Verifiable Zero-Knowledge Order Queries and Updates for Fully Dynamic Lists and Trees

Esha Ghosh[1(✉)], Michael T. Goodrich[2], Olga Ohrimenko[3], and Roberto Tamassia[1]

[1] Department Computer Science, Brown University, Providence, USA
{esha_ghosh,roberto_tamassia}@brown.edu

[2] Department Computer Science, University of California, Irvine, USA
goodrich@uci.edu

[3] Microsoft Research, Cambridge, UK
oohrim@microsoft.com

Abstract. We propose a three-party model for maintaining a dynamic data structure that supports verifiable and privacy-preserving (zero-knowledge) queries. We give efficient constructions supporting this model for order queries on data organized in lists, trees, and partially-ordered sets of bounded dimension.

1 Introduction

Cloud computing enables clients to outsource storage, computation, and services to online service providers, thus benefiting from scalability, availability, and usage-driven pricing. However, there are also some challenges that arise from cloud computing, such as the difficulty of maintaining assurances of data integrity and privacy as physical possession of data is delegated to a cloud storage provider. Thus, we are interested in the study of technical solutions that allow a cloud storage provider to prove the integrity of client data and the adherence to privacy policies concerning this data. Of course, in order for any such technical solution to be practically viable, the efficiency of such cloud-based integrity and privacy solutions should be a major factor in evaluating them.

The need for simultaneously providing efficiency, integrity, and privacy in cloud-based outsourced storage has motivated a considerable amount of recent research on a three-party model, where a *data owner* uploads a database to a *cloud server* so that a group of *clients* can interact with the server to execute queries on the outsourced database (e.g., see [14–16,21,22,29]). This approach has resulted in some interesting solutions, but most existing techniques appear to be limited to static datasets, where data is uploaded only once by the data owner and never updated, in spite of the fact that data changes over time in many practical applications. There is only a small amount of prior work that addresses integrity and privacy on dynamic outsourced data, and, to be best

Research supported in part by the U.S. National Science Foundation and by the Kanellakis Fellowship at Brown University.

DOI: 10.1007/978-3-319-44618-9_12

of our knowledge, all of this prior work considers only set membership and set algebra queries (e.g., see [7,13,25,33]).

In this paper, we provide a framework and formal security definitions for the problem of ensuring integrity and privacy in cloud services operating on dynamic data, with an efficient construction that can process a rich set of queries. The queries include membership and order queries on dynamic lists, trees, and partially-ordered sets of bounded dimension.

Our first main contribution is to formally define a model we call *dynamic privacy-preserving authenticated data structure* (DPPADS). In this model, a data *owner* outsources his data structure to a *server* who answers queries issued by *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 or the updates besides what can be inferred from the answers. In other words, the privacy property ensures that even a malicious client learns nothing about any update, unless she specifically queries for the updated item before and after the update. For example, consider the case when a new element, x, is inserted into a dataset. If a client queried for x before and after this update, she will *only* learn that x has been added to the database, but nothing else about the database.

The dynamic behavior of data structures raises challenges from a definitional point of view since an adversary (i.e., a malicious client) may choose update operations and later query the system to see their effect on nearby data, for example. Pöhls and Samelin [33] consider updates in the three-party model only for positive membership queries and their definition is specific to their data structure and cannot be easily generalized to support richer data structures and queries. Thus, we feel a richer framework, like DPPADS, is warranted.

Our DPPADS model strives to capture realistic uses of cloud services from the data owner's perspective. First, the owner's online presence is required only if he needs to update the data. Hence, clients' queries are performed solely by the server. Second, access control policies on the data can be seamlessly integrated with our model since integrity tokens sent to the clients to enable verification of query answers do not leak any information about the non-queried data.

Our second main contribution is to show how to efficiently instantiate DPPADS with membership and *order queries* on lists, trees, and partial orders of bounded dimension. Order queries are fundamental mechanisms for seeking relative order information about the elements of a partial order. In the simplest case, an order query for two elements, u and v, of a list L asks whether u precedes or follows v in L. Consider now an ordered tree, T, i.e., a rooted tree where a left-to-right ordering is defined among the descendants of each node. An order query on two nodes u and v, of T asks which one of the following relations holds: u is above v; u is below v; u is to the left of v; or u is to the right of v. The first two cases occur when one of u and v is a descendant of the other and the last two cases occur when u and v are in distinct subtrees of their least common ancestor. We note that none of the previous works supported dynamic behavior on such a rich set of structural data with integrity and privacy guarantees.

Our model implies that the client should be able to verify the data she queried with authentication tokens produced in part by the server (as the owner is not present in the query phase). Building constructions with such authentication tokens is challenging for several reasons. Beside being hard to forge, these tokens should bare no information about the rest of the data, not even the size of the data; only in this case we can protect privacy of non-queried data. Furthermore, dynamic datasets require corresponding updates to these tokens as a consequence. Hence, these tokens should be easily updatable.

Existing work comes short in achieving all of the requirements above at once. For example, commitments and zero-knowledge proofs [17] present a naive but unfortunately inefficient solution. In particular, a naive approach would require space and setup cost quadratic in the list/tree size and, hence, very inefficient. Though more efficient solutions exist [10,14], they are set in the case where data does not change after the owner uploads it and they consider only sets and lists as underlying datasets. An attempt to cover dynamic datasets with integrity and privacy guarantees was made by Liskov [25] and Catalano and Fiore [7]. These constructions are set in a weaker privacy model which we elaborate further in Sect. 2. In fact, the authors comment on this limitation themselves: "Ideally, the adversary should learn nothing more than the values of elements for which a proof has been obtained (and possibly updated), and that updates have occurred. However, we have not been able to realize this full level of security, and instead offer a weaker but acceptable notion of security." [25] (We also note that the question of efficient constructions of zero-knowledge data structures more complex than sets is left open in [33]; we answer it affirmatively here.)

Due to these limitations, we take an algorithmic approach to this problem and identify efficient dynamic constructions for lists and trees. Then we integrate the lightweight cryptographic primitive developed in [14] with our algorithmic constructions in a novel way. But this alone was not sufficient to achieve our strong notion of privacy for updates that are influenced by a strong adversary. To this end, we have developed a technique of systematic, periodic re-randomization to achieve strong privacy guarantees. However, for some application, even this technique is not sufficient as the information that "an update has occurred" can itself be regarded as sensitive information. This leakage is out of the scope of the privacy definitions, but can be crucial for some applications from a practical point of view. We address this issue further and propose a technique that can be executed periodically in order to hide the existence of an update.

