit has changed and infer that a new element was inserted nearby. This violates the zero-knowledge property of DPPADS: the client should not learn information about updates to elements he did not query explicitly.

UpdateOwner achieves the zero-knowledge property as follows. We set f to be a function that takes the client queries since the last update and returns a set of elements accessed by them; these are the elements whose authentication units are known to the client. Given these elements in UpdateOwner's input SID_t, the owner can recompute the member-witnesses of each of them using fresh randomness, update their signatures and the list digest using using the subroutine refresh (Algorithm 5). Since the member-witnesses and signatures of the elements in SID_t are changed independently of u_t, seeing refreshed units after the update reveals no information to the client. We define f this way for optimization. In a naive implemention, where f is defined as the a constant function, or where SID_t is not used, the UpdateOwner algorithm has to randomize member-witnesses and signatures for *all* the elements in the list.

Finally, the owner updates state_O and sends u_t and authentication units (updated due to u_t and refresh) in Upd_{t+1} to the server and updated list digest $\sigma_{\mathcal{L}_{t+1}}$ to the client. The server runs UpdateServer (Algorithm 6) to propagate the update using Upd_{t+1} to add/substitute/remove units in her digest and u_t to update the list.

Query Phase Given an order query δ, the server executes the following instantiation of Query (Algorithm 7). It reorders the elements in δ according to their order in $\mathcal{L}_t$, sets answer to $\pi_{\mathcal{L}_t}(\delta)$ and computes proof that consists of units to prove membership and order of elements in δ. For every element $y_i \in \delta$ its member witness $t_{y_j \in \mathcal{L}_t}$ and σ_{y_j} are included in proof. Note that sending $t_{y_j \in \mathcal{L}_t}$ to the client does not reveal its tag since the witness was blinded using secret randomness. The server also has to prove that δ is indeed a part of the source list $\mathcal{L}_t$. She computes the authentication digest for $\mathcal{L}' = \mathcal{L}_t \setminus \delta$, denoted as $\lambda_{\mathcal{L}'}$. For every pair of adjacent elements y_j, y_{j+1} in answer, the server computes an order witness $t_{y_j < y_{j+1}} := (g^{s^d})^{r''/r'}$, where $d = \mathsf{tag}(y_{j+1}) - \mathsf{tag}(y_j)$ and r' and r'' are randomness of y_j and y_{j+1} and g^{s^d} is part of server's digest.

Verification Phase Given (answer, proof), the client uses Verify (Algorithm 8) and his copy of the list digest signature to verify answer. He checks the membership of elements in answer by using the properties of bilinear aggregate signatures. In particular he can verify the relationship of $\mathcal{L}' = \mathcal{L}_t \setminus \delta$

Algorithm 6 $(\mathsf{digest}_S^{t+1}, \mathcal{L}_{t+1}) \leftarrow$ **UpdateServer**$(\mathsf{digest}_S^t, \mathsf{Upd}_{t+1}, \mathcal{L}_t, u_t)$, where u_t is an update to perform on $\mathcal{L}_t$ and Upd_{t+1} contains updates on authentication information. Upon receiving update messages u_t and Upd_{t+1} from the owner, the server executes UpdateServer to propagate the update on her copy of list $\mathcal{L}_t$ and her digest digest_S.

1: Update the list: $\mathcal{L}_{t+1} := U(\mathcal{L}_t, u_t)$ where $|\mathcal{L}_{t+1}| = n'$.
2: Parse Upd_{t+1} as a 4-tuple: $\langle \sigma_{\mathcal{L}_{t+1}}, \mathcal{T}, \Sigma_{\mathsf{Upd}}, \Omega_{\mathsf{Upd}} \rangle$.
3: Compute $\Sigma_{\mathcal{L}_{t+1}}$: add/replace/delete elements from Σ_{Upd} in $\Sigma_{\mathcal{L}_t}$.
4: Compute $\Omega_{\mathcal{L}_{t+1}}$: add/replace/delete elements from Ω_{Upd} in $\Omega_{\mathcal{L}_t}$.
5: **If** $\mathcal{T} \neq \perp$: % *u_t caused regeneration of tags for all elements, hence authenticated information needs to be replaced with new one.*
6: $\quad \mathsf{digest}_S^{t+1} := \langle \mathsf{pk}, \sigma_{\mathcal{L}_{t+1}}, \langle g, g^s, g^{s^2}, \ldots, g^{s^{n'}} \rangle, \Sigma_{\mathcal{L}_{t+1}}, \Sigma_{\mathcal{L}_{t+1}} \rangle$.
7: **Else** % *u_t does not cause regeneration of tags for all elements*
8: $\quad \mathsf{digest}_S^{t+1} := (\mathsf{pk}, \sigma_{\mathcal{L}_{t+1}}, \langle g, g^s, g^{s^2}, \ldots, g^{s^n} \rangle, \Sigma_{\mathcal{L}_{t+1}}, \Omega_{\mathcal{L}_{t+1}})$ % *where g^{s^i}'s are from* digest_S^t
9: **return** $(\mathcal{L}_{t+1}, \mathsf{digest}_S^{t+1})$

Algorithm 7 $(\mathsf{answer}, \mathsf{proof}) \leftarrow$ **Query**$(\mathsf{digest}_S^t, \mathcal{L}_t, \delta)$, where $\delta = (z_1, \ldots, z_m)$, s.t. $z_i \in \mathcal{L}_t$, , is the queried sublist and $\mathcal{L}_t$ is the most recent list. Query is executed by the server to generate answer answer to an order query on the elements of the list, δ, and a proof proof of the answer

1: $\mathsf{answer} = \pi_{\mathcal{L}_t}(\delta) = \{y_1, \ldots, y_m\}$;
2: $\mathsf{proof} = \langle \Sigma_{\mathsf{answer}}, \Omega_{\mathsf{answer}} \rangle$:
3: $\quad \Sigma_{\mathsf{answer}} := \langle \sigma_{\mathsf{answer}}, T, \lambda_{\mathcal{L}'} \rangle$ where $\mathcal{L}' = \mathcal{L}_t \setminus \delta$ and:
4: $\quad \sigma_{\mathsf{answer}} \leftarrow \prod_{y_j \in \mathsf{answer}} \sigma_{y_j}$. % *Digest signature for the query elements.*
5: $\quad\quad T = (t_{y_1 \in \mathcal{L}_t}, \ldots, t_{y_m \in \mathcal{L}_t})$. % *Member witnesses for query elements.*
6: $\quad\quad$ Let $\mathcal{S}$ be a set of random elements without any corresponding tag, that were introduced in $\Omega_{\mathcal{L}_t}$ due to Idelete.
7: $\quad\quad$ The member verification unit: $\lambda_{\mathcal{L}'} \leftarrow \mathcal{H}(\omega) \times g^{\sum_{r \in \mathcal{S}} r} \times \prod_{x \in \mathcal{L}'} \mathcal{H}(t_{x \in \mathcal{L}_t} || x)$ where $\mathcal{H}(\omega)$ comes from digest_S^t.
8: $\quad \Omega_{\mathsf{answer}} = (t_{y_1 < y_2}, t_{y_2 < y_3}, \ldots, t_{y_{m-1} < y_m})$:
9: $\quad\quad$ For every $j \in [1, m-1]$: Let $i' := \mathsf{tag}(y_j)$ and $i'' := \mathsf{tag}(y_{j+1})$, 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 $d = |i' - i''|$.
10: **return** $(\mathsf{answer}, \mathsf{proof})$

by knowing elements in δ and their signatures, authentic list digest signature $\sigma_{\mathcal{L}_t}$ (received from the owner) and server computed authentication digest $\lambda_{\mathcal{L}'}$. We note that the client cannot tell if δ is the whole list or not, because of the blinding factor salt used in computing $\sigma_{\mathcal{L}_t}$. The client then uses bilinear map (Section 3) to verify order witnesses. Recall that it lets him verify algebraic properties of the exponents, i.e., that $d = \mathsf{tag}(y_{j+1}) - \mathsf{tag}(y_j)$ for $t_{y_i \in \mathcal{L}_t} = g^{r' s^{\mathsf{tag}(y_j)}}$, $t_{y_{j+1} \in \mathcal{L}_t} = g^{r'' s^{\mathsf{tag}(y_{j+1})}}$ and $t_{y_j < y_{j+1}}$ as defined above.

5.3 Extensions

Batch updates For simplicity our construction describes the UpdateOwner algorithm for a single update operation u_t. However, UpdateOwner can be easily generalized to batch updates, where a series of updates $u_1, u_2, \ldots, u_m$ happen together. In this case, the subroutine refresh on line 25 in Algorithm 4 needs to be called only once after all updates are performed. In this case, $\mathcal{Y}$ will contain all the elements whose tags were regenerated by any of the updates $u_1, u_2, \ldots, u_m$.

Algorithm 8 $b \leftarrow$ **Verify**$(\mathsf{pk}, \mathsf{digest}_C^t, \delta, \mathsf{answer}, \mathsf{proof})$. Verify is executed by the client to verify the integrity of an answer answer to an order query δ on the elements of the list. the verifier uses proof to verify the answer wrt the owner's public key pk and the most updated list digest digest_C^t.

1: Compute $\xi \leftarrow \prod_{y_j \in \delta} \mathcal{H}(t_{y_j \in \mathcal{L}_t} || y_j)$
2: $e(\sigma_{\mathsf{answer}}, g) \stackrel{?}{=} e(\xi, \mathsf{pk})$ % *Verify the* answer *digest is signed by the owner*
3: $e(\sigma_{\mathcal{L}_t}, g) \stackrel{?}{=} e(\sigma_{\mathsf{answer}}, g) \times e(\lambda_{\mathcal{L}'}, \mathsf{pk})$. % *Verify* answer *is a part of the source list*
4: $\forall j \in [1, m-1]$: $e(t_{y_j \in \mathcal{L}_t}, t_{y_j < y_{j+1}}) \stackrel{?}{=} e(t_{y_{j+1} \in \mathcal{L}_t}, g)$. % *Verify the returned order is correct*
5: If (2), (3), (4) equalities hold, then accept. Else reject.

