# UC Updatable Databases and Applications*

Aditya Damodaran[0000−0003−4030−6859] and Alfredo Rial[0000−0003−1107−4841]

SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
firstname.lastname@uni.lu

**Abstract.** We define an ideal functionality $\mathcal{F}_{\text{UD}}$ and a construction $\Pi_{\text{UD}}$ for an updatable database (UD). UD is a two-party protocol between an updater and a reader. The updater sets the database and updates it at any time throughout the protocol execution. The reader computes zero-knowledge (ZK) proofs of knowledge of database entries. These proofs prove that a value is stored at a certain position in the database, without revealing the position or the value.
(Non-)updatable databases are implicitly used as building block in priced oblivious transfer, privacy-preserving billing and other privacy-preserving protocols. Typically, in those protocols the updater signs each database entry, and the reader proves knowledge of a signature on a database entry. Updating the database requires a revocation mechanism to revoke signatures on outdated database entries.
Our construction $\Pi_{\text{UD}}$ uses a non-hiding vector commitment (NHVC) scheme. The updater maps the database to a vector and commits to the database. This commitment can be updated efficiently at any time without needing a revocation mechanism. ZK proofs for reading a database entry have communication and amortized computation cost independent of the database size. Therefore, $\Pi_{\text{UD}}$ is suitable for large databases. We implement $\Pi_{\text{UD}}$ and our timings show that it is practical.
In existing privacy-preserving protocols, a ZK proof of a database entry is intertwined with other tasks, e.g., proving further statements about the value read from the database or the position where it is stored. $\mathcal{F}_{\text{UD}}$ allows us to improve modularity in protocol design by separating those tasks. We show how to use $\mathcal{F}_{\text{UD}}$ as building block of a hybrid protocol along with other functionalities.

**Keywords:** Vector commitments, ZK proofs, universal composability

## 1 Introduction

In priced oblivious transfer (POT) [3], a provider offers $N$ messages to a user. Each message $m_i$ is associated with a price $p_i$ ($\forall i \in [1, N]$). The user purchases a message $m_i$ without disclosing $i$ or $p_i$.

In privacy-preserving billing (PPB) [24], a user receives meter readings from a meter that measures the consumption $c$ of some service. The provider defines a

* This research is supported by the Luxembourg National Research Fund (FNR) CORE project "Stateful Zero-Knowledge" (Project code: C17/11650748).

tariff policy that typically consists of several functions. For example, a different rate $r_i$ is applied depending on the time interval $i$ of consumption. The user pays a price $p = r_i c$ for her consumption at time interval $i$ and proves that $p_i$ is correct without revealing $c$, $r_i$ or $i$. Usually, multiple prices $p$ are aggregated and paid together so that the aggregate reveals little information about each ($c$, $r_i$, $i$).

In POT [40,9] (resp. PPB [38,39]) protocols, the user frequently uses a zero-knowledge (ZK) proof to prove that $p_i$ (resp. $r_i$) is correctly associated with $i$. The user discloses neither $p_i$ (resp. $r_i$) nor $i$. Nevertheless, the user needs to prove in ZK statements about $i$ and $p_i$ (resp. $r_i$), such as proving that she retrieves $m_i$ and that she has enough funds to pay $p_i$.

We can generalize the task of associating $i$ with $p_i$ (resp. $r_i$) as the task of proving that an entry is read from a database. Consider a database $\mathsf{DB}$ of $N$ entries of the form $[i, vr_i]$ ($\forall i \in [1, N]$), where $i$ is the position and $vr_i$ the value stored at that position. The provider establishes the contents of $\mathsf{DB}$, which are revealed to the user. Then the user proves knowledge of a database entry $[i, vr_i]$. The provider does not learn $[i, vr_i]$ but is guaranteed that $[i, vr_i]$ is stored in $\mathsf{DB}$.