Our constructions strive to achieve good performance for all the three parties. In particular, all the parties run in *optimal time* except for a logarithmic (in the size of the source data structure) runtime overhead for the server. The client-server interaction consists of a single round: the client sends a query and the server returns the answer and the proof. The proof size and client verification time are proportional to the answer. The owner interacts with the server only when he needs to make an update to his data. We consider two types of owners: one that can keep a copy of the data structure (DPPADS) and one that prefers not to (due to limited storage resources) (SE-DPPADS). In the first case, the owner performs an update operation himself, in time linear in the size of a batch update. In the second case, he outsources the update operation to the server and later verifies it, incurring a logarithmic multiplicative cost in the size of the source data structure. In both cases, the owner performs constant-time updates, this time is amortized over the number of elements queried or updated since the last update. Our contributions can be summarized as follows:

– We formally define the DPPADS model for a dynamic privacy-preserving authenticated data structure that supports *zero-knowledge* proofs for queries and *zero-knowledge* updates (Sect. 3).
– We give an efficient construction of a DPPADS in the Random Oracle model for a list that supports order queries and updates (Sect. 4).
– We give an overview of a space-efficient variant of the DPPADS model in Sect. 4 and defer the detailed description to the technical report [12].
– We present an efficient extension of our DPPADS construction to trees and partial orders of bounded dimension (Sect. 5).

2 Related Work

We describe the related primitives and discuss how we compare with them. Detailed comparison of privacy properties and the asymptotic complexity of our constructions with the most efficient constructions in the literature is in Table 1.

Traditional authenticated data structures (ADS) [11,18,31,37] are often set in the three party model with a trusted owner, a *trusted* client and a malicious server; the owner outsources the data to the server and later the client interacts with the server to run queries on the data. The security requirement of such constructions is data authenticity for the client against the server. This integrity requirement is the same in our model. However, since the client is trusted, the strong privacy requirement of our model is usually violated by the ADS proofs. For example, Merkle Hash Tree (MHT) [26] reveals the number of elements in the dataset and the proof path in a MHT reveals order information.

Authenticity and privacy together were considered in the two-party model of *zero knowledge set* (ZKS) [8,10,24,27] introduced in [27] and later used in knowledge lists [14], statistically hiding sets [34] and consistent query protocols [30]. In this model a malicious prover commits to a database in the setup phase and later a malicious verifier queries it. The prover and the verifier are non-colluding. The prover may try to give answers inconsistent with the committed database, while the verifier may try to learn information beyond query answers. In this paper, we study a three-party model which can be seen as a relaxation of the ZKS model with similar privacy and integrity guarantees. That is, the committer is "honest" and the (malicious) prover is different from the committer. The three party model leads to efficiency enhancements since one can use primitives with a trapdoor (like bilinear aggregate signature in our case) as opposed to trapdoorless hash and commitments and generic zero-knowledge proofs. As a result, efficient constructions were proposed for positive membership queries [1,2,38], dictionary queries on sets [16,29], range queries [15], order queries and statistics on lists [6,9,14,22,23,32,35]. However, all of these models consider a *static* dataset. We enhance this three-party model to support a fully dynamic dataset and formalize the notion of privacy and integrity.

Updates in both of the above models have received only limited attention. The notion of updatable zero knowledge set was first proposed in [25]with two definitions: 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 [7,25]. Our zero-knowledge definition in Sect. 3 supports opaque updates in the three-party model, which is also satisfied by our constructions.

In the three-party model, updates on a set were considered in the recent work of [33]. This work supports privacy-preserving verification of positive membership only (i.e., a proof is returned only when the queried elements are members of the given set). Their formal definition for updates is based on an indistinguishability game and is specific to their data structure and cannot be easily extended to support richer data structures and queries. In comparison, we propose simulation based definition and our definition are not tailored to any specific data structure. Moreover, the construction of [33] supports only two update operations: addition of new elements and merge of two sets. Here, we consider operations on lists, trees and support addition, deletion and replace operations.

We compare privacy properties and the asymptotic complexity of our constructions with the static [14] and updatable [33] constructions in Table 1. We note that, the *static* construction for order queries in [14] was shown to outperform the existing static constructions of [6,9,20,21,32,35,36]. We show that the performance of our construction for queries is the same as that of [14]. Moreover, our list construction is the first to support 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 phase are optimal.

Finally, we note that our work on privacy-preserving updates is not to be confused with *history independent data structures (HIDSs)* [28]. HIDS is concerned with the leakage one obtains when she looks at the *layout* of a data structure before and after a sequence of updates on it. In our model, the client (i.e., the adversary) obtains only some *content* of the data structure and not the layout. Furthermore, we require the client to be able to verify that query answers are correct and not leak any information about the rest of the content.

3 Dynamic Privacy Preserving Authenticated Data Structure (DPPADS)

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

Table 1. 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, m is the query 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. Wlog 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); (SE-)DPPAL/T denotes (space efficient) Dynamic Privacy Preserving Authenticated Lists and Trees. We use $\tilde{n}$ to denote $\min(m \log n, n)$.

	[14]	[33]	DPPAL/T	SE-DPPAL/T
Zero-knowledge update			✓	✓
Transparent update		✓	✓	✓
Owner's state size		n	n	1
Server storage size	n	n	n	n
Order query time	$\tilde{n}$		$\tilde{n}$	$\tilde{n}$
Order verification time	m		m	m
Positive membership query time	$\tilde{n}$	m	$\tilde{n}$	$\tilde{n}$
Positive membership verif. time	m	m	m	m
Proof size	m	m	m	m
Insertion time		L	$L + M$	$L \log n + M$
Deletion time			$L + M$	$L \log n + M$
Assumptions	ROM, nBDHI	DIHF, SRSA	ROM, nBDHI	ROM, nBDHI
Assumptions	ROM	DIHF	ROM	ROM
	nBDHI	SRSA	nBDHI	nBDHI

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. (In this model the owner is required to keep a copy of $\mathcal{D}$ to perform updates, while in the space efficient version the owner keeps only a small digest.) 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 $\mathcal{D}$.