Hiding occurrence of an update The construction in Section 5.2 satisfies the Zero-Knowledge property of Definition 4.3 (as we show in Section 5.5.3), i.e., the updated list digest does not reveal any information about the update. However, if the DPPAL system is implemented such that, the UpdateOwner algorithm is invoked only when updates to the list occur, then by the mere fact that a list digest has changed, the client learns that an update has occurred, even though he cannot tell which one. This leakage is beyond the scope of Definition 4.3 and, depending on the intended application, can be tolerated. If not, we can hide the information that an update has occurred as follows. Instead of calling UpdateOwner only when an update is required, the owner calls it periodically. He saves update operations such that the next time he executes UpdateOwner he can perform a batch update. If there are no updates scheduled when UpdateOwner has to be invoked, the owner runs UpdateOwner with an empty update operation. In this case, refresh (line 25 in Algorithm 4) is still invoked with elements that were queried since the last time UpdateOwner was executed (information stored in SID_t). Hence, client's digest, member witnesses and the individual signatures of these elements are updated. Since client's digest changes periodically independent of the updates, he cannot distinguish a refresh from an update operation.

5.4 Efficiency Analysis

Our construction uses efficient cryptographic operations: multiplication and exponentiation in prime order groups, and evaluation of a cryptographic hash function and bilinear map. As is standard, we assume they take constant time. Moreover, we use at most four of these operations per element. We also note that a member/order witness and a signature is a group element and is represented using $O(1)$ space[2]. Theorem 5.1 summarizes the security and performance of our construction.

Here we analyze the asymptotic running time and space complexity for each party.

Owner For a list of size n, the Setup algorithm instantiates $\mathsf{OD}(n)$ and makes n insertafter calls to generate tags. The amortized cost of insertion is $O(1)$ from Theorem 3.1. Generating the key pair takes time $O(1)$. The generation of each member-witness and signature takes time $O(1)$ and generating the public list digest signature involves $O(n)$ multiplications. Therefore the overall *setup time* is $O(n)$. The *space* required to store the list and generate the server digest is $O(n)$, since each member witness and signature is a group element (and, hence, of constant size) and from Theorem 3.1, we have, the space required to store $\mathsf{OD}(n)$ is $O(n)$.[3] Therefore, the overall space required for the Setup algorithm is $O(n)$.

We now analyze the time it takes the owner to perform an update in UpdateOwner in Algorithm 4. This algorithm can be split in two parts. In the first part the owner updates the

[2]By standard convention, the word size is $\log(\mathsf{poly}(\mathsf{k}))$ and k is the security parameter.

[3]Following the standard convention, in our analysis, we ignore the element representation which takes $O(\log \mathsf{poly}(\mathsf{k}))$ per element.

list and changes corresponding authenticated information. In the next part he refreshes the authenticated information of all distinct elements that were queried since the last update, i.e., elements in SID.

The amortized cost of a single list update is $O(1)$ (Theorem 3.1). Hence, an update of a batch of L elements takes $O(L)$ time. The authenticated information of elements affected by the update, i.e., elements in $\mathcal{Y}$, is update by the refresh algorithm, which takes $O(1)$ time per element. Since the amortized size of $\mathcal{Y}$ is $O(1)$, the update takes $O(1)$ amortized time.

Let M be the number of elements in SID. For each of these elements, the owner generates new randomness, a member-witness and a signature, where each takes time $O(1)$ by refresh in Algorithm 5. Hence, the update phase requires $O(L + M)$ time for the owner, or $O(1)$ amortized over the total number of elements queried or updated since the last update.

Server The Query algorithm, takes time $O(\min\{m \log n, n\})$ with $O(n)$ preprocessing as in [28] and space $O(n)$, where the list size is between $n/2$ and $2n$.

For every update that the owner makes, the server runs the UpdateServer algorithm. This algorithm requires the server to update the list and update authenticated information (i.e., randomness, member-witnesses, signatures) that has changed due to the changes in the list. The server updates the list in time $O(L)$ where L is the number of elements in a batch update. Updating authenticated information takes time proportional to the size of the update string Upd that the server receives from the owner, i.e., the output of UpdateOwner. Recall that the number of elements to update is $O(L+M)$ where M is the number of elements in SID. Hence, the overall *update time* is $O(L + M)$, or $O(1)$ amortized over the total number of elements queried or updated since the last update.

Client The client requires $O(1)$ space to store digest_C and pk.

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, where m is the size of the query. 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.

5.5 Security Analysis

In order to show that our construction of DPPAL in Section 5.2 is secure we need to show that it satisfies completeness (Definition 4.1), soundness (Definition 4.2) and zero-knowledge (Definition 4.3) properties.

5.5.1 Proof of Completeness

If all the parties are honest, all the equations in Verify (in Algorithm 8) evaluate to true. This is easy to see just by expanding the equations as follows:

Lemma 5.1 *The DPPAL scheme in Section 5 satisfies completeness in Definition 4.1.*

Proof We will expand all the equality checks in the verification algorithm (Algorithm 8).

Equation $e(\sigma_{\mathsf{answer}}, g) \stackrel{?}{=} e(\xi, \mathsf{pk} = g^v)$: Let $\mathsf{answer} = \{y_1, \ldots, y_m\} = \pi_{\mathcal{L}_t}(\delta)$ and $\mathcal{L}' = \mathcal{L}_t \setminus \delta$ then

$$e(\sigma_{\mathsf{answer}}, g) = e(\prod\nolimits_{y \in \mathsf{answer}} \sigma_y, g) = e(\prod\nolimits_{y \in \mathsf{answer}} \mathcal{H}(t_{y \in \mathcal{L}_t} || y)^v, g) =$$
$$e(\prod\nolimits_{y \in \mathsf{answer}} \mathcal{H}(t_{y \in \mathcal{L}_t} || y), g^v) = e(\xi, g^v).$$

Equation $e(\sigma_{\mathcal{L}_t}, g) \stackrel{?}{=} e(\sigma_{\mathsf{answer}}, g) \times e(\lambda_{\mathcal{L}'}, \mathsf{pk} = g^v)$: We start with the right hand side. Let $\mathcal{L}_0$ be the initial list and $u_0, \ldots, u_{t-1}$ be the sequence of updates that resulted in the most recent list, $\mathcal{L}_t$. Let $\mathcal{S}$ be a set of random elements introduced in $\Omega_{\mathcal{L}_t}$ for u_i's that were $\mathsf{Idelete}$ ($\mathcal{S}$ is empty in case of no delete operations).

$$
\begin{aligned}
& e(\sigma_{\mathsf{answer}}, g) \times e(\lambda_{\mathcal{L}'}, g^v) \\
&= e(\prod\nolimits_{y \in \mathsf{answer}} \mathcal{H}(t_{y \in \mathcal{L}_t} || y)^v, g) \times e(\mathcal{H}(\omega) \times g^{\sum_{r \in \mathcal{S}} r} \times \prod\nolimits_{x \in \mathcal{L}'} \mathcal{H}(t_{x \in \mathcal{L}_t} || x), g^v) \\
&= e(\prod\nolimits_{y \in \mathsf{answer}} \mathcal{H}(t_{y \in \mathcal{L}_t} || y), g^v) \times e(\mathcal{H}(\omega) \times g^{\sum_{r \in \mathcal{S}} r} \times \prod\nolimits_{x \in \mathcal{L}'} \mathcal{H}(t_{x \in \mathcal{L}_t} || x), g^v) \\
&= e(\mathcal{H}(\omega)^v \times g^{v \sum_{r \in \mathcal{S}} r} \times \prod\nolimits_{x \in \mathcal{L}_t} \mathcal{H}(t_{x \in \mathcal{L}_t} || x)^v, g) = e(\sigma_{\mathcal{L}_t}, g).
\end{aligned}
$$

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

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

This concludes our the proof of completeness. ∎

5.5.2 Proof of Soundness

Let n be the size of the initial list $\mathcal{L}_0$ that Adv picks. Since Adv is allowed to make $\mathsf{poly}(\mathsf{k})$ number of update requests, the list can grow to the size polynomial in n. Let $P/2$ be the maximum size that the list can grow to. We prove soundness of the DPPAL scheme in Section 5.2 by reduction from the P-Bilinear Diffie Hellman (P-BDHI) assumption (see Definition 3.1 for details).

To the contrary of the Soundness Definition 4.2, let us assume that the malicious server, Adv, requests a series of updates, $u_0, \ldots, u_l$ and then produces a forgery answer on a $\mathcal{L}_i$ where $\mathcal{L}_i = U(U(\ldots(U(\mathcal{L}_0, u_0), u_1)\ldots), u_{i-1})$. That is, Adv produces a non-trivial sublist $\delta = \{x_1, x_2, \ldots, x_m\}$, where $m \geq 2$, such that $\mathsf{answer} \neq \pi_{\mathcal{L}_i}(\delta)$ and corresponding order proof is accepted by the client, i.e., by algorithm Verify. Since $|\delta| > 1$, there exists at least one inversion pair (x_i, x_j) in answer where $i, j \in [1, m]$. Let us assume, wlog, $x_i < x_j$ is the order in $\mathcal{L}_i$. This implies $x_j < x_i$ is the forged order for which Adv has successfully generated a valid proof, i.e., $e(t_{x_j \in \mathcal{L}}, t_{x_j < x_i}) = e(t_{x_i \in \mathcal{L}}, g)$ was verified by Verify since it accepted the corresponding proof. We show we can construct a PPT adversary $\mathcal{A}$ that successfully solves the *P-BDHI Problem* [12] by invoking Adv and using its forged witness $t_{x_j < x_i}$. This in turn contradicts the assumption. Below we present a formal reduction.

Lemma 5.2 *If P-Bilinear Diffie Hellman assumption holds, then DPPAL scheme in Section 5 satisfies soundness in Definition 4.2.*

Proof We construct a PPT adversary $\mathcal{A}$ that successfully solves the *P-BDHI Problem* [12], if he can invoke Adv who can forge a proof that passes Verify in Algorithm 8.

Algorithm $\mathcal{A}$ is given the public parameters $(p, G, G_1, e, g, \mathcal{H})$ and $\mathcal{T} = \langle g, g^s, g^{s^2}, \ldots, g^{s^P} \rangle$, where $P = \mathsf{poly}(\mathsf{k})$.

Setup $\mathcal{A}$ first executes the steps in Algorithm 1 with the following changes. In line 1, $\mathcal{A}$ only picks $\mathsf{sk} = \langle v \stackrel{\$}{\leftarrow} \mathbb{Z}_p^*, \omega \stackrel{\$}{\leftarrow} \{0,1\}^* \rangle$ and sets $\mathsf{pk} = g^v$ and sends pk to Adv. Given pk, let $\mathcal{L}_0$ be the original list picked by Adv and n be its size.