To allow the user to prove knowledge of an entry $[i, vr_i]$ from $\mathsf{DB}$, $\mathsf{DB}$ needs to be stored into some data structure that allows for efficient ZK proofs. POT [40,9] and PPB [38,39] protocols typically use a signature scheme with efficient ZK proofs of signature possession. The provider computes signatures $s_i$ on tuples $[i, vr_i]$ ($\forall i \in [1, N]$) and sends them to the user. Then the user proves knowledge of a signature $s_i$ on $[i, vr_i]$ to prove that $i$ and $vr_i$ are stored together in $\mathsf{DB}$.

Practical POT and PPB protocols require that the provider be able to update $\mathsf{DB}$, so the data structure should allow efficient updates. However, if signatures are used, each time a database entry is updated, a signature revocation mechanism would be needed to revoke the signatures that sign old database entries.

In addition to proving that $[i, vr_i] \in \mathsf{DB}$, the user needs to prove other statements about $i$ and $vr_i$. Very frequently, in cryptographic protocol design, these two types of statements are intertwined. I.e, protocols use ZK proofs that involve both statements to prove that the witness is stored in a data structure and statements to prove something else about the witness. To improve modularity in protocol design, we propose to separate those tasks.

*Our Contribution:* $\mathcal{F}_{\text{UD}}$. We use the universal composability (UC) framework [14] and define an ideal functionality $\mathcal{F}_{\text{UD}}$ for an updatable database (UD) in §3. We define UD as a two-party task between a reader $\mathcal{R}$ and an updater $\mathcal{U}$. $\mathcal{U}$ sets a database $\mathsf{DB}$ and updates it at any time. Both $\mathcal{R}$ and $\mathcal{U}$ know the content of $\mathsf{DB}$. $\mathcal{R}$ reads in ZK an entry $[i, vr_i]$ from $\mathsf{DB}$. $\mathcal{F}_{\text{UD}}$ ensures that it is not possible to prove that $[i, vr_i]$ is stored in $\mathsf{DB}$ if that is not the case.

In the UC framework, modular protocol design can be achieved by describing hybrid protocols. In a hybrid protocol, the protocol building blocks are described by their ideal functionalities, and parties in the real world invoke those ideal functionalities. We show how to use $\mathcal{F}_{\text{UD}}$ as building block in a protocol where $\mathcal{F}_{\text{UD}}$ handles the tasks of storing a database $\mathsf{DB}$ and proving that an entry $[i, vr_i]$ is stored in $\mathsf{DB}$, while the ideal functionality $\mathcal{F}_{\text{ZK}}^{R}$ for zero-knowledge is used to prove further statements about $i$ and $vr_i$. One challenge when defining a hybrid

protocol is to ensure that two functionalities receive the same input. To this end, $\mathcal{F}_{\text{UD}}$ uses the method proposed in [10], which consists in receiving committed inputs produced by a functionality $\mathcal{F}_{\text{NIC}}$ for non-interactive commitments. We show how to use $\mathcal{F}_{\text{UD}}$ as building block in a protocol designed modularly in §6.

The advantages of our modular design are threefold. First, it simplifies the security analysis because security proofs in the hybrid model are simpler and because, by separating the handling of the database from ZK proofs about other statements, each building block becomes simpler to analyze. Second, it allows multiple instantiations by replacing each of the ideal functionalities by any protocols that realize them. Third, it allows the study of the task of creating an updatable database in isolation, which eases the comparison of different constructions for it.

*Our Contribution:* $\Pi_{\text{UD}}$. In §4, we propose a construction $\Pi_{\text{UD}}$ for $\mathcal{F}_{\text{UD}}$. $\Pi_{\text{UD}}$ is based on non-hiding vector commitments (NHVC) [32,15]. A NHVC scheme allows us to compute a commitment *com* to a vector $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[N])$. To open the value $\mathbf{x}[i]$ committed at position $i$, an opening $w_i$ is computed. The size of $w_i$ is independent of $N$.