Model. DPPADS is a tuple of six probabilistic polynomial time algorithms $(\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{UpdateOwner}, \mathsf{UpdateServer}, \mathsf{Query}, \mathsf{Verify})$. We describe how these algorithms are used between the three parties and 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 itself 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 its digest, it also prepares a proof of the answer. The client then uses Verify to verify the query answer against proof and the digest she 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 a session information and is set to the output of a function f on the queries invoked since the last update (Setup is counted as the 0^{th} update).

$\mathsf{UpdateOwner}$ 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.

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

Our model also supports the execution of a batch of updates as a single operation, which may be used to optimize overall performance (Sect. 4). We note that SID and f are introduced for efficiency reasons only. Intuitively, function f can be instantiated in a way that helps reduce the owner's work for maintaining zero-knowledge property of each update. We leave f to be defined by a particular instantiation. Once defined, f remains fixed for the instantiation. Since the function is public, anybody, who has access to the list of (authentic) queries performed since the last update, can compute it.

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 her query from the server. Here, honest behavior implies that whenever the owner updates the data structure and its public digest, the server updates $\mathcal{D}$ and its digest accordingly and replies client's queries faithfully w.r.t. the latest data structure and digest.

Definition 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 $\mathcal{D}_L$:*

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

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, it can pick any point of time (w.r.t. updates) to create a forgery.

Since, given the server digest, the server can compute answers to queries herself, it is superfluous to give Adv explicit access to Query algorithm.

Definition 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 *adaptively chooses a series of updates $u_1, u_2, \ldots, u_L$ and corresponding* SID*s, where $L = \mathsf{poly(k)}$. 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.,* $(\mathsf{state}_O, \mathsf{digest}_C^{i+1}, \mathsf{Upd}_{i+1}, \mathcal{D}_{i+1}, u_i) \leftarrow \mathsf{UpdateOwner}(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^i, \mathsf{digest}_S^i, \mathcal{D}_i, u_i, \mathsf{SID}_i)$.

Response: *Finally,* Adv *outputs* $(\mathcal{D}_j, \delta, \mathsf{answer}, \mathsf{proof})$, $0 \le j \le L$, *and wins the game if* $\mathsf{answer} \neq Q(\mathcal{D}_j, \delta)$ *and* $\mathsf{Verify}(\mathsf{pk}, \mathsf{digest}_C^j, \delta, \mathsf{answer}, \mathsf{proof}) = \mathsf{accept}$.

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 she also receives an updated digest, due to the owner making changes to $\mathcal{D}$. Informally, (1) the proofs should reveal nothing beyond the query answer, and (2) an updated digest should reveal nothing about update operations performed on $\mathcal{D}$. This security property guarantees that the client does not learn which elements were updated, unless she queries for an updated element (deleted or replaced), before and after the update.

Definition 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 asks only for valid data and update queries.*[1]

$\mathsf{Real}_{\mathcal{E},\mathsf{Adv}}(1^k)$	$\mathsf{Ideal}_{\mathcal{E},\mathsf{Adv},\mathsf{Sim}}(1^k)$
The challenger, $\mathcal{C}$, runs $\mathsf{KeyGen}(1^k)$ to generate sk, pk, sends pk to Adv_1.	Sim_1 generates a public key, pk, sends it to Adv_1 and keeps a state, $\mathsf{state}_{\mathcal{S}}$.
Given pk, Adv_1 picks an ADT $(\mathcal{D}_0, Q, U)$ of its choice.	
Given $\mathcal{D}_0$, $\mathcal{C}$ runs $\mathsf{Setup}(\mathsf{sk}, \mathsf{pk}, \mathcal{D}_0)$ and sends $\mathsf{digest}_{\mathcal{C}}^0$ to Adv_1.	Sim_1 generates $\mathsf{digest}_{\mathcal{C}}^0$, sends it to Adv_1, updates $\mathsf{state}_{\mathcal{S}}$. (It is not given $\mathcal{D}_0$.)
With access to Adv_1's state, Adv_2 adaptively queries $\{q_1, q_2, \ldots, q_M\}$, $M = \mathsf{poly}(\mathsf{k})$:	
Let $\mathcal{D}_{t-1}$ denote the state of the data structure at the time of q_i.)	
On data query q_i:	
$\mathcal{C}$ runs Query algorithm for the query on $\mathcal{D}_{t-1}$ and the corresponding digest as its parameters. $\mathcal{C}$ returns answer and proof to Adv_2	Given the answer to the query, $Q(\mathcal{D}_{t-1}, q_i)$, and $\mathsf{state}_{\mathcal{S}}$, Sim_2 generates answer and proof, sends them to Adv_2 and updates its state.
On update query q_i:	
$\mathcal{C}$ runs $\mathsf{UpdateOwner}$ algorithm on q_i and returns the public digest $\mathsf{digest}_{\mathcal{C}}^t$	Given $\mathsf{state}_{\mathcal{S}}$, Sim_2 returns updated digest $\mathsf{digest}_{\mathcal{C}}^t$ and updates its state. (It is not given the update query q_i.)
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.*

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

We note that SID argument to $\mathsf{UpdateOwner}$ need not be used explicitly in the definition: Adv implicitly controls the input of f by choosing queries on $\mathcal{D}$ (recall that $\mathsf{SID} = f(\ldots)$), while the challenger and the simulator know all the queries and can compute f themselves.

Comparison with the update definitions of [7,25]**:** Liskov [25] introduced two notions of update (w.r.t. a zero-knowledge database): *Opaque:* an adversary should learn nothing more than the values of queried elements and the fact that an update has occurred. *Transparent:* in addition to what is revealed in the opaque update, an adversary learns the pseudonym of an updated key; pseudonyms are generated deterministically when keys are added to the database and do not change. The constructions in [25] and [7] achieve only the weaker

[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.

of the two, that is, they satisfy the transparent definition. Our zero-knowledge definition is close to the opaque definition where an updated client digest is indistinguishable from a fresh digest, and old proofs are not valid after an update.