$\mathcal{A}$ proceeds as follows. In order to compute the member-witnesses (as in Algorithm 2), $\mathcal{A}$ uses its input tuple $\mathcal{T}$ to get the $g^{s^{\mathsf{tag}(.)}}$ components. Finally, $\mathcal{A}$ returns $\mathsf{digest}_S^0 := \langle^v, \sigma_{\mathcal{L}_0}, \langle g, g^s, g^{s^2}, \ldots, g^{s^n}\rangle, \Sigma_{\mathcal{L}_0}, \Omega_{\mathcal{L}_0}\rangle$ to Adv, where $n = |\mathcal{L}_0|$.

Query For an update request, u_t on $\mathcal{L}_t$, $\mathcal{A}$ runs $\mathsf{UpdateOwner}$ in Algorithm 4 with the following changes. Since the game is concerned with updates only, and no order query is generates, SID_t is empty and so in line 25, $\mathsf{refresh}$ is called with parameter $\mathcal{Y}$ instead of $\mathsf{SID}_t \cup \mathcal{Y}$. Subsequently, $\mathsf{SID}_t \cup \mathcal{Y}$ is replaced with $\mathcal{Y}$ in steps 26 and 27. Recall that $\mathcal{A}$ uses its input tuple $\mathcal{T}$ to (re-)compute member witnesses. Finally, Upd_{t+1} is returned to Adv.

Response Finally Adv outputs a forged order answer for some non-trivial sublist, $\delta = \{x_1, x_2, \ldots, x_m\}$ on list $\mathcal{L}_j$. As discussed before, let (x_i, x_j) be an inversion pair such that $x_i < x_j$ is the order in $\mathcal{L}_j$ and $\mathsf{tag}(x_i) < \mathsf{tag}(x_j)$. This implies $x_j < x_i$ is the forged order for which Adv has successfully generated a valid proof $t_{x_j<x_i} = \left(g^{s^{(\mathsf{tag}(x_i)-\mathsf{tag}(x_j))}}\right)^{r_{x_i} r_{x_j}^{-1}}$.

Now $\mathcal{A}$ outputs $e(t_{x_j<x_i}, (g^{s^{\mathsf{tag}(x_j)-\mathsf{tag}(x_i)-1}})^{{r_{x_i}}^{-1} r_{x_j}}) = e(g,g)^{\frac{1}{s}}$. $\mathcal{A}$ inherits success probability of Adv, therefore if Adv succeeds with non-negligible advantage, so does $\mathcal{A}$. Hence, a contradiction. ∎

5.5.3 Proof of Zero-Knowledge

Lemma 5.3 *The DPPAL scheme in Section 5 satisfies zero-knowledge in Definition 4.3.*

Proof We define the simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ from Definition 4.3 as follows. Sim has access to the system parameters, $(p, G, G_1, e, g, \mathcal{H})$.

Setup Sim_1 picks $v \xleftarrow{\$} \mathbb{Z}_p^*, \omega \xleftarrow{\$} \{0,1\}^*$ and publishes $\mathsf{pk} = g^v$ and sends it to Adv_1 and keeps $\mathsf{sk} = \langle v, \omega\rangle$ as the secret key. Given pk, Adv_1 picks a list of his choice $\mathcal{L}$. Sim_1 picks a random element $g_1 \xleftarrow{\$} G$ and sends $\mathsf{digest}_C^0 := g_1^v$ to Adv_1.

Query For every query Sim_2 receives $\mathcal{F}(q)$ as input informing it whether q is an update or an order query. In order to simulate consistent answers to the queries, it maintains a table of its previous answers (in the order of their arrival). Sim_2 simulates a reply to each query as follows.

If $\mathcal{F}(q) = \perp$ (i.e., q is an update request):

- Sim_2 picks a random $r \xleftarrow{\$} \mathbb{Z}_p^*$ and inserts $\mathcal{F}(q), r$ in the table.
- Sim_2 computes $\mathsf{digest}_C^{'} \leftarrow (g_1^v)(g^{rv})$ and returns $\mathsf{digest}_C^{'}$.

If $\mathcal{F}(q) = q$ (i.e., q is an order query):

- Let $q = (x_1, \ldots, x_m)$. Sim_2 makes an oracle call to the most recent list, $\mathcal{L}$ to get the current order of the elements in $\mathcal{L}$. Let us call it $\mathsf{answer} = \pi_{\mathcal{L}}(q) = \{y_1, y_2, \ldots, y_m\}$.
- For each $y_i \in \mathsf{answer}$, Sim_2 sets the randomness r_i as follows. It finds the most recent query that included y. If this query was an update Sim_2 uses the random element r corresponding to this row in the table. If the most recent query on y was an order query and there was no update query (on y or other elements) after that, then Sim_2 uses the same randomness as he used last time. Otherwise, Sim_2 picks a fresh random element $r' \xleftarrow{\$} \mathbb{Z}_p^*$ and inserts $\mathcal{F}(q), x, r'$ to the table.
- Sim_2 computes the member witness as $t_{y_i \in \mathcal{L}} := g^{r_i}$ and computes $\sigma_{y_i} \leftarrow \mathcal{H}(t_{y_i \in \mathcal{L}} || y_i)^v$.
- Sim_2 sets $\sigma_{\mathsf{answer}} := \prod_{y_i \in \mathsf{answer}} \sigma_{y_i}$ and $\lambda_{\mathcal{L}'} := g_1 / \prod_{y_i \in \mathsf{answer}} \mathcal{H}(t_{y_i \in \mathcal{L}} || y_i)$.
- For every pair of elements y_i, y_{i+1} in answer, Sim_2 computes $t_{y_i < y_{i+1}} \leftarrow g^{r_{i+1}/r_i}$.
- Finally, Sim_2 returns $(\mathsf{answer}, \mathsf{proof})$, where $\mathsf{proof} = \langle \Sigma_{\mathsf{answer}}, \Omega_{\mathsf{answer}}\rangle$, $\Sigma_{\mathsf{answer}} = \langle \sigma_{\mathsf{answer}}, T, \lambda_{\mathcal{L}'}\rangle$, $T = (t_{y_1 \in \mathcal{L}}, \ldots, t_{y_m \in \mathcal{L}})$ and $\Omega_{\mathsf{answer}} = (t_{y_1 < y_2}, t_{y_2 < y_3}, \ldots, t_{y_{m-1} < y_m})$.

We now argue that Sim produces a sequence of answers that is identically distributed to the one produced by the real challenger (i.e., the owner and the server). We consider each phase. During the Setup Sim_1 produces initial digest digest_C^0 and pk. v is a random element picked exactly as the owner does in Figure 2. The client list digest computed by the owner in Figure 2, has at least one multiplicative random element from the group G and g_1 is picked randomly by Sim_1. Therefore digest_C^0 and pk are distributed identically as in the real experiment. (This follows from a simple argument. 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.)

For an update request, we want to show that the client receives a list digest signature in digest_C that is indistinguishable from a random element in the same space (i.e., that is how Sim_2 performs an update). We argue that a signature after the update is a product of the signature before the update multiplied by at least one random element from the group. If this is the case then the result of this product, i.e., the new signature, is distributed identically to a random element from the same group (following the argument above).

If u is $\mathsf{linsertafter}$ or $\mathsf{lreplace}$, then at least one member witness gets added or refreshed (line 20, Algorithm 4). Then the signature of the corresponding new or updated element is hashed using $\mathcal{H}$ (line 21, Algorithm 4). Then element's signature is used to update the old signature which corresponds to the multiplication by a random value since $\mathcal{H}$ is viewed as a random oracle (line 28, Algorithm 4). If u is $\mathsf{ldelete}$, then the old signature is updated explicitly in the construction by adding fresh randomness (line 15, Algorithm 4).

For an order query, we need to show that all the units of $\mathsf{proof} = \langle \Sigma_{\mathsf{answer}}, \Omega_{\mathsf{answer}} \rangle$ are distributed identically in both games. Notice that all the components of Σ_{answer} and Ω_{answer} have a multiplicative random group element. Sim_2 generates the components as random group elements. Therefore, following the same argument, as for signatures, all the units of $\mathsf{proof} = \langle \Sigma_{\mathsf{answer}}, \Omega_{\mathsf{answer}} \rangle$ are distributed identically in both games.

Now we need to show that Adv's views before and after updates are identical. The signature and member-witness for an element touched by some previous query before an update, and by some query after the update gets refreshed in the real game. Sim_2 also refreshes the signature and member-witness for every element touched by the queries before and after an update. The refresh is done identically in both cases. So in Adv's view, the signature and member-witness before and after an update are identically distributed in both the games. This concludes our proof that the DPPAL scheme presented in Section 5.2 is simulatable and the Zero-Knowledge is perfect. ∎

Theorem 5.1 *The dynamic privacy-preserving authenticated list (DPPAL) construction of Section 5 satisfies the security properties of DPPADS including completeness, soundness (under the P-BDHI assumption [12]) and zero-knowledge in the random oracle model (inherited from [13]). The construction has the following performance, where n is the list size, m is the query size, L is the number of updates in a batch and M is the number of distinct elements that have been queried since the last update:*

- *The owner uses $O(n)$ time and space for setup, and keeps $O(n)$ state;*
- *In the update phase the owner sends a message of size $O(L+M)$ to the server and a message of size $O(1)$ to the client;*
- *The update phase requires $O(L+M)$ time for the owner and the server, or $O(1)$ amortized over the number of elements queried or updated since the last update;*
- *The server uses $O(n)$ space and performs the preprocessing in $O(n)$ time;*
- *The server computes the answer to a query and its proof in time $O(\min\{m \log n, n\})$;*

- *The proof size is $O(m)$;*
- *The client verifies the proof in $O(m)$ time and space.*

6 Space Efficient DPPADS (SE-DPPADS)

The model of Section 4 assumes the owner himself updates his data structure and sends information to the server to propagate the changes. Hence, the owner is required to keep the most recent version of $\mathcal{D}_t$ and any associated auxiliary information. Since this may not be ideal for an owner with small memory requirement, we propose a model that is space efficient and relies on an authenticated data structure (ADS) protocol executed between the owner and the server.

An ADS protocol is incorporated in the space efficient model as follows. The owner keeps a short digest (ideally of $O(1)$ size) of the data structure and its auxiliary information. He then outsources the data structure as well as any auxiliary information to the server. To perform an update u_t, the owner requests the server to update the data structure and to send back any part that has changed due to the update (ideally, proportional to the size of the update). Since the server can be malicious, the owner also requests a proof that the information he receives is authentic, i.e., the update was performed correctly. Then the owner incorporates the update in authenticated information for the server and digest for the client. (Note that the server cannot perform these updates since she does not have the secret key.) The owner updates his local digest in order to verify authenticity of future updates.