$\Pi_{\text{UD}}$ works as follows. $\mathcal{U}$ sends a database $\mathsf{DB}$ to $\mathcal{R}$, and both $\mathcal{U}$ and $\mathcal{R}$ map $\mathsf{DB}$ to a vector $\mathbf{x}$ and compute a commitment *com* to $\mathbf{x}$. To update an entry $[i, vr_i]$ to $[i, vr'_i]$, $\mathcal{U}$ sends $[i, vr'_i]$ to $\mathcal{R}$, and both $\mathcal{U}$ and $\mathcal{R}$ update *com* to obtain a commitment *com*$'$ to a vector $\mathbf{x}'$ such that $\mathbf{x}'[i] = vr'_i$, while the other positions remain unchanged. Therefore, updates do not need any revocation mechanism. To prove in ZK that an entry $[i, vr_i]$ is in $\mathsf{DB}$, $\mathcal{R}$ computes an opening $w_i$ for position $i$ and a ZK proof of knowledge of $(w_i, i, vr_i)$ that proves that $\mathbf{x}[i] = vr_i$.

We discuss a variant of $\mathcal{F}_{\text{UD}}$ and $\Pi_{\text{UD}}$ where $\mathcal{R}$ reads several entries simultaneously. We also discuss a variant where the database is 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.

We describe an efficient instantiation of $\Pi_{\text{UD}}$ (and its variants) that uses a NHVC scheme based on the DHE assumption, similar to the mercurial VC scheme in [32]. The size of the public parameters of the scheme grows linearly with $N$. The size of *com* and $w_i$ is constant and independent of $i$ and $N$. The computation cost of *com* and $w_i$ grows linearly with $N$. However, the cost of updating *com* and $w_i$ grows only with the number of updated positions and is independent of $N$. Also, after $w_i$ is computed, it can be reused to compute multiple ZK proofs. In our efficiency analysis in §5, we show that the size of a ZK proof that $[i, vr_i] \in \mathsf{DB}$ is independent of the size $N$ of the database. Moreover, when $w_i$ is already computed (after the first proof for position $i$), the computation cost is also independent of $N$. We implement our instantiation of $\Pi_{\text{UD}}$ and report timings for updating and reading $\mathsf{DB}$, which attest that our solution is practical.

$\Pi_{\text{UD}}$ can be regarded as an efficient way of implementing an OR proof, i.e., a ZK proof for a disjunction of statements. Namely, proving that $[i, vr_i]$ is in $\mathsf{DB}$ is equivalent to computing an OR proof where the prover proves that he knows at least one of the entries. Typically, the size of an OR proof would grow with $N$, while our proof is of size independent of $N$. In fact, $\Pi_{\text{UD}}$ is suitable for databases of large sizes. We compare our construction with related work in §7.

## 2 Modular Design and $\mathcal{F}_{\text{NIC}}$

We summarize the UC framework in §A. We use the method in [10] to allow $\mathcal{F}_{\text{UD}}$ to be used as building block in modularly-designed protocol. This method allows us to ensure, when needed, that $\mathcal{F}_{\text{UD}}$ and other functionalities receive the same input. In [10], a functionality $\mathcal{F}_{\text{NIC}}$ for non-interactive commitments is proposed. $\mathcal{F}_{\text{NIC}}$ consists of four interfaces:

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

To ensure that a party $\mathcal{P}_i$ sends the same input $m$ to several functionalities, $\mathcal{P}_i$ first uses com.commit to get a commitment $com$ to $m$ with opening $open$. Then $\mathcal{P}_i$ sends $(com, m, open)$ to each functionality, and each functionality runs COM.Verify to verify $com$. Finally, other parties receive $com$ from each functionalities and use com.validate to validate $com$. Then, if $com$ received from all the functionalities is the same, the binding property provided by $\mathcal{F}_{\text{NIC}}$ ensures that all the functionalities received the same input $m$. $\mathcal{F}_{\text{UD}}$ receives committed inputs as described in [10].