4 Dynamic Privacy-Preserving Authenticated List

In this section we instantiate a DPPADS with a list (an ordered set of distinct elements) and refer to it as *dynamic privacy-preserving authenticated list.* We first describe the cryptographic primitives and assumptions that our construction relies on for security. We then explain how to maintain labeling of a dynamic list and how we use it to achieve efficient updates in our construction.

Preliminaries. Let $\mathcal{L}$ denote a list and $\mathsf{Elements}(\mathcal{L})$ denote the unordered set corresponding to $\mathcal{L}$. We refer to element's position in the list as it's rank. 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., $\mathsf{answer} = \pi_{\mathcal{L}}(\delta)$. For example, with $\mathcal{L} = \{a, b, c, d, e\}$ and $\delta = (d, a, e)$, answer is $\{a, d, e\}$. An update operation on a list can be one of the following: $\mathsf{linsertafter}(x, y)$: insert element $x \notin \mathcal{L}$ after element $y \in \mathcal{L}$; $\mathsf{ldelete}(x)$: delete element x from $\mathcal{L}$; $\mathsf{lreplace}(x', x)$: replace element $x' \in \mathcal{L}$ with element $x \notin \mathcal{L}$.

Bilinear Maps: Let k be the security parameter, 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 $e : G \times G \rightarrow G_1$ is a map with properties: (1) $\forall u, v \in G$ and $\forall a, b \in \mathbb{Z}$, $e(u^a, v^b) = e(u, v)^{ab}$; (2) $e(g, g) \neq 1$ where g is a generator of G. As is standard, we assume that group action on G, G_1 and the bilinear map e can be computable in one unit time. We measure time complexity in terms of number of group actions.

Bilinear Aggregate Signature Scheme [5]*:* Given signatures $\sigma_1, \ldots, \sigma_n$ on *distinct* messages $M_1, \ldots, M_n$ from a user, 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 user indeed signed the n original messages. The scheme guarantees that σ is valid iff the aggregator used all σ_i's to construct it.

Hardness Assumption: Let p be a large k-bit prime where $k \in \mathbb{N}$ is a security parameter. Let $P \in \mathbb{N}$ be polynomial in k, $p = \mathsf{poly(k)}$. Let $e : G \times G \rightarrow G_1$ be a bilinear map (as defined above) and g be a random generator of G. We denote a probabilistic polynomial time (PPT) adversary $\mathcal{A}$ as a probabilistic polynomial time Turing Machine running in time $\mathsf{poly(k)}$.

Definition 4 (P-Bilinear Diffie Hellman Inversion (P-BDHI) [4]). *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: $Pr[s \xleftarrow{\$} \mathbb{Z}_p^*; y \leftarrow \mathcal{A}(\langle g, g^s, g^{s^2}, \ldots, g^{s^P}\rangle) : y = e(g, g)^{\frac{1}{s}}] \leq \nu(k)$.*

Online List Labeling (or File Maintenance) Problem [3,19,39]: In *online list labeling*, a 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. Moreover, the mapping has to be maintained efficiently as the list changes.

We use the order data structure OD presented in [3] for online list labeling. We briefly describe OD and summarize its performance here. 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 encloses 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. $\mathsf{OD}(n)$ supports the following operations:
$\mathsf{insertafter}(x, y)$: To insert an element x after y, do the following: (1) Examine the enclosing tag ranges of y. (2) Calculate the density of a tag range by traversing the elements within the tag range. (3) Relabel the smallest enclosing tag range that is not overflowing. (4) Return the relabeled tags and the tag of x.
$\mathsf{delete}(x)$: Delete x from the list and mark the corresponding tag as unoccupied.
$\mathsf{tag}(x)$: Return the tag of element x.

We note that the original list together with a sequence of updates on it deterministically define the tags of all elements in OD.

Complexity: Initially, we set $N = (2n)^{\frac{1}{1-\log\alpha}}$, $n_{\min} = n/2$ and $n_{\max} = 2n$, where n is the number of elements. As elements are inserted into or deleted from the list, the data structure can generate tags while $n_{\min} \le n \le n_{\max}$. If at any point, the current number of elements, n, falls below $n_{\min}$ or exceeds $n_{\max}$, the data structure is rebuilt for the new value of n and N is recomputed. Hence, the algorithm needs $\frac{\log n}{1-\log\alpha}$ bits to represent a tag. The rebuild introduces a constant amortized overhead (over the insert and delete operations). Hence, $\mathsf{OD}(n)$ uses $O(\log n)$ bits per tag and $O(n \log n)$ bits for storing all tags. $\mathsf{OD}(n)$ has $O(1)$ amortized insert and delete time, and $O(1)$ time for tag.

Dynamic Construction. Our construction of DPPAL uses as a starting point a static privacy-preserving authenticated list [14]. 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 above) 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 the 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 efficient for static lists, data structures are often dynamic. The intuition behind our modifications is as follows. We first notice that the rank information of each element used in the construction of [14] can actually be replaced with *any tag* that respects the rank ordering. For example, let the rank of elements x and y be 5 and 6, respectively. We can replace this information with $\mathsf{tag}(x)$ and $\mathsf{tag}(y)$ as long as the following hold: 1) $\mathsf{tag}(x) < \mathsf{tag}(y)$ and 2) there is no other element in the list whose tag falls between $\mathsf{tag}(x)$ and $\mathsf{tag}(y)$. The tag generation algorithm of the order labeling data structure $\mathsf{OD}(n)$ has exactly this property: elements' tags respect the order of elements' ranks in the list. Hence, we use $\mathsf{OD}(n)$ to generate tags for the elements (instead of their ranks) to maintain list order. This enables efficient updates, albeit, in a non privacy-preserving way (e.g., information about ranks as well as which elements were updated is revealed). To this end, we develop a re-randomization method (explained in the subsequent *Update Phase*) to preserve privacy.