6.1 Space Efficient DPPADS (SE-DPPADS) Model

The original model in Section 4 remains the same except for the $\mathsf{UpdateOwner}$ that is split into two algorithms. The first algorithm, $\mathsf{UpdateDS}$, is executed by the server. It updates $\mathcal{D}_t$, extracts auxiliary information and computes the proof for the owner. The second algorithm, $\mathsf{VerifyUpdate}$, is executed by the owner to verify the authenticity of the server's output. If the verification succeeds, the owner generates an update string for the server and a new digest for the client.

We describe $\mathsf{UpdateDS}$ and $\mathsf{VerifyUpdate}$ below.

$(\mathcal{D}_{t+1}, \mathsf{aux}_t, \mathsf{auxproof}_t) \leftarrow \mathbf{UpdateDS}(\mathcal{D}_t, u_t, \mathsf{digest}_S^t)$ where u_t is an update operation to be performed on $\mathcal{D}_t$ and digest_S^t is the corresponding server digest. aux_t is the auxiliary information generated by the server that contains any part of $\mathcal{D}_t$ that has changed due to the update u_t, along with its associated authentication information from digest_S^t (generated by owner during setup or update). $\mathsf{auxproof}_t$ is the proof of authenticity of aux_t and $\mathcal{D}_{t+1}$ is the updated DS, i,e., $\mathcal{D}_{t+1} := U(\mathcal{D}_t, u_t)$.

$(\mathsf{state}_O, \mathsf{digest}_C^{t+1}, \mathsf{Upd}_{t+1}) \leftarrow \mathbf{VerifyUpdate}(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^t, \mathsf{aux}_t, \mathsf{auxproof}_t, u_t, \mathsf{SID}_t)$ where sk is the secret key of the owner, state_O is the internal state variable of the owner, digest_C^t is the client digest corresponding to DS $\mathcal{D}_t$ and SID_t is set to the output of a function f on the queries invoked since the last update (Setup for the 0^{th} update). $\mathsf{auxproof}_t$, aux_t and u_t are as defined above. $\mathsf{auxproof}_t$ is used to verify the authenticity of aux_t. If it verifies, aux_t is used to generate an update string Upd_{t+1} for the server, an updated digest digest_C^{t+1} for the client, and the state variable state_O is updated. If the authenticity of aux_t does not verify, then the algorithm $\mathsf{VerifyUpdate}$ outputs $(\perp, \perp, \perp)$ and stops.

6.2 Security Properties

We need to accommodate the new interaction between the owner and the server only in the completeness and the soundness definitions. The **zero-knowledge** definition for updates and queries

remains the same as in Definition 4.3, since this change in the model is opaque from the client's perspective. In other words, the client's view is exactly the same in both DPPADS and SE-DPPADS, regardless of how the update phase is implemented.

The new **completeness** definition describes the following. If all three parties are honest, then for any ADT, the owner accepts auxiliary information about an update as returned by the server using UpdateDS; and the client accepts an answer and a proof to his query as returned by the server using Query. We note that this definition augments Definition 4.1 to capture the verification step performed by the owner during the update phase.

To this end, we define a predicate $\mathsf{checkAux}(\mathcal{D}, \mathsf{digest}_S, u, \mathsf{aux})$ that takes a data structure $\mathcal{D}$, the corresponding server digest digest_S, an update request u, and auxiliary information aux. If aux contains the portion of $\mathcal{D}$ that changes due to update u, along with the relevant portion of digest_S, then checkAux outputs 1 and it outputs 0, otherwise. We note that this predicate is not part of the model and is not executed by any of the participating parties. It is merely used to express correctness of aux w.r.t. a DS $\mathcal{D}$, its corresponding authentication information digest_S, and an update u.

Definition 6.1 (Completeness) *For an ADT $(\mathcal{D}_0, Q, U)$, any sequence of updates $u_0, u_1, \ldots, u_L$ on the data structure $\mathcal{D}_0$, and for all queries δ on $\mathcal{D}_L$:*

$$
\begin{aligned}
&\Pr[(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^k); (\mathsf{state}_O, \mathsf{digest}_C^0, \mathsf{digest}_S^0) \leftarrow \mathsf{Setup}(\mathsf{sk},\mathsf{pk},\mathcal{D}_0);\\
&\quad \big\{(\mathcal{D}_{t+1}, \mathsf{aux}_t, \mathsf{auxproof}_t) \leftarrow \mathsf{UpdateDS}(\mathcal{D}_t, u_t, \mathsf{digest}_S^t);\\
&\quad\ (\mathsf{state}_O, \mathsf{digest}_C^{t+1}, \mathsf{Upd}_{t+1}) \leftarrow \mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^t, \mathsf{aux}_t, \mathsf{auxproof}_t, u_t, \mathsf{SID}_t);\\
&\quad\ (\mathsf{digest}_S^{t+1}, \mathcal{D}_{t+1}) \leftarrow \mathsf{UpdateServer}(\mathsf{digest}_S^t, \mathsf{Upd}_{t+1}, \mathcal{D}_t, u_t); \big\}_{0\le t\le L}\\
&\quad \big\{(\mathcal{D}_{L+1}, \mathsf{aux}_L, \mathsf{auxproof}_L) \leftarrow \mathsf{UpdateDS}(\mathcal{D}_L, u_L, \mathsf{digest}_S^L) :\\
&\quad\ \mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^L, \mathsf{aux}_L, \mathsf{auxproof}_L, u_L, \mathsf{SID}_L) = (\mathsf{state}_O, \mathsf{digest}_C^{L+1}, \mathsf{Upd}_{L+1})\\
&\quad\ \wedge\, \mathsf{checkAux}(\mathcal{D}_L, \mathsf{digest}_S^L, u, \mathsf{aux}_L) = 1\big\} \wedge\\
&\quad \big\{(\mathsf{answer}, \mathsf{proof}) \leftarrow \mathsf{Query}(\mathsf{digest}_S^L, \mathcal{D}_L, \delta) :\\
&\quad\ \mathsf{Verify}(\mathsf{pk}, \mathsf{digest}_C^L, \delta, \mathsf{answer}, \mathsf{proof}) = \mathsf{accept} \wedge \mathsf{answer} = Q(\mathcal{D}_L, \delta)\big\}] = 1
\end{aligned}
$$

The new **soundness** definition protects the client as well as the owner against a malicious server (Adv in the definition). Recall that Definition 4.2 considered only the former case since the update phase was performed by a trusted party: the owner. The additional protection ensures that if the server forges the auxiliary information for an update request from the owner, the owner will accept it with at most negligible probability.

Definition 6.2 (Soundness) *For all PPT adversaries,* Adv *and for all possible valid queries δ on the data structure $\mathcal{D}_j$ of an ADT, there exists a negligible unction $\nu(.)$ such that, the probability of winning the following game is negligible:*

Setup Adv *is given* pk *where* $(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^k)$. *Given* pk, Adv *picks an ADT of its choice,* $(\mathcal{D}_0, Q, U)$ *and receives the server digest* digest_S^0 *for* $\mathcal{D}_0$, *where where* $(\mathsf{state}_O, \mathsf{digest}_C^0, \mathsf{digest}_S^0) \leftarrow \mathsf{Setup}(\mathsf{sk},\mathsf{pk},\mathcal{D}_0)$.

Query Adv *requests a series of updates $u_1, u_2, \ldots, u_L$, where $L = \mathsf{poly}(\mathsf{k})$, of his choice. For $0 \le t \le L$, the following steps are executed.*

1. Adv *runs* UpdateDS. *Let* $(\mathcal{D}_{t+1}, \mathsf{aux}_t, \mathsf{auxproof}_t) \leftarrow \mathsf{UpdateDS}(\mathcal{D}_t, u_t, \mathsf{digest}_S^t)$. *Outputs* aux_t *and* $\mathsf{proofaux}_t$ *are sent to the challenger.*
2. *The challenger runs* VerifyUpdate. *Let* $(\mathsf{state}_O, \mathsf{digest}_C^{t+1}, \mathsf{Upd}_{t+1}) \leftarrow \mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^t, \mathsf{aux}_t, \mathsf{auxproof}_t, u_t, \mathsf{SID}_t)$ *where* $\mathsf{SID}_t = \perp$. Upd_{t+1} *is sent back to* Adv.

Response *Finally,* Adv *outputs at least one of the following:*

- $(\mathcal{D}_j, u, \mathsf{aux}, \mathsf{auxproof})$, $0 \leq j \leq L$;
- $(\mathcal{D}_j, \delta, \mathsf{answer}, \mathsf{proof})$, $0 \leq j \leq L$.

Adv *wins the game if the following holds:*

$$\{\mathsf{checkAux}(\mathcal{D}_j, \mathsf{digest}_S^j, u, \mathsf{aux}) = 0 \wedge \mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{state}_O, u, \mathsf{aux}, \mathsf{auxproof}, \mathsf{digest}_S^j, \perp) \neq (\perp, \perp, \perp)\} \vee$$
$$\{\mathsf{answer} \neq Q(\mathcal{D}_j, \delta) \wedge \mathsf{Verify}(\mathsf{pk}, \mathsf{digest}_C^j, \delta, \mathsf{answer}, \mathsf{proof}) = \mathsf{accept}\}.$$

Recall that VerifyUpdate *returns* $(\perp, \perp, \perp)$ *only when auxiliary information is not verified.*

6.3 Space Efficient DPPAL

The construction for DPPAL presented in Section 5 can be adapted to the space efficient model by using an authenticated version of the order data structure, $\mathsf{AOD} = (\mathsf{setupAOD}, \mathsf{proveAOD}, \mathsf{verifyAOD})$, instantiated below.

6.3.1 Authenticated Order Data Structure

We give an instantiation of an authenticated version of the order data structure, $\mathsf{AOD} = (\mathsf{setupAOD}, \mathsf{proveAOD}, \mathsf{verifyAOD})$ based on Merkle Hash Tree (MHT) [42]. We will use this construction as a black box for our space efficient DPPAL construction. AOD is executed between the trusted owner and the untrusted server. The owner generates a digest and authenticated information of the order data structure O using setupAOD. He keeps the digest and sends authenticated information and O to the server. Given an update operation, the server runs proveAOD using O and authenticated information to generate the reply and a proof. The owner executes verifyAOD to verify that the reply was authentic w.r.t. his order data structure and to update his digest.

