Aditya Damodaran and Alfredo Rial*

Unlinkable Updatable Hiding Databases and Privacy-Preserving Loyalty Programs

Abstract: Loyalty programs allow vendors to profile buyers based on their purchase histories, which can reveal privacy sensitive information. Existing privacy-friendly loyalty programs force buyers to choose whether their purchases are linkable. Moreover, vendors receive more purchase data than required for the sake of profiling. We propose a privacy-preserving loyalty program where purchases are always unlinkable, yet a vendor can profile a buyer based on her purchase history, which remains hidden from the vendor. Our protocol is based on a new building block, an unlinkable updatable hiding database (HD), which we define and construct. HD allows the vendor to initialize and update databases stored by buyers that contain their purchase histories and their accumulated loyalty points. Updates are unlinkable and, at each update, the database is hidden from the vendor. Buyers can neither modify the database nor use old versions of it. Our construction for HD is practical for large databases.

Keywords: UC framework, vector commitments

DOI 10.2478/popets-2021-0039
Received 2020-11-30; revised 2021-03-15; accepted 2021-03-16.

1 Introduction

Loyalty programs (LPs) are marketing strategies implemented by vendors ($\mathcal{V}$s) to establish lasting relationships with buyers ($\mathcal{B}$s). LPs offer accumulating benefits to $\mathcal{B}$s who purchase certain brands or buy in certain shops [19]. LPs benefit $\mathcal{V}$s in two ways. First, they improve customer retention. Second, by storing $\mathcal{B}$s' purchase histories (PHs) and personal identifiable information (PII), and thanks to data mining techniques, LPs allow $\mathcal{V}$s to profile $\mathcal{B}$s. Those buyer profiles (BPs) are used for market research and targeted advertising.

LPs have raised privacy concerns because BPs can reveal sensitive PII, which $\mathcal{V}$s could sell to third-party advertisers and data brokers [21]. Those privacy concerns have been shown to have an impact on the participation in LPs [19].

Previous Work. LPs can be classified depending on the type of PII and PH collected, on whether they involve a single or multiple $\mathcal{V}$s, on the type of rewards given to $\mathcal{B}$s, on whether those rewards are transferable to other $\mathcal{B}$s, etc. For privacy's sake, the key issue is whether $\mathcal{V}$ requires collection of PHs or not.

Some privacy-preserving LPs (PPLPs) do not allow $\mathcal{V}$ to collect PHs [5, 6, 16, 28]. In [16], blind signatures are used to provide unlinkability between issuance and redemption of loyalty points (*lpts*). In [28], $\mathcal{B}$s can anonymously download a batch of loyalty cards and use them in an unlinkable manner. In [5], an updatable anonymous credential scheme is proposed that allows $\mathcal{B}$s to get $\mathcal{V}$ to add *lpts* at each purchase without revealing the total number of *lpts* accumulated. Recently, in [6], the scheme in [5] is improved to add some features such as recoverability of failed spent *lpts* or backward unlinkability. Despite offering privacy protection to $\mathcal{B}$s, these solutions prevent any form of buyer profiling, and thus they limit the benefits that $\mathcal{V}$s obtain from LPs.

Other solutions do allow $\mathcal{V}$ to obtain some information on PHs [4, 29, 32]. In [29, 32], $\mathcal{B}$ obtains an anonymous credential at registration. At each purchase, $\mathcal{B}$ shows her anonymous credential and chooses whether her purchases are linkable or unlinkable to previous purchases. In [4], $\mathcal{B}$, at each purchase, obtains a blind signature. This signature signs purchase data along with a value chosen by $\mathcal{B}$. To obtain *lpts*, $\mathcal{B}$ can link signatures of different purchases when they sign the same $\mathcal{B}$-chosen value. $\mathcal{B}$ then reveals those signatures to $\mathcal{V}$. Therefore, in [4, 29, 32], at each purchase, $\mathcal{B}$s need to decide whether they want the purchases to be linkable to previous purchases. The PPLPs in [4, 29, 32] have two drawbacks.

Privacy of PHs. Profiling consists in running a classification algorithm over a $\mathcal{B}$'s PH. A PPLP should minimize the disclosure of PHs. However, in [4, 29, 32], when $\mathcal{B}$s choose to link their purchases in order to benefit from the LP, $\mathcal{V}$ receives more information on PHs than is actually needed to create a BP. In [29, 32], all the PH is revealed, while in [4] a

Aditya Damodaran, Alfredo Rial: SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg.
E-mail: firstname.lastname@uni.lu

taxonomy of products and product types is defined and $\mathcal{B}$s can reveal the concrete product bought or the product category.

Unlinkability of Purchases. In [4, 29, 32], $\mathcal{B}$s need to decide whether purchases are linkable or unlinkable at the moment of purchase. It would be desirable to have a scheme where purchases are always unlinkable, and yet at a later stage $\mathcal{B}$s can be profiled based on their PH and benefit from *lpts*.

Our Contribution. We propose a PPLP that addresses those drawbacks. First, it protects $\mathcal{B}$'s privacy by avoiding the disclosure of PHs to $\mathcal{V}$. Second, it guarantees that purchases are always unlinkable. PHs and *lpts* are stored on $\mathcal{B}$'s side and can later be used for profiling or to redeem *lpts*.

Our PPLP involves a vendor $\mathcal{V}$ and multiple buyers $\mathcal{B}_k$ and can be used in e-commerce and physical shops. $\mathcal{B}$s are equipped with an electronic device (e.g., a smartphone). The PPLP consists of the following phases.

Registration. $\mathcal{B}_k$ signs up for the PPLP.

Purchase. $\mathcal{B}_k$'s PH and *lpts* are stored on $\mathcal{B}_k$'s side. Each purchase is unlinkable to previous ones. Despite of unlinkability, at each purchase, $\mathcal{V}$ updates $\mathcal{B}_k$'s PH and *lpts* without learning any of them.

Redemption. $\mathcal{B}_k$ proves that she has accumulated a number of *lpts*.

Profiling. $\mathcal{B}_k$ reveals to $\mathcal{V}$ the result of running a profiling algorithm on her PH (or, in general, any computation run on her PH) and proves that this result is correct without disclosing her PH. We describe a variant of this phase where $\mathcal{V}$ does not learn the profiling result either, but is able to use it to, e.g., give $\mathcal{B}_k$ more *lpts*.

To describe our PPLP, we use a new building block, an unlinkable updatable hiding database (HD), which could be used in other privacy-preserving protocols. An HD is a protocol between an updater $\mathcal{U}$ and multiple readers $\mathcal{R}_k$. (In our PPLP, $\mathcal{V}$ (resp. $\mathcal{B}_k$) plays the role of $\mathcal{U}$ (resp. $\mathcal{R}_k$).) An HD consists of an update phase and a read phase. In an update phase, $\mathcal{U}$ updates the database stored by a reader $\mathcal{R}_k$ identified by a pseudonym P. We consider simple databases DB with entries of the form $[i, vr_i]$, where i is the position and vr_i the value stored at position i. (In §E, we show how to modify our definition and construction for HD to use databases of the form $[i, vr_{i,1}, \dots, vr_{i,m}]$, i.e., databases where a tuple of values is stored in each entry.) $\mathcal{U}$ does not learn the contents of DB and cannot link it to previous updates. However, $\mathcal{U}$ is ensured that he is updating the last version of DB given to $\mathcal{R}_k$. In the read phase, $\mathcal{R}_k$ commits to some of the data from the database and proves to $\mathcal{U}$ that it is stored in DB. Later, $\mathcal{R}_k$ can use those commitments to prove in zero-knowledge (ZK) other statements about the data in DB. For instance, $\mathcal{R}_k$ can prove the correctness of the result of running an algorithm on input those data. One key property of HD is that $\mathcal{R}_k$ is able to prove in ZK statements about both vr_i and i.

In §3.1, we define security for HD in the universal composability (UC) framework [13]. We define an ideal functionality $\mathcal{F}_{\text{HD}}$ for HD that is suitable for modular design, i.e., that can be used as a building block of a protocol. In the UC framework, a protocol can be described modularly in the hybrid model, where parties use ideal functionalities for the building blocks of the protocol. However, many functionalities in the literature cannot be used for modular design whenever we must guarantee that two or more functionalities receive the same input. To solve this issue, we use the method in [8], which is based on sending committed inputs to functionalities.

We propose a construction Π_{HD} for HD in §3.3. Π_{HD} uses a vector commitment (VC) scheme as main building block. A VC scheme allows us to compute a vector commitment vc to a vector $\mathbf{x}$ of values, where each value $\mathbf{x}[i]$ is stored at a given position i. Additionally, vc can be opened to $\mathbf{x}[i]$ with communication cost independent of the vector length. We use a VC scheme that is additively homomorphic. Π_{HD} also uses a pseudonymous channel that provides unlinkability in communications between any $\mathcal{R}_k$ and $\mathcal{U}$.

A VC vc is used to store the database DB by committing to a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vr_i$. Initially, $\mathcal{R}_k$ obtains a signed vc to an initial database DB from $\mathcal{U}$. To read DB in an unlinkable manner, $\mathcal{R}_k$ rerandomizes vc to vc', reveals vc' to $\mathcal{U}$ and proves in ZK that vc' is a rerandomized version of a commitment signed by $\mathcal{U}$. Then $\mathcal{R}_k$ can prove statements about the database committed in vc'. To update DB, $\mathcal{U}$ uses the homomorphic property of the VC scheme to update the database in vc' (without learning its contents) and produce an updated VC vc_u, which $\mathcal{U}$ signs and sends to $\mathcal{R}_k$. Π_{HD} implements a double-spending detection mechanism to ensure that, in the next read phase, $\mathcal{R}_k$ uses vc_u and not previous commitments received from $\mathcal{U}$.

To construct a PPLP based on HD, we store $\mathcal{B}_k$'s PH in the form of a database DB that, for each product (or product category, depending on the detail needed to create BPs), stores the number of purchases or the amount of money spent by $\mathcal{B}_k$ on that product. Therefore, DB is a vector $\mathbf{x}$ where each position i represents a product and $\mathbf{x}[i]$ is the number of purchases or the amount of money spent on that product. An additional position is used to store the accumulated *lpts*. When $\mathcal{B}_k$ purchases some products, to update DB, $\mathcal{V}$ computes another VC to a vector $\mathbf{x}_u$ that stores the information regarding the products purchased and additional *lpts*. Then, by using the homomorphic property, $\mathcal{V}$ multiplies both vector commitments to get a VC to $\mathbf{x} + \mathbf{x}_u$ and sends the new VC to $\mathcal{B}_k$. We describe our PPLP in §4.

Our PPLP guarantees unlinkability between purchases and minimizes the PH disclosure towards $\mathcal{V}$. Thanks to double-spending detection, our PPLP ensures that $\mathcal{B}_k$ uses the last version of the PH at the next update or when reading it for profiling or redeeming *lpts*. Nevertheless, we note that $\mathcal{B}_k$ can refuse to use the PPLP for certain purchases or create more than one profile. Allowing users to curate their own profiles might yield profiles better suited for some purposes, such as providing personalized discounts, but, if needed, it can be discouraged (see §4).

Π_{HD} benefits from the efficiency properties of VCs. Particularly, updating the database is very efficient thanks to the homomorphic property. In §3.4, we show that the communication cost of read and update operations, as well as the computation cost of update operations, is independent of the database size N. Read operations have amortized computation cost that is also independent of N. Therefore, Π_{HD} is suitable for large databases. In §4.3, we implement our PPLP and analyze its efficiency.

2 Modular Design in UC

We summarize the UC framework in §A. We use the method in [8] to allow $\mathcal{F}_{\mathrm{HD}}$ to be used as building block in our PPLP protocol in §4, which we describe modularly in the hybrid model. This method allows us to ensure, when needed, that $\mathcal{F}_{\mathrm{HD}}$ and other functionalities receive the same input. In [8], a functionality $\mathcal{F}_{\mathrm{NIC}}$ for non-interactive commitments is proposed. $\mathcal{F}_{\mathrm{NIC}}$ consists of four interfaces:

1. Any party $\mathcal{P}_i$ uses the nic.setup interface to set up the functionality.
2. Any party $\mathcal{P}_i$ uses the nic.commit interface to send a message m and obtain a commitment c and an opening o. A commitment c consists of $(c', pnic, \mathsf{NIC.Vf})$, where c' is the commitment, $pnic$ are the public parameters, and $\mathsf{NIC.Vf}$ is the verification algorithm.
3. Any party $\mathcal{P}_i$ uses the nic.validate interface to send a commitment c to check that c contains the correct $pnic$ and $\mathsf{NIC.Vf}$.
4. Any party $\mathcal{P}_i$ uses the nic.verify interface to send (c, m, o) to verify that c is a commitment to m with opening o.

$\mathcal{F}_{\mathrm{NIC}}$ can be realized by a perfectly hiding commitment scheme, such as Pedersen commitments [8]. To ensure that a party $\mathcal{P}_i$ sends the same input m to several ideal functionalities, $\mathcal{P}_i$ first uses nic.commit to get a commitment c to m with opening o. Then $\mathcal{P}_i$ sends (c, m, o) as input to each of the functionalities, and each functionality runs $\mathsf{NIC.Vf}$ to verify the commitment. Finally, other parties in the protocol receive the commitment c from each of the functionalities and use the nic.validate interface to validate c. Then, if c received from all the functionalities is the same, the binding property provided by $\mathcal{F}_{\mathrm{NIC}}$ ensures that all the functionalities received the same input m.

3 Our Hiding Database

3.1 Ideal Functionality for HD

$\mathcal{F}_{\mathrm{HD}}$ interacts with readers $\mathcal{R}_k$ and an updater $\mathcal{U}$. $\mathcal{F}_{\mathrm{HD}}$ maintains one database DB per reader. Each DB consists of N entries of the form $[i, vr_i]$. $\mathcal{F}_{\mathrm{HD}}$ has two interfaces hd.read in Fig. 1 and hd.update in Fig. 2:

1. The reader $\mathcal{R}_k$ sends the hd.read.ini message on input a pseudonym P and $(i, vr_i, c_i, o_i, cr_i, or_i)_{i \in \mathbb{S}}$, where $[i, vr_i]$ is a DB entry and (c_i, o_i) and (cr_i, or_i) are commitments and openings to i and vr_i.
   - If no DB is stored for $\mathcal{R}_k$, $\mathcal{F}_{\mathrm{HD}}$ sends P and $f = 1$ to $\mathcal{U}$ to inform $\mathcal{U}$ that a reader with pseudonym P requests a database be set up.
   - If a DB for $\mathcal{R}_k$ is stored, and $\mathbb{S} = \emptyset$, $\mathcal{F}_{\mathrm{HD}}$ sends P and $f = 0$ to $\mathcal{U}$. In this case, $\mathcal{R}_k$ does not read any entry and simply wants that $\mathcal{U}$ updates her database.
   - If a DB for $\mathcal{R}_k$ is stored, and $\mathbb{S} \neq \emptyset$, for all $i \in \mathbb{S}$, $\mathcal{F}_{\mathrm{HD}}$ verifies that $[i, vr_i]$ is stored in DB and verifies the commitments (c_i, o_i) and (cr_i, or_i). Then $\mathcal{F}_{\mathrm{HD}}$ sends P, $f = 0$ and $(c_i, cr_i)_{i \in \mathbb{S}}$ to $\mathcal{U}$. In this case, $\mathcal{R}_k$ reads some entries from her database and sends the commitments to the entries read to $\mathcal{U}$.
2. $\mathcal{U}$ sends the hd.update.ini message on input P and $(i, vu_i)_{i \in [1,N]}$. This interface can only be invoked by $\mathcal{U}$ as a response to a previous invocation of the read interface by the reader with pseudonym P. $\mathcal{F}_{\mathrm{HD}}$ recovers the flag f sent previously to $\mathcal{U}$.

$\mathcal{F}_{\mathrm{HD}}$ is parameterised by a database size N, a universe of pseudonyms $\mathbb{U}_p$, a universe of values $\mathbb{U}_v$ and an operator $\odot : \mathbb{U}_v \times \mathbb{U}_v \to \mathbb{U}_v$.

1. On input $(\mathsf{hd.read.ini}, sid, P, (i, vr_i, c_i, o_i, cr_i, or_i)_{i \in \mathbb{S}})$ from $\mathcal{R}_k$:
   - Abort if $sid \notin (\mathcal{U}, sid')$, or if $P \notin \mathbb{U}_p$.
   - Abort if there is a tuple $(sid, P', \mathcal{R}'_k, f)$ stored such that $P' = P$, or such that $\mathcal{R}'_k = \mathcal{R}_k$ and $f \neq \perp$.
   - If a tuple $(sid, \mathcal{R}_k, \mathsf{DB})$ is not stored, set $f \leftarrow 1$, else set $f \leftarrow 0$.
   - If $f = 0$ and $\mathbb{S} \neq \emptyset$, for all $i \in \mathbb{S}$, do the following:
     - Abort if $i \notin [1, N]$, or if $vr_i \notin \mathbb{U}_v$, or if $[i, \mathbb{U}_v]$ is not stored in DB.
     - Parse c_i as $(c'_i, pnic, \mathsf{NIC.Vf})$ and cr_i as $(cr'_i, pnic, \mathsf{NIC.Vf})$.
     - Abort if $\mathsf{NIC.Vf}$ is not a ppt algorithm.
     - Abort if $1 \neq \mathsf{NIC.Vf}(pnic, c'_i, i, o_i)$ or if $1 \neq \mathsf{NIC.Vf}(pnic, cr'_i, vr_i, or_i)$.
   - Create fresh qid, store $(qid, P, \mathcal{R}_k, f, (c_i, cr_i)_{i \in \mathbb{S}})$ and send $(\mathsf{hd.read.sim}, sid, qid, (c_i, cr_i)_{i \in \mathbb{S}})$ to $\mathcal{S}$.

S. On input $(\mathsf{hd.read.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{R}_k, f, (c_i, cr_i)_{i \in \mathbb{S}})$ such that $qid = qid'$ is not stored.
   - Store $(sid, P, \mathcal{R}_k, f)$, delete $(qid, P, \mathcal{R}_k, f, (c_i, cr_i)_{i \in \mathbb{S}})$ and send $(\mathsf{hd.read.end}, sid, P, f, (c_i, cr_i)_{i \in \mathbb{S}})$ to $\mathcal{U}$.

Fig. 1. Functionality $\mathcal{F}_{\mathrm{HD}}$: interface hd.read

- If $f = 1$, $\mathcal{F}_{\mathrm{HD}}$ stores $(i, vu_i)_{i \in [1,N]}$ as the database of the reader $\mathcal{R}_k$ associated with P and sends $(i, vu_i)_{i \in [1,N]}$ to $\mathcal{R}_k$.
- If $f = 0$, $\mathcal{F}_{\mathrm{HD}}$ replaces the database $(i, vr_i)_{i \in [1,N]}$ of the reader $\mathcal{R}_k$ associated with P by $(i, vr_i \odot vu_i)_{i \in [1,N]}$. $\odot$ represents a generic operation to be instantiated in each construction for $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{F}_{\mathrm{HD}}$ sends $(i, vu_i)_{i \in [1,N]}$ to $\mathcal{R}_k$. $(i, vu_i)_{i \in [1,N]}$ could equal $(i, 0)_{i \in [1,N]}$, where 0 is the identity element for $\odot$. In this case, the database is not updated after the read operation. Still, $\mathcal{U}$ must invoke the update interface after a read operation so that $\mathcal{R}_k$ can perform the next read operation.

When invoked by $\mathcal{U}$ or $\mathcal{R}_k$, $\mathcal{F}_{\mathrm{HD}}$ first checks the correctness of the input and aborts if it does not belong to the correct domain. $\mathcal{F}_{\mathrm{HD}}$ also aborts if an interface is invoked at an incorrect moment in the protocol. For example, $\mathcal{R}_k$ cannot invoke hd.read if $\mathcal{R}_k$ is awaiting an update operation.

The session identifier sid has the structure $(\mathcal{U}, sid')$ so that each updater can have its instance of $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{F}_{\mathrm{HD}}$ implicitly checks that sid in a message equals the one received in the first invocation. Before $\mathcal{F}_{\mathrm{HD}}$ queries the simulator $\mathcal{S}$, $\mathcal{F}_{\mathrm{HD}}$ saves its state, which is recovered when receiving a response from $\mathcal{S}$. To match a query to a response, $\mathcal{F}_{\mathrm{HD}}$ creates a query identifier qid.

$\mathcal{F}_{\mathrm{HD}}$ guarantees the following properties:

Unlinkability. The read operations remain unlinkable towards $\mathcal{U}$. $\mathcal{F}_{\mathrm{HD}}$ reveals to $\mathcal{U}$ a pseudonym P rather than the identifier $\mathcal{R}_k$, and $\mathcal{F}_{\mathrm{HD}}$ checks that P is unique for each read operation.