Our construction consists of instantiating the algorithms of DPPADS: Setup, UpdateOwner, UpdateServer, Query and Verify. We describe each algorithm in this section and give their pseudo-code in Algorithms 1–6. We use the following notation. $\mathcal{H} : \{0,1\}^* \rightarrow G$: cryptographic hash function that will be modeled as a random oracle in the security analysis; all arithmetic/group operations are performed $\mod p$. System parameters are $(p, G, G_1, e, g, \mathcal{H})$, where p, G, G_1, e, g are defined in Sect. 4. $\mathcal{L}_0$ 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 Phases: These algorithms proceed as follows. The owner randomly picks $s, v \in \mathbb{Z}_p^*$ and ω as part of his secret key sk and publishes $\mathsf{pk} = g^v$ as his public key as in [14]. But instead of using rank information, the owner 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 uses the following GenAuthTokens procedure: it generates fresh randomness r_i to blind $\mathsf{tag}(x_i)$; computes member witness $t_{x_i \in \mathcal{L}_0}$ as $g^{s^{\mathsf{tag}(x_i)} r_i}$ and its signature σ_{x_i}

Algorithm 1. $(\mathsf{state}_O, \mathsf{digest}^0_C, \mathsf{digest}^0_S) \leftarrow \mathbf{Setup}(\mathsf{sk}, \mathsf{pk}, \mathcal{L}_0)$ where $\mathcal{L}_0 = \{x_1, \ldots, x_n\}$.

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

as $\mathcal{H}(t_{x_i \in \mathcal{L}_t} || x_i)^v$. The owner then executes standard signature aggregation by multiplying element signatures into a list signature $\sigma_{\mathcal{L}_0}$. To preserve privacy of the size of the list, he further multiplies the list signature with $\mathsf{salt} = (\mathcal{H}(\omega))^v$.

The owner sends $\sigma_{\mathcal{L}_0}$ to the client, as the client digest digest^0_C. To the server, he sends $\mathcal{L}_0$ and a digest digest^0_S which contains the tag, the random value used for blinding, the member witness and the signature for every element in the list (we refer to these four units as *authentication units* of an element). He also sends $g^{s^0}, \ldots, g^{s^n}$, that help the server compute proofs for the client during the query phase. The owner saves $\mathcal{L}_0, \mathsf{O}, \mathsf{digest}^0_S$ in his state variable state_O, which he later uses to perform updates.

Update Phase: UpdateOwner (Algorithm 3) lets the owner perform update u_t on his outsourced data structure and propagate the update in the digests. For the actual update, the owner uses O to efficiently compute the new tag of an element and update the tags of the elements affected by the update. (We note this may include rebuild of O itself when the size of the list either falls below $n/2$ or grows above $2n$.) The owner then updates all the digests and authentication units that have to be updated due to the tag change. For the server, the owner computes the member witnesses and signatures since these operations rely on the secret keys. As we argue later, updating only elements whose tags have been modified due to the update is not sufficient to obtain zero-knowledge update. To this end, the owner has to also rerandomize any authentication units that were sent to the client. We elaborate on each step in the update below.

The update of authentication units depends on which one of the three update operations was performed on the list. If a new element x has been inserted in the list (i.e., linsertafter operation), then the owner recomputes all witnesses and signatures of elements in $\mathcal{Y}$ where $\mathcal{Y}$ is a set of elements whose tags were

Algorithm 2. $(\sigma_{\mathcal{L}_t}, \mathsf{O}, \Sigma_{\mathcal{L}_t}, \Omega_{\mathcal{L}_t}) \leftarrow \mathsf{build}(\mathsf{sk}, \mathsf{state}_O, \mathcal{L}_t)$ where sk contains v and ω and $\mathcal{L}_t = \{x_1, \ldots, x_{n'}\}$.

```
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 $i < i \leq n'$: $\mathsf{O}.\mathsf{insertafter}(x_{i-1}, x_i)$.
3: For every $x_i \in \mathcal{L}_t$: $r_i, t_{x_i \in \mathcal{L}_0}, \sigma_{x_i} \leftarrow \mathsf{GenAuthTokens}(x_i)$
4: Compute list digest signature $\sigma_{\mathcal{L}_t} \leftarrow \mathsf{salt} \times \prod_{x_i \in \mathcal{L}_t} \sigma_{x_i}$, where $\mathsf{salt} = (\mathcal{H}(\omega))^v$.
5: $\Sigma_{\mathcal{L}_t} := \langle \forall x_i \in \mathcal{L}_t : (t_{x_i \in \mathcal{L}_t}, \sigma_{x_i}), \mathcal{H}(\omega) \rangle$ and $\Omega_{\mathcal{L}_t} := \langle \forall x_i \in \mathcal{L}_t : (r_i, \mathsf{tag}(x_i)) \rangle$
6: return $(\sigma_{\mathcal{L}_t}, \mathsf{O}, \Sigma_{\mathcal{L}_t}, \Omega_{\mathcal{L}_t})$
```

Algorithm 3. $(\mathsf{state}_O, \mathsf{digest}_C^{t+1}, \mathsf{Upd}_{t+1}, \mathcal{L}_{t+1}, u_t) \leftarrow$ **UpdateOwner**$(\mathsf{sk}, \mathsf{state}_O, \mathsf{digest}_C^t, \mathsf{digest}_S^t, \mathcal{L}_t, u_t, \mathsf{SID}_t)$, where 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 linsertafter, ldelete or lreplace); and SID_t contains all the elements that were accessed by queries since update operation u_{t-1}.

```
$\mathcal{L}_{t+1} := U(\mathcal{L}_t, u_t)$                                          % 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 $r' \xleftarrow{\$} \mathbb{Z}_p^*$
        % $\Sigma_{\mathsf{Upd}}(+), \Omega_{\mathsf{Upd}}(+)$ contain information of elements to be added/replaced:
        $\Sigma_{\mathsf{Upd}}(+) := \langle\rangle$ and $\Omega_{\mathsf{Upd}}(+) := \langle(r', \perp)\rangle$
        % $\Sigma_{\mathsf{Upd}}(-), \Omega_{\mathsf{Upd}}(-)$ contain information of elements to be deleted:
        $\Sigma_{\mathsf{Upd}}(-) := \langle(t_{z\in\mathcal{L}_t}, \sigma_z)\rangle$ and $\Omega_{\mathsf{Upd}}(-) := \langle(r_z, \mathsf{tag}(z))\rangle$.
    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 $(\mathcal{L}_{t+1}, \mathsf{digest}_C^{t+1}, u_t, \mathsf{Upd}_{t+1}, \mathsf{state}_O)$
```

updated due to insertion (recall that $\mathcal{Y}$ is of amortized size $O(1)$). The owner also computes the member witness and a signature for the new x from scratch (called, x_{new} in the pseudo-code). Note that this step is equivalent to the steps in Setup for generating authentication units, only in this case it is for elements in $\mathcal{Y}$ and element x_{new} instead of the whole list. The owner then propagates these changes to the list digest signature $\sigma_{\mathcal{L}_t}$ as follows: (1) replaces signatures on the elements that have changed (i.e., elements in $\mathcal{Y}$); (2) adds a signature for x_{new}.