We sketch an instantiation of each algorithm of AOD here.

setupAOD: For a given $\mathsf{O} = \mathsf{OD}(n)$, this algorithm builds a Merkle Hash Tree (MHT) [42] on the tag space of O, which is of size $2n$. For each tag, the corresponding leaf of the MHT stores the list element associated with this tag, if any, and $\perp$ otherwise. setupAOD returns root_O: the root of the MHT which represents the digest information for O.

proveAOD: AOD supports four queries and authenticates them as follows:

- $\mathsf{tag}(x)$**:** The proof returns the element x at leaf $\mathsf{tag}(x)$ and the authentication path $\mathsf{auth}(\mathsf{tag}(x))$ of the MHT. Recall that auth of a leaf in MHT is a sequence of siblings of every node on the path from the leaf to the root.
- $\mathsf{delete}(x)$**:** The proof returns the element x at leaf $\mathsf{tag}(x)$ and the authentication path, $\mathsf{auth}(\mathsf{tag}(x))$, in the MHT.
- $\mathsf{replace}(x', x)$**:** The proof returns the element x' at leaf $\mathsf{tag}(x')$ and the authentication path, $\mathsf{auth}(\mathsf{tag}(x'))$, in the MHT.
- $\mathsf{inserafter}(x, y)$**:** The proof of this operation is slightly more involved. Let $\mathcal{Y} \leftarrow \mathsf{O.insertafter}(x, y)$. The server returns $\mathcal{Y}$ and $w, \mathsf{auth}(\mathsf{tag}(w))$ for all w (possibly empty) in the smallest tag range that is not overflowing (i.e., has room for a new element) and $\mathsf{tag}(x)$. It also returns a tag to be assigned to y, $\mathsf{tag}(y)$, and $\mathsf{auth}(\mathsf{tag}(y))$.

verifyAOD: The verification involves two steps: (1) check the portion of O that has changed due to the update is indeed the correct portion that needed to be changed, and (2) the returned

portion of O is authentic. The verification involves verifying the authentication paths auth of the returned tag values. Recall that in MHT it is done by hashing the returned value with its sibling, then hashing the result with the sibling of the parent, and so on till the hash of the root is computed. The verification succeeds if this value equals $\mathsf{root}_{\mathsf{O}}$. The owner can use standard MHT verification for each operation except for $\mathsf{insertafter}(x, y)$. In this case, the owner has to verify that the tag range returned by the server is indeed the smallest tag range that needed to be relabeled for the insert operation. In particular, the correct range has the following properties: it is the smallest tag range that encloses $\mathsf{tag}(x)$ and it is not in overflow in O (i.e., the density of the range is above the threshold as described in Section 3).
Before going through verification of $\mathcal{Y}$ and the tag range, we recall that an enclosing range of $\mathsf{tag}(x)$ can be computed using $\mathsf{tag}(x)$ and n, the size of the original list. The first enclosing tag range is just $\mathsf{tag}(x)$. The next enclosing tag range consists of all the leaves in the subtree rooted at the parent of $\mathsf{tag}(x)$ (i.e., if $\mathsf{tag}(x)$ is odd, then this enclosing tag range consists of integers between $\mathsf{tag}(x)-1$ and $\mathsf{tag}(x)$, and integers between $\mathsf{tag}(x)+1$ and $\mathsf{tag}(x)$, otherwise). This corresponds to the tag range at level 2^1 of the implicit full binary tree over the universe of tags $[0, N]$, where N is set to $2n$ in our implementation. The next enclosing tag range consists of all the leaves in the subtree rooted at the parent of the parent of $\mathsf{tag}(x)$ and so on. Thus all the enclosing tag ranges can be enumerated by walking up the implicit binary tree until the smallest range not in overflow is found. The verification steps are given in Algorithm 9.
Note that an insertion or deletion might cause the number of elements in the list to become less than $\frac{n}{2}$ or greater than $2n$. This will require rebuild of O and the underlying tree. In this case, the server has to return the whole MHT and the owner can verify the correctness by checking number of leaves that are assigned to elements in the MHT using their authentication paths. We describe the verification steps when the insertion does not cause rebuild in Algorithm 9.

Security and Efficiency: The security of theAOD presented above follows from the security of MHT and properties of the relabeling algorithm of OD. For a single leaf, MHT returns a proof

Algorithm 9 Verification of $\mathsf{insertafter}(x, y)$ operation.

1: % *Let l and u be the leftmost and rightmost points of the current tag range enclosing* $\mathsf{tag}(x)$*, and l' and u' be the leftmost and rightmost points of the previous (smaller) tag range enclosing* $\mathsf{tag}(x)$.

2: Set $l' := -1$, $u' := -1$, and parent to be a parent of $\mathsf{tag}(x)$.
3: **while** ($\mathsf{parent} \neq \perp$) % *enclosing range rooted at* parent *is still in the tree*
4: Let l and u be the leftmost and rightmost leaves of the range enclosed by parent.
5: **for** every tag i in range $[l, u]/[l', u']$ % *exclude elements processed in smaller tag ranges*
6: Find element (possibly empty) corresponding to tag i among w's returned by the server.
7: If not found, **return** reject.
8: Verify this element using the corresponding auth path.
9: If not verified, **return** reject.
10: Remember the old tag range as: $l' := l$ and $u' := u$.
11: **if** (tag range $[l, u]$ is not in overflow) % *there is an unoccupied* tag *in this range*
12: Verify the authenticity of $\mathsf{tag}(y)$ using $\mathsf{auth}(\mathsf{tag}(y))$.
13: If verified, **return** accept. Otherwise, reject.
14: Set parent to the parent node of old parent.
15: % *Insert caused a rebuild of the underlying tree.*

of size $O(\log n)$ since it returns a node for every level in the tree. The size of $\mathcal{Y}$ is of amortized size $O(1)$, hence, proveAOD and verifyAOD require $O(\log n)$ amortized time for every query.

6.3.2 DPPAL Construction

In the setup, the owner executes Setup in Algorithm 2 and calls setupAOD with order data structure O as input to generate the digest of AOD. The owner's state state_O now only stores the digest of AOD and the list digest digest_C^0 instead of the list $\mathcal{L}_0$, O and associated authenticated information. As before, the owner sends digest_C^0 to the client, while digest_S^0 is augmented to contain authenticated information of AOD.

The update phase is changed as follows. UpdateOwner in Algorithm 4 is split between two algorithms: UpdateDS, executed by the server, and VerifyUpdate, executed by the owner. Informally this split proceeds as follows. Given an update request u_t from the owner, the server runs UpdateDS. This algorithm updates $\mathcal{L}_t$ and O and returns to the owner elements that were changed during the update, their tags and signatures, and a tag of the new element in case of linsertafter and lreplace. UpdateDS also returns proofs of authenticity of her output computed using proveAOD.

Given the updated content and a corresponding proof from UpdateDS, the owner executes VerifyUpdate which first invokes verifyAOD to verify that the server returned the correct tags that correspond to the update. The owner also verifies the signatures. If the verification passes, then parts of the UpdateOwner (Algorithm 4) that involve generation of fresh randomness, member witnesses and new signatures on the elements are executed. Finally, VerifyUpdate updates the state variable, state_O, the updated digests for O and the list. It outputs the new list digest and an update string for the server. The server updates her digest as in Section 5.

The algorithms UpdateServer, Query and Verify in Section 5 remain unchanged.

Security: The completeness and soundness of this space efficient DPPAL construction follow from the security of the underlying AOD and DPPAL construction of Section 5. Zero-knowledge follows from server-client protocol of Section 5.

Efficiency: This construction improves the performance of update phase in Section 5. The space complexity for the owner is $O(1)$ since he keeps a digest of O. Update phase incurs $\log n$ multiplicative cost due to a hash based AOD construction. We highlight below the changes in efficiency as compared to Theorem 5.1.

Theorem 6.1 *The space efficient dynamic privacy-preserving authenticated list construction has the following performance, where n is the list size, L is the number of updates in a batch and M is the number of distinct elements that have been queried since the last update:*

- *The owner uses $O(n)$ time and space for setup, and keeps $O(1)$ state;*
- *The update phase requires one round of interaction between the owner and the server where they exchange a message of size $(L \log n + M)$;*
- *The update phase requires $O(L \log n + M)$ time for the owner and the server, or $O(\log n)$ amortized over the number of queried or updated elements.*

7 Dynamic Privacy-Preserving Authenticated Tree (DPPAT)

In this section, we propose another instantiation of DPPADS: a fully dynamic privacy-preserving tree, DPPAT, where the ADT is a rooted ordered tree instead of a list. In particular, we first discuss in detail how a tree can be uniquely represented using two lists. Then, we show that

order queries and dynamic operations on a tree can be translated into dynamic operations on those two lists. Given this list representation of a tree, it becomes easy to instantiate a DPPAT using instantiations of DPPAL as described in Section 5 or 6. As a consequence, the security properties of the resulting DPPAT construction follow from the security properties of the underlying DPPAL scheme as discussed in Section 5.5 or Section 6.3.2.

Data structure: Let $\mathcal{T}$ be a rooted tree of non-repeated elements where each vertex, or node, stores an element. Recall that a *rooted tree* is a connected, undirected, acyclic graph where one node is designated as *root*. We implicitly consider each edge as a directed edge and all the edges are directed away from the root. We denote each element as a binary string $\{0,1\}^*$. Two nodes in a tree can be related in exactly one of the following ways:

Above/below: A node x is above a node y if x is an ancestor of y.

Left/right: If two nodes x and y are not related by ancestry relation, then one is to the left of the other (they may or may not be siblings) with respect to their lowest common ancestor (LCA).

In the example in Figure 1a, node F is to the *left* of node H, and node E is to the *right* of node K.

Tree Representation via L-Order and R-Order: We define two recursive traversal patterns for traversing a (non-empty) rooted tree:

Left to right traversal (L-Order): Start from the root. At node u do the following: (1) Process node u; (2) Recursively traverse its subtrees in left to right order.

Right to left traversal (R-Order): Start from the root. At node u do the following: (1) Process node u; (2) Recursively traverse its subtrees in right to left order.

See Figure 1a for an example of the above traversals.

From the theory of Partial Orders it is well known that a rooted tree $\mathcal{T}$ can be uniquely represented as two lists, L-Order$_{\mathcal{T}}$ and R-Order$_{\mathcal{T}}$ (Lemma 7.1). Following the standard definition, $\mathsf{rank}(\mathcal{L}, x)$ is a function that associates a natural number to every element x of a list $\mathcal{L}$ such that $\mathsf{rank}(\mathcal{L}, x) < \mathsf{rank}(\mathcal{L}, y)$ if x precedes y is the list, $\mathcal{L}$.