Hiding. $\mathcal{U}$ does not learn the contents of any DB. $\mathcal{U}$ sets the initial values of each DB but, because read operations are unlinkable, afterwards $\mathcal{U}$ cannot keep track of the updates performed on each DB and thus does not know their contents. Additionally, when reading, $\mathcal{R}_k$ sends $\mathcal{U}$ commitments to database entries, which are hiding as guaranteed by $\mathcal{F}_{\mathrm{NIC}}$. When $\mathcal{F}_{\mathrm{HD}}$ is used as building block of a protocol, $\mathcal{R}_k$ can prove in ZK statements about the committed values rather than revealing them.

Unforgeability. $\mathcal{R}_k$ cannot modify her database DB or read from DB entries that were not stored and updated by $\mathcal{U}$. Additionally, $\mathcal{R}_k$ cannot make $\mathcal{U}$ update an old version of DB.

3.2 Building Blocks of our Construction

We describe here the building blocks of Π_{HD}. We define their security in §B.

$\underline{\mathcal{F}_{\mathrm{REG}}}$. Π_{HD} uses the functionality $\mathcal{F}_{\mathrm{REG}}$ for key registration in [13]. $\mathcal{F}_{\mathrm{REG}}$ interacts with any party $\mathcal{T}$ that registers a message v and with any party $\mathcal{P}$ that retrieves the registered message. $\mathcal{F}_{\mathrm{REG}}$ consists of two interfaces reg.register and reg.retrieve. $\mathcal{T}$ uses reg.register to register a message v with $\mathcal{F}_{\mathrm{REG}}$. $\mathcal{P}$ uses reg.retrieve to retrieve v from $\mathcal{F}_{\mathrm{REG}}$.

2. On input $(\mathsf{hd.update.ini}, sid, P, (i, vu_i)_{i\in[1,N]})$ from $\mathcal{U}$:
   - Abort if $(sid, P', \mathcal{R}_k, f)$ such that $P' = P$ and $f \neq \perp$ is not stored, or if, for $i \in [1, N]$, $vu_i \notin \mathbb{U}_v$.
   - If $f = 1$, set $\mathsf{DB} \leftarrow (i, vu_i)_{i\in[1,N]}$ and store a tuple $(sid, \mathcal{R}_k, \mathsf{DB})$.
   - If $f = 0$, take the stored tuple $(sid, \mathcal{R}_k, \mathsf{DB})$, parse DB as $(i, vr_i)_{i\in[1,N]}$ and update $\mathsf{DB} \leftarrow (i, vr_i \odot vu_i)_{i\in[1,N]}$ in the tuple $(sid, \mathcal{R}_k, \mathsf{DB})$.
   - Update $f \leftarrow \perp$ in the tuple $(sid, P', \mathcal{R}_k, f)$ with $P' = P$.
   - Create fresh qid, store $(qid, \mathcal{R}_k, P, (i, vu_i)_{i\in[1,N]})$ and send $(\mathsf{hd.update.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{hd.update.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if a tuple $(qid', \mathcal{R}_k, P, (i, vu_i)_{i\in[1,N]})$ such that $qid = qid'$ is not stored.
   - Delete $(qid', \mathcal{R}_k, P, (i, vu_i)_{i\in[1,N]})$ such that $qid = qid'$ and send $(\mathsf{hd.update.end}, sid, P, (i, vu_i)_{i\in[1,N]})$ to $\mathcal{R}_k$.

Fig. 2. Functionality $\mathcal{F}_{\mathrm{HD}}$: interface hd.update

$\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. Π_{HD} uses the functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ for common reference string generation in [13]. $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ interacts with any parties $\mathcal{P}$ that obtain the common reference string, and consists of one interface crs.get. $\mathcal{P}$ uses crs.get to request and receive crs from $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. In the first invocation, $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ generates crs by running algorithm CRS.Setup.

$\mathcal{F}_{\mathrm{NYM}}$. Π_{HD} uses the functionality $\mathcal{F}_{\mathrm{NYM}}$, which models an idealized secure pseudonymous channel. We use $\mathcal{F}_{\mathrm{NYM}}$ to describe our protocol in order to abstract away the details of real-world pseudonymous channels. $\mathcal{F}_{\mathrm{NYM}}$ is similar to the functionality for anonymous secure message transmission in [11]. $\mathcal{F}_{\mathrm{NYM}}$ interacts with senders $\mathcal{T}_k$ and a replier $\mathcal{R}$ and consists of two interfaces nym.send and nym.reply. $\mathcal{T}_k$ uses nym.send to send a message m and a pseudonym P to $\mathcal{R}$. $\mathcal{R}$ uses nym.reply to send a message m and a pseudonym P. $\mathcal{F}_{\mathrm{NYM}}$ checks if there is a party $\mathcal{T}_k$ associated with P that is awaiting a reply, and in that case sends m and P to $\mathcal{T}_k$. Therefore, $\mathcal{R}$ replies to messages from $\mathcal{T}_k$ by specifying P.

$\mathcal{F}_{\mathrm{ZK}}^{R}$. Let R be a polynomial time computable binary relation. For tuples $(wit, ins) \in R$ we call wit the witness and ins the instance. Π_{HD} uses a functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$ for zero-knowledge. $\mathcal{F}_{\mathrm{ZK}}^{R}$ runs with multiple provers $\mathcal{P}_k$ and a verifier $\mathcal{V}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ follows the functionality for ZK in [13], except that a prover $\mathcal{P}_k$ is identified by a pseudonym P towards $\mathcal{V}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ consists of one interface zk.prove. $\mathcal{P}_k$ uses zk.prove to send a witness wit, an instance ins and a pseudonym P to $\mathcal{F}_{\mathrm{ZK}}^{R}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ checks whether $(wit, ins) \in R$, and, in that case, sends ins and P to $\mathcal{V}$.

Commitment Schemes. A commitment scheme consists of algorithms C.Setup, C.Com and C.Vf. $\mathsf{C.Setup}(1^k)$ generates the parameters par_c, which include a description of the message space $\mathcal{M}$. $\mathsf{C.Com}(par_c, x)$ outputs a commitment c to $x \in \mathcal{M}$ and an opening o. $\mathsf{C.Vf}(par_c, c, x, o)$ outputs 1 if c is a commitment to x with opening o or 0 otherwise. A commitment scheme must be hiding and binding. We use a commitment scheme that is additively homomorphic. Given two commitments c_1 and c_2 to messages x_1 and x_2 with openings o_1 and o_2, there exists an operation $\cdot$ such that $c \leftarrow c_1 \cdot c_2$ is a commitment to $x = x_1 + x_2$ with opening $o = o_1 + o_2$. If $x_2 = 0$, then c is a rerandomization of c_1, which we denote by $c \leftarrow \mathsf{C.Rerand}(c_1, o_2)$.

Vector Commitments. Vector commitments (VC) allow us to commit to a vector of messages and to open the commitment to one of the messages with an opening size independent of the vector length [14, 27]. A VC scheme consists of the algorithms VC.Setup, VC.Com, VC.Open and VC.Vf. On input an upper bound ℓ for the vector length, $\mathsf{VC.Setup}(1^k, \ell)$ generates the parameters par, which include a description of the message space $\mathcal{M}$ and of the randomness space $\mathcal{R}$. $\mathsf{VC.Com}(par, \mathbf{x}, r)$ outputs a commitment vc to a vector $\mathbf{x} \in \mathcal{M}^{\ell}$ with randomness $r \in \mathcal{R}$. $\mathsf{VC.Open}(par, i, \mathbf{x}, r)$ computes an opening w for $\mathbf{x}[i]$. $\mathsf{VC.Vf}(par, vc, x, i, w)$ outputs 1 if w is a valid opening for x being at position i and 0 otherwise. A VC scheme must be hiding and binding. We use VC schemes that are additively homomorphic. Given two commitments vc_1 and vc_2 to vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ with randomness r_1 and r_2, there exists an operation $\cdot$ such that $vc \leftarrow vc_1 \cdot vc_2$ is a commitment to $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ with $r = r_1 + r_2$. (If vc_2 is a commitment to the vector $\mathbf{x}[i] = 0$ for all $i \in [1, \ell]$, then vc is a rerandomization of vc_1, which we denote by $vc = \mathsf{VC.Rerand}(vc_1, r_2)$.) In addition, if $w_{1,i}$ is a witness for $\mathbf{x}_1[i]$ being a position i in the vector $\mathbf{x}_1$ committed in vc_1, and $w_{2,i}$ is an opening for $\mathbf{x}_2[i]$ being a position i in the vector $\mathbf{x}_2$ committed

Π_{HD} is parameterized by a database size N, a universe of pseudonyms $\mathbb{U}_p$, which parameterizes $\mathcal{F}_{\mathrm{NYM}}$, and a universe of values $\mathbb{U}_v$ which is given by the message space of the VC scheme.

1. On input $(\mathsf{hd.read.ini}, sid, P, (i, vr_i, c_i, o_i, cr_i, or_i)_{i \in \mathbb{S}})$, $\mathcal{R}_k$ and $\mathcal{U}$ do the following:
   - If a tuple $(sid, par, par_c, pk, vc, \mathbf{x}, r, c, s, o, sig)$ is not stored:
     - $\mathcal{R}_k$ uses $\mathsf{crs.get}$ to obtain the VC parameters par and the commitment parameters par_c.
     - $\mathcal{R}_k$ picks a random value $s_1 \leftarrow \mathcal{M}$, computes a commitment $(c_1, o_1) \leftarrow \mathsf{C.Com}(par_c, s_1)$ and stores $(sid, par, par_c, c_1, s_1, o_1)$.
     - $\mathcal{R}_k$ picks a random pseudonym $P \leftarrow \mathbb{U}_p$ and uses the $\mathsf{nym.send}$ interface of $\mathcal{F}_{\mathrm{NYM}}$ on input P to send c_1 to $\mathcal{U}$.
     - If P was received before, $\mathcal{U}$ aborts.
     - $\mathcal{U}$ sets a flag $f \leftarrow 1$ (because a single commitment c_1 is received) and $(c_i, cr_i)_{i \in \mathbb{S}} \leftarrow \perp$.
     - $\mathcal{U}$ stores (sid, P, f, c_1).
   - If a tuple $(sid, par, par_c, pk, vc, \mathbf{x}, r, c, s, o, sig)$ is stored:
     - $\mathcal{R}_k$ picks randomly $r_2 \leftarrow \mathcal{R}$, sets $r' \leftarrow r + r_2$ and runs $vc' \leftarrow \mathsf{VC.Rerand}(vc, r_2)$ to rerandomize vc.
     - For all $i \in \mathbb{S}$, $\mathcal{R}_k$ does the following:
       * Abort if $\mathbf{x}[i] \neq vr_i$.
       * Compute an opening $w_i \leftarrow \mathsf{VC.Open}(par, i, \mathbf{x}, r')$.
       * Parse the commitments c_i as $(c'_i, pnic, \mathsf{NIC.Vf})$ and cr_i as $(cr'_i, pnic, \mathsf{NIC.Vf})$.
     - $\mathcal{R}_k$ picks random o_2, sets $o' \leftarrow o + o_2$ and runs $c' \leftarrow \mathsf{C.Rerand}(c, o_2)$ to rerandomize c.
     - $\mathcal{R}_k$ sets as follows the witness $wit_r \leftarrow (sig, vc, c, r_2, o_2, \langle i, vr_i, w_i, o_i, or_i \rangle_{i \in \mathbb{S}})$ and $ins_r \leftarrow (pk, par, par_c, vc', c', pnic, \langle c'_i, cr'_i \rangle_{i \in \mathbb{S}})$.
     - $\mathcal{R}_k$ replaces vc by vc' and r by r' in the stored tuple $(sid, par, par_c, pk, vc, \mathbf{x}, r, c, s, o, sig)$.
     - $\mathcal{R}_k$ picks randomly a pseudonym $P \leftarrow \mathbb{U}_p$ and uses the $\mathsf{zk.prove}$ interface to send wit_r, ins_r and P to $\mathcal{F}_{\mathrm{ZK}}^{R_r}$, where R_r is

$$R_r = \{(wit_r, ins_r):$$
$$1 = \mathsf{S.Vf}(pk, sig, \langle vc, c \rangle) \ \wedge \ vc' = \mathsf{VC.Rerand}(vc, r_2) \ \wedge \ c' = \mathsf{C.Rerand}(c, o_2) \ \wedge \quad (1)$$
$$\{1 = \mathsf{NIC.Vf}(pnic, c'_i, i, o_i) \ \wedge \ 1 = \mathsf{NIC.Vf}(pnic, cr'_i, vr_i, or_i) \ \wedge \quad (2)$$
$$1 = \mathsf{VC.Vf}(par, vc', vr_i, i, w_i)\ \}_{\forall i \in \mathbb{S}}\ \} \quad (3)$$

       In equation 1, $\mathcal{R}_k$ proves that vc and c were signed by $\mathcal{U}$, and that vc' and c' are rerandomizations of vc and c. In equation 2, $\mathcal{R}_k$ proves that c'_i and cr'_i commit to i and vr_i respectively. In equation 3, $\mathcal{R}_k$ proves that $\mathbf{x}[i] = vr_i$, where $\mathbf{x}$ is the vector committed in vc'.
     - $\mathcal{U}$ receives ins_r and P. $\mathcal{U}$ aborts if P was received before. $\mathcal{U}$ checks that the signing public key, VC parameters and commitment parameters in ins_r are equal to those stored in the tuple $(sid, par, par_c, pk, sk)$. $\mathcal{U}$ sets a flag $f \leftarrow 0$ and stores (sid, P, f, vc', c').
     - $\mathcal{U}$ sends a message (Open c') to $\mathcal{U}$ by using the $\mathsf{nym.reply}$ interface of $\mathcal{F}_{\mathrm{NYM}}$ on input P. (Here we consider that any construction for $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ uses $\mathcal{F}_{\mathrm{NYM}}$, so $\mathcal{U}$ can reply through $\mathcal{F}_{\mathrm{NYM}}$.)
     - $\mathcal{R}_k$ picks a random value $s_1 \leftarrow \mathcal{M}$, computes a commitment $(c_1, o_1) \leftarrow \mathsf{C.Com}(par_c, s_1)$ and stores (sid, c_1, s_1, o_1).
     - $\mathcal{R}_k$ uses the $\mathsf{nym.send}$ interface of $\mathcal{F}_{\mathrm{NYM}}$ on input P to send the message and the opening (s, o') of c' and a new commitment c_1 to $\mathcal{U}$.
     - $\mathcal{U}$ verifies the opening by running $\mathsf{C.Vf}(par_c, c', s, o')$. $\mathcal{U}$ checks that s has never been received before, which ensures that the reader uses the last version of the database $\mathbf{x}$ in vc. $\mathcal{U}$ replaces c' by c_1 in the stored tuple (sid, P, f, vc', c_1).
   - $\mathcal{U}$ outputs $(\mathsf{hd.read.end}, sid, P, f, (c_i, cr_i)_{i \in \mathbb{S}})$.

Fig. 3. Construction Π_{HD}: interface hd.read

2. On input $(\mathsf{hd.update.ini}, sid, P, (i, vu_i)_{i\in[1,N]})$, $\mathcal{U}$ and $\mathcal{R}_k$ do the following
   - $\mathcal{U}$ aborts if there is not a tuple $(sid, P', f, \ldots)$ such that $P = P'$.
   - If $(sid, par, par_c, pk, sk)$ is not stored, $\mathcal{U}$ obtains par and par_c from $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. $\mathcal{U}$ computes a signing key pair $(pk, sk) \leftarrow \mathsf{S.KG}(1^k)$ and uses $\mathsf{reg.register}$ to register pk. $\mathcal{U}$ stores $(sid, par, par_c, pk, sk)$.
   - If $f = 1$, do the following:
     - $\mathcal{U}$ takes the stored tuple (sid, P, f, c_1).
     - $\mathcal{U}$ sets a vector $\mathbf{x}[i] \leftarrow vu_i$ (for all $i \in [1, N]$) and runs $vc \leftarrow \mathsf{VC.Com}(par, \mathbf{x}, 0)$ with 0 as randomness.
     - $\mathcal{U}$ picks a random value s_2, computes a commitment $(c_2, 0) \leftarrow \mathsf{C.Com}(par_c, s_2)$ with 0 as opening, and computes $c \leftarrow c_1 \cdot c_2$.
     - $\mathcal{U}$ computes a signature $sig \leftarrow \mathsf{S.Sign}(sk, \langle vc, c\rangle)$ on vc and c.
     - $\mathcal{U}$ uses $\mathsf{nym.reply}$ on input P to send $(\mathbf{x}, s_2, sig)$ to $\mathcal{R}_k$.
     - $\mathcal{R}_k$ takes the stored $(sid, par, par_c, c_1, s_1, o_1)$.
     - $\mathcal{R}_k$ computes $vc \leftarrow \mathsf{VC.Com}(par, \mathbf{x}, 0)$ with 0 as randomness.
     - $\mathcal{R}_k$ computes $(c_2, 0) \leftarrow \mathsf{C.Com}(par_c, s_2)$ with 0 as opening and sets $c \leftarrow c_1 \cdot c_2$.
     - $\mathcal{R}_k$ uses the $\mathsf{reg.retrieve}$ interface of $\mathcal{F}_{\mathrm{REG}}$ to retrieve the public key pk of $\mathcal{U}$ and verifies the signature sig by running $\mathsf{S.Vf}(pk, sig, \langle vc, c\rangle)$.
     - $\mathcal{R}_k$ stores $(sid, par, par_c, pk, vc, \mathbf{x}, r \leftarrow 0, c, s \leftarrow s_1 + s_2, o \leftarrow o_1, sig)$.
   - $\mathcal{R}_k$ outputs $(\mathsf{hd.update.end}, sid, P, \mathbf{x})$.

Fig. 4. Construction Π_{HD}: interface hd.update (Case $f = 1$)

in vc_2, then $w_i \leftarrow w_{1,i} \cdot w_{2,i}$ is an opening for $\mathbf{x}_1[i] + \mathbf{x}_2[i]$ being at position i in the vector $\mathbf{x}$ committed in vc.

Signature Schemes. A signature scheme consists of the algorithms S.KG, S.Sign and S.Vf. $\mathsf{S.KG}(1^k)$ outputs a secret key sk and a public key pk, which include a description of the message space $\mathcal{M}$. $\mathsf{S.Sign}(sk, m)$ outputs a signature sig on the message $m \in \mathcal{M}$. $\mathsf{S.Vf}(pk, sig, m)$ outputs 1 if sig is a valid signature on m and 0 otherwise. This definition can be extended to blocks of messages $\bar{m} = (m_1, \ldots, m_n)$. In this case, $\mathsf{S.KG}(1^k, n)$ receives the maximum number n of messages as input. A signature scheme must be existentially unforgeable [18].

3.3 Construction for HD

We depict Π_{HD} in Fig. 3, Fig. 4 and Fig. 5. In Π_{HD}, a VC vc is used to commit to the databases DB, where each DB consists of N entries of the form $[i, vr_i]$. To this end, vc commits to a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vr_i$ for $i \in [1, N]$.

In the first interaction with the updater $\mathcal{U}$, the reader $\mathcal{R}_k$ obtains a VC vc to an initial DB. $\mathcal{U}$ signs vc and gives the signature sig to $\mathcal{R}_k$. In subsequent interactions, $\mathcal{R}_k$ may want to read entries from DB, to make $\mathcal{U}$ update the database, or both. To read entries $[i, vr_i]$, $\mathcal{R}_k$ gets as input the commitments and openings $(c_i, o_i, cr_i, or_i)_{i\in\mathbb{S}}$. $\mathcal{R}_k$ uses $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ to prove in ZK that c_i and cr_i commit to an entry $[i, vr_i]$ in DB. $\mathcal{U}$ receives a randomized version vc' of vc. To update DB with $(i, vu_i)_{i\in[1,N]}$, $\mathcal{U}$ computes a VC vc_2 to $(i, vu_i)_{i\in[1,N]}$ and then uses the homomorphic property of the VC scheme to set $vc_u \leftarrow vc' \cdot vc_2$. The VC vc_u that commits to the updated database is signed and sent to $\mathcal{R}_k$.

To prevent $\mathcal{R}_k$ from reading an old database, Π_{HD} uses a double-spending detection mechanism. A commitment c is signed along with every vc. c commits to a random value $s = s_1 + s_2$, where s_1 and s_2 are random values contributed by $\mathcal{R}_k$ and $\mathcal{U}$ respectively. In a read operation, $\mathcal{R}_k$ reveals a randomized version c' of c and opens c' to s. $\mathcal{U}$ checks that s has never been received before. The message space of the commitment scheme should be sufficiently big to avoid collisions.

To provide unlinkability, in addition to rerandomizing vc' and c', $\mathcal{R}_k$ communicates with $\mathcal{U}$ through a secure pseudonymous channel modeled by $\mathcal{F}_{\mathrm{NYM}}$. $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ is used by $\mathcal{R}_k$ and $\mathcal{U}$ to retrieve the VC parameters and the commitment parameters. (We note that the commitments $(c_i, o_i, cr_i, or_i)_{i\in\mathbb{S}}$ received as input by $\mathcal{R}_k$ are computed outside the protocol by $\mathcal{F}_{\mathrm{NIC}}$ and may have other parameters.) $\mathcal{F}_{\mathrm{REG}}$ is used to register the signing public key of $\mathcal{U}$.