If an element x has been replaced with x' (i.e., lreplace operation), then the new member witness and signature is computed for x'. The owner propagates

Algorithm 4. $(\mathsf{digest}_S^{t+1}, \mathcal{L}_{t+1}) \leftarrow \mathbf{UpdateServer}(\mathsf{digest}_S^t, \mathsf{Upd}_{t+1}, \mathcal{L}_t, u_t)$, where u_t is an update to perform on $\mathcal{L}_t$ and Upd_{t+1} contains updates on authentication information generated by the owner.

```
1: Update the list: $\mathcal{L}_{t+1} := U(\mathcal{L}_t, u_t)$ where $|\mathcal{L}_{t+1}| = n'$.
2: Parse $\mathsf{Upd}_{t+1}$ as $\langle \sigma_{\mathcal{L}_{t+1}}, \mathcal{T}, \Sigma_{\mathsf{Upd}}, \Omega_{\mathsf{Upd}} \rangle$.
3: Compute $\Sigma_{\mathcal{L}_{t+1}}$: add/replace/delete elements from $\Sigma_{\mathsf{Upd}}$ in $\Sigma_{\mathcal{L}_t}$.
4: Compute $\Omega_{\mathcal{L}_{t+1}}$: add/replace/delete elements from $\Omega_{\mathsf{Upd}}$ 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:     $\mathsf{digest}_S^{t+1} := \langle \mathsf{pk}, \sigma_{\mathcal{L}_{t+1}}, \langle g, g^s, g^{s^2}, \ldots, g^{s^{n'}} \rangle, \Sigma_{\mathcal{L}_{t+1}}, \Sigma_{\mathcal{L}_{t+1}} \rangle$.
7: Else                    % $u_t$ does not cause regeneration of tags for all elements
8:     $\mathsf{digest}_S^{t+1} := (\mathsf{pk}, \sigma_{\mathcal{L}_{t+1}}, \langle g, g^s, g^{s^2}, \ldots, g^{s^n} \rangle, \Sigma_{\mathcal{L}_{t+1}}, \Omega_{\mathcal{L}_{t+1}})$ % where $g^{s^i}$ are from $\mathsf{digest}_S^t$
9: return $(\mathcal{L}_{t+1}, \mathsf{digest}_S^{t+1})$
```

these change to the list digest signature $\sigma_{\mathcal{L}_t}$ as follows: (1) adds a signature for x'; (2) removes the signature of the old element x.

In case when element z is deleted, the owner removes the signature of the old element in the list digest signature $\sigma_{\mathcal{L}_t}$ and re-randomizes the list digest signature with fresh randomness r'. Notice that, in case of insert and replace operations, the list digest signature gets re-randomized implicitly, since the new membership witness gets refreshed with fresh randomness.

As described so far, UpdateOwner has a viable leakage channel. Recall that an update operation changes authentication units of elements in the update u_t and $\mathcal{Y}$. Hence, if the client accesses an element in $\mathcal{Y}$, before and after the update, she 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 she did not query explicitly.

UpdateOwner achieves the zero-knowledge property as follows. We set f to be a function that takes client queries that have occurred since the last update and returns a set of elements accessed by them; recall that 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 with GenAuthTokens. We define a subroutine refresh which calls GenAuthTokens for each element in $\mathcal{Y}$ and SID_t, and returns $\sigma_{\mathsf{refresh}}$ which contains old signatures to be removed and new ones to be added to $\sigma_{\mathcal{L}_{t+1}}$. 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 implementation, where f is defined as a constant function, or where SID_t is not used, the UpdateOwner algorithm has to randomize member-witnesses and signatures for *all* the list elements.

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 4) to

Algorithm 5. $(\mathsf{answer}, \mathsf{proof}) \leftarrow \mathbf{Query}(\mathsf{digest}^t_S, \mathcal{L}_t, \delta)$, 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.

```
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:     $\Sigma_{\mathsf{answer}} := \langle \sigma_{\mathsf{answer}}, T, \lambda_{\mathcal{L}'} \rangle$ where $\mathcal{L}' = \mathcal{L}_t \setminus \delta$ and:
4:     $\sigma_{\mathsf{answer}} \leftarrow \prod_{y_j \in \mathsf{answer}} \sigma_{y_j}$.    % Digest signature for the query elements.
5:         $T = (t_{y_1 \in \mathcal{L}_t}, \ldots, t_{y_m \in \mathcal{L}_t})$.    % Member witnesses for query elements.
6:         Let $\mathcal{S}$ be a set of random elements w/o tags, i.e., introduced in $\Omega_{\mathcal{L}_t}$ due to Idelete.
7:         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 $\mathsf{digest}^t_S$.
8:     $\Omega_{\mathsf{answer}} = (t_{y_1 < y_2}, t_{y_2 < y_3}, \ldots, t_{y_{m-1} < y_m})$:
9:         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 $(\mathsf{answer}, \mathsf{proof})$
```

Algorithm 6. $b \leftarrow \mathbf{Verify}(\mathsf{pk}, \mathsf{digest}^t_C, \delta, \mathsf{answer}, \mathsf{proof})$.

```
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 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
5: If all equalities hold, then accept. Else reject.
```

propagate the update at its end. It uses u_t to update the list and Upd_{t+1} to add/substitute/remove authentication units in its digest.

Query Phase: The server has to perform two tasks when it receives query δ. It has to answer the query and compute the proof that the answer is correct. For the former step, it simply reorders the elements in δ according to their order in $\mathcal{L}_t$, sets answer to $\pi_{\mathcal{L}_t}(\delta)$. The latter step consists of proving that elements in answer are in $\mathcal{L}_t$ (i.e., membership) and that they are ordered correctly.