Lemma 7.1 *[11] Given the L-Order and the R-Order ordering of a rooted tree, the relation between two nodes x and y of the tree is uniquely determined by their relative orders in L-Order and R-Order:*

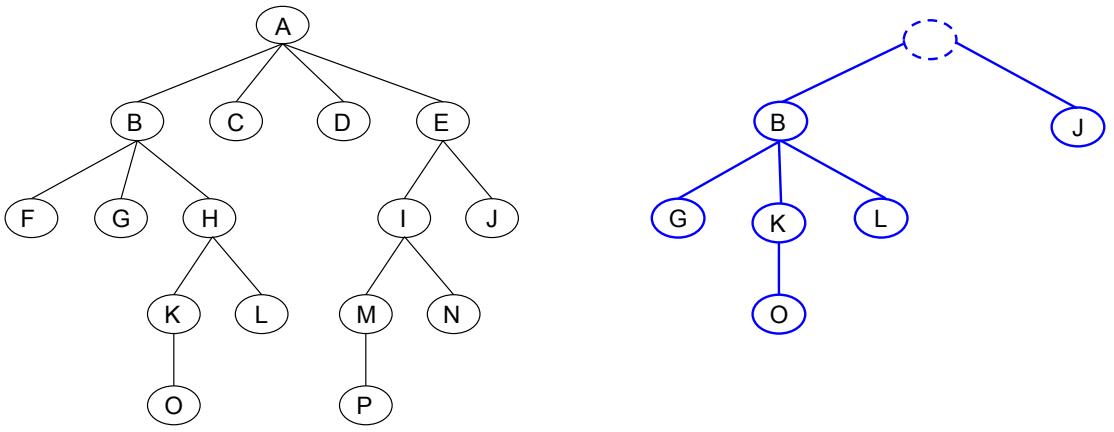
A node x is above *a node y iff* $\mathsf{rank}(L\text{-}Order, x) < \mathsf{rank}(L\text{-}Order, y)$ *and* $\mathsf{rank}(R\text{-}Order, x) < \mathsf{rank}(R\text{-}Order, y)$.

A node x is to the left *of a node y w.r.t. their lowest common ancestor iff* $\mathsf{rank}(L\text{-}Order, x) < \mathsf{rank}(L\text{-}Order, y)$ *and* $\mathsf{rank}(R\text{-}Order, x) > \mathsf{rank}(R\text{-}Order, y)$.

By Lemma 7.1, we see that the relation between any two nodes in a rooted tree can be uniquely represented using order relations on two lists. We will discuss how the order queries and updates on a tree translate to order queries on its representative lists L-Order and R-Order in the subsequent sections. Given that, a DPPAT scheme can be instantiated with the DPPAL schemes we described in Sections 5.5 or 6.3.2.

7.1 Tree Preliminaries

In this section we recall some standard terminologies for trees (Definition 7.1 and Definition 7.2) and introduce new terminology (Definition 7.3 and Definition 7.4) that we would need in order to express order queries on a tree.



(a) (b)

Figure 1: (a) A tree $\mathcal{T}$ with L-Order $= \{A, B, F, G, H, K, O, L, C, D, E, I, M, P, N, J\}$ and R-Order $= \{A, E, J, I, N, M, P, D, C, B, H, L, K, O, G, F\}$. (b) A forest $\mathcal{F}_\delta = (V(\mathcal{F}_\delta), E(\mathcal{F}_\delta))$ induced on $\mathcal{T}$ by $\delta = \{B, J, G, K, O, L\}$, where $V(\mathcal{F}_\delta) = \{B, J, G, K, O, L\}$ and $E(\mathcal{F}_\delta) = \{(B, G), (B, K), (B, L), (K, O)\}$).. The dotted node is a dummy node that expresses the left to write order between the subtrees rooted at node B and J.

Definition 7.1 (Short-circuiting two vertices) *We define the following operation as short circuiting two vertices: given two vertices $u, v \in V(\mathcal{T})$, if there is a directed path from u to v in $\mathcal{T}$, then replace the path with a directed edge from u to v.*

Definition 7.2 (Induced forest) *Given a set of vertices, $\delta = \{v_1, \ldots, v_m\} \in V(\mathcal{T})$, we define the forest $\mathcal{F}_\delta = (V(\mathcal{F}_\delta), E(\mathcal{F}_\delta))$ induced by δ as follows:*

- *Set $V(\mathcal{F}_\delta) := \{v_1, \ldots, v_m\}$ and initialize $E(\mathcal{F}_\delta) := \emptyset$.*
- *$\forall v_i, v_j \in V(\mathcal{F}_\delta)$, if edge $(v_i, v_j) \in E(\mathcal{T})$, then add that edge to the forest, i.e., $E(\mathcal{F}_\delta) := E(\mathcal{F}_\delta) \cup (v_i, v_j)$.*
- *$\forall v_i, v_j \in V(\mathcal{F}_\delta)$, if there exists a directed path from v_i to v_j in $\mathcal{T}$, then short circuit v_i, v_j in $\mathcal{F}_\delta$, i.e., $E(\mathcal{F}_\delta) := E(\mathcal{F}_\delta) \cup (v_i, v_j)$.*

See Figure 1b for an example.

Definition 7.3 (Order Representation of a tree) *We define the representative set $\mathsf{Rep}(\mathcal{T})$ for a tree $\mathcal{T} = (V(\mathcal{T}), E(\mathcal{T}))$ recursively, as follows:*

- *Initialize $\mathsf{Rep}(\mathcal{T}) := \emptyset$.*
- *Explore $\mathcal{T}$ in breadth-first search (BFS) order starting at the root and augment each $v \in V(\mathcal{T})$ with its BFS level number $0, 1, \ldots, h(\mathcal{T})$ where $h(\mathcal{T})$ is the height of $\mathcal{T}$.*
- *Now update $\mathsf{Rep}(\mathcal{T})$ as follows: for level i, $1 \leq i \leq h(\mathcal{T}) - 1$:*
  - *$\mathsf{Rep}(\mathcal{T}) := \mathsf{Rep}(\mathcal{T}) \cup (u, v, 0)$ where $(u, v) \in E(\mathcal{T})$ and u has level $i - 1$, v has level i.*
  - *$\mathsf{Rep}(\mathcal{T}) := \mathsf{Rep}(\mathcal{T}) \cup (u_1, u_2, 1) \cup (u_2, u_3, 1) \cup \ldots \cup (u_{l-1}, u_l, 1)$ where $u_1, \ldots, u_l$ are nodes at level i ordered as follows: let $u_1, \ldots, u_j$ be the children of node w. Here, the children nodes, $u_1, \ldots, u_j$ are arranged according to the total order induced by their parent w.*

Let $\mathsf{LCA}(a, b)$ denote the lowest common ancestor of nodes a and b.

Claim 7.1 *Let $\mathcal{T} = (V(\mathcal{T}), E(\mathcal{T}))$ be a rooted tree and $a, b, c, x_1, x_2 \in V(\mathcal{T})$ where $x_1 = \mathsf{LCA}(a, b)$ and $x_2 = \mathsf{LCA}(b, c)$. Then $\mathsf{LCA}(a, c) = \mathsf{LCA}(x_1, x_2)$.*

Proof If possible, let $\mathsf{LCA}(x_1, x_2) = x_3$, $\mathsf{LCA}(a, c) = x_4$ and $x_3 \neq x_4$. Since x_3 is an ancestor of both x_1 and x_2, therefore x_3 is an ancestor of both a and c. By definition, x_4 is also an ancestor of both a and c. Therefore either x_3 is an ancestor of x_4 or x_4 is an ancestor of x_3. But since $x_4 = \mathsf{LCA}(a, c)$, it must be the case that x_3 is an ancestor of x_4. This, in turn, implies that x_4 is an ancestor of x_1 (since x_1 and x_4 are ancestors of a, and x_3 is an ancestor of x_1 and x_4) and x_2 (similarly). Therefore x_4 is lower that x_3 and this contradicts the fact that $x_3 = \mathsf{LCA}(x_1, x_2)$. Hence $x_3 = x_4$. ∎

Observation 7.1 *Let $\mathcal{F}_\delta$ be the forest induced on tree $\mathcal{T}$ by a subset of its vertices δ. Let $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$ be the components of $\mathcal{F}_\delta$ and $r_1, r_2, \ldots, r_k$ be the root vertices of these trees, respectively. Then:*

- *For each pair of root vertices r_i, r_j, either r_i is to the left or to the right of r_j w.r.t. their LCA in the source tree $\mathcal{T}$;*
- *If r_i is to the left of r_j w.r.t. their LCA in $\mathcal{T}$, then $\forall u, v$ where $u \in \mathcal{T}_i, v \in \mathcal{T}_j$, u is to the left of v w.r.t. their LCA in $\mathcal{T}$;*
- *By Claim 7.1 and the fact that tree is planar, it follows that the left/right relation w.r.t. LCA induces a total order on $r_1, r_2, \ldots r_k$.*

Definition 7.4 (Order Representation of a forest) *We define the representative set $\mathsf{Rep}(\mathcal{F})$ for a forest $\mathcal{F} = (V(\mathcal{F}), E(\mathcal{F}))$ recursively, as follows:*

- *Initialize $\mathsf{Rep}(\mathcal{F}) := \emptyset$.*
- *Let $\mathcal{F} = \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$ be the trees of the forest and let $r_1, r_2, \ldots, r_k$ be their roots ordered as per the left/right relation. Update $\mathsf{Rep}(\mathcal{F}) := \mathsf{Rep}(\mathcal{F}) \cup (r_1, r_2, 1) \cup (r_2, r_3, 1) \cup \ldots \cup (r_{k-1}, r_k, 1)$. Note that order between any two roots can be inferred given the $k-1$ pairwise orders $(r_1, r_2), (r_2, r_3), \ldots, (r_{k-1}, r_k)$ by transitivity of the left/right relation w.r.t. LCA as discussed in Observation 7.1.*
- *Then, for each $\mathcal{T}_i, 1 \leq 1 \leq k$, update $\mathsf{Rep}(\mathcal{F}) := \mathsf{Rep}(\mathcal{F}) \cup \mathsf{Rep}(\mathcal{T}_i)$.*

Note that $\mathsf{Rep}(\mathcal{F})$ denotes the partial order induced by $\mathcal{T}$ on the vertices of the forest, $V(\mathcal{F}_\delta)$.