Theorem 3.1. The construction Π_{HD} securely realizes $\mathcal{F}_{\mathrm{HD}}$ in the $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\mathrm{REG}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_r}$-hybrid

2. On input $(\mathsf{hd.update.ini}, sid, P, (i, vu_i)_{i\in[1,N]})$, $\mathcal{U}$ and $\mathcal{R}_k$ do the following
   - $\mathcal{U}$ aborts if there is not a tuple $(sid, P', f, \ldots)$ such that $P = P'$.
   - If $(sid, par, par_c, pk, sk)$ is not stored, $\mathcal{U}$ obtains par and par_c from $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. $\mathcal{U}$ computes a signing key pair $(pk, sk) \leftarrow \mathsf{S.KG}(1^k)$ and uses $\mathsf{reg.register}$ to register pk. $\mathcal{U}$ stores $(sid, par, par_c, pk, sk)$.
   - If $f = 0$, do the following:
     - $\mathcal{U}$ takes the stored tuple (sid, P, f, vc', c_1).
     - $\mathcal{U}$ sets a vector $\mathbf{x}_u[i] \leftarrow vu_i$ (for all $i \in [1, N]$), computes a commitment $vc_2 \leftarrow \mathsf{VC.Com}(par, \mathbf{x}_u, 0)$ with 0 as randomness and sets $vc_u \leftarrow vc' \cdot vc_2$.
     - $\mathcal{U}$ picks a random value s_2, computes a commitment $(c_2, 0) \leftarrow \mathsf{C.Com}(par_c, s_2)$ with 0 as opening, and computes $c_u \leftarrow c_1 \cdot c_2$.
     - $\mathcal{U}$ sets a signature $sig \leftarrow \mathsf{S.Sign}(sk, \langle vc_u, c_u\rangle)$ on vc_u and c_u.
     - $\mathcal{U}$ uses $\mathsf{nym.reply}$ on input P to send $(\mathbf{x}_u, s_2, sig)$ to $\mathcal{R}_k$.
     - $\mathcal{R}_k$ takes the stored tuple (sid, c_1, s_1, o_1).
     - $\mathcal{R}_k$ computes $(c_2, 0) \leftarrow \mathsf{C.Com}(par_c, s_2)$ with 0 as opening and sets $c_u \leftarrow c_1 \cdot c_2$.
     - $\mathcal{R}_k$ takes vc from the stored tuple $(sid, par, par_c, pk, vc, \mathbf{x}, r, c, s, o, sig)$, runs $vc_2 \leftarrow \mathsf{VC.Com}(par, \mathbf{x}, 0)$ with 0 as randomness and sets $vc_u \leftarrow vc \cdot vc_2$.
     - $\mathcal{R}_k$ verifies the signature sig by running $\mathsf{S.Vf}(pk, sig, \langle vc_u, c_u\rangle)$.
     - $\mathcal{R}_k$ updates the tuple $(sid, par, par_c, pk, vc \leftarrow vc_u, \mathbf{x} \leftarrow \mathbf{x} + \mathbf{x}_u, r, c, s \leftarrow s_1 + s_2, o \leftarrow o_1, sig)$.
   - $\mathcal{R}_k$ outputs $(\mathsf{hd.update.end}, sid, P, \mathbf{x})$.

Fig. 5. Construction Π_{HD}: interface hd.update (Case $f = 0$).

model if the VC scheme is hiding and binding, the commitment scheme is hiding and binding, and the signature scheme is existentially unforgeable.

Security Analysis. We analyze in detail the security of Π_{HD} in §C. The use of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ to generate the parameters guarantees that they are generated honestly and that $\mathcal{R}_k$ and $\mathcal{U}$ do not know any trapdoor.

When $\mathcal{R}_k$ is corrupt, the binding property of the VC scheme guarantees that $\mathcal{R}_k$ is not able to open a VC vc to a value vr_i at position i if $vr_i \neq \mathbf{x}[i]$, where $\mathbf{x}$ is the vector committed by the honest $\mathcal{U}$ in vc. The binding property of the commitment scheme guarantees that the commitment c cannot be opened to a random value s different from the one committed, which ensures that $\mathcal{R}_k$ cannot open the same commitment twice without being detected. The unforgeability property of the signature scheme ensures that $\mathcal{R}_k$ can only use vc and c that were signed by $\mathcal{U}$.

When $\mathcal{U}$ is corrupt, the hiding property of the VC scheme guarantees that the database in vc remains hidden from $\mathcal{U}$. The use of $\mathcal{F}_{\mathrm{NYM}}$ guarantees unlinkability between read operations, so that $\mathcal{U}$ cannot keep track of the updates for a particular reader. The hiding property of the commitment scheme hides the random value s committed in c during the update phase, which is needed to provide unlinkability between the update phase and the next read phase. The use of $\mathcal{F}_{\mathrm{REG}}$ guarantees that $\mathcal{U}$ does not set up a different public key for each reader, which would also break unlinkability.

Discussion. In Π_{HD}, an update operation always follows a read operation, and vice versa. We remark that, when only an update is required, the tuple $(i, vr_i, c_i, o_i, cr_i, or_i)_{i\in\mathbb{S}}$ can be empty, and when only a read operation is needed, the tuple $(i, vu_i)_{i\in[1,N]}$ can equal $(i, 0)_{i\in[1,N]}$.

$\mathcal{U}$ updates databases without knowing their content. The update operator is the sum. In our instantiation in §3.4, the sum is done modulo p. We assume p is big enough so that the sums accumulated are always smaller.

3.4 Instantiation and Efficiency

In §D, we describe an instantiation of Π_{HD} that uses a VC scheme similar to the one in [27], the Pedersen commitment scheme [31], the signature scheme in [1] and the UC ZK proof protocol in [10]. Here we analyze the storage, communication, and computation costs of our instantiation.

Storage Cost. $\mathcal{R}_k$ stores the common reference string, which consists of the VC and commitment parameters.

The VC parameters grow linearly with the maximum size N of the database. $\mathcal{R}_k$ also stores the public key pk, which has constant size. Throughout the protocol execution, $\mathcal{R}_k$ stores the last update of sig, vc and c, the committed values and their openings. In conclusion, the storage cost for $\mathcal{R}_k$ grows linearly with N. $\mathcal{U}$ stores the common reference string and the signing key pair and thus his storage cost also grows linearly with N.

Communication Cost. In a read operation, $\mathcal{R}_k$ and $\mathcal{U}$ run a ZK proof for R_r where wit_r and ins_r grow linearly with the number $|\mathbb{S}|$ of positions read, but are independent of N. Therefore, the communication cost grows linearly with $|\mathbb{S}|$ but is independent of N. $\mathcal{R}_k$ sends a Pedersen commitment c_1 and an opening (s, o') to $\mathcal{U}$, which have constant size. In an update, $\mathcal{U}$ sends $\mathcal{R}_k$ an update $\mathbf{x}_u$ for vc', a signature on vc' and c' and the random value s_2. The signature and s_2 are of constant size. Although the size of $\mathbf{x}_u$ could be N, in practice, $\mathbf{x}_u$ only needs to contain the values for positions that are updated. Consequently, the communication cost of read and update operations grows linearly with the number of positions read or updated but is independent of N.

Computation Cost. In a read operation, $\mathcal{R}_k$ needs to compute VC openings w_i for the positions read. If w_i is computed for the first time, the computation cost grows linearly with N. However, by using the homomorphic property, when vc is updated, each opening w_i can be updated with cost that grows linearly with the number of updated positions but is independent of N. The remaining steps in the computation and verification of the proof for R_r are also independent of N. $\mathcal{R}_k$ also computes a commitment c_1, which entails constant cost. In the first update operation, $\mathcal{U}$ computes a commitment vc to an initial database with cost linear in N. (In practice, if the database is initialized to a vector of zeroes, the cost is constant.) Subsequently, $\mathcal{U}$ updates vc by using the homomorphic property of the VC scheme. The computation cost grows with the number of positions updated.

In summary, the communication cost is independent of N and the amortized computation cost is also independent of N, which makes our instantiation of Π_{HD} practical for large databases.

Implementation and Timings. We implement our instantiation of Π_{HD} in the Python programming language, using the Charm cryptographic framework [2], on a computer equipped with an Intel Core i5-7300U CPU clocked at 2.60 GHz, and 8 gigabytes of RAM. The BN256 curve is used for the pairing group setup. To compute the UC ZK proofs for R_r, we use the compiler in [10]. The cost of a proof depends on the number of elements in the witness and on the number of equations composed by Boolean ANDs. The computation cost for the prover of a Σ-protocol for R_r involves one evaluation of each of the equations and one multiplication per value in the witness. The compiler in [10] extends a Σ-protocol and requires, additionally, a computation of a multi-integer commitment that commits to the values in the witness, an evaluation of a Paillier encryption for each of the values in the witness, a Σ-protocol to prove that the commitment and the encryptions are correctly generated, and 3 exponentiations in the DSA group. The computation cost for the verifier, as well as the communication cost, also depends on the number of values in the witness and on the number of equations. Therefore, as the number of values in the witness and of equations is independent of N in our proof for relation R_r, the computation and communication costs of our proof do not depend on N. The storage costs for storing the common reference string in this implementation, for databases of size 15000 and 65000 were found to be about 10MB and 42MB respectively.

Table 1. Π_{HD} execution times in seconds

Interface	$N = 15000$		$N = 65000$	
	Time (s)	SD	Time (s)	SD
First Update	88.97	0.5	386.20	3
Computation of vc or w_i	0.23	0.002	1.0028	0.008
1 entry Update of vc or w_i	0.00003	0.00001	0.00003	0.00001
1 entry Read	6.9564	0.4	7.6770	0.3
5 entry Read	23.0339	0.4	26.9555	0.4

Table 1 lists the mean execution times (and the standard deviation) of the interfaces of the protocol, in seconds, over 100 trial runs, evaluated against N. The Paillier key size is 2048 bits. In the first update, the public parameters of all building blocks are computed and the database is set up by computing vc. In the second row of Table 1, we show the cost of just computing vc, which is virtually the same as that of computing an opening w_i, and these costs are very small. In a single entry update, one database entry is modified and vc is updated, and the cost of updating an opening w_i is virtually the same. As can be seen, the cost of the first update grows linearly with N, as does the cost of setting up vc or w_i, whereas the cost of updating vc or w_i is very small and independent of N. (We note that the cost of setting up vc or w_i is constant when the database is initialized to 0 in every position, which is the case in our PPLP protocol.) The timings for the read interface increase linearly with the number of entries read $|\mathbb{S}|$. However, the timings are independent of N.

4 Our PPLP

Our PPLP involves a vendor $\mathcal{V}$ and multiple buyers $\mathcal{B}_k$. It can be applied to both online and physical shops. Buyers are equipped with an electronic device such as a smartphone. To provide unlinkability, our PPLP uses the secure pseudonymous channel modeled by $\mathcal{F}_{\text{NYM}}$. In addition, it requires payment methods that provide unlinkability, such as cash in physical shops or anonymous e-cash [9] in online shops. In an online setting, the purchase of physical goods would also require an anonymous delivery system [3].

Our PPLP consists of registration, purchase, redemption and profiling phases. After the registration phase for a buyer $\mathcal{B}_k$ is run, the remaining phases are interleaved throughout the protocol execution. Those four phases are initiated by $\mathcal{B}_k$, and $\mathcal{V}$ finalizes them with an update database operation.

When $\mathcal{B}_k$ purchases products, $\mathcal{V}$ gives $\mathcal{B}_k$ a number of loyalty points *lpts*, which $\mathcal{B}_k$ can later redeem to obtain any sort of discount or gift. This general setting follows many existing LPs. Our PPLP can be adapted to variations or extensions of this general setting (e.g., applying discounts at every purchase, adding an expiration date to *lpts*, ...).

$\mathcal{V}$ profiles $\mathcal{B}_k$ based on her PH. To store the PH of $\mathcal{B}_k$, we use arrays of the form [*product, value*], where *product* is a specific product or category of product (depending on the granularity needed to profile buyers), and *value* is the amount of money spent by $\mathcal{B}_k$ on that product in one or more purchases, the number of times that the product was bought, or any other data required to profile buyers. Without loss of generality, products are identified by an index, i.e. *product* is in $[1, M]$, where M is the number of products offered by $\mathcal{V}$.

We consider a profiling algorithm defined by a function family $\mathcal{F}$. A function $f \in \mathcal{F}$ is defined by $f : (i, v)_{i \in \mathbb{S}} \rightarrow res$, i.e., a function that takes as input a subset $\mathbb{S} \subseteq [1, M]$ of the entries of the PH of $\mathcal{B}_k$ and outputs a result *res* that classifies $\mathcal{B}_k$ into a category. In our PPLP, $\mathcal{B}_k$ runs f on input her PH and proves in ZK that the result is correct.

We remark that the concrete function $f \in \mathcal{F}$ (and the set $\mathbb{S}$) used by $\mathcal{B}_k$ at a profiling phase remains hidden from $\mathcal{V}$. As a simple example, in our implementation in §4.3, we use a family where each function f specifies a set $\mathbb{S}$ and requires $\mathcal{B}_k$ to prove that the total amount paid for the products in $\mathbb{S}$ surpasses a threshold. $\mathcal{B}_k$ needs to prove in ZK that a correct subset $\mathbb{S}$ of positions is read without revealing $\mathbb{S}$. For this purpose, $\mathcal{V}$ computes signatures on the indices of each of the subsets in the family $\mathbb{S}$. $\mathcal{B}_k$ proves in ZK knowledge of a signature and proves that the positions read from the database are equal to the signed subset indices, as described in the ZK set membership and range proof in [7].

Previous work [15] has shown how to prove in ZK correctness of the evaluation of two popular classification methods: random forests and hidden Markov models. The schemes in [15] follow a similar approach to our example above. For each classification method, a function family is defined, and $\mathcal{V}$ provides $\mathcal{B}_k$ with signatures that sign the parameters of each concrete function that $\mathcal{V}$ considers. $\mathcal{B}_k$ uses those signatures to prove in ZK the correctness of the profiling result *res* without revealing to $\mathcal{V}$ the concrete function being used.

The conditions under which $\mathcal{V}$ can request $\mathcal{B}_k$ to reveal her classification result should be defined in the terms and conditions of the PPLP and agreed upon by $\mathcal{B}_k$. $\mathcal{V}$ could incentivize $\mathcal{B}_k$ to be profiled by e.g. offering $\mathcal{B}_k$ more *lpts*.

Our PPLP only discloses to $\mathcal{V}$ the profiling result. However, depending on the profiling function used, this result can be privacy sensitive. To counter this problem, we show that for some purposes $\mathcal{V}$ can use the profiling result without learning it.

4.1 Ideal Functionality for a PPLP

We depict an ideal functionality $\mathcal{F}_{\text{LP}}$ for a PPLP in Fig. 6. $\mathcal{F}_{\text{LP}}$ interacts with a vendor $\mathcal{V}$ and multiple buyers $\mathcal{B}_k$. $\mathcal{F}_{\text{LP}}$ maintains one database DB per buyer. Each DB is of size $N = M + 1$ and stores the PH for M products and the *lpts* accumulated by $\mathcal{B}_k$. $\mathcal{F}_{\text{LP}}$ consists of the interfaces lp.register, lp.purchase, lp.redeem, lp.profile and lp.updatedb.

1. A buyer $\mathcal{B}_k$ sends the lp.register.ini message on input a pseudonym P. $\mathcal{F}_{\text{LP}}$ checks that P is unique and that $\mathcal{B}_k$ did not send a registration request before. In that case, after being prompted by the simulator $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P to $\mathcal{V}$.
2. A buyer $\mathcal{B}_k$ sends the lp.purchase.ini message on input a pseudonym P. $\mathcal{F}_{\text{LP}}$ checks that $\mathcal{B}_k$ is already registered, that P is unique and that $\mathcal{B}_k$ has not initiated another purchase, redeem or profile phase that

is still unfinished. In that case, after being prompted by $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P to $\mathcal{V}$.

3. A buyer $\mathcal{B}_k$ sends the lp.redeem.ini message on input a pseudonym P and a number of loyalty points p. $\mathcal{F}_{\text{LP}}$ checks that $\mathcal{B}_k$ is already registered, that P is unique, that $\mathcal{B}_k$ has not initiated another purchase, redeem or profile phase that is still unfinished, and that $\mathcal{B}_k$ has accumulated at least p loyalty points. In that case, after being prompted by $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P and p to $\mathcal{V}$.
4. A buyer $\mathcal{B}_k$ sends the lp.profile.ini message on input a pseudonym P and a profiling function f. $\mathcal{F}_{\text{LP}}$ checks that $\mathcal{B}_k$ is already registered, that P is unique, and that $\mathcal{B}_k$ has not initiated another purchase, redeem or profile phase that is still unfinished. In that case, $\mathcal{F}_{\text{LP}}$ evaluates f on input the purchase history of $\mathcal{B}_k$. After being prompted by $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P and the profiling result *res* to $\mathcal{V}$.
5. The vendor $\mathcal{V}$ sends the lp.updatedb.ini message on input a pseudonym P and a database $(i, vu_i)_{i\in[1,N]}$. $\mathcal{F}_{\text{LP}}$ checks that there is a request pending for pseudonym P, which is associated to a buyer $\mathcal{B}_k$. For a registration request, $\mathcal{F}_{\text{LP}}$ initializes the database DB of $\mathcal{B}_k$ contain 0 at every position. For a purchase request, $\mathcal{F}_{\text{LP}}$ uses $(i, vu_i)_{i\in[1,N]}$ to update the PH and the loyalty points stored in DB. For a redeem request, $\mathcal{F}_{\text{LP}}$ updates DB by subtracting p loyalty points redeemed by $\mathcal{B}_k$. For a profile request, $\mathcal{F}_{\text{LP}}$ does not update the database. After being prompted by $\mathcal{S}$, $\mathcal{F}_{\text{LP}}$ sends P and $(i, vu_i)_{i\in[1,N]}$ to $\mathcal{B}_k$.

$\mathcal{F}_{\text{LP}}$ guarantees that the requests made by buyers are *unlinkable* for $\mathcal{V}$. As can be seen, $\mathcal{F}_{\text{LP}}$ reveals to $\mathcal{V}$ a pseudonym that is unique per request. $\mathcal{F}_{\text{LP}}$ also ensures that the PH and *lpts* of a buyer are *hidden* from $\mathcal{V}$. $\mathcal{F}_{\text{LP}}$ never reveals a buyer's database to $\mathcal{V}$ and, thanks to unlinkability, $\mathcal{V}$ is not able to link the updates of a database. $\mathcal{F}_{\text{LP}}$ also ensures *unforgeabilty* of databases, i.e. buyers are not able to modify their databases. This guarantees that *lpts* can only be redeemed if they were accumulated, and that the profiling result is correct.

4.2 Construction for a PPLP

We depict a construction Π_{LP} for $\mathcal{F}_{\text{LP}}$ in Fig. 7 and Fig. 8. Π_{LP} is described modularly in the hybrid model, where $\mathcal{V}$ and $\mathcal{B}_k$ use the functionalities $\mathcal{F}_{\text{HD}}$, $\mathcal{F}_{\text{ZK}}^{R_{rd}}$, $\mathcal{F}_{\text{ZK}}^{R_{pr}}$, $\mathcal{F}_{\text{NYM}}$ and $\mathcal{F}_{\text{NIC}}$. $\mathcal{F}_{\text{ZK}}^{R_{rd}}$ and $\mathcal{F}_{\text{ZK}}^{R_{pr}}$ are ZK functionalities used in the redemption and profiling phases respectively. This modular design allows multiple constructions of Π_{LP} by choosing different instantiations for those functionalities. It also eases its security analysis.

$\mathcal{F}_{\text{HD}}$ is used to store the buyers' databases DB. During registration, $\mathcal{B}_k$ invokes the hd.read interface for the first time, and $\mathcal{V}$ invokes the hd.update interface on input $(i, 0)_{i\in[1,N]}$ to initialize DB. In the purchase phase, $\mathcal{B}_k$ invokes the hd.read interface without reading the database, and $\mathcal{V}$ invokes the hd.update interface on input $(i, vu_i)_{i\in[1,N]}$ to update the PH and *lpts* stored in DB.

To redeem p *lpts*, $\mathcal{B}_k$ invokes the hd.read interface to read the database entry $[N, v_N]$, which stores the accumulated *lpts*. $\mathcal{B}_k$ uses $\mathcal{F}_{\text{ZK}}^{R_{rd}}$ to prove that $p \in [0, v_N]$. Both $\mathcal{F}_{\text{ZK}}^{R_{rd}}$ and $\mathcal{F}_{\text{HD}}$ receive as input commitments output by $\mathcal{F}_{\text{NIC}}$ to ensure that the value v_N read is used in the ZK proof. $\mathcal{V}$ invokes the hd.update interface on input $(i, 0)_{i\in[1,M]}\|[N, -p]$ to subtract the redeemed loyalty points from DB.