The detailed query phase is presented in Algorithm 5. In order to prove membership of every element in answer, the server uses its digest to obtain member witnesses $t_{y_j \in \mathcal{L}_t}$ and signatures σ_{y_j}, for each element $y_i \in \delta$, and includes them in the proof. It then proves that these y_is are indeed part of the source list $\mathcal{L}_t$ by computing the authentication digest for all elements not in the query. Let $\mathcal{L}' = \mathcal{L}_t \setminus \delta$. Then, computing the authentication digest, $\lambda_{\mathcal{L}'}$, is very similar to the computation of the list signature (i.e., client digest) by the owner albeit without using secret key v. That is, $\lambda_{\mathcal{L}'}$ is a product of hashed tags of elements in $\mathcal{L}'$ along with the hash of ω, given in the server digest.

The server proves the order condition as follows. 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. This part of the server digest, $t_{y_j < y_{j+1}}$, is used for verification in the equation $e(g^{\mathsf{tag}(y_j)^{r'}}, t_{y_j < y_{j+1}}) = e(g^{\mathsf{tag}(y_{j+1})^{r''}}, g)$.

The above steps preserve privacy and integrity of the scheme. In particular, $t_{y_j \in \mathcal{L}_t}$'s do not reveal element ranks since the witnesses have blinded using secret randomness during the setup. Furthermore, it is hard for the server to compute an invalid order witnesses $t_{y_j < y_{j+1}}$ as this would require computing $(g^{s^{-d}})^{r''/r'}$. This, in turn is (almost) equivalent to computing an inverse in the exponent, violating the P-BDHI assumption as a result.

Verification Phase: Given $(\mathsf{answer}, \mathsf{proof})$, the client uses Verify (Algorithm 6) and her copy of the list digest signature to verify answer. She checks the membership of elements in answer by using the properties of bilinear aggregate signatures. In particular she can verify the relationship of $\mathcal{L}' = \mathcal{L}_t \backslash \delta$ 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 to verify order witnesses as it lets her verify algebraic properties of the exponents, i.e., that $d = \mathsf{tag}(y_{j+1}) - \mathsf{tag}(y_j)$ for $t_{y_j \in \mathcal{L}_t} = g^{r' s^{\mathsf{tag}(y_j)}}$ and $t_{y_{j+1} \in \mathcal{L}_t} = g^{r'' s^{\mathsf{tag}(y_{j+1})}}$.

Extensions: $\mathsf{UpdateOwner}$ can be easily generalized to *batch updates* for optimization. Our construction can also hide from the client the fact that *an update has happened* via periodic updates and refreshes. The details are in [12].

Efficiency: Our construction uses efficient cryptographic operations: multiplication and exponentiation in prime order groups, 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. A member/order witness and a signature is a group element and is represented using $O(1)$ space (by standard convention, the word size is $\log(\mathsf{poly}(\mathsf{k}))$ and k is the security parameter). Theorem 1 summarizes the security and performance.

Theorem 1. *The dynamic privacy-preserving authenticated list (DPPAL) construction of Sect. 4 satisfies the security properties of DPPADS including completeness, soundness (under the P-BDHI assumption* [4]*) and zero-knowledge in the random oracle model (inherited from* [5]*). 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.*

Space Efficient DPPADS: The model of Sect. 3 assumes the owner himself updates his data structure and sends information to the server to propagate the changes. So, the owner is required to keep the most recent version of $\mathcal{D}_t$ and any associated auxiliary information. He gets the advantage of remaining offline during the query phase and gets online only during an update. But this may not be ideal for an owner with small memory requirement. So we propose a model that is space efficient and relies on an authenticated data structure (ADS) protocol executed between the owner and the server and give an instantiation in [12]. We summarize the performance in Theorem 2 below.

Theorem 2. *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.*

5 Dynamic Privacy-Preserving Authenticated Tree

We now propose a tree instantiation of DPPADS: a dynamic privacy-preserving tree (DPPAT) using a dynamic privacy-preserving authenticated list (Sect. 4). We only give the summary of the performance here, in Theorem 3, and defer the details of the construction to [12].

Order Queries: An *order query* on $\mathcal{T}$ is a pair of elements (x, y) from a tree $\mathcal{T}$. The corresponding answer is the pair rearranged according to their order in $\mathcal{T}$ along with a bit b indicating if the relation in ancestry or left-right (i.e., one node is to the left of the other with respect to their lowest common ancestor). For generality, the data structure also supports a *batch order query* where the returned answer is an *induced forest* of the queried elements.

DPPAT using DPPAL: A rooted tree $\mathcal{T}$ can be uniquely represented as two lists, L-Order$_{\mathcal{T}}$ and R-Order$_{\mathcal{T}}$, where the lists correspond to two different traversals of the tree constructed as follows. Both traversals start from the root, process each node they encounter, and recur on the subtrees of the current node. L-Order traverses subtrees left to right while R-Order traverses subtrees right to left. To construct a DPPAT, we construct two DPPAL's: on L-Order$_{\mathcal{T}}$ and R-Order$_{\mathcal{T}}$. DPPAT uses DPPAL to augment the answer with proofs of membership of x and y, and a proof of order. For a batch query of size m, the proof size is linear in the answer size, i.e., it is sufficient to prove $O(m)$ pairwise orders as we show in [12]. DPPAT can support all the dynamic operations, namely, link, cut and replace on $\mathcal{T}$ by making a *constant* number of update queries to the DPPALs of L-Order$_{\mathcal{T}}$ and R-Order$_{\mathcal{T}}$. We give the details in [12].

Theorem 3. *A dynamic privacy-preserving authenticated tree (DPPAT) can be implemented using a DPPAL. This scheme satisfies the security properties of a DPPADS: completeness, soundness and zero-knowledge. The runtime, space, and message size for every party is proportional to the corresponding runtime, space, and message size in the DPPAL scheme.*

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 special case of $d = 2$).