Lemma 7.2 *Let $\mathcal{T}$ be a tree and $\mathcal{F}_\delta$ be the forest induced by a subset of vertices of $\mathcal{T}$, δ. Then, the order between each pair of vertices $x, y \in \delta$ is inferrable from $\mathsf{Rep}(\mathcal{F}_\delta)$.*

Proof For any pair of vertices $x, y \in V(\mathcal{F}_\delta)$, there are two possible cases:
Case 1: x, y belong to the same component of the forest, say $\mathcal{T}_i$. We can split this case further:

- x, y are related through ancestory: Wlog, assume that x is an ancestor of y and let $x, v_1, v_2, \ldots, v_l, y$ be the path from x to y in $\mathcal{T}_i$. By construction of $\mathsf{Rep}(\mathcal{T}_i)$, for every parent child pair $v_p, v_c \in V(\mathcal{T}_i)$, $(v_p, v_c, 0) \in \mathsf{Rep}(\mathcal{T}_i)$. Therefore, $(x, v_1, 0), (v_1, v_2, 0), \ldots, (v_{l-1}, v_l, 0) \in \mathsf{Rep}(\mathcal{T}_i)$ and hence the order between x and y is inferrable by transitivity.
- x and y are related as left/right w.r.t. $\mathsf{LCA}(x, y)$. Wlog, assume that x is to the left of y w.r.t. $\mathsf{LCA}(x, y)$. If x and y are siblings, then $(x, y, 1) \in \mathsf{Rep}(\mathcal{T}_i)$ by construction. Otherwise, let x be in the subtree rooted at r_x and y be in the subtree rooted at r_y, s.t., r_x and r_y are children of $\mathsf{LCA}(x, y)$. Therefore, by construction, $(r_x, r_y, 1) \in \mathsf{Rep}(\mathcal{T}_i)$ and therefore the order between x and y is inferrable from the tuple $(r_x, r_y, 1) \in \mathsf{Rep}(\mathcal{T}_i)$.

Hence, the order between $x, y \in \mathcal{T}_i$ is inferrable from $\mathsf{Rep}(\mathcal{T}_i) \subseteq \mathsf{Rep}(\mathcal{F}_\delta)$.

Case 2: x, y belong to different components of the forest, say $x \in \mathcal{T}_i$ and $y \in \mathcal{T}_j$, $i \neq j$. In this case, the order between x and y is the same as the order between the corresponding roots of $\mathcal{T}_i$ and $\mathcal{T}_j$ (as discussed in Observation 7.1). Since the order between every pair of roots of the component trees is inferrable from $\mathsf{Rep}(\mathcal{F}_\delta)$, the order between x, y is also inferrable from $\mathsf{Rep}(\mathcal{F}_\delta)$.

Lemma 7.3 *Let $\mathcal{T}$ be a tree and $\mathcal{F}_\delta$ be the forest induced by a subset of vertices of $\mathcal{T}$, δ. Then, the size of $\mathsf{Rep}(\mathcal{F}_\delta)$ is $O(|\delta|)$.*

Proof Let $\mathcal{F}_\delta$ consist of k components: $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$. We can compute the size of $\mathsf{Rep}(\mathcal{T}_i)$, $1 \leq i \leq k$, as follows. Consider a node of $\mathcal{T}_i$ at level j of the BFS order and the number of tuples in $\mathsf{Rep}(\mathcal{T}_i)$ it has in common with nodes at level j or above. Each such node participates in at most three tuples in $\mathsf{Rep}(\mathcal{T}_i)$: one tuple with its parent (at BFS level $j-1$) and at most two tuples with its two siblings (the one on its left and the one on its right, both at BFS level j). Then, the total number of tuples in $\mathsf{Rep}(\mathcal{T}_i)$ is $3(|V(\mathcal{T}_i)| - 1)$, since the root is included in the tuples of its children.

$\mathsf{Rep}(\mathcal{F}_\delta)$ includes $\mathsf{Rep}(\mathcal{T}_i)$ for every tree and $k-1$ tuples for the roots of adjacent trees. Therefore, $|\mathsf{Rep}(\mathcal{F}_\delta)| = (k-1) + \sum_{1 \leq i \leq k} 3(|V(\mathcal{T}_i)| - 1)$. Since $|V(\mathcal{T}_1)| + |V(\mathcal{T}_2)| + \ldots + |V(\mathcal{T}_k)| = |\delta|$ and $k \leq |\delta|$, $|\mathsf{Rep}(\mathcal{F}_\delta)| = O(|\delta|)$. ∎

7.2 Order Queries on Trees

An *order query* on a tree $\mathcal{T}$ of distinct elements is defined as follows: given a pair of query elements (x, y) of $\mathcal{T}$, the server returns the pair with its elements rearranged according to their order in $\mathcal{T}$ along with a bit b indicating the relationship between x and y. If $b = 0$, then the returned order of x, y indicates that x, y are in the *above/below* relationship. Otherwise, if $b = 1$, then the returned order of x, y indicates that x, y are in the *left/right* relationship w.r.t. their lowest common ancestor. For example, if y is an ancestor of x in $\mathcal{T}$, then the pair $(y, x, 0)$ is returned as an answer. If y is to the left of x, w.r.t. their lowest common ancestor, then $(y, x, 1)$ is returned as an answer.

For generality, the data structure also supports *batch order query*. Given a non-trivial set of query elements δ, i.e., $|\delta| \geq 2$, the server returns the forest induced by δ, $\mathcal{F}_\delta = (V(\mathcal{F}_\delta), E(\mathcal{F}_\delta))$, and its representative set $\mathsf{Rep}(\mathcal{F}_\delta)$ (as defined in Definition 7.3).

The server proves authenticity of his answers to order queries by providing a proof of its answer. The proof consists of pairwise orders of the form (x, y, b). Recall that, for a query of size two only one such tuple is returned, while for a larger query a sequence of tuples is returned in $\mathsf{Rep}(\mathcal{F}_\delta)$. A proof for each tuple in $\mathsf{Rep}(\mathcal{F}_\delta)$ is computed using the result in Lemma 7.1. For the order $(x, y, 0)$ the server returns two proofs: $\{\mathsf{rank}(\text{L-Order}, x) < \mathsf{rank}(\text{L-Order}, y)$ and $\mathsf{rank}(\text{R-Order}, x) < \mathsf{rank}(\text{R-Order}, y)\}$. Likewise, for the order $(x, y, 1)$ she returns two proofs, $\{\mathsf{rank}(\text{L-Order}, x) < \mathsf{rank}(\text{L-Order}, y)$ and $\mathsf{rank}(\text{R-Order}, y) < \mathsf{rank}(\text{R-Order}, x)$. In order to compute the proofs of order of x, y in L-Order and R-Order, the server uses DPPAL version of L-Order and R-Order.

Example Consider the tree in Figure 1a and a query $\delta = \{B, J, G, K, O, L\}$. Then the server returns the induced forest in Figure 1b and $\mathsf{Rep}(\mathcal{F}_\delta)$ where $\mathsf{Rep}(\mathcal{F}_\delta) = \{(B, J, 1), (B, G, 0), (B, K, 0), (B, L, 0), (G, K, 1), (K, L, 1), (K, O, 0)\}$. He also returns a proof for every order in $\mathsf{Rep}(\mathcal{F}_\delta)$. The proof of order includes two permutations of δ:

- $\delta' = \{B, G, K, O, L, J\}$ (elements of δ permuted according to their order in L-Order);
- $\delta'' = \{J, B, L, K, O, G\}$ (elements of δ permuted according to their order in R-Order).

and a proofs of each order:

$\boldsymbol{(B, J, 1)}$: $\{(\mathsf{rank}(\text{L-Order}, B) < \mathsf{rank}(\text{L-Order}, J)) \wedge (\mathsf{rank}(\text{R-Order}, B) > \mathsf{rank}(\text{R-Order}, J))\}$;

$(\boldsymbol{B}, \boldsymbol{G}, \mathbf{0})$: $\{(\mathsf{rank}(\text{L-Order}, B) < \mathsf{rank}(\text{L-Order}, G)) \wedge (\mathsf{rank}(\text{R-Order}, B) < \mathsf{rank}(\text{R-Order}, G))\}$;
$(\boldsymbol{B}, \boldsymbol{K}, \mathbf{0})$: $\{(\mathsf{rank}(\text{L-Order}, B) < \mathsf{rank}(\text{L-Order}, K)) \wedge (\mathsf{rank}(\text{R-Order}, B) < \mathsf{rank}(\text{R-Order}, K))\}$;
$(\boldsymbol{B}, \boldsymbol{L}, \mathbf{0})$: $\{(\mathsf{rank}(\text{L-Order}, B) < \mathsf{rank}(\text{L-Order}, L)) \wedge (\mathsf{rank}(\text{R-Order}, B) < \mathsf{rank}(\text{R-Order}, L))\}$;
$(\boldsymbol{G}, \boldsymbol{K}, \mathbf{1})$: $\{(\mathsf{rank}(\text{L-Order}, G) < \mathsf{rank}(\text{L-Order}, K)) \wedge (\mathsf{rank}(\text{R-Order}, G) > \mathsf{rank}(\text{R-Order}, K))\}$;
$(\boldsymbol{K}, \boldsymbol{L}, \mathbf{1})$: $\{(\mathsf{rank}(\text{L-Order}, K) < \mathsf{rank}(\text{L-Order}, L)) \wedge (\mathsf{rank}(\text{R-Order}, K) > \mathsf{rank}(\text{R-Order}, L))\}$;
$(\boldsymbol{K}, \boldsymbol{O}, \mathbf{0})$: $\{(\mathsf{rank}(\text{L-Order}, K) < \mathsf{rank}(\text{L-Order}, O)) \wedge (\mathsf{rank}(\text{R-Order}, K) < \mathsf{rank}(\text{R-Order}, O))\}$.