In the profiling phase, $\mathcal{B}_k$ invokes the hd.read interface to read the subset $\mathbb{S}$ of database entries required by the function f. $\mathcal{B}_k$ uses $\mathcal{F}_{\text{ZK}}^{R_{pr}}$ to prove correctness of the evaluation of f. Commitments produced by $\mathcal{F}_{\text{NIC}}$ are again used to guarantee that the entries read are used in $\mathcal{F}_{\text{ZK}}^{R_{pr}}$. $\mathcal{V}$ invokes the hd.update interface without updating the database to conclude this phase.

Theorem 4.1. Π_{LP} securely realizes $\mathcal{F}_{\text{LP}}$ in the $\mathcal{F}_{\text{HD}}$, $\mathcal{F}_{\text{ZK}}^{R_{rd}}$, $\mathcal{F}_{\text{ZK}}^{R_{pr}}$, $\mathcal{F}_{\text{NYM}}$ and $\mathcal{F}_{\text{NIC}}$-hybrid model.

Security Analysis. We analyze in detail the security of Π_{LP} in §F . Π_{LP} provides the unlinkability and hiding properties of $\mathcal{F}_{\text{LP}}$, which protect buyers' privacy. Regarding unlinkability, $\mathcal{F}_{\text{HD}}$, $\mathcal{F}_{\text{ZK}}^{R_{rd}}$, $\mathcal{F}_{\text{ZK}}^{R_{pr}}$ and $\mathcal{F}_{\text{NYM}}$ reveal to $\mathcal{V}$ a pseudonym and not the identifier $\mathcal{B}_k$. $\mathcal{B}_k$ chooses a different pseudonym for each phase. Regarding the hiding property, in the purchase phase $\mathcal{V}$ does not learn any information on the PH or on the *lpts* of $\mathcal{B}_k$. In the redemption phase, $\mathcal{V}$ only learns that $\mathcal{B}_k$ accumulated at least p points. In the profiling phase, $\mathcal{V}$ only learns the result of evaluating f on input the PH of $\mathcal{B}_k$. We recall that $\mathcal{F}_{\text{NIC}}$ ensures that commitments are hiding.

Π_{LP} prevents an adversarial $\mathcal{B}_k$ from forging her PH or her *lpts*. $\mathcal{F}_{\text{HD}}$ ensures that only $\mathcal{V}$ sets up and modifies databases. $\mathcal{F}_{\text{HD}}$ also ensures that $\mathcal{B}_k$ cannot reuse her PH or her *lpts*. Additionally, the binding property ensured by $\mathcal{F}_{\text{NIC}}$ guarantees that commitments sent to $\mathcal{F}_{\text{HD}}$ and to the ZK functionalities are opened to the same value. Therefore, $\mathcal{B}_k$ can only prove knowledge of the PH or *lpts* that were stored by $\mathcal{V}$.

Extensions. There are many LPs with different features and requirements. In general, we think that our

$\mathcal{F}_{\mathrm{LP}}$ is parameterised by a database size N, a universe of pseudonyms $\mathbb{U}_p$, a function family $\mathcal{F}$ and a universe of values $\mathbb{U}_v$.

1. On input $(\mathsf{lp.register.ini}, sid, P)$ from $\mathcal{B}_k$:
   - Abort if $sid \notin (\mathcal{V}, sid')$, or if $P \notin \mathbb{U}_p$. Abort if there is a tuple $(sid, P', \mathcal{B}'_k, \ldots)$ stored such that $P' = P$, or a tuple $(sid, P', \mathcal{B}'_k, 0)$ such that $\mathcal{B}'_k = \mathcal{B}_k$.
   - Store $(sid, P, \mathcal{B}_k, 0)$, create a fresh qid, store $(qid, P, \mathcal{B}_k)$ and send $(\mathsf{lp.register.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.register.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{B}_k)$ such that $qid = qid'$ is not stored.
   - Store $(sid, P, \mathcal{B}_k, \mathsf{lp.register})$, delete $(qid, P, \mathcal{B}_k)$ and send $(\mathsf{lp.register.end}, sid, P)$ to $\mathcal{V}$.

2. On input $(\mathsf{lp.purchase.ini}, sid, P)$ from $\mathcal{B}_k$:
   - Abort if $sid \notin (\mathcal{V}, sid')$, or if $P \notin \mathbb{U}_p$. Abort if a tuple $(sid, P', \mathcal{B}'_k, 1)$ such that $\mathcal{B}'_k \neq \mathcal{B}_k$ is not stored. Abort if there is a tuple $(sid, P', \mathcal{B}'_k, \ldots)$ stored such that $P' = P$, or $(sid, P', \mathcal{B}'_k, 0)$ such that $\mathcal{B}'_k = \mathcal{B}_k$.
   - Store $(sid, P, \mathcal{B}_k, 0)$, create a fresh qid, store $(qid, P, \mathcal{B}_k)$ and send $(\mathsf{lp.purchase.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.purchase.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{B}_k)$ such that $qid = qid'$ is not stored.
   - Store $(sid, P, \mathcal{B}_k, \mathsf{lp.purchase})$, delete $(qid, P, \mathcal{B}_k)$ and send $(\mathsf{lp.purchase.end}, sid, P)$ to $\mathcal{V}$.

3. On input $(\mathsf{lp.redeem.ini}, sid, P, p)$ from $\mathcal{B}_k$:
   - Abort if $sid \notin (\mathcal{V}, sid')$, or if $P \notin \mathbb{U}_p$. Abort if a tuple $(sid, P', \mathcal{B}'_k, 1)$ such that $\mathcal{B}'_k \neq \mathcal{B}_k$ is not stored. Abort if there is a tuple $(sid, P', \mathcal{B}'_k, \ldots)$ stored such that $P' = P$, or a tuple $(sid, P', \mathcal{B}'_k, 0)$ such that $\mathcal{B}'_k = \mathcal{B}_k$. Take the stored tuple $(sid, \mathcal{B}_k, \mathsf{DB})$ and take the entry $[N, v_N] \in \mathsf{DB}$. Abort if $p \notin [0, v_N]$.
   - Store $(sid, P, \mathcal{B}_k, 0)$, create a fresh qid, store $(qid, P, \mathcal{B}_k, p)$ and send $(\mathsf{lp.redeem.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.redeem.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{B}_k, p)$ such that $qid = qid'$ is not stored.
   - Store $(sid, P, \mathcal{B}_k, \mathsf{lp.redeem}, p)$, delete $(qid, P, \mathcal{B}_k, p)$ and send $(\mathsf{lp.redeem.end}, sid, P, p)$ to $\mathcal{V}$.

4. On input $(\mathsf{lp.profile.ini}, sid, P, f)$ from $\mathcal{B}_k$:
   - Abort if $sid \notin (\mathcal{V}, sid')$, or if $P \notin \mathbb{U}_p$, of if $f \notin \mathcal{F}$. Abort if a tuple $(sid, P', \mathcal{B}'_k, 1)$ such that $\mathcal{B}'_k \neq \mathcal{B}_k$ is not stored. Abort if there is a tuple $(sid, P', \mathcal{B}'_k, \ldots)$ stored such that $P' = P$, or a tuple $(sid, P', \mathcal{B}'_k, 0)$ such that $\mathcal{B}'_k = \mathcal{B}_k$.
   - Take $(sid, \mathcal{B}_k, \mathsf{DB})$ and compute $res \leftarrow f((i, v)_{i \in \mathbb{S}})$, where $(i, v)_{i \in \mathbb{S}} \subseteq \mathsf{DB}$ and $\mathbb{S}$ is defined in f.
   - Store $(sid, P, \mathcal{B}_k, 0)$, create a fresh qid, store $(qid, P, \mathcal{B}_k, res)$ and send $(\mathsf{lp.profile.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.profile.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{B}_k, res)$ such that $qid = qid'$ is not stored.
   - Store $(sid, P, \mathcal{B}_k, \mathsf{lp.profile})$, delete $(qid, P, \mathcal{B}_k, res)$ and send $(\mathsf{lp.profile.end}, sid, P, res)$ to $\mathcal{V}$.

5. On input $(\mathsf{lp.updatedb.ini}, sid, P, (i, vu_i)_{i \in [1,N]})$ from $\mathcal{V}$:
   - Abort if $(sid, P', \mathcal{B}_k, \mathsf{lp.x}, \ldots)$ such that $P' = P$ is not stored, or if, for $i \in [1, N]$, $vu_i \notin \mathbb{U}_v$.
   - If $\mathsf{lp.x} = \mathsf{lp.register}$, set $\mathsf{DB} \leftarrow (i, 0)_{i \in [1,N]}$ and store a tuple $(sid, \mathcal{B}_k, \mathsf{DB})$.
   - If $\mathsf{lp.x} = \mathsf{lp.purchase}$, take the stored tuple $(sid, \mathcal{B}_k, \mathsf{DB})$, parse DB as $(i, vr_i)_{i \in [1,N]}$ and update $\mathsf{DB} \leftarrow (i, vr_i + vu_i)_{i \in [1,N]}$ in the tuple $(sid, \mathcal{B}_k, \mathsf{DB})$.
   - If $\mathsf{lp.x} = \mathsf{lp.redeem}$, take p from the tuple $(sid, P', \mathcal{B}_k, \mathsf{lp.purchase}, p)$. Take the stored tuple $(sid, \mathcal{B}_k, \mathsf{DB})$, parse DB as $(i, vr_i)_{i \in [1,N]}$ and update the database entry $[N, vr_i + p]$ in the tuple $(sid, \mathcal{B}_k, \mathsf{DB})$.
   - Delete the tuple $(sid, P', \mathcal{B}_k, \mathsf{lp.x}, \ldots)$. If $\mathsf{lp.x} \neq \mathsf{lp.purchase}$, set $(i, vu_i)_{i \in [1,N]} \leftarrow \perp$.
   - Create fresh qid, store $(qid, \mathcal{B}_k, P, \mathsf{lp.x}, (i, vu_i)_{i \in [1,N]})$ and send $(\mathsf{lp.updatedb.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{lp.updatedb.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if a tuple $(qid', \mathcal{B}_k, P, \mathsf{lp.x}, (i, vu_i)_{i \in [1,N]})$ such that $qid = qid'$ is not stored.
   - In the tuple $(sid, P', \mathcal{B}_k, 0)$ such that $P' = P$, if $\mathsf{lp.x} = \mathsf{lp.register}$, replace 0 by 1, else replace 0 by $\perp$.
   - Delete $(qid', \mathcal{B}_k, P, \mathsf{lp.x}, (i, vu_i)_{i \in [1,N]})$ and send $(\mathsf{lp.updatedb.end}, sid, P, (i, vu_i)_{i \in [1,N]})$ to $\mathcal{B}_k$.

Fig. 6. Functionality $\mathcal{F}_{\mathrm{LP}}$

Π_{LP} is parameterised by a database size N, a universe of pseudonyms $\mathbb{U}_p$, a function family $\mathcal{F}$ and a universe of values $\mathbb{U}_v$.

1. On input $(\mathsf{lp.register.ini}, sid, P)$, $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ uses the $\mathsf{nic.setup}$ interface of $\mathcal{F}_{\mathrm{NIC}}$ and sends $(\mathsf{hd.read.ini}, sid, P, \perp)$ to $\mathcal{F}_{\mathrm{HD}}$.
   - $\mathcal{V}$ receives $(\mathsf{hd.read.end}, sid, P, f, \perp)$ from $\mathcal{F}_{\mathrm{HD}}$. If $f = 1$, $\mathcal{V}$ stores $(sid, P, \mathsf{lp.register})$ and outputs $(\mathsf{lp.register.end}, sid, P)$. In its first activation, $\mathcal{V}$ uses the $\mathsf{nic.setup}$ interface of $\mathcal{F}_{\mathrm{NIC}}$.
2. On input $(\mathsf{lp.purchase.ini}, sid, P)$, $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ sends $(\mathsf{hd.read.ini}, sid, P, \perp)$ to $\mathcal{F}_{\mathrm{HD}}$.
   - $\mathcal{V}$ receives $(\mathsf{hd.read.end}, sid, P, f, \perp)$ from $\mathcal{F}_{\mathrm{HD}}$. If $f = 0$ and no commitments to read data are received, $\mathcal{V}$ stores $(sid, P, \mathsf{lp.purchase})$ and outputs $(\mathsf{lp.purchase.end}, sid, P)$.
3. On input $(\mathsf{lp.redeem.ini}, sid, P, p)$, $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ takes the stored $(i, v_i)_{i\in[1,N]}$ and checks that $p \in [0, v_N]$.
   - $\mathcal{B}_k$ uses the $\mathsf{nic.commit}$ interface of $\mathcal{F}_{\mathrm{NIC}}$ on input N and v_N to get two commitments and openings $(c_N,\ o_N,\ c_v,\ o_v)$ to N and v_N.
   - $\mathcal{B}_k$ sets a witness $wit_{rd} \leftarrow (v_N, o_v)$ and $ins_{rd} \leftarrow (p, c_v, c_N, N, o_N)$. $\mathcal{B}_k$ uses the $\mathsf{zk.prove}$ interface to send wit_{rd}, ins_{rd} and P to $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$, where R_{rd} is

     $$R_{rd} = \{(wit_{rd}, ins_{rd}) : 1 = \mathsf{NIC.Vf}(pnic, c_v, v_N, o_v) \ \wedge\ p \in [0, v_N]\}$$

     $\mathcal{B}_k$ proves that v_N is committed in c_v and that $p \le v_N$. For the latter, $\mathcal{B}_k$ uses a range proof [7].
   - $\mathcal{V}$ uses the $\mathsf{nic.verify}$ interface of $\mathcal{F}_{\mathrm{NIC}}$ on input (c_N, N, o_N) to verify that c_N commits to the position N. $\mathcal{V}$ uses the $\mathsf{nic.validate}$ interface on input c_v to check that c_v is a commitment computed by $\mathcal{F}_{\mathrm{NIC}}$. $\mathcal{V}$ stores the commitments (c_N, c_v).
   - $\mathcal{V}$ sends a message (Read DB) to $\mathcal{B}_k$ by using the $\mathsf{nym.reply}$ interface of $\mathcal{F}_{\mathrm{NYM}}$ on input P. (Here we consider that any construction for $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$ uses $\mathcal{F}_{\mathrm{NYM}}$, so $\mathcal{V}$ can reply through $\mathcal{F}_{\mathrm{NYM}}$.)
   - $\mathcal{B}_k$ stores (sid, P, p) and sends the message $(\mathsf{hd.read.ini}, sid, P, (N, v_N, c_N, o_N, c_v, o_v))$ to $\mathcal{F}_{\mathrm{HD}}$.
   - $\mathcal{V}$ receives $(\mathsf{hd.read.end}, sid, P, f, (c_N, c_v))$ from $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{V}$ checks that the commitments (c_N, c_v) equal the ones received from $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$. $\mathcal{V}$ stores $(sid, P, \mathsf{lp.redeem}, p)$ and outputs $(\mathsf{lp.redeem.end}, sid, P, p)$.
4. On input $(\mathsf{lp.profile.ini}, sid, P, f)$, $\mathcal{B}_k$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}_k$ takes her purchase history $(i, v_i)_{i\in[1,M]}$. Let $\mathbb{S} \subseteq [1, M]$ contain the indices needed to evaluate f. $\mathcal{B}_k$ commits to $(i, v_i)_{i\in\mathbb{S}}$, i.e., for all $i \in \mathbb{S}$, $\mathcal{B}_k$ uses $\mathsf{nic.commit}$ on input i and v_i to get (c_i, o_i, cr_i, or_i).
   - $\mathcal{B}_k$ computes $res \leftarrow f((i, v_i)_{i\in\mathbb{S}})$.
   - $\mathcal{B}_k$ sets a witness $wit_{pr} \leftarrow (\langle i, o_i, v_i, or_i\rangle_{i\in\mathbb{S}})$ and $ins_{pr} \leftarrow (res, \langle c_i, cr_i\rangle_{i\in\mathbb{S}})$. $\mathcal{B}_k$ uses the $\mathsf{zk.prove}$ interface to send wit_{pr}, ins_{pr} and P to $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$, where R_{pr} is

     $$\begin{aligned} R_{pr} = \{&(wit_{pr}, ins_{pr}) : \\ &\{1 = \mathsf{NIC.Vf}(pnic, c_i, i, o_i) \ \wedge\ i \in \mathbb{S} \ \wedge\ 1 = \mathsf{NIC.Vf}(pnic, cr_i, v_i, or_i)\}_{\forall i\in\mathbb{S}} \ \wedge\ res = f((i, v_i)_{i\in\mathbb{S}})\}\end{aligned}$$

     $\mathcal{B}_k$ proves that i and v_i are committed in c_i and cr_i, and that $i \in \mathbb{S}$. $\mathcal{B}_k$ also proves that res is the result of evaluating f on input $(i, v_i)_{i\in\mathbb{S}}$.
   - $\mathcal{V}$ uses $\mathsf{nic.validate}$ to check that $(c_i, cr_i)_{i\in\mathbb{S}}$ are commitments computed by $\mathcal{F}_{\mathrm{NIC}}$ and stores $(c_i, cr_i)_{i\in\mathbb{S}}$.
   - $\mathcal{V}$ sends a message (Read DB) to $\mathcal{B}_k$ by using the $\mathsf{nym.reply}$ interface of $\mathcal{F}_{\mathrm{NYM}}$. (Here we consider that any construction for $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$ uses $\mathcal{F}_{\mathrm{NYM}}$, so $\mathcal{V}$ can reply through $\mathcal{F}_{\mathrm{NYM}}$.)
   - $\mathcal{B}_k$ sends $\mathcal{F}_{\mathrm{HD}}$ the message $(\mathsf{hd.read.ini}, sid, P, (i, v_i, c_i, o_i, cr_i, or_i)_{i\in\mathbb{S}})$.
   - $\mathcal{V}$ receives $(\mathsf{hd.read.end}, sid, P, f, (c_i, cr_i)_{i\in\mathbb{S}})$ from $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{V}$ checks that the commitments $(c_i, cr_i)_{i\in\mathbb{S}}$ equal the ones received from $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$. $\mathcal{V}$ stores $(sid, P, \mathsf{lp.profile})$ and outputs $(\mathsf{lp.profile.end}, sid, P, res)$.

Fig. 7. Construction Π_{LP}: interfaces lp.register, lp.purchase, lp.redeem and lp.profile.

5. On input $(\mathsf{lp.updatedb.ini}, sid, P, (i, vu_i)_{i\in[1,N]})$, $\mathcal{V}$ and $\mathcal{B}_k$ do the following:
   - $\mathcal{V}$ finds the tuple $(sid, P', \mathsf{lp.x}, \ldots)$, such that $P' = P$.
   - If $\mathsf{lp.x} = \mathsf{lp.register}$, $\mathcal{V}$ initializes a database $(i, 0)_{i\in[1,N]}$ and sends $(\mathsf{hd.update.ini}, sid, P, (i, 0)_{i\in[1,N]})$ to $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{B}_k$ receives $(\mathsf{hd.update.end}, sid, (i, 0)_{i\in[1,N]})$ from $\mathcal{F}_{\mathrm{HD}}$, checks that $(i, 0)_{i\in[1,N]}$ contains 0 in all positions, stores $(i, 0)_{i\in[1,N]}$ and outputs $(\mathsf{lp.updatedb.end}, sid, P, \perp)$.
   - If $\mathsf{lp.x} = \mathsf{lp.purchase}$, $\mathcal{V}$ sends $(\mathsf{hd.update.ini}, sid, P, (i, vu_i)_{i\in[1,N]})$ to $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{B}_k$ receives $(\mathsf{hd.update.end}, sid, (i, vu_i)_{i\in[1,N]})$ from $\mathcal{F}_{\mathrm{HD}}$ and checks that $(i, vu_i)_{i\in[1,N]}$ is correct, i.e., the database was updated by using the correct prices paid by $\mathcal{B}_k$ for the products purchased and the correct number of *lpts* was added. $\mathcal{B}_k$ updates her database $(i, v'_i)_{i\in[1,N]}$ by setting $(i, v'_i + vu_i)_{i\in[1,N]}$ and outputs $(\mathsf{lp.updatedb.end}, sid, P, (i, vu_i)_{i\in[1,N]})$.
   - If $\mathsf{lp.x} = \mathsf{lp.redeem}$, $\mathcal{V}$ takes p from the tuple $(sid, P, \mathsf{lp.redeem}, p)$ and sets a database $(i, v_i)_{i\in[1,N]}$ where $v_i = 0$ for $i \in [1, M]$ and $v_N = -p$. $\mathcal{V}$ sends $(\mathsf{hd.update.ini}, sid, P, (i, v_i)_{i\in[1,N]})$ to $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{B}_k$ receives $(\mathsf{hd.update.end}, sid, (i, v_i)_{i\in[1,N]})$ from $\mathcal{F}_{\mathrm{HD}}$ and checks that $v_i = 0$ for $i \in [1, M]$ and $v_N = -p$, where p is stored in the tuple (sid, P, p). $\mathcal{B}_k$ updates her database $(i, v'_i)_{i\in[1,N]}$ by setting $(N, v'_N + v_N)$ and outputs $(\mathsf{lp.updatedb.end}, sid, P, \perp)$.
   - If $\mathsf{lp.x} = \mathsf{lp.profile}$, $\mathcal{V}$ initializes a database $(i, 0)_{i\in[1,N]}$ and sends $(\mathsf{hd.update.ini}, sid, P, (i, 0)_{i\in[1,N]})$ to $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{B}_k$ receives $(\mathsf{hd.update.end}, sid, (i, 0)_{i\in[1,N]})$ from $\mathcal{F}_{\mathrm{HD}}$ and checks that $(i, 0)_{i\in[1,N]}$ contains 0 in all positions. $\mathcal{B}_k$ outputs $(\mathsf{lp.updatedb.end}, sid, P, \perp)$.