## 3 Functionality $\mathcal{F}_{\text{UD}}$

$\mathcal{F}_{\text{UD}}$ interacts with a reader $\mathcal{R}$ and an updater $\mathcal{U}$. $\mathcal{F}_{\text{UD}}$ maintains a database DB that consists of $N$ entries $[i, vr_i]$. $\mathcal{F}_{\text{UD}}$ has two interfaces ud.update and ud.read:

1. $\mathcal{U}$ sends the ud.update.ini message on input $(i, vu_i)_{\forall i \in [1,N]}$. For all $i \in [1, N]$, $\mathcal{F}_{\text{UD}}$ updates DB to contain value $vu_i$ at position $i$. If $vu_i = \perp$, no update at position $i$ takes place. $\mathcal{F}_{\text{UD}}$ sends $(i, vu_i)_{\forall i \in [1,N]}$ to $\mathcal{R}$.
2. $\mathcal{R}$ sends ud.read.ini on input $(i, vr_i, com_i, open_i, comr_i, openr_i)$, where $[i, vr_i]$ is a DB entry and $(com_i, open_i)$ and $(comr_i, openr_i)$ are commitments and openings to $i$ and $vr_i$. $\mathcal{F}_{\text{UD}}$ verifies the commitments and checks that there is an entry $[i, vr_i]$ in DB. $\mathcal{F}_{\text{UD}}$ sends $(com_i, comr_i)$ to $\mathcal{U}$.

$\mathcal{F}_{\text{UD}}$ stores a counter $cr$ for $\mathcal{R}$ and a counter $cu$ for $\mathcal{U}$. These counters are used to check that $\mathcal{R}$ and $\mathcal{U}$ have the same version of DB. When $\mathcal{U}$ initiates the ud.update interface, $cu$ is incremented. When $\mathcal{F}_{\text{UD}}$ sends the update to $\mathcal{R}$, $\mathcal{F}_{\text{UD}}$ checks that $cu = cr + 1$ and then increments $cr$. In the ud.read interface, $\mathcal{F}_{\text{UD}}$ checks that $cu = cr$, which ensures that they have the same DB.

When invoked by $\mathcal{U}$ or $\mathcal{R}$, $\mathcal{F}_{\text{UD}}$ first checks the correctness of the input and aborts if it does not belong to the correct domain. $\mathcal{F}_{\text{UD}}$ also aborts if an interface

is invoked at an incorrect moment in the protocol. For example, $\mathcal{R}$ cannot invoke ud.read if ud.update was never invoked.

The session identifier *sid* has the structure $(\mathcal{R},\mathcal{U}, sid')$. Including the identities in *sid* ensures that any reader can initiate an instance of $\mathcal{F}_{\text{UD}}$ with any updater. $\mathcal{F}_{\text{UD}}$ implicitly checks that *sid* in a message equals the one received in the first invocation. Before $\mathcal{F}_{\text{UD}}$ queries the simulator $\mathcal{S}$, $\mathcal{F}_{\text{UD}}$ saves its state, which is recovered when receiving a response from $\mathcal{S}$. To match a query to a response, $\mathcal{F}_{\text{UD}}$ creates a query identifier *qid*.

*Description of* $\mathcal{F}_{\text{UD}}$. $\mathcal{F}_{\text{UD}}$ is parameterised by a universe of values $\mathbb{U}_v$ and by a database size $N$.

1. On input $(\mathsf{ud.update.ini}, sid, (i, vu_i)_{\forall i\in[1,N]})$ from $\mathcal{U}$:
   - Abort if $sid \notin (\mathcal{R},\mathcal{U}, sid')$.
   - For all $i \in [1, N]$, abort if $vu_i \notin \mathbb{U}_v$.
   - If $(sid, \mathsf{DB}, cu)$ is not stored:
     - For all $i \in [1, N]$, abort if $vu_i = \perp$.
     - Set $\mathsf{DB} \leftarrow (i, vu_i)_{\forall i\in[1,N]}$ and $cu \leftarrow 0$ and store $(sid, \mathsf{DB}, cu)$.
   - Else:
     - For all $i \in [1, N]$, if $vu_i \neq \perp$, update DB with $[i, vu_i]$.
     - Increment $cu$ and update DB and $cu$ in $(sid, \mathsf{DB}, cu)$.
   - Create a fresh *qid* and store $(qid, (i, vu_i)_{\forall i\in[1,N]}, cu)$.
   - Send $(\mathsf{ud.update.sim}, sid, qid, (i, vu_i)_{\forall i\in[1,N]})$ to $\mathcal{S}$.