7.3 Dynamic Operations on Trees

DPPAT can support link, cut and replace on a tree $\mathcal{T}$. It does so by making the following *constant* number of update queries to the DPPAL representations of L-Order$_{\mathcal{T}}$ and R-Order$_{\mathcal{T}}$ (formalized in Lemma 7.4):

link(v, w)**:** Insert new node $v \notin \mathcal{T}$ by making v a child of $w \in \mathcal{T}$, if w was a leaf. Using DPPAL this can be done as: L-Order$_{\mathcal{T}}$.linsertafter(v, w) and R-Order$_{\mathcal{T}}$.linsertafter(v, w).

link(v, w, x)**:** If w is not a leaf in $\mathcal{T}$, the third argument x specifies a child of w; due to the link, v becomes a child of w in $\mathcal{T}$, succeeding x. Using DPPAL: L-Order$_{\mathcal{T}}$.linsertafter(v, x_{right}) and R-Order$_{\mathcal{T}}$.linsertafter(v, y_{left}), where y is the sibling to the right of x, x_{right} is the rightmost node of the subtree rooted at x and y_{left} is the leftmost node of the subtree rooted at y.

cut(v)**:** Delete a leaf node $v \in \mathcal{T}$ by disconnecting v from its parent in $\mathcal{T}$. Using DPPAL: L-Order$_{\mathcal{T}}$.ldelete(v) and R-Order$_{\mathcal{T}}$.ldelete(v).

replace(v', v)**:** Replace the content of node v' with new content v (v is not content of $\mathcal{T}$). Using DPPAL: L-Order$_{\mathcal{T}}$.lreplace(v', v) and R-Order$_{\mathcal{T}}$.lreplace(v', v).

Lemma 7.4 *All the dynamic operations for a Dynamic Privacy-Preserving Authenticated Tree (DPPAT) can be implemented using one call on the L-Order and one call on the R-Order representing the tree.*

We summarize the security properties and efficiency of our DPPAT scheme in Theorem 7.1.

Theorem 7.1 *A dynamic privacy-preserving authenticated tree (DPPAT) can be implemented using a DPPAL. This scheme satisfies the security properties of a DPPADS: completeness (Definition 4.1), soundness (Definition 4.2) and zero-knowledge (Definition 4.3). The runtime, space, and message size for every party is proportional to the corresponding runtime, space, and message size in the DPPAL scheme.*

Proof We showed in Lemma 7.1 that a tree can be completely and uniquely represented using two lists: L-Order and R-Order. We then showed in Lemma 7.2 that the answer and proof size for an order query on a tree, is proportional to the size of the query. The initial generation of two lists involves traversing the tree twice which takes time proportional to the size of the tree and, hence, does not add any additional overhead for the owner. By Lemma 7.4, each dynamic operation on a DPPAT can be implemented using one call on each of L-Order and R-Order. Completeness, soundness and zero-knowledge of the DPPAT construction follow from DPPAL security properties (as proved in Theorem 5.1). ∎

Remark: The technique used for DPPAT can be further extended to d-dimensional Partial Orders (POs) for some constant d. The extension relies on the unique intersection of d total ordered lists of a PO. Hence, the dynamic privacy-preserving version can be implemented using d DPPALs (e.g., a tree is a PO with $d = 2$).

8 Acknowledgment

This research was supported in part by the National Science Foundation under grant CNS–1228485.

References

[1] Ahn, J.H., Boneh, D., Camenisch, J., Hohenberger, S., Shelat, A., Waters, B.: Computing on authenticated data. In: TCC (2012)

[2] Atallah, M.J., Blanton, M., Fazio, N., Frikken, K.B.: Dynamic and efficient key management for access hierarchies. ACM Trans. Inf. Syst. Secur. (2009)

[3] Atallah, M.J., Blanton, M., Frikken, K.B.: Efficient techniques for realizing geo-spatial access control. In: ASIACCS (2007)

[4] Attrapadung, N., Libert, B., Peters, T.: Computing on authenticated data: New privacy definitions and constructions. In: ASIACRYPT (2012)

[5] Babka, M., Bulánek, J., Cunát, V., Koucký, M., Saks, M.: On online labeling with polynomially many labels. In: ESA (2012)

[6] Benabbas, S., Gennaro, R., Vahlis, Y.: Verifiable delegation of computation over large datasets. In: CRYPTO (2011)

[7] Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two simplified algorithms for maintaining order in a list. In: ESA (2002)

[8] Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious b-trees. SIAM J. Comput. (2005)

[9] Bender, M.A., Duan, Z., Iacono, J., Wu, J.: A locality-preserving cache-oblivious dynamic dictionary. J. Algorithms (2004)

[10] Bird, R.S., Sadnicki, S.: Minimal on-line labelling. Inf. Proc. Let. (2007)

[11] Birkhoff, G.: Lattice theory, vol. 25. American Mathematical Soc. (1967)

[12] Boneh, D., Boyen, X.: Efficient selective-ID secure identity based encryption without random oracles. In: EUROCRYPT (2004)

[13] Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: EUROCRYPT (2003)

[14] Brodal, G.S., Fagerberg, R., Jacob, R.: Cache oblivious search trees via binary trees of small height. In: SODA (2002)

[15] Brzuska, C., Busch, H., Dagdelen, Ö., Fischlin, M., Franz, M., Katzenbeisser, S., Manulis, M., Onete, C., Peter, A., Poettering, B., Schröder, D.: Redactable signatures for tree-structured data: Definitions and constructions. In: ACNS (2010)

[16] Bukhatwa, F., Patel, A.: Effects of ordered access lists in firewalls. In: IADIS WWW/Internet International Conference (2004)

[17] Camacho, P., Hevia, A.: Short transitive signatures for directed trees. In: CT-RSA (2012)

[18] Catalano, D., Fiore, D.: Vector commitments and their applications. In: PKC (2013)

[19] Catalano, D., Fiore, D., Messina, M.: Zero-knowledge sets with short proofs. In: EUROCRYPT (2008)

[20] Chang, E.C., Lim, C.L., Xu, J.: Short redactable signatures using random trees. In: CT-RSA (2009)

[21] Chase, M., Healy, A., Lysyanskaya, A., Malkin, T., Reyzin, L.: Mercurial commitments with applications to zero-knowledge sets. In: EUROCRYPT (2005)

[22] Chase, M., Kohlweiss, M., Lysyanskaya, A., Meiklejohn, S.: Malleable signatures: Complex unary transformations and delegatable anonymous credentials. IACR ePrint Archive (2013)

[23] Dietz, P.F.: Maintaining order in a linked list. In: STOC (1982)

[24] Dietz, P.F., Seiferas, J.I., Zhang, J.: A tight lower bound for on-line monotonic list labeling. In: SWAT (1994)

[25] Dietz, P.F., Sleator, D.D.: Two algorithms for maintaining order in a list. In: STOC (1987)

[26] Dietz, P.F., Zhang, J.: Lower bounds for monotonic list labeling. In: SWAT (1990)

[27] Emek, Y., Korman, A.: New bounds for the controller problem. In: DISC (2009)

[28] Ghosh, E., Ohrimenko, O., Tamassia, R.: Verifiable member and order queries on a list in zero-knowledge. IACR ePrint Archive Report 2014/632 (2014)

[29] Goldberg, S., Naor, M., Papadopoulos, D., Reyzin, L., Vasant, S., Ziv, A.: NSEC5: Provably preventing DNSSEC zone enumeration. ePrint Archive, Report 2014/582

[30] Goodrich, M.T.: Pipelined algorithms to detect cheating in long-term grid computations. Theoretical Computer Science (2008)

[31] Goodrich, M.T., Kerschbaum, F.: Privacy-enhanced reputation-feedback methods to reduce feedback extortion in online auctions. In: CODASPY (2011)

[32] Hamed, H., Al-Shaer, E.: Dynamic rule-ordering optimization for high-speed firewall filtering. In: ASIACCS (2006)

[33] Itai, A., Konheim, A.G., Rodeh, M.: A sparse table implementation of priority queues. In: Akko (1981)

[34] Johnson, R., Molnar, D., Song, D.X., Wagner, D.: Homomorphic signature schemes. In: CT-RSA (2002)

[35] Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: FC (2013)

[36] Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: ASIACRYPT (2010)

[37] Kundu, A., Atallah, M.J., Bertino, E.: Leakage-free redactable signatures. In: CODASPY (2012)

[38] Kundu, A., Bertino, E.: Structural signatures for tree data structures. PVLDB (2008)

[39] Kundu, A., Bertino, E.: Privacy-preserving authentication of trees and graphs. Int. J. Inf. Sec. (2013)

[40] Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In: TCC (2010)

[41] Liskov, M.: Updatable zero-knowledge databases. In: ASIACRYPT (2005)

[42] Merkle, R.C.: A certified digital signature. In: CRYPTO (1989)

[43] Micali, S., Rabin, M.O., Kilian, J.: Zero-knowledge sets. In: FOCS (2003)

[44] Micali, S., Rivest, R.L.: Transitive signature schemes. In: CT-RSA (2002)

[45] Miyazaki, K., Hanaoka, G., Imai, H.: Digitally signed document sanitizing scheme based on bilinear maps. In: ASIACCS (2006)

[46] Naveed, M., Prabhakaran, M., Gunter, C.A.: Dynamic searchable encryption via blind storage. In: IEEE Symp. on Security and Privacy (2014)

[47] Ostrovsky, R., Rackoff, C., Smith, A.: Efficient consistency proofs for generalized queries on a committed database. In: ICALP (2004)

[48] Poehls, H.C., Samelin, K., Posegga, J., De Meer, H.: Length-hiding redactable signatures from one-way accumulators in $O(n)$. Tech. Rep. MIP-1201, FIM. University of Passau (2012)

[49] Pöhls, H.C., Samelin, K.: On updatable redactable signatures. In: ACNS (2014)

[50] Prabhakaran, M., Xue, R.: Statistically hiding sets. In: CT-RSA (2009)

[51] Samelin, K., Poehls, H.C., Bilzhause, A., Posegga, J., De Meer, H.: Redactable signatures for independent removal of structure and content. In: ISPEC (2012)

[52] Steinfeld, R., Bull, L., Zheng, Y.: Content extraction signatures. In: ICISC (2001)

[53] Tapdiya, A., Fulp, E.: Towards optimal firewall rule ordering utilizing directed acyclical graphs. In: ICCCN (2009)

[54] Tsakalidis, A.K.: Maintaining order in a generalized linked list. Acta Inf. (1984)

[55] Wang, Z.: Improvement on Ahn et al.'s RSA P-homomorphic signature scheme. In: SecureComm (2012)

[56] Willard, D.E.: Maintaining dense sequential files in a dynamic environment (extended abstract). In: STOC (1982)

[57] Willard, D.E.: Good worst-case algorithms for inserting and deleting records in dense sequential files. In: ACM SIGMOD (1986)

[58] Willard, D.E.: A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. Inf. Comp. (1992)

[59] Yi, X.: Directed transitive signature scheme. In: CT-RSA (2006)