Fig. 8. Construction Π_{LP}: interfaces lp.updatedb.

PPLP can be extended to implement them. In the following, we discuss some popular features:

- **Learning vs Using Profiles.** In Π_{LP}, $\mathcal{V}$ learns the result of evaluating a profiling function f on input the PH of $\mathcal{B}_k$. Depending on f, that result can be privacy sensitive, and thus disclosing it must be avoided whenever possible. There are applications in which $\mathcal{V}$ does not need to learn the buyer's profile in order to use it. For example, if $\mathcal{V}$'s goal is to offer extra *lpts* to buyers with certain profiles, Π_{LP} can be modified to add those *lpts* without $\mathcal{V}$ learning the buyer's profile.

  Consider a profiling function f. We define a function $f' : (i, v)_{i\in\mathbb{S}} \to lpts$ as $f' = g \circ f$, i.e. f' first evaluates f on input PH to obtain a profiling result, and second evaluates g on input that result to get a number of *lpts*. In the profiling phase of Π_{LP}, in $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$, f must be replaced by f'. To allow $\mathcal{B}_k$ to prove in ZK correctness of the evaluation of g, $\mathcal{V}$ can follow the same approach as for f. (In the simplest case, for each profiling result, $\mathcal{V}$ computes a signature that signs the result along with the number of *lpts* awarded to that result.) Additionally, $\mathcal{V}$ must be able to add *lpts* to the database without learning the amount of *lpts* added. Let p be the output of g. To do that, $\mathcal{B}_k$ computes a commitment c to p. The relation R_{pr} is extended to prove in ZK that the output of g is committed in c. Then $\mathcal{V}$ uses c to update the database by making use of the homomorphic property of the VC scheme.
- **Expiration.** To improve customer retention, many LPs require that *lpts* expire if $\mathcal{B}_k$ did not purchase anything in the last weeks or months. Our PPLP can be extended to support expiration. In the purchase phase, $\mathcal{V}$ signs a commitment to the date of purchase along with the VC *vc* and the commitment c. In the next purchase, or when redeeming *lpts*, $\mathcal{B}_k$ proves in ZK that the committed date is recent enough.
- **Database Consistency.** In Π_{LP}, a user can register several buyer identities and thus hold several databases. This frequently leads to the creation of better profiles. Nevertheless, if needed, authentication during registration can be applied to allow $\mathcal{V}$ to reject multiple registrations by a single user. Also, in Π_{LP}, several users can share a buyer identity. When this is undesirable, Π_{HD} can be extended with existing mechanisms to discourage transferability of anonymous credentials [12].
- **Revocation.** Many LPs require the ability of revoking registrations. Our PPLP can be extended with existing mechanisms proposed for revocation

of anonymous credentials [12]. These mechanisms, like the non-transferability mechanisms discussed above, would require $\mathcal{V}$ to blindly sign some user secret along with vc and c.

– **Multiple Vendors.** Many LPs involve multiple vendors from the same or from different organizations. Unlinkability should be maintained when a buyer interacts with more than one vendor, which means that the public key of a vendor cannot be revealed when interacting with the next vendor. The simplest way to extend our PPLP to support multiple vendors, also used in the recent work in [5], is that vendors share their signing secret key. This is undesirable especially when different organizations are involved. Another possibility is to allow $\mathcal{B}_k$ to prove in ZK that she uses a signature signed by a vendor in the LP without revealing his public key. To this end, signatures on the public keys of the vendors in the LP are published. With structure preserving signatures, which we use in our instantiation of Π_{HD}, $\mathcal{B}_k$ can prove in ZK that the public key for a signature on vc and c is signed in one of the published signatures, without revealing that public key.
  Double-spending detection with multiple vendors would require that they have online access to a central database. In [5], a method for off-line detection is proposed, which our PPLP can implement.

4.3 Implementation and Efficiency

We have implemented Π_{LP} in the Python programming language, using the Charm cryptographic framework [2], on a computer equipped with an Intel Core i5-7300U CPU clocked at 2.60 GHz, and 8 gigabytes of RAM. Although we have used a single core implementation of the protocol for our evaluations, we comment that the protocol may benefit from parallelization in the setup phase. The BN256 curve has been used for the pairing group setup. As described in §3.4, we use the compiler in [10] to compute UC ZK proofs for R_{rd} and R_{pr}, and the Paillier encryption scheme uses a 2048 bit key in both cases. We also employ the range proof scheme described in [7] in the redemption and profiling phases, which requires the computation and storage of a CRS of about 152 bytes. For the purposes of evaluating the profiling phase, we have used a profiling function f which checks whether the sum of the amounts paid by $\mathcal{B}_k$ for products represented by a specified set of database entries surpasses a specified threshold value. Table 2 depicts the mean computation times (and the standard deviation) for all phases of the protocol over 100 trial runs, against a database size $N = \{15000, 65000\}$, which we consider adequate for shops of average size.[1] The timings take into account the database update operation performed by $\mathcal{V}$ for each of the phases. The last three rows of Table 2 list the mean computation times for the profiling phase when sets $\mathbb{S}$ of 1, 5, and 10 database entries were passed as input to the profiling function. The execution times for the profiling phase grow with the size of $\mathbb{S}$. However, the computation cost is independent of N. We remark that the purchase and redemption phases do not depend on $\mathbb{S}$, and that the profiling phase does not have real-time requirements.

Table 2. PPLP phase execution times in seconds

Phase	N = 15000		N = 65000	
	Time (s)	SD	Time (s)	SD
Registration	0.03	0.0004	0.04	0.0009
Purchase	4.08	0.5	4.89	0.5
Redemption	11.32	0.4	12.92	0.4
1 entry Profiling	11.51	0.4	13.08	0.4
5 entry Profiling	57.55	0.8	65.42	0.9
10 entry Profiling	115.08	1.2	130.91	1.1

5 Related Work

VC Schemes. VC schemes [14, 27] can be based on different assumptions such as CDH, RSA and DHE. We could use VCs secure under the more standard CDH or RSA assumptions, but the VCs based on DHE have efficiency advantages. A mercurial VC scheme based on DHE was proposed in [27], and subsequently DHE VC schemes were used in [20, 24, 25]. In our instantiation of Π_{HD}, we extend the VC scheme with signatures to enable ZK proofs of an opening w_i.

Polynomial commitments (PCs) allow us to commit to a polynomial and open the commitment to an evaluation of the polynomial. PCs can be used as VCs by committing to a polynomial that interpolates the vector to be committed. In [22], a construction of PCs from the

1 The number of SKUs of a typical supermarket, which is a kind of shop that frequently uses loyalty programs, is anywhere from 15000 to 60000 (https://www.fmi.org/our-research/supermarket-facts)

SDH assumption is proposed. A further generalization of VCs and PCs are functional commitments [26].

ZK Proofs for Large Datasets. In most ZK proofs, the computation and communication costs grow linearly with the size of the witness, which is inadequate for proofs about datasets of large size N. However, some techniques attain costs sublinear in N. Probabilistically checkable proofs [23] achieve verification cost sublinear in N, but the cost for the prover is linear in N. In succinct non-interactive arguments of knowledge [17], verification cost is independent of N, but the cost for the prover is still linear in N. ZK proofs for oblivious RAM programs [30] consist of a setup phase where the prover commits to the dataset, with cost linear in N for the prover and constant for the verifier. After setup, multiple proofs can be computed about the dataset with cost sublinear (proportional to the runtime of an ORAM program) for prover and verifier.

Our construction is somehow similar to [30], i.e. a database is committed, and then ZK proofs are computed. Storage cost is linear in N. However, the verification cost of a ZK proof is constant an independent of N. To compute a ZK proof, only the cost of computing an opening w_i is linear in N, but w_i can be reused and updated with cost independent of N. Therefore, computing a ZK proof has an amortized cost independent of N, which makes our construction practical for large databases.

Updatable Anonymous Credentials. In an anonymous credential (AC) [12] scheme, issuers sign user attributes. Users show that they possess a signature from an issuer on their attributes and selectively disclose, or prove in ZK statements, about their attributes. Unlinkability ensures that shows of a credential cannot be linked to each other or to the issuance of the credential. Recently, an updatable AC scheme has been proposed in [5] and applied to LPs. An updatable AC scheme allows users to update their signed attributes blindly, i.e., without the issuer knowing the attribute values. An HD can thus also be used to construct an updatable AC scheme. In [5], the cost of proving and verifying statements about attributes grows with the number of attributes signed in a credential. In their LP, only the number of $lpts$ is signed, and vendors do not receive any information on buyer profiles. Our construction Π_{HD} provides ZK proofs of cost independent of N. This allows us to design a PPLP where, in addition to $lpts$, $\mathcal{V}$ signs and updates the whole purchase history of buyers, which could consist of hundreds or thousands of products, with a computation and computation cost comparable to [5]. Buyers can then be profiled based on the result of a profiling algorithm, without disclosing further purchase data to $\mathcal{V}$.

6 Acknowledgements

The authors wish to thank the anonymous reviewers for their comments. This research is supported by the Luxembourg National Research Fund (FNR) CORE project "Stateful Zero-Knowledge" (Project code: C17/11650748).

References

[1] Masayuki Abe, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Optimal structure-preserving signatures in asymmetric bilinear groups. In *CRYPTO 2011*, pages 649–666.

[2] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. Charm: a framework for rapidly prototyping cryptosystems. *J. Cryptographic Engineering*, 3(2):111–128, 2013.

[3] Roy L Anderson, Joan M Ziegler, and Jacob Y Wong. Anonymous merchandise delivery system, 2010. US Patent 7,693,798.

[4] Alberto Blanco-Justicia and Josep Domingo-Ferrer. Privacy-preserving loyalty programs. In *DPM 2014, SETOP 2014, QASA 2014*, pages 133–146.

[5] Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. Updatable anonymous credentials and applications to incentive systems. In *ACM CCS 2019*, pages 1671–1685.

[6] Jan Bobolz, Fabian Eidens, Stephan Krenn, Daniel Slamanig, and Christoph Striecks. Privacy-preserving incentive systems with highly efficient point-collection. *IACR Cryptol. ePrint Arch.*, 2020:382.

[7] Jan Camenisch, Rafik Chaabouni, and Abhi Shelat. Efficient protocols for set membership and range proofs. In *ASIACRYPT 2008*, pages 234–252.

[8] Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. UC commitments for modular protocol design and applications to revocation and attribute tokens. In *CRYPTO 2016*, pages 208–239.

[9] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *EUROCRYPT 2005*, pages 302–321.

[10] Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In *ASIACRYPT 2011*, pages 449–467.

[11] Jan Camenisch, Anja Lehmann, Gregory Neven, and Alfredo Rial. Privacy-preserving auditing for attribute-based credentials. In *ESORICS 2014*, pages 109–127.

[12] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *EUROCRYPT 2001*, pages 93–118.

[13] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS 2001*, pages 136–145.

[14] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *PKC 2013*, pages 55–72.

[15] George Danezis, Markulf Kohlweiss, Benjamin Livshits, and Alfredo Rial. Private client-side profiling with random forests and hidden markov models. In *PETS 2012*, pages 18–37.

[16] Matthias Enzmann and Markus Schneider. A privacy-friendly loyalty system for electronic marketplaces. In *2004 IEEE International Conference on e-Technology, e-Commerce, and e-Services (EEE 04)*, pages 385–393.

[17] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT 2013*, pages 626–645.

[18] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[19] Oliver Hinz, Eva Gerstmeier, Omid Tafreschi, Matthias Enzmann, and Markus Schneider. Customer loyalty programs and privacy concerns. *BLED 2007 Proceedings*, page 32, 2007.

[20] Malika Izabachène, Benoît Libert, and Damien Vergnaud. Block-wise p-signatures and non-interactive anonymous credentials with efficient attributes. In *Cryptography and Coding - 13th IMA International Conference, IMACC 2011*, pages 431–450.

[21] Tun-Min Catherine Jai and Nancy J King. Privacy versus reward: Do loyalty programs increase consumers' willingness to share personal information with third-party advertisers and data brokers? *Journal of Retailing and Consumer Services*, 28:296–303, 2016.

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

[23] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *ACM STOC 1992*, pages 723–732.

[24] Markulf Kohlweiss and Alfredo Rial. Optimally private access control. In *WPES 2013*, pages 37–48, 2013.

[25] Benoît Libert, Thomas Peters, and Moti Yung. Group signatures with almost-for-free revocation. In *CRYPTO 2012*, pages 571–589.

[26] Benoît Libert, Somindu C. Ramanna, and Moti Yung. Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In *ICALP 2016*, pages 30:1–30:14.

[27] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *TCC 2010*, pages 499–517.

[28] Philip Marquardt, David Dagon, and Patrick Traynor. Impeding individual user profiling in shopper loyalty programs. In *FC 2011*, pages 93–101.

[29] Milica Milutinovic, Italo Dacosta, Andreas Put, and Bart De Decker. An advanced, privacy-friendly loyalty system. In *Privacy and Identity Management for Emerging Services and Technologies - 8th IFIP WG 9.2, 9.5, 9.6/11.7, 11.4, 11.6 International Summer School*, pages 128–138.

[30] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. Sublinear zero-knowledge arguments for RAM programs. In *EUROCRYPT 2017*, pages 501–531.

[31] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO '91*, pages 129–140.

[32] Arrianto Mukti Wibowo, Kwok-Yan Lam, and Gary S. H. Tan. Loyalty program scheme for anonymous payment system. In *Electronic Commerce and Web Technologies, EC-Web 2000*, pages 253–265.

A UC Security

We prove our protocol secure in the universal composability framework [13]. The UC framework allows one to define and analyze the security of cryptographic protocols so that security is retained under an arbitrary composition with other protocols. The security of a protocol is defined by means of an ideal protocol that carries out the desired task. In the ideal protocol, all parties send their inputs to an ideal functionality $\mathcal{F}$ for the task. $\mathcal{F}$ locally computes the outputs of the parties and provides each party with its prescribed output.

The security of a protocol φ is analyzed by comparing the view of an environment $\mathcal{Z}$ in a real execution of φ against that of $\mathcal{Z}$ in the ideal protocol defined in $\mathcal{F}_\varphi$. $\mathcal{Z}$ chooses the inputs of the parties and collects their outputs. In the real world, $\mathcal{Z}$ can communicate freely with an adversary $\mathcal{A}$ who controls both the network and any corrupt parties. In the ideal world, $\mathcal{Z}$ interacts with dummy parties, who simply relay inputs and outputs between $\mathcal{Z}$ and $\mathcal{F}_\varphi$, and a simulator $\mathcal{S}$. We say that a protocol φ securely realizes $\mathcal{F}_\varphi$ if $\mathcal{Z}$ cannot distinguish the real world from the ideal world, i.e., $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and parties running protocol φ or with $\mathcal{S}$ and dummy parties relaying to $\mathcal{F}_\varphi$.

A protocol $\varphi^{\mathcal{G}}$ securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model when φ is allowed to invoke the ideal functionality $\mathcal{G}$. Therefore, for any protocol ψ that securely realizes $\mathcal{G}$, the composed protocol φ^{ψ}, which is obtained by replacing each invocation of an instance of $\mathcal{G}$ with an invocation of an instance of ψ, securely realizes $\mathcal{F}$.

In the ideal functionalities described in this paper, we consider static corruptions. When describing ideal functionalities, we use the following conventions as in [8].

Interface Naming Convention. An ideal functionality can be invoked by using one or more interfaces. The name of a message in an interface consists of three fields separated by dots, e.g., hd.read.ini in

$\mathcal{F}_{\mathrm{HD}}$ in §3.1. The first field indicates the name of the functionality and is the same in all interfaces of the functionality. This field is useful for distinguishing between invocations of different functionalities in a hybrid protocol that uses two or more different functionalities. The second field indicates the kind of action performed by the functionality and is the same in all messages that the functionality exchanges within the same interface. The third field distinguishes between the messages that belong to the same interface, and can take the following different values. A message hd.read.ini is the incoming message received by the functionality, i.e., the message through which the interface is invoked. A message hd.read.end is the outgoing message sent by the functionality, i.e., the message that ends the execution of the interface. The message hd.read.sim is used by the functionality to send a message to $\mathcal{S}$, and the message hd.read.rep is used to receive a message from $\mathcal{S}$.

Network vs Local Communication. Identities of interactive Turing machine instances (ITI) consist of a party identifier *pid* and a session identifier *sid*. A set of parties in an execution of a system of interactive Turing machines is a protocol instance if they have the same session identifier *sid*. ITIs can pass direct inputs to and outputs from "local" ITIs that have the same *pid*. An ideal functionality $\mathcal{F}$ has $pid = \perp$ and is considered local to all parties. An instance of $\mathcal{F}$ with the session identifier *sid* only accepts inputs from and passes outputs to machines with the same session identifier *sid*. Some functionalities require the session identifier to have some structure. Those functionalities check whether the session identifier possesses the required structure in the first message that invokes the functionality. For the subsequent messages, the functionality implicitly checks that the session identifier equals the session identifier used in the first message. Communication between ITIs with different party identifiers must take place over the network. The network is controlled by $\mathcal{A}$, meaning that he can arbitrarily delay, modify, drop, or insert messages.

Query Identifiers. Some interfaces in a functionality can be invoked more than once. When the functionality sends a message hd.read.sim to $\mathcal{S}$ in such an interface, a query identifier *qid* is included in the message. The query identifier must also be included in the response hd.read.rep sent by $\mathcal{S}$. The query identifier is used to identify the message hd.read.sim to which $\mathcal{S}$ replies with a message hd.read.rep. We note that, typically, $\mathcal{S}$ in the security proof may not be able to provide an immediate answer to the functionality after receiving a message hd.read.sim. The reason is that $\mathcal{S}$ typically needs to interact with the copy of $\mathcal{A}$ it runs in order to produce the message hd.read.rep, but $\mathcal{A}$ may not provide the desired answer or may provide a delayed answer. In such cases, when the functionality sends more than one message hd.read.sim to $\mathcal{S}$, $\mathcal{S}$ may provide delayed replies, and the order of those replies may not follow the order of the messages received.

Aborts. When an ideal functionality $\mathcal{F}$ aborts after being activated with a message sent by a party, we mean that $\mathcal{F}$ halts the execution of its program and sends a special abortion message to the party that invoked the functionality. When an ideal functionality $\mathcal{F}$ aborts after being activated with a message sent by $\mathcal{S}$, we mean that $\mathcal{F}$ halts the execution of its program and sends a special abortion message to the party that receives the outgoing message from $\mathcal{F}$ after $\mathcal{F}$ is activated by $\mathcal{S}$.

B Security Definitions

Description of $\mathcal{F}_{\mathrm{REG}}$. $\mathcal{F}_{\mathrm{REG}}$ is parameterized by a message space $\mathcal{M}$.

1. On input (reg.register.ini, sid, v) from a party $\mathcal{T}$:
   - Abort if $sid \neq (\mathcal{T}, sid')$, or if $v \notin \mathcal{M}$ or if there is a tuple $(sid, v', 0)$ stored.
   - Store $(sid, v, 0)$.
   - Send (reg.register.sim, sid, v) to $\mathcal{S}$.

S. On input (reg.register.rep, sid) from the simulator $\mathcal{S}$:
   - Abort if $(sid, v, 0)$ is not stored or if $(sid, v, 1)$ is already stored.
   - Store $(sid, v, 1)$ and parse sid as $(\mathcal{T}, sid')$.
   - Send (reg.register.end, sid) to $\mathcal{T}$.

2. On input (reg.retrieve.ini, sid) from any party $\mathcal{P}$:
   - If $(sid, v, 1)$ is stored, set $v' \leftarrow v$; else set $v' \leftarrow \perp$.
   - Create a fresh qid and store $(qid, \mathcal{P}, v')$.
   - Send (reg.retrieve.sim, sid, qid, v') to $\mathcal{S}$.

S. On input (reg.retrieve.rep, sid, qid) from $\mathcal{S}$:
   - Abort if $(qid, \mathcal{P}, v')$ is not stored.
   - Delete the record $(qid, \mathcal{P}, v')$.
   - Send (reg.retrieve.end, sid, v') to $\mathcal{P}$.