S. On input $(\mathsf{ud.update.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', (i, vu_i)_{\forall i\in[1,N]}, cu')$ such that $qid = qid'$ is not stored.
   - If $(sid, \mathsf{DB}, cr)$ is not stored, set $\mathsf{DB} \leftarrow (i, vu_i)_{\forall i\in[1,N]}$ and $cr \leftarrow 0$ and store $(sid, \mathsf{DB}, cr)$.
   - Else:
     - Abort if $cu' \neq cr + 1$.
     - For all $i \in [1, N]$, if $vu_i \neq \perp$, update DB with $[i, vu_i]$.
     - Increment $cr$ and update $cr$ and DB in $(sid, \mathsf{DB}, cr)$.
   - Delete the record $(qid, (i, vu_i)_{\forall i\in[1,N]}, cu')$.
   - Send $(\mathsf{ud.update.end}, sid, (i, vu_i)_{\forall i\in[1,N]})$ to $\mathcal{R}$.

2. On input $(\mathsf{ud.read.ini}, sid, i, vr_i, com_i, open_i, comr_i, openr_i)$ from $\mathcal{R}$:
   - Abort if $(sid, \mathsf{DB}, cr)$ is not stored.
   - Abort if $i \notin [1, N]$, or if $vr_i \notin \mathbb{U}_v$, or if $[i, vr_i] \notin \mathsf{DB}$.
   - Parse the commitment $com_i$ as $(com_i', parcom, \mathsf{COM.Verify})$.
   - Parse the commitment $comr_i$ as $(comr_i', parcom, \mathsf{COM.Verify})$.
   - Abort if COM.Verify is not a ppt algorithm.
   - Abort if $1 \neq \mathsf{COM.Verify}(parcom, com_i', i, open_i)$.
   - Abort if $1 \neq \mathsf{COM.Verify}(parcom, comr_i', vr_i, openr_i)$.
   - Create a fresh *qid* and store $(qid, com_i, comr_i, cr)$.
   - Send $(\mathsf{ud.read.sim}, sid, qid, com_i, comr_i)$ to $\mathcal{S}$.

S. On input $(\mathsf{ud.read.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', com_i, comr_i, cr')$ such that $qid = qid'$ is not stored, or if $cr' \neq cu$, where $cu$ is in $(sid, \mathsf{DB}, cu)$.
   - Delete the record $(qid, com_i, comr_i, cr')$.
   - Send $(\mathsf{ud.read.end}, sid, com_i, comr_i)$ to $\mathcal{U}$.

*Variants of* $\mathcal{F}_{\text{UD}}$. It is straightforward to modify the ud.read interface of $\mathcal{F}_{\text{UD}}$ to allow $\mathcal{R}$ to read a tuple $(i, vr_i, com_i, open_i, comr_i, openr_i)_{\forall i \in \mathbb{S}}$ ($\mathbb{S} \subseteq [1, N]$) of database entries simultaneously. This variant of $\mathcal{F}_{\text{UD}}$ allows us to reduce communication rounds when a party in a protocol that uses $\mathcal{F}_{\text{UD}}$ needs to read more than one value simultaneously, e.g. a buyer that purchases several items at once and reads the prices of those items from the database.

$\mathcal{F}_{\text{UD}}$ can also be modified to store a database 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 ud.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 ud.read interface, $\mathcal{R}$ sends $(i, vr_{i,1}, \ldots, vr_{i,m})$ along with commitments and openings to the position and values, 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 $\mathcal{F}_{\text{UD}}$ 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, e.g. a user that consumes some utility and reads a pricing function that is represented by a tuple of values.

$\mathcal{F}_{\text{UD}}$ can also be modified to interact with two parties such that both of them can read and update the database, or such that a party reads and updates and the other party receives read and update operations. $\Pi_{\text{UD}}$ can be easily adapted to realize the variants of $\mathcal{F}_{\text{UD}}$ discussed here.