References

1. Ahn, J.H., Boneh, D., Camenisch, J., Hohenberger, S., Waters, B.: Computing on authenticated data. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 1–20. Springer, Heidelberg (2012)
2. Attrapadung, N., Libert, B., Peters, T.: Computing on authenticated data: new privacy definitions and constructions. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 367–385. Springer, Heidelberg (2012)
3. Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two simplified algorithms for maintaining order in a list. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 152–164. Springer, Heidelberg (2002)
4. Boneh, D., Boyen, X.: Efficient selective-ID secure identity-based encryption without random oracles. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 223–238. Springer, Heidelberg (2004)
5. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Biham, Eli (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 416–432. Springer, Heidelberg (2003)
6. Brzuska, C., et al.: Redactable signatures for tree-structured data: definitions and constructions. In: Zhou, J., Yung, M. (eds.) ACNS 2010. LNCS, vol. 6123, pp. 87–104. Springer, Heidelberg (2010)
7. Catalano, D., Fiore, D.: Vector commitments and their applications. In: PKC (2013)
8. Catalano, D., Fiore, D., Messina, M.: Zero-knowledge sets with short proofs. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 433–450. Springer, Heidelberg (2008)
9. Chang, E.-C., Lim, C.L., Xu, J.: Short redactable signatures using random trees. In: Fischlin, M. (ed.) CT-RSA 2009. LNCS, vol. 5473, pp. 133–147. Springer, Heidelberg (2009)
10. Chase, M., Healy, A., Lysyanskaya, A., Malkin, T., Reyzin, L.: Mercurial commitments with applications to zero-knowledge sets. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 422–439. Springer, Heidelberg (2005)
11. Devanbu, P.T., Gertz, M., Martel, C.U., Stubblebine, S.G.: Authentic third-party data publication. In: DBSec (2000)
12. Ghosh, E., Goodrich, M.T., Ohrimenko, O., Tamassia, R.: Fully-dynamic verifiable zero-knowledge order queries for network data. ePrint 2015/283 (2015)
13. Ghosh, E., Ohrimenko, O., Papadopoulos, D., Tamassia, R., Triandopoulos, N.: Zero-knowledge accumulators and set operations. ePrint 2015/404 (2015)

14. Ghosh, E., Ohrimenko, O., Tamassia, R.: Verifiable member and order queries on a list in zero-knowledge. In: ACNS (2015)
15. Ghosh, E., Ohrimenko, O., Tamassia, R.: Efficient verifiable range and closest point queries in zero-knowledge. PoPETs **2016**(4) (2016)
16. Goldberg, S., Naor, M., Papadopoulos, D., Reyzin, L., Vasant, S., Ziv, A.: NSEC5: provably preventing DNSSEC zone enumeration. In: NDSS (2015)
17. Goldreich, O.: The Foundations of Cryptography - Basic Applications, vol. 2. Cambridge University Press, Cambridge (2004)
18. Goodrich, M.T., Nguyen, D., Ohrimenko, O., Papamanthou, C., Tamassia, R., Triandopoulos, N., Lopes, C.V.: Efficient verification of web-content searching through authenticated web crawlers. PVLDB **5**(10), 920–931 (2012)
19. Itai, A., Konheim, A.G., Rodeh, M.: A sparse table implementation of priority queues. In: Even, S., Kariv, O. (eds.) Automata, Languages and Programming. LNCS, vol. 115, pp. 417–431. Springer, Heidelberg (1981)
20. Johnson, R., Molnar, D., Song, D., Wagner, D.: Homomorphic signature schemes. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 244–262. Springer, Heidelberg (2002)
21. Kundu, A., Atallah, M.J., Bertino, E.: Leakage-free redactable signatures. In: CODASPY (2012)
22. Kundu, A., Bertino, E.: Structural signatures for tree data structures. In: PVLDB (2008)
23. Kundu, A., Bertino, E.: Privacy-preserving authentication of trees and graphs. Int. J. Inf. Secur. **12**, 467–494 (2013)
24. Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In: Micciancio, D. (ed.) TCC 2010. LNCS, vol. 5978, pp. 499–517. Springer, Heidelberg (2010)
25. Liskov, M.: Updatable zero-knowledge databases. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 174–198. Springer, Heidelberg (2005)
26. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
27. Micali, S., Rabin, M.O., Kilian, J.: Zero-knowledge sets. In: FOCS (2003)
28. Naor, M., Teague, V.: Anti-presistence: history independent data structures. In: Proceedings on 33rd Annual ACM Symposium on Theory of Computing, 6–8 July 2001 (2001)
29. Naor, M., Ziv, A.: Primary-secondary-resolver membership proof systems. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015, Part II. LNCS, vol. 9015, pp. 199–228. Springer, Heidelberg (2015)
30. Ostrovsky, R., Rackoff, C., Smith, A.: Efficient consistency proofs for generalized queries on a committed database. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1041–1053. Springer, Heidelberg (2004)
31. Papadopoulos, D., Papamanthou, C., Tamassia, R., Triandopoulos, N.: Practical authenticated pattern matching with optimal proof size. PVLDB **8**(7), 750–761 (2015)
32. Poehls, H.C., Samelin, K., Posegga, J., De Meer, H.: Length-hiding redactable signatures from one-way accumulators in $O(n)$. Technical report MIP-1201, FIM. University of Passau (2012)
33. Pöhls, H.C., Samelin, K.: On updatable redactable signatures. In: Boureanu, I., Owesarski, P., Vaudenay, S. (eds.) ACNS 2014. LNCS, vol. 8479, pp. 457–475. Springer, Heidelberg (2014)
34. Prabhakaran, M., Xue, R.: Statistically hiding sets. In: Fischlin, M. (ed.) CT-RSA 2009. LNCS, vol. 5473, pp. 100–116. Springer, Heidelberg (2009)

35. Samelin, K., Pöhls, H.C., Bilzhause, A., Posegga, J., de Meer, H.: Redactable signatures for independent removal of structure and content. In: Ryan, M.D., Smyth, B., Wang, G. (eds.) ISPEC 2012. LNCS, vol. 7232, pp. 17–33. Springer, Heidelberg (2012)
36. Steinfeld, R., Bull, L., Zheng, Y.: Content extraction signatures. In: Kim, K. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 285–304. Springer, Heidelberg (2002)
37. Tamassia, R.: Authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003)
38. Wang, Z.: Improvement on Ahn et al.'s RSA P-homomorphic signature scheme. In: Keromytis, A.D., Di Pietro, R. (eds.) SecureComm 2012. LNICST, vol. 106, pp. 19–28. Springer, Heidelberg (2013)
39. Willard, D.E.: A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. Inf. Comput. **97**, 150–204 (1992)