Description of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ is parameterized by a ppt alg. CRS.Setup.

1. On input (crs.get.ini, sid) from any party $\mathcal{P}$:
   - If (sid, crs) is not stored, run $crs \leftarrow$ CRS.Setup and store (sid, crs).

- Create a fresh qid and store $(qid, \mathcal{P})$.
- Send $(\mathsf{crs.get.sim}, sid, qid, crs)$ to $\mathcal{S}$.

S. On input $(\mathsf{crs.get.rep}, sid, qid)$ from the simulator $\mathcal{S}$:
- Abort if $(qid, \mathcal{P})$ is not stored.
- Delete the record $(qid, \mathcal{P})$.
- Send $(\mathsf{crs.get.end}, sid, crs)$ to $\mathcal{P}$.

Description of $\mathcal{F}_{\mathrm{NYM}}$. $\mathcal{F}_{\mathrm{NYM}}$ is parameterized by a message space $\mathcal{M}$, a universe of pseudonyms $\mathbb{U}_p$, and a leakage function l, which leaks the message length.

1. On input $(\mathsf{nym.send.ini}, sid, m, P)$ from $\mathcal{T}_k$:
   - Abort if $sid \neq (\mathcal{R}, sid')$, or if $m \notin \mathcal{M}$, or if $P \notin \mathbb{U}_p$.
   - Create a fresh qid and store $(qid, P, \mathcal{T}_k, m)$.
   - Send $(\mathsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{S}$.

S. On input $(\mathsf{nym.send.rep}, sid, qid)$ from $\mathcal{S}$:
- Abort if $(qid, P, \mathcal{T}_k, m)$ is not stored.
- Store $(sid, P, \mathcal{T}_k)$.
- Delete the record $(qid, P, \mathcal{T}_k, m)$.
- Parse sid as $(\mathcal{R}, sid')$.
- Send $(\mathsf{nym.send.end}, sid, m, P)$ to $\mathcal{R}$.

2. On input $(\mathsf{nym.reply.ini}, sid, m, P)$ from $\mathcal{R}$:
   - Abort if $sid \neq (\mathcal{R}, sid')$, or if $m \notin \mathcal{M}$, or if $P \notin \mathbb{U}_p$.
   - Abort if there is not a tuple $(sid, P', \mathcal{T}_k)$ stored such that $P' = P$.
   - Create a fresh qid and store $(qid, P, \mathcal{T}_k, m)$.
   - Delete the tuple $(sid, P, \mathcal{T}_k)$.
   - Send $(\mathsf{nym.reply.sim}, sid, qid, l(m))$ to $\mathcal{S}$.

S. On input $(\mathsf{nym.reply.rep}, sid, qid)$ from $\mathcal{S}$:
- Abort if $(qid, P, \mathcal{T}_k, m)$ is not stored.
- Delete the record $(qid, P, \mathcal{T}_k, m)$.
- Send $(\mathsf{nym.send.end}, sid, m, P)$ to $\mathcal{T}_k$.

Description of $\mathcal{F}_{\mathrm{ZK}}^{R}$. The functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$ is parameterized by a description of a relation R and by a universe of pseudonyms $\mathbb{U}_p$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ interacts with provers $\mathcal{P}_k$ and a verifier $\mathcal{V}$.

1. On input $(\mathsf{zk.prove.ini}, sid, wit, ins, P)$ from $\mathcal{P}_k$:
   - Abort if $sid \neq (\mathcal{V}, sid')$, or if $(wit, ins) \notin R$, or if $P \notin \mathbb{U}_p$.
   - Create a fresh qid and store (qid, ins, P).
   - Send $(\mathsf{zk.prove.sim}, sid, qid, ins)$ to $\mathcal{S}$.

S. On input $(\mathsf{zk.prove.rep}, sid, qid)$ from $\mathcal{S}$:
- Abort if (qid, ins, P) is not stored.
- Parse sid as $(\mathcal{V}, sid')$.
- Delete the record (qid, ins, P).
- Send $(\mathsf{zk.prove.end}, sid, ins, P)$ to the verifier $\mathcal{V}$.

Commitment Schemes. A commitment scheme must be hiding and binding. The hiding property ensures that a commitment c to x does not reveal any information about x, whereas the binding property ensures that c cannot be opened to another value x'.

Definition B.1. [Hiding Property] For any PPT adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{l} par_c \xleftarrow{\$} \mathsf{C.Setup}(1^k);\\ (x_0, st) \xleftarrow{\$} \mathcal{A}(par_c);\\ x_1 \xleftarrow{\$} \mathcal{M};\ b \xleftarrow{\$} \{0,1\};\\ (c, o) \xleftarrow{\$} \mathsf{C.Com}(par_c, x_b);\\ b' \xleftarrow{\$} \mathcal{A}(st, c) :\ x_0 \in \mathcal{M}\ \wedge\ b = b' \end{array}\right] \leq \frac{1}{2} + \epsilon(k)\ .$$

Definition B.2. [Binding Property] For any PPT adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{l} par_c \xleftarrow{\$} \mathsf{C.Setup}(1^k);\\ (c, x, o, x', o') \xleftarrow{\$} \mathcal{A}(par_c) :\\ x \in \mathcal{M}\ \wedge\ x' \in \mathcal{M}\ \wedge\\ 1 = \mathsf{C.Vf}(par_c, c, x, o)\ \wedge\\ 1 = \mathsf{C.Vf}(par_c, c, x', o')\ \wedge\ x \neq x' \end{array}\right] \leq \epsilon(k)\ .$$

Vector Commitments. A VC scheme must be hiding and binding. Informally, the hiding property ensures that a VC vc does not reveal any information about the committed vector $\mathbf{x}$, while the binding property ensures that vc cannot be opened to two different messages $\mathbf{x}[i]$ and $\mathbf{x}'[i]$ for any $i \in [1, \ell]$.

Definition B.3. [Hiding Property] For any PPT $\mathcal{A}$ and ℓ polynomial in k,

$$\Pr\left[\begin{array}{l} par \leftarrow \mathsf{VC.Setup}(1^k, \ell);\\ (\mathbf{x}_0, st) \leftarrow \mathcal{A}(par);\ r \leftarrow \mathcal{R};\\ \mathbf{x}_1 \leftarrow \mathcal{M}^\ell;\ b \leftarrow \{0,1\};\\ vc \leftarrow \mathsf{VC.Com}(par, \mathbf{x}_b, r);\\ b' \leftarrow \mathcal{A}(st, vc) :\ b = b'\ \wedge\ \mathbf{x}_0 \in \mathcal{M}^\ell \end{array}\right] = \frac{1}{2} + \epsilon(k)\ .$$

Definition B.4. [Binding Property] For any PPT $\mathcal{A}$ and ℓ polynomial in k,

$$\Pr\left[\begin{array}{l} par \leftarrow \mathsf{VC.Setup}(1^k, \ell);\\ (vc, i, x, x', w, w') \leftarrow \mathcal{A}(par) :\\ \mathsf{VC.Vf}(par, vc, x, i, w) = 1\ \wedge\ x \neq x'\ \wedge\\ \mathsf{VC.Vf}(par, vc, x', i, w') = 1\ \wedge\\ i \in [1, \ell]\ \wedge x, x' \in \mathcal{M} \end{array}\right] \leq \epsilon(k)\ .$$

Signature Schemes. A signature scheme must be existentially unforgeable, which ensures that it is not feasible to output a signature on a message without knowledge of the secret key or of another signature on that message.

Definition B.5. [Existential Unforgeability] Let $\mathcal{O}_s$ be an oracle that, on input sk and a message $m \in \mathcal{M}$, outputs $\mathsf{S.Sign}(sk, m)$, and let S_s be a set that contains the messages sent to $\mathcal{O}_s$. For any ppt adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{l}(sk, pk) \xleftarrow{\$} \mathsf{S.KG}(1^k);\\ (m, sig) \xleftarrow{\$} \mathcal{A}(pk)^{\mathcal{O}_s(sk,\cdot)} :\\ 1 = \mathsf{S.Vf}(pk, sig, m) \wedge\\ m \in \mathcal{M} \ \wedge \ m \notin S_s\end{array}\right] \leq \epsilon(k) \ .$$

C Security Analysis

To prove that Π_{HD} securely realizes $\mathcal{F}_{\mathrm{HD}}$, we must show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{S}$ thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\mathrm{HD}}$ for all corrupt parties in the ideal world.

Our simulator $\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\mathrm{REG}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_r}$. When any of the copies of those functionalities aborts, $\mathcal{S}$ implicitly forwards the abortion message to $\mathcal{A}$ if the functionality sends the abortion message to a corrupt party.

First, we analyze the security of Π_{HD} when (a subset of) readers $\mathcal{R}_k$ are corrupt. Second, we analyze the security of Π_{HD} when the updater $\mathcal{U}$ is corrupt. We do not analyze in detail the security of Π_{HD} when $\mathcal{U}$ and (a subset of) readers $\mathcal{R}_k$ are corrupt. We note that, in Π_{HD}, honest readers communicate with $\mathcal{U}$ but not with other readers. Therefore, for this case the simulator and the security proof are very similar to the case where only $\mathcal{U}$ is corrupt.

Description of $\mathcal{S}$ for Corrupt $\mathcal{R}_k$. We describe $\mathcal{S}$ for the case in which (a subset of) $\mathcal{R}_k$ are corrupt.

$\mathcal{A}$ requests parameters. On input from $\mathcal{A}$ the message ($\mathsf{crs.get.ini}$, sid), the simulator $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ sends ($\mathsf{crs.get.sim}$, sid, qid, $\langle par, par_c\rangle$), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ receives parameters. On input from $\mathcal{A}$ the message ($\mathsf{crs.get.rep}$, sid, qid), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ sends ($\mathsf{crs.get.end}$, sid, $\langle par, par_c\rangle$), $\mathcal{S}$ sends ($\mathsf{crs.get.end}$, sid, $\langle par, par_c\rangle$) to $\mathcal{A}$.

$\mathcal{A}$ sends a ZK proof. On input ($\mathsf{zk.prove.ini}$, sid, wit_r, ins_r, P) from $\mathcal{A}$, $\mathcal{S}$ stores (wit_r, ins_r, P) and runs $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ on that input. When $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ sends the message ($\mathsf{zk.prove.sim}$, sid, qid, ins_r), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest $\mathcal{U}$ receives ZK proof. On input the message ($\mathsf{zk.prove.rep}$, sid, qid) from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ on that input. When the copy of $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ sends ($\mathsf{zk.prove.end}$, sid, ins_r, P), $\mathcal{S}$ parses the stored wit_r as ($sig, vc, c, r_2, o_2, \langle i, vr_i, w_i, o_i, or_i\rangle_{i\in\mathbb{S}}$) and does the following:

- If the pseudonym P was received before, $\mathcal{S}$ aborts. (We recall that the honest updater also aborts if a pseudonym is reused.)
- Else, if $\mathcal{A}$ did not receive a signature sig on (vc, c), $\mathcal{S}$ outputs failure.
- Else, $\mathcal{S}$ finds the stored tuple ($vc, \mathbf{x}, r, c, s, o$) that contains vc and c. If, for any $i \in \mathbb{S}$, $\mathbf{x}[i] \neq vr_i$, $\mathcal{S}$ outputs failure.
- Else, $\mathcal{S}$ takes c' from ins_r and runs $\mathcal{F}_{\mathrm{NYM}}$ on input ($\mathsf{nym.reply.ini}$, sid, (Open c'), P). When $\mathcal{F}_{\mathrm{NYM}}$ sends ($\mathsf{nym.reply.sim}$, sid, qid, l(Open c')), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ receives (Open c') message. When $\mathcal{A}$ sends the message ($\mathsf{nym.reply.rep}$, sid, qid), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{NYM}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{NYM}}$ sends ($\mathsf{nym.send.end}$, sid, (Open c'), P), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ sends the commitment (and opening). On input the message ($\mathsf{nym.send.ini}$, sid, m, P) from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{NYM}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{NYM}}$ sends ($\mathsf{nym.send.sim}$, sid, qid, $l(m)$), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest $\mathcal{U}$ gets the commitment (and opening). On input the message ($\mathsf{nym.send.rep}$, sid, qid) from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{NYM}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{NYM}}$ sends ($\mathsf{nym.send.end}$, sid, m, P), $\mathcal{S}$ parses m as either c_1 or (c_1, s, o') and proceeds as follows.

- If $m = c_1$, $\mathcal{S}$ aborts if P was already received, else stores ($wit_r \leftarrow \perp, ins_r \leftarrow \perp, P$) and sends ($\mathsf{hd.read.ini}$, sid, P, $\perp$) to $\mathcal{F}_{\mathrm{HD}}$.
- If $m = (c_1, s, o')$, $\mathcal{S}$ finds the stored tuple (wit_r, ins_r, P) with the same pseudonym previously received when $\mathcal{A}$ sends a ZK proof. If no such tuple exists, $\mathcal{S}$ aborts. Otherwise $\mathcal{S}$ parses ins_r as (pk, $par, par_c, vc', c', pnic, \langle c'_i, cr'_i\rangle_{i\in\mathbb{S}}$) and aborts if ($s, o'$) is not a valid opening for c'. If the opening is valid, $\mathcal{S}$ parses wit_r as (sig, vc, c, r_2, $o_2, \langle i, vr_i, w_i, o_i, or_i\rangle_{i\in\mathbb{S}}$) and finds the stored tuple ($vc, \mathbf{x}, r, c, s', o$) that contains vc and c.
  - If $s = s'$, $\mathcal{S}$ aborts (in this case, $\mathcal{A}$ double-spent a VC).
  - If $s \neq s'$, $\mathcal{S}$ outputs failure.
  - If $s' = \perp$, the simulator $\mathcal{S}$ stores ($s, o' - o_2$) in that tuple and sends ($\mathsf{hd.read.ini}$, sid, $P, (i, vr_i, c_i, o_i, cr_i, or_i)_{i\in\mathbb{S}}$) to $\mathcal{F}_{\mathrm{HD}}$. When

$\mathcal{F}_{\mathrm{HD}}$ sends $(\mathsf{hd.read.sim}, sid, qid, (c_i, cr_i)_{i \in \mathbb{S}})$, $\mathcal{S}$ sends $(\mathsf{hd.read.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{HD}}$.

Honest $\mathcal{U}$ sends update. When $\mathcal{F}_{\mathrm{HD}}$ sends the simulator $\mathcal{S}$ the message $(\mathsf{hd.update.sim}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{hd.update.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{HD}}$. When $\mathcal{F}_{\mathrm{HD}}$ sends $(\mathsf{hd.update.end}, sid, P, (i, vu_i)_{i \in [1,N]})$, $\mathcal{S}$ sets $\mathbf{x}_u[i] \leftarrow vu_i$ (for all $i \in [1, N]$) and picks a random value s_2. $\mathcal{S}$ picks the stored tuple (wit_r, ins_r, P) and the tuple $(vc, \mathbf{x}, r, c, s, o)$ that contains vc and c in wit_r. $\mathcal{S}$ follows Π_{HD} to compute the new values of vc, $\mathbf{x}$, and c and updates the tuple $(vc, \mathbf{x}, r, c, s, o)$ accordingly (r is updated by using wit_r). If a signing key pair (pk, sk) is not stored, $\mathcal{S}$ computes and stores (pk, sk) and registers pk with the copy of $\mathcal{F}_{\mathrm{REG}}$. $\mathcal{S}$ computes a signature sig on vc and c and runs $\mathcal{F}_{\mathrm{NYM}}$ on input $(\mathsf{nym.reply.ini}, sid, \langle \mathbf{x}_u, s_2, sig \rangle, P)$. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.reply.sim}, sid, qid, l(\langle \mathbf{x}_u, s_2, sig \rangle))$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ receives update. When $\mathcal{A}$ sends $(\mathsf{nym.reply.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on that input. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.reply.end}, sid, \langle \mathbf{x}_u, s_2, sig \rangle, P)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ requests public key. On input $(\mathsf{reg.retrieve.ini}, sid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{REG}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{REG}}$ sends $(\mathsf{reg.retrieve.sim}, sid, qid, pk)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ receives public key. On input $(\mathsf{reg.retrieve.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{REG}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{REG}}$ sends $(\mathsf{reg.retrieve.end}, sid, pk)$, $\mathcal{S}$ sends $(\mathsf{reg.retrieve.end}, sid, pk)$ to $\mathcal{A}$.

Theorem C.1. When (a subset of) $\mathcal{R}_k$ are corrupt, Π_{HD} securely realizes $\mathcal{F}_{\mathrm{HD}}$ in the $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\mathrm{REG}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_r}$-hybrid model if the VC scheme is binding, the commitment scheme is binding, and the signature scheme is existentially unforgeable.

Proof of Theorem C.1. We show by means of a series of hybrid games that $\mathcal{Z}$ cannot distinguish between the real-world protocol and our simulation with non-negligible probability. $\Pr[\mathbf{Game}\ i]$ is the probability that $\mathcal{Z}$ distinguishes $\mathbf{Game}\ i$ from the real-world protocol.

Game 0: This game corresponds the real-world protocol. Therefore, $\Pr[\mathbf{Game}\ 0] = 0$.

Game 1: Game 1 follows **Game** 0, with the exception that **Game** 1 outputs failure when $\mathcal{A}$ sends a signature sig on vc and c and $\mathcal{A}$ did not receive a signature on those values before. The probability that **Game** 1 outputs failure is bound by theorem C.2.

Theorem C.2. Under the existential unforgeability property of the signature scheme, we have that $|\Pr[\mathbf{Game}\ 1] - \Pr[\mathbf{Game}\ 0]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{unf-sig}}$.

We omit a formal proof of theorem C.2. In a nutshell, we can construct an algorithm B that, given an $\mathcal{A}$ that makes **Game** 1 output failure with non-negligible probability, wins the existential unforgeability game with that probability. B receives the public key from the challenger and registers it with $\mathcal{F}_{\mathrm{REG}}$. To compute signatures on vc and c, B uses the signing oracle. When $\mathcal{A}$ sends a signature sig on vc and c as part of the witness wit_r sent to $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ such that B did not compute before a signature on those values, B submits sig along with (vc, c) in order to win the existential unforgeability game.

Game 2: Game 2 follows **Game** 1, with the exception that **Game** 2 outputs failure when $\mathcal{A}$ sends a witness $(sig, vc, c, r_2, o_2, \langle i, vr_i, w_i, o_i, or_i \rangle_{i \in \mathbb{S}})$ and an instance $(pk, par, par_c, vc', c', pnic, \langle c'_i, cr'_i \rangle_{i \in \mathbb{S}})$ such that w_i opens the commitment vc' to a value $vr_i \neq \mathbf{x}[i]$ at position i, where $\mathbf{x}$ is the vector committed in vc (vc' is a rerandomization of vc). The probability that **Game** 2 outputs failure is bound by theorem C.3.

Theorem C.3. If the binding property of the VC scheme holds, $|\Pr[\mathbf{Game}\ 2] - \Pr[\mathbf{Game}\ 1]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{bin-vc}}$.

We omit a formal proof of theorem C.3. We can construct an algorithm B that, given an $\mathcal{A}$ that makes **Game** 2 output failure with non-negligible probability, wins the binding game of the VC scheme with that probability. B receives the parameters of the VC scheme from the challenger and stores them in the copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. When $\mathcal{A}$ sends an opening w_i that opens vc' to vr_i as described above, B computes another opening w'_i that opens vc' to $\mathbf{x}[i]$. We note that wit_r contains r_2, so B knows the randomness change of vc' with respect to vc and is able to compute w'_i. B sends vc' along with i, w_i, vr_i, w'_i, and $\mathbf{x}[i]$ to win the binding game of the VC scheme.

Game 3: Game 3 follows **Game** 2, with the exception that **Game** 3 outputs failure when $\mathcal{A}$ sends an opening (s, o') for a commitment c' such that a previous opening $(\hat{s}, \hat{o}')$ was received for a commitment $\hat{c}'$ and both c' and $\hat{c}'$ are rerandomizations of c. The probability that **Game** 3 outputs failure is bound by theorem C.4.

Theorem C.4. Under the binding property of the commitment scheme, we have that $|\Pr[\mathbf{Game}\ 3] - \Pr[\mathbf{Game}\ 2]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{bin-com}}$.

We omit a formal proof of theorem C.4. We can construct an algorithm B that, given an $\mathcal{A}$ that makes **Game** 3 output failure with non-negligible probability, wins the binding game of the commitment scheme with that probability. B receives the parameters of the commitment scheme from the challenger and stores them in the copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. When $\mathcal{A}$ sends an opening (s, o') for a commitment c' such that a previous opening $(\hat{s}, \hat{o}')$ was received for a commitment $\hat{c}'$ and both c' and $\hat{c}'$ are both rerandomizations of a commitment c, B computes $(s, o' - o_2)$ and $(\hat{s}, \hat{o}' - \hat{o}_2)$. The values o_2 and $\hat{o}_2$ are the randomness used to rerandomize c into c' and $\hat{c}'$ respectively, which B obtains through the witnesses wit_r sent to $\mathcal{F}_{\mathrm{ZK}}^{R_r}$. B sends c along with $(s, o' - o_2)$ and $(\hat{s}, \hat{o}' - \hat{o}_2)$ to win the binding game of the commitment scheme.

The distribution of **Game** 3 is identical to our simulation. This concludes the proof of theorems C.1.

<u>Description of $\mathcal{S}$ when $\mathcal{U}$ is Corrupt.</u> We describe $\mathcal{S}$ for the case in which $\mathcal{U}$ is corrupt.

$\mathcal{A}$ requests or receives parameters. $\mathcal{S}$ runs as in the case where (a subset of) $\mathcal{R}_k$ are corrupt.

$\mathcal{A}$ registers public key. On input $(\mathsf{reg.register.ini}, sid, pk)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{REG}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{REG}}$ sends $(\mathsf{reg.register.sim}, sid, pk)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ ends registration of public key. On input from $\mathcal{A}$ the message $(\mathsf{reg.register.rep}, sid)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{REG}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{REG}}$ sends $(\mathsf{reg.register.end}, sid)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest $\mathcal{R}_k$ starts ZK proof. On input from functionality $\mathcal{F}_{\mathrm{HD}}$ the message $(\mathsf{hd.read.sim}, sid, qid, (c_i, cr_i)_{i\in\mathbb{S}})$, the simulator $\mathcal{S}$ sends $(\mathsf{hd.read.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{HD}}$ and receives $(\mathsf{hd.read.end}, sid, P, f, (c_i, cr_i)_{i\in\mathbb{S}})$ from $\mathcal{F}_{\mathrm{HD}}$. If $f = 0$, $\mathcal{S}$ does the following:
- $\mathcal{S}$ computes a VC vc' to a random vector and a commitment c' to a random value s with opening o'.
- Parse the commitment c_i as $(c'_i, pnic, \mathsf{NIC.Vf})$.
- Parse the commitment cr_i as $(cr'_i, pnic, \mathsf{NIC.Vf})$.
- $\mathcal{S}$ sets the instance as $ins_r \leftarrow (pk, par, par_c, vc', c', pnic, \langle c'_i, cr'_i\rangle_{i\in\mathbb{S}})$ and stores (qid, ins_r, P) and (sid, P, ins_r, s, o').
- $\mathcal{S}$ sends $(\mathsf{zk.prove.sim}, sid, qid, ins_r)$ to $\mathcal{A}$.

$\mathcal{A}$ receives ZK proof. On input $(\mathsf{zk.prove.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ does the following:
- $\mathcal{S}$ sends an abortion message to $\mathcal{A}$ if (qid, ins_r, P) is not stored.
- $\mathcal{S}$ deletes the record (qid, ins_r, P).
- $\mathcal{S}$ sends $(\mathsf{zk.prove.end}, sid, ins_r, P)$ to $\mathcal{A}$.

$\mathcal{A}$ requests opening. On input from the adversary $\mathcal{A}$ the message $(\mathsf{nym.reply.ini}, sid, (\text{Open } c'), P)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input $(\mathsf{nym.reply.ini}, sid, (\text{Open } c'), P)$. When $\mathcal{F}_{\mathrm{NYM}}$ sends the message $(\mathsf{nym.reply.sim}, sid, qid, l(\text{Open } c'))$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest $\mathcal{R}_k$ receives request. On input the message $(\mathsf{nym.reply.rep}, sid, qid)$ from $\mathcal{A}$, the simulator $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{NYM}}$ on that input. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.end}, sid, (\text{Open } c'), P)$, $\mathcal{S}$ continues with the case "Honest $\mathcal{R}_k$ sends commitment (and opening)".

Honest $\mathcal{R}_k$ sends commitment (and opening). On input $(\mathsf{hd.read.sim}, sid, qid, (c_i, cr_i)_{i\in\mathbb{S}})$ from the functionality $\mathcal{F}_{\mathrm{HD}}$, $\mathcal{S}$ sends $(\mathsf{hd.read.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{HD}}$ and receives $(\mathsf{hd.read.end}, sid, P, f, (c_i, cr_i)_{i\in\mathbb{S}})$ from $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{S}$ computes a commitment c_1 to a random value. If $f = 1$, $\mathcal{S}$ sets $m \leftarrow c_1$. Else, after following the case "Honest $\mathcal{R}_k$ starts ZK proof", $\mathcal{S}$ picks the stored tuple (sid, P, ins_r, s, o') and sets $m \leftarrow \langle c_1, s, o'\rangle$. $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{NYM}}$ on input $(\mathsf{nym.send.ini}, sid, m, P)$. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ receives commitment (and opening). On input $(\mathsf{nym.send.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{NYM}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.end}, sid, m, P)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ sends update. On input from $\mathcal{F}_{\mathrm{NYM}}$ the message $(\mathsf{nym.reply.ini}, sid, m, P)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{NYM}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.reply.sim}, sid, qid, l(m))$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest $\mathcal{R}_k$ receives update. On input from $\mathcal{A}$ the message $(\mathsf{nym.reply.rep}, sid, qid)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{NYM}}$ on that input. When the copy of $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.end}, sid, m, P)$, $\mathcal{S}$ parses m as $(\mathbf{x}, s_2, sig)$, picks the stored tuple (sid, P, ins_r, s, o') and follows the steps described in Π_{HD} to verify sig. Then $\mathcal{S}$ sends $(\mathsf{hd.update.ini}, sid, P, \mathbf{x})$ to $\mathcal{F}_{\mathrm{HD}}$. When $\mathcal{F}_{\mathrm{HD}}$ sends the message $(\mathsf{hd.update.sim}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{hd.update.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{HD}}$.

Theorem C.5. When $\mathcal{U}$ is corrupt, Π_{HD} securely realizes $\mathcal{F}_{\mathrm{HD}}$ in the $\mathcal{F}_{\mathrm{REG}}$, $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_r}$-hybrid model if the VC scheme is hiding and the commitment scheme is hiding.

Proof of Theorem C.5. We show by means of a series of hybrid games that $\mathcal{Z}$ cannot distinguish between the real-world protocol and our simulation with non-negligible probability.

Game 0**:** This game corresponds to the real-world protocol. Therefore, $\Pr[\textbf{Game } 0] = 0$.

Game 1**: Game** 1 follows **Game** 0, with the exception that **Game** 1 does not run a copy of $\mathcal{F}_{\mathrm{ZK}}^{R_r}$. Instead, **Game** 1 sets the messages ($\mathsf{zk.prove.sim}$, sid, qid, ins_r) and ($\mathsf{zk.prove.end}$, sid, ins_r, P) directly. This change does not alter the view of $\mathcal{Z}$. Therefore, $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| = 0$

Game 2**: Game** 2 follows **Game** 1, with the exception that **Game** 2 replaces the VC vc' in ins_r by a commitment to a random vector. The probability that **Game** 1 and **Game** 2 are distinguished by $\mathcal{Z}$ is bound by theorem C.6.

Theorem C.6. Let M be the number of read operations sent by honest readers after their first read operation. Under the hiding property of the vector commitment scheme, we have that $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]| \leq M \cdot \mathrm{Adv}_{\mathcal{A}}^{\mathsf{hid-vc}}$.

Proof of Theorem C.6. We define a sequence of games. In **Game** 1.i, for the last i read operations (after their first read operation) sent by honest readers, the vector commitment vc' is replaced by a vector commitment to a random vector. Therefore, **Game** 1.0 corresponds to **Game** 1, while **Game** 1.M corresponds to **Game** 2.

We construct an algorithm B that, given an $\mathcal{A}$ that distinguishes **Game** 1.i from **Game** 1.$(i+1)$ with non-negligible probability, breaks the hiding property of the vector commitment scheme with non-negligible probability. B works as follows. When the challenger sends the parameters par, B stores par as common reference string in the copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. B replaces the vector commitment vc' by a vector commitment to a random vector in the last $i+1$ read operations by the honest readers. For the read operation i, B sends to the challenger the vector $\mathbf{x}$ that should be committed in vc'. The challenger sends back a commitment vc'. As can be seen, if vc' is a commitment to a random vector, the situation corresponds to **Game** 1.$(i+1)$, while otherwise the situation corresponds to **Game** 1.i. Therefore, if $\mathcal{A}$ distinguishes **Game** 1.i from **Game** 1.$(i+1)$ with non-negligible probability, B can use $\mathcal{A}$'s guess to break the hiding property of the VC scheme. This concludes the proof of Theorem C.6.

Game 3**: Game** 3 follows **Game** 2, with the exception that **Game** 3 replaces the commitment c' in ins_r by a commitment to a random value. The probability that **Game** 2 and **Game** 3 are distinguished by $\mathcal{Z}$ is bound by theorem C.7.

Theorem C.7. Let M be the number of read operations sent by honest readers. Under the hiding property of the commitment scheme, we have that $|\Pr[\textbf{Game } 3] - \Pr[\textbf{Game } 2]| \leq M \cdot \mathrm{Adv}_{\mathcal{A}}^{\mathsf{hid-com}}$.

Proof of Theorem C.7. We define a sequence of games. In **Game** 2.i, for the last i read operations sent by honest readers, the commitment c' is replaced by a commitment to a random value. Therefore, **Game** 2.0 corresponds to **Game** 2, while **Game** 2.M corresponds to **Game** 3.

We construct an algorithm B that, given an $\mathcal{A}$ that distinguishes **Game** 2.i from **Game** 2.$(i+1)$ with non-negligible probability, breaks the hiding property of the commitment scheme with non-negligible probability. B works as follows. When the challenger sends the parameters par_c, B stores par_c as common reference string in the copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. B replaces the commitment c' by a commitment to a random value in the last $i+1$ read operations by the honest readers. For the proof i, B sends to the challenger the value s that should be committed in c'. The challenger sends back a commitment c'. As can be seen, if c' is a commitment to a random value, the situation corresponds to **Game** 2.$(i+1)$, while otherwise the situation corresponds to **Game** 2.i. Therefore, if $\mathcal{A}$ distinguishes **Game** 2.i from **Game** 2.$(i+1)$ with non-negligible probability, B can use the $\mathcal{A}$'s guess to break the hiding property of the commitment scheme. This concludes the proof of Theorem C.7.

The distribution of **Game** 3 is identical to our simulation. This concludes the proof of Theorem C.5.

D Instantiation of HD

D.1 Building Blocks of Our Instantiation

Bilinear Maps. Let $\mathbb{G}$, $\tilde{\mathbb{G}}$ and $\mathbb{G}_t$ be groups of prime order p. A map $e : \mathbb{G} \times \tilde{\mathbb{G}} \to \mathbb{G}_t$ must satisfy bilinearity, i.e., $e(g^x, \tilde{g}^y) = e(g, \tilde{g})^{xy}$; non-degeneracy, i.e., for all generators $g \in \mathbb{G}$ and $\tilde{g} \in \tilde{\mathbb{G}}$, $e(g, \tilde{g})$ generates $\mathbb{G}_t$; and efficiency, i.e., there exists an efficient algorithm $\mathcal{G}(1^k)$ that outputs the pairing group setup $grp \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and an efficient algorithm to compute $e(a, b)$ for any $a \in \mathbb{G}$, $b \in \tilde{\mathbb{G}}$.

ℓ-DH Exponent (DHE) Assumption. Let $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$ and $\alpha \leftarrow \mathbb{Z}_p$. Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and a tuple $(g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$ such that

$g_i = g^{(\alpha^i)}$ and $\tilde{g}_i = \tilde{g}^{(\alpha^i)}$, for any p.p.t. adversary $\mathcal{A}$, $\Pr[g^{(\alpha^{\ell+1})} \leftarrow \mathcal{A}(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})] \leq \epsilon(k)$.

VC Scheme. We use a VC scheme that is secure under the ℓ-DHE assumption [27].

$\mathsf{VC.Setup}(1^k, \ell)$**.** Generate groups $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$, pick $\alpha \leftarrow \mathbb{Z}_p$, and compute $(g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$, where $g_i = g^{(\alpha^i)}$ and $\tilde{g}_i = \tilde{g}^{(\alpha^i)}$. Output the parameters $par = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell}, \mathcal{M} = \mathbb{Z}_p, \mathcal{R} = \mathbb{Z}_p)$.

$\mathsf{VC.Com}(par, \mathbf{x}, r)$**.** Let $|\mathbf{x}| = n \leq \ell$. Output $vc = g^r \cdot \prod_{j=1}^{n} g_{\ell+1-j}^{\mathbf{x}[j]} = g^r \cdot g_\ell^{\mathbf{x}[1]} \cdots g_{\ell+1-n}^{\mathbf{x}[n]}$.

$\mathsf{VC.Open}(par, i, \mathbf{x}, r)$**.** Let $|\mathbf{x}| = n \leq \ell$. Output $w = g_i^r \cdot \prod_{j=1, j\neq i}^{n} g_{\ell+1-j+i}^{\mathbf{x}[j]}$.

$\mathsf{VC.Vf}(par, vc, x, i, w)$**.** Output 1 if $e(vc, \tilde{g}_i) = e(w, \tilde{g}) \cdot e(g_1, \tilde{g}_\ell)^x$, else output 0.

Theorem D.1. This vector commitment scheme is hiding and binding under as defined in §B under the ℓ-DHE assumption.

This vector commitment scheme fulfills the hiding property in an information-theoretic way. We show that this vector commitment scheme fulfills the binding property under the ℓ-DHE assumption. Given an adversary $\mathcal{A}$ that breaks the binding property with non-negligible probability ν, we construct an algorithm $\mathcal{T}$ that breaks the ℓ-DHE assumption with non-negligible probability ν. First, $\mathcal{T}$ receives an instance $(e, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, p, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$ of the ℓ-DHE assumption. $\mathcal{T}$ sets $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$ and sends par to $\mathcal{A}$. $\mathcal{A}$ returns (vc, i, x, x', w, w') such that $\mathsf{VC.Vf}(par, vc, x, i, w) = 1$, $\mathsf{VC.Vf}(par, vc, x', i, w') = 1$, $i \in [1, \ell]$, $x, x' \in \mathcal{M}$, and $x \neq x'$. $\mathcal{T}$ computes $g_{\ell+1}$ as follows:

$$\begin{aligned} e(w, \tilde{g}) e(g_1, \tilde{g}_\ell)^x &= e(w', \tilde{g}) e(g_1, \tilde{g}_\ell)^{x'} \\ e(w/w', \tilde{g}) &= e(g_1, \tilde{g}_\ell)^{x'-x} \\ e((w/w')^{1/(x'-x)}, \tilde{g}) &= e(g_1, \tilde{g}_\ell) \\ e((w/w')^{1/(x'-x)}, \tilde{g}) &= e(g_{\ell+1}, \tilde{g}) \ . \end{aligned}$$

The last equation implies that $g_{\ell+1} = (w/w')^{1/(x'-x)}$. $\mathcal{T}$ returns $(w/w')^{1/(x'-x)}$ as a solution for the ℓ-DHE problem.

Commitment Scheme. We use the Pedersen commitment scheme [31]. $\mathsf{C.Setup}(1^k)$ takes a group $\mathbb{G}$ of prime order p with generator g, picks random α, computes $h \leftarrow g^\alpha$ and sets the parameters $par_c \leftarrow (\mathbb{G}, g, h)$, which include a description of the message space $\mathcal{M} \leftarrow \mathbb{Z}_p$. $\mathsf{C.Com}(par_c, x)$ picks random $o \leftarrow \mathbb{Z}_p$ and outputs a commitment $c \leftarrow g^x h^o$ to $x \in \mathcal{M}$ and an opening o. $\mathsf{C.Vf}(par_c, c, x, o)$ outputs 1 if $c = g^x h^o$. This scheme is perfectly hiding and computationally binding. In [8], it is shown that Pedersen commitments realize $\mathcal{F}_{\mathrm{NIC}}$. Therefore, we use Pedersen commitments to instantiate both the commitments computed in Π_{HD} and those computed by $\mathcal{F}_{\mathrm{NIC}}$ and received as input to Π_{HD}.

Signature Scheme. We use the structure-preserving signature (SPS) scheme in [1]. In SPSs, the public key, the messages, and the signatures are group elements in $\mathbb{G}$ and $\tilde{\mathbb{G}}$, and verification must consist purely in the checking of pairing product equations. We employ SPSs to sign group elements, while still supporting efficient ZK proofs of signature possession. In this SPS scheme, a elements in $\mathbb{G}$ and b elements in $\tilde{\mathbb{G}}$ are signed.

$\mathsf{S.KG}(grp, a, b)$**.** Let $grp \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ be the bilinear map parameters. Pick at random $u_1, \ldots, u_b, v, w_1, \ldots w_a, z \leftarrow \mathbb{Z}_p^*$ and compute $U_i = g^{u_i}$, $i \in [1..b]$, $V = \tilde{g}^v$, $W_i = \tilde{g}^{w_i}$, $i \in [1..a]$ and $Z = \tilde{g}^z$. Return the verification key $pk \leftarrow (grp, U_1, \ldots, U_b, V, W_1, \ldots, W_a, Z)$ and the signing key $sk \leftarrow (pk, u_1, \ldots, u_b, v, w_1, \ldots, w_a, z)$.

$\mathsf{S.Sign}(sk, \langle m_1, \ldots, m_{a+b}\rangle)$**.** Pick $r \leftarrow \mathbb{Z}_p^*$, set $R \leftarrow g^r$, $S \leftarrow g^{z-rv} \prod_{i=1}^{a} m_i^{-w_i}$, and $T \leftarrow (\tilde{g} \prod_{i=1}^{b} m_{a+i}^{-u_i})^{1/r}$, and output the signature $sig \leftarrow (R, S, T)$.

$\mathsf{S.Vf}(pk, sig, \langle m_1, \ldots, m_{a+b}\rangle)$**.** Output 1 if it is satisfied that $e(R, V) e(S, \tilde{g}) \prod_{i=1}^{a} e(m_i, W_i) = e(g, Z)$ and $e(R, T) \prod_{i=1}^{b} e(U_i, m_{a+i}) = e(g, \tilde{g})$.

Functionality $\mathcal{F}_{\mathrm{ZK}}^R$. To instantiate $\mathcal{F}_{\mathrm{ZK}}^R$, we use the scheme in [10]. In [10], a UC ZK protocol proving knowledge of exponents $(w_1, \ldots, w_n)$ that satisfy the formula $\phi(w_1, \ldots, w_n)$ is described as

$$\text{⋊}\, w_1, \ldots, w_n : \phi(w_1, \ldots, w_n) \qquad (4)$$

The formula $\phi(w_1, \ldots, w_n)$ consists of conjunctions and disjunctions of "atoms". An atom expresses *group relations*, such as $\prod_{j=1}^{k} g_j^{\mathcal{F}_j} = 1$, where the g_j's are elements of prime order groups and the $\mathcal{F}_j$'s are polynomials in the variables $(w_1, \ldots, w_n)$.

A proof system for (4) can be transformed into a proof system for more expressive statements about secret exponents *sexps* and secret bases *sbases*:

$$\text{⋊}\, sexps, sbases : \phi(sexps, bases \cup sbases) \qquad (5)$$

The transformation adds an additional base h to the public bases. For each $g_j \in sbases$, the transformation picks a random exponent ρ_j and computes a blinded base $g'_j = g_j h^{\rho_j}$. The transformation adds g'_j to the public bases *bases*, ρ_j to the secret exponents *sexps*, and rewrites $g_j^{\mathcal{F}_j}$ into ${g'_j}^{\mathcal{F}_j} h^{-\mathcal{F}_j \rho_j}$.

The proof system supports pairing product equations $\prod_{j=1}^{k} e(g_j, \tilde{g}_j)^{\mathcal{F}_j} = 1$ in groups of prime order with a bilinear map e, by treating the target group $\mathbb{G}_t$ as the group of the proof system. The embedding for secret bases is unchanged, except for the case in which both bases in a pairing are secret. In this case, $e(g_j, \tilde{g}_j)^{\mathcal{F}_j}$ must be transformed into $e(g'_j, \tilde{g}'_j)^{\mathcal{F}_j} e(g'_j, \tilde{h})^{-\mathcal{F}_j \tilde{\rho}_j} e(h, \tilde{g}'_j)^{-\mathcal{F}_j \rho_j} e(h, \tilde{h})^{\mathcal{F}_j \rho_j \tilde{\rho}_j}$.

D.2 UC ZK Proof for the Read Relation

To instantiate $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ with the protocol in [10], we need to instantiate R_r with our chosen VC, commitment and signature schemes. Then we need to express R_r following the notation for UC ZK proofs described above. We use ZK proof similar to the one in [24].

In R_r, we need to prove that the position i committed in c'_i equals the position opened in the VC vc through the verification equation $e(vc, \tilde{g}_i) = e(w, \tilde{g}) \cdot e(g_1, \tilde{g}_\ell)^x$. In our VC scheme, α is secret, which makes the relation between $\tilde{g}_i = \tilde{g}^{\alpha^i}$ and i not efficiently provable. To solve this problem, the updater computes SPSs that bind g^i with $\tilde{g}_i$. Given the parameters $par = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell}, \mathcal{M} = \mathbb{Z}_p, \mathcal{R} = \mathbb{Z}_p)$, and the key pair (sk, pk), the updater, for $i \in [1, \ell]$, computes $sig_i \leftarrow \mathsf{S.Sign}(sk, \langle g^{sid}, \tilde{g}_i, \tilde{g}^i \rangle)$.

Let $\tilde{h} \leftarrow \tilde{\mathbb{G}}$. Let $(g, h) \in \mathbb{G}^2$ and $(\tilde{g}, \tilde{h}) \in \mathbb{G}^2$ be two sets of parameters of the Pedersen commitment scheme for the commitments (c'_i, cr'_i) and c respectively. R_r involves proofs about secret bases and we use the transformation described above for those proofs. The base h is also used to randomize secret bases in $\mathbb{G}$, and $\tilde{h}$ is added to randomize bases in $\tilde{\mathbb{G}}$.

Let (U_1, U_2, V, W_1, Z) be the public key of the signature scheme for signing 1 element of $\mathbb{G}$ and 2 element of $\tilde{\mathbb{G}}$. Let (R_i, S_i, T_i) be a signature on $(g^{sid}, \tilde{g}_i, \tilde{g}^i)$. Let (R, S, T) be a signature on vc, c and $\tilde{g}^{sid}$. (When verifying a proof, the updater can distinguish between the two types of signatures because sid is signed in different positions.) We describe the ZK proof for R_r as follows.

$$\begin{aligned}
&\mathsf{K}\ vc, c, R, S, T, \{i, o_i, vr_i, or_i, \tilde{g}_i, w_i, R_i, S_i, T_i\}_{\forall i \in \mathbb{S}} : \\
&\quad e(R, V) e(S, \tilde{g}) e(vc, W_1) e(g, Z)^{-1} = 1 \wedge && (6)\\
&\quad e(R, T) e(U_1, c) e(U_2, \tilde{g}^{sid}) e(g, \tilde{g})^{-1} = 1 \wedge && (7)\\
&\quad \{c'_i = g^i h^{o_i} \wedge cr'_i = g^{vr_i} h^{or_i} \wedge && (8)\\
&\quad e(R_i, V) e(S_i, \tilde{g}) e(g^{sid}, W_1) e(g, Z)^{-1} = 1 \wedge && (9)\\
&\quad e(R_i, T_i) e(U_1, \tilde{g}_i) e(U_2, \tilde{g})^i e(g, \tilde{g})^{-1} = 1 \wedge && (10)\\
&\quad e(vc, \tilde{g}_i)^{-1} e(w_i, \tilde{g}) e(g_1, \tilde{g}_\ell)^{vr_i} = 1\}_{\forall i \in \mathbb{S}} && (11)
\end{aligned}$$

Equation 6 and equation 7 prove knowledge of a signature (R, S, T) on vc, c and $\tilde{g}^{sid}$. Equation 8 proves knowledge of the openings of the Pedersen commitments c'_i and cr'_i. Equation 9 and Equation 10 prove knowledge of a signature (R_i, S_i, T_i) on a message $(g^{sid}, \tilde{g}_i, \tilde{g}^i)$. Equation 11 proves that the value vr_i in cr'_i is equal to the value committed in the position i of the vector commitment vc.

To prove knowledge of the secret bases $(vc, c, R, S, T, \{\tilde{g}_i, w_i, R_i, S_i, T_i\}_{\forall i \in \mathbb{S}})$, the equations need to be modified following the transformation described above. This means that each of the bases is rerandomized and added to the instance. Therefore, a reader rerandomizes c and vc, and the fact that they are a rerandomization of the values signed in the signature (R, S, T) is proven via equation 6 and equation 7. In the case of vc, we use g instead of h as the base used for randomization.

E Variants of HD

Variants of $\mathcal{F}_{\mathrm{HD}}$. $\mathcal{F}_{\mathrm{HD}}$ can be modified to store a database DB of the form $[i, vr_{i,1}, \ldots, vr_{i,m}]$, i.e., a database where a tuple of values is stored in each entry. In the hd.update interface, $\mathcal{U}$ sends $(i, vu_{i,1}, \ldots, vu_{i,m})_{\forall i \in [1,N]}$, and each value $vu_{i,j}$ $(j \in [1, m])$ can be updated or not independently of other values in the same entry. In the hd.read interface, $\mathcal{R}_k$ sends $(i, vr_{i,1}, \ldots, vr_{i,m})_{i \in \mathbb{S}}$ along with commitments and openings to the position and values of each of the entries read, i.e., all the values in an entry are read. The position $j \in [1, m]$ of each value $vr_{i,j}$ is not hidden from $\mathcal{U}$. This variant of $\mathcal{F}_{\mathrm{HD}}$ is useful for protocols where a party needs to read a tuple of values and prove that they are stored in the same entry and that each $vr_{i,j}$ is stored at a certain position j within the entry.

$\mathcal{F}_{\mathrm{HD}}$ cannot be modified so that it authenticates $\mathcal{R}_k$ towards $\mathcal{U}$. If all $\mathcal{R}_k$ were authenticated, $\mathcal{U}$ would be able to track all the changes performed on each of the databases, and thus the contents of the databases would not be hidden from $\mathcal{U}$.

Variants of Π_{HD}. To construct the variant described above, vc commits to a vector $\mathbf{x}$ of length $N \times m$ such that $\mathbf{x}[(i-1)m + j] = vr_{i,j}$ for all $i \in [1, N]$ and $j \in [1, m]$. In the update phase, each vector component can be updated independently of others regardless of whether they belong to the same database entry. To read the database entry i, $\mathcal{R}_k$ needs to compute openings $(w_{(i-1)m+1}, \ldots, w_{im})$ to open the positions $[(i-1)m+1, im]$ of the committed vector $\mathbf{x}$. $\mathcal{R}_k$ must also prove that those positions belong to the database entry i. To this end, the relation R_r is modified to involve a witness $wit_r \leftarrow (sig, vc, c, r_2, o_2, \langle i, o_i, \{w_{(i-1)m+j}, vr_{i,j}, or_{i,j}\}_{\forall j \in [1,m]}\rangle_{i \in \mathbb{S}})$ and $ins_r \leftarrow (pk, par, par_c, vc', c', pnic, \langle c'_i, \{cr'_{i,j}\}_{\forall j \in [1,m]}\rangle_{i \in \mathbb{S}})$.

$$\begin{aligned} R_r =\{ & (wit_r, ins_r) : \\ & 1 = \mathsf{S.Vf}(pk, sig, \langle vc, c\rangle) \wedge \\ & vc' = \mathsf{VC.Rerand}(vc, r_2) \wedge c' = \mathsf{C.Rerand}(c, o_2) \wedge \\ & \{1 = \mathsf{NIC.Vf}(pnic, c'_i, i, o_i) \wedge \\ & \{1 = \mathsf{NIC.Vf}(pnic, cr'_{i,j}, vr_{i,j}, or_{i,j}) \wedge \\ & 1 = \mathsf{VC.Vf}(par, vc, vr_{i,j}, (i-1)m + j, w_{(i-1)m+j}) \\ & \}_{\forall j \in [1,m]} \}_{\forall i \in \mathbb{S}} \} \end{aligned}$$

Instantiations of Variants of Π_{HD}. In order to instantiate this variant of Π_{HD}, we compute signatures $sig_i \leftarrow \mathsf{S.Sign}(sk, \langle g^i, g^{sid}, \tilde{g}_{(i-1)m+1}, \ldots, \tilde{g}_{im}\rangle)$ to bind the entry i to the positions $[(i-1)m + 1, im]$ that need to be opened in the committed vector. The public key of the signature scheme is now $(U_1, \ldots, U_m, V, W_1, W_2, Z)$. The ZK proof for relation R is:

$$\begin{aligned} \text{Ӽ}\ & vc, c, R, S, T, \{i, o_i, \{vr_{i,j}, or_{i,j}, \\ & \tilde{g}_{(i-1)m+j}, w_{(i-1)m+j}\}_{\forall j \in [1,m]}, R_i, S_i, T_i\}_{\forall i \in \mathbb{S}} : \\ & e(R, V)e(S, \tilde{g})e(vc, W_1)e(g, Z)^{-1} = 1 \wedge \\ & e(R, T)e(U_1, c)e(U_2, \tilde{g}^{sid})e(g, \tilde{g})^{-1} = 1 \wedge \\ & \{c'_i = g^i h^{o_i} \wedge \{cr'_{i,j} = g^{vr_{i,j}} h^{or_{i,j}}\}_{\forall j \in [1,m]} \wedge \\ & e(R_i, V)e(S_i, \tilde{g})e(g, W_1)^i e(g^{sid}, W_2)e(g, Z)^{-1} = 1 \wedge \\ & e(R_i, T_i)e(U_1, \tilde{g}_{(i-1)m+1}) \cdots e(U_m, \tilde{g}_{im})e(g, \tilde{g})^{-1} = 1 \wedge \\ & \{e(vc, \tilde{g}_{(i-1)m+j})^{-1} e(w_{(i-1)m+j}, \tilde{g})e(g_1, \tilde{g}_\ell)^{vr_{i,j}} = 1 \\ & \}_{\forall j \in [1,m]}\}_{\forall i \in \mathbb{S}} \end{aligned}$$

The signature on $\langle g^i, g^{sid}, \tilde{g}_{(i-1)m+1}, \ldots, \tilde{g}_{im}\rangle$ also binds the positions of the database entry i together and reveals the position $j \in [1, m]$ of each value $vr_{i,j}$ within the entry.

F Security Analysis of Our PPLP

To prove that Π_{LP} securely realizes $\mathcal{F}_{\mathrm{LP}}$, we must show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{LP}}$. $\mathcal{S}$ thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\mathrm{LP}}$ for all corrupt parties in the ideal world.

Our simulator $\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\mathrm{HD}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{NIC}}$. When any of the copies of those functionalities aborts, $\mathcal{S}$ implicitly forwards the abortion message to $\mathcal{A}$ if the functionality sends the abortion message to a corrupt party.

First, we analyze the security of Π_{LP} when (a subset of) buyers $\mathcal{B}_k$ are corrupt. Second, we analyze the security of Π_{LP} when the vendor $\mathcal{V}$ is corrupt. We do not analyze in detail the security of Π_{LP} when $\mathcal{V}$ and (a subset of) readers $\mathcal{B}_k$ are corrupt. We note that, in Π_{LP}, honest buyers communicate with $\mathcal{V}$ but not with other buyers. Therefore, for this case the simulator and the security proof are very similar to the case where only $\mathcal{V}$ is corrupt.

Description of $\mathcal{S}$ for Corrupt $\mathcal{B}_k$. We describe $\mathcal{S}$ for the case in which (a subset of) $\mathcal{B}_k$ are corrupt. Basically, $\mathcal{S}$ runs copies of $\mathcal{F}_{\mathrm{HD}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{NIC}}$ as well as a copy of $\mathcal{V}$ in Π_{LP} to simulate the protocol towards the adversary $\mathcal{A}$. We refer to that as "running protocol Π_{LP}". In the following, we indicate when $\mathcal{S}$ needs to deviate from protocol Π_{LP} and we detail the communication between $\mathcal{S}$ and $\mathcal{F}_{\mathrm{LP}}$.

Register. When the adversary $\mathcal{A}$ uses the nic.setup interface and the hd.read interface on input $(P, \perp)$, $\mathcal{S}$ runs protocol Π_{LP}. When the copy of $\mathcal{V}$ outputs (lp.register.end, sid, P), $\mathcal{S}$ sends (lp.register.ini, sid, P) to $\mathcal{F}_{\mathrm{LP}}$. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.register.sim, sid, qid), $\mathcal{S}$ sends (lp.register.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$.

Purchase. When $\mathcal{A}$ uses the hd.read interface on input $(P, \perp)$, $\mathcal{S}$ runs protocol Π_{LP}. When the copy of $\mathcal{V}$ outputs (lp.purchase.end, sid, P), $\mathcal{S}$ sends (lp.purchase.ini, sid, P) to $\mathcal{F}_{\mathrm{LP}}$. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.purchase.sim, sid, qid), $\mathcal{S}$ sends (lp.purchase.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$.

Redeem. When the adversary $\mathcal{A}$ uses the nic.commit interface on input N and v_N, the zk.prove interface on input $wit_{rd} \leftarrow (v_N, o_v)$, $ins_{rd} \leftarrow (p, c_v, c_N, N, o_N)$ and P, and the hd.read interface on input P and $(N, v_N, c_N, o_N, c_v, o_v)$, $\mathcal{S}$ runs protocol Π_{LP}. (This includes a nym.reply message (Read DB) sent to $\mathcal{A}$.) $\mathcal{S}$ outputs failure if the values (v_N, o_v) sent by $\mathcal{A}$ to the zk.prove interface and to the hd.read interface are different but they are both valid openings of c_v. When the copy of $\mathcal{V}$ outputs (lp.redeem.end, sid, P, p), $\mathcal{S}$ sends (lp.redeem.ini, sid, P, p) to $\mathcal{F}_{\mathrm{LP}}$. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.redeem.sim, sid, qid), $\mathcal{S}$ sends (lp.redeem.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$.

Profile. When $\mathcal{A}$ uses the nic.commit interface on input i and v_i (for all $i \in \mathbb{S}$), the zk.prove interface on input $wit_{pr} \leftarrow (\langle i, o_i, v_i, or_i\rangle_{i \in \mathbb{S}})$, $ins_{pr} \leftarrow (res, \langle c_i, cr_i\rangle_{i \in \mathbb{S}})$ and P, and the hd.read interface on input P and $(i, v_i, c_i, o_i, cr_i, or_i)_{i \in \mathbb{S}}$, $\mathcal{S}$ runs protocol Π_{LP}. (This includes a nym.reply message (Read DB) sent to $\mathcal{A}$.) $\mathcal{S}$ outputs failure if any of the commitment openings $(\langle i, o_i, v_i, or_i\rangle_{i\in\mathbb{S}})$ sent by $\mathcal{A}$ to the zk.prove interface and to the hd.read interface are different but they are both valid openings of c_i or of cr_i. When the copy of $\mathcal{V}$ outputs (lp.profile.end, sid, P, res), $\mathcal{S}$ sends (lp.profile.ini, sid, P, f) to $\mathcal{F}_{\mathrm{LP}}$. (We note that f is determined by the set $\mathbb{S}$ that $\mathcal{S}$ learns from $\mathcal{A}$.) When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.profile.sim, sid, qid), $\mathcal{S}$ sends (lp.profile.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$.

Update. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.updatedb.sim, sid, qid), $\mathcal{S}$ sends (lp.updatedb.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.updatedb.end, $sid, P, (i, vu_i)_{i\in[1,N]}$), $\mathcal{S}$ runs protocol Π_{LP} on input (lp.updatedb.ini, sid, P, $(i, vu_i)_{i\in[1,N]}$) and communicates with $\mathcal{A}$ following protocol Π_{LP}.

Theorem F.1. Π_{LP} securely realizes $\mathcal{F}_{\mathrm{LP}}$ in the $\mathcal{F}_{\mathrm{HD}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{NIC}}$-hybrid model when (a subset of) $\mathcal{B}_k$ are corrupt.

Proof of Theorem F.1. We omit a formal proof of this theorem. We note that, when (a subset of) $\mathcal{B}_k$ are corrupt, the simulator $\mathcal{S}$ runs protocol Π_{LP}. $\mathcal{S}$ is able to do that because it receives from $\mathcal{A}$ and from $\mathcal{F}_{\mathrm{LP}}$ all the information needed to run Π_{LP}. The difference between $\mathcal{S}$ and protocol Π_{LP} is that $\mathcal{S}$ outputs failure when $\mathcal{A}$ sends two different valid openings for the same commitment. The probability that $\mathcal{S}$ outputs failure is negligible thanks to the binding property of commitments provided by $\mathcal{F}_{\mathrm{NIC}}$.

Description of $\mathcal{S}$ for Corrupt $\mathcal{V}$. We describe $\mathcal{S}$ for the case in which $\mathcal{V}$ is corrupt. Basically, $\mathcal{S}$ runs copies of $\mathcal{F}_{\mathrm{HD}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{NIC}}$ as well as a copies of $\mathcal{B}_k$ in Π_{LP} to simulate the protocol towards the adversary $\mathcal{A}$. We refer to that as "running protocol Π_{LP}". In the following, we indicate when $\mathcal{S}$ needs to deviate from protocol Π_{LP} and we detail the communication between $\mathcal{S}$ and $\mathcal{F}_{\mathrm{LP}}$.

Register. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.register.sim, sid, qid), $\mathcal{S}$ sends (lp.register.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.register.end, sid, P), $\mathcal{S}$ runs protocol Π_{LP} on input (lp.register.ini, sid, P). ($\mathcal{S}$ creates a new copy of $\mathcal{B}_k$.) $\mathcal{S}$ interacts with $\mathcal{A}$ following protocol Π_{LP}.

Purchase. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.purchase.sim, sid, qid), $\mathcal{S}$ sends (lp.purchase.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.purchase.end, sid, P), $\mathcal{S}$ runs protocol Π_{LP} on input (lp.purchase.ini, sid, P). ($\mathcal{S}$ chooses the first copy of $\mathcal{B}_k$.) $\mathcal{S}$ interacts with $\mathcal{A}$ following protocol Π_{LP}.

Redeem. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.redeem.sim, sid, qid), $\mathcal{S}$ sends (lp.redeem.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.redeem.end, sid, P, p), $\mathcal{S}$ runs protocol Π_{LP} on input (lp.redeem.ini, sid, P, p). ($\mathcal{S}$ chooses the first copy of $\mathcal{B}_k$. We note that, in the purchase and update phases, all the loyalty points were accumulated in this copy, so it is guaranteed that at least p loyalty points are accumulated.) $\mathcal{S}$ interacts with $\mathcal{A}$ following protocol Π_{LP}.

Profile. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.profile.sim, sid, qid), the simulator $\mathcal{S}$ sends (lp.profile.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.profile.end, sid, P, res), $\mathcal{S}$ chooses a database and a function f such that the evaluation of f on input the database is res. $\mathcal{S}$ creates a new copy of $\mathcal{B}_k$, stores that database on the copy, and runs protocol Π_{LP} on input (lp.profile.ini, sid, P, f) by using that copy of $\mathcal{B}_k$. $\mathcal{S}$ interacts with $\mathcal{A}$ following protocol Π_{LP}.

Update. When $\mathcal{A}$ uses the hd.update interface on input P and $(i, vu_i)_{i\in[1,N]}$, $\mathcal{S}$ runs protocol Π_{LP}. ($\mathcal{S}$ chooses the copy of $\mathcal{B}_k$ that was associated with P in a previous phase.) When the copy of $\mathcal{B}_k$ outputs (lp.updatedb.end, $sid, P, (i, vu_i)_{i\in[1,N]}$), $\mathcal{S}$ sends (lp.updatedb.ini, $sid, P, (i, vu_i)_{i\in[1,N]}$) to $\mathcal{F}_{\mathrm{LP}}$. When $\mathcal{F}_{\mathrm{LP}}$ sends (lp.updatedb.sim, sid, qid), $\mathcal{S}$ sends (lp.updatedb.rep, sid, qid) to $\mathcal{F}_{\mathrm{LP}}$.

Theorem F.2. Π_{LP} securely realizes $\mathcal{F}_{\mathrm{LP}}$ in the $\mathcal{F}_{\mathrm{HD}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$, $\mathcal{F}_{\mathrm{NYM}}$ and $\mathcal{F}_{\mathrm{NIC}}$-hybrid model when $\mathcal{V}$ is corrupt.

Proof of Theorem F.2. We omit a formal proof of this theorem. We note that, when $\mathcal{V}$ is corrupt, the simulator $\mathcal{S}$ runs protocol Π_{LP} with the following changes.

– In the purchase and redeem phases, $\mathcal{S}$ picks the first copy of $\mathcal{B}_k$, while in the profile phase, it creates a new copy. This change is indistinguishable because $\mathcal{F}_{\mathrm{HD}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$ and $\mathcal{F}_{\mathrm{NYM}}$ never reveal the identity $\mathcal{B}_k$ of the buyer to the adversary. They only reveal a pseudonym.
– In the redeem phase, $\mathcal{S}$ uses a copy of $\mathcal{B}_k$ that stores a database where the number of accumulated loyalty points is likely different. In the profile phase, $\mathcal{S}$ uses a copy of $\mathcal{B}_k$ that stores a newly created database. This change is indistinguishable because commitments provided by $\mathcal{F}_{\mathrm{NIC}}$ are hiding, $\mathcal{F}_{\mathrm{ZK}}^{R_{rd}}$ or $\mathcal{F}_{\mathrm{ZK}}^{R_{pr}}$ do not reveal the witness to the adversary, and $\mathcal{F}_{\mathrm{HD}}$ only reveals commitments to the positions and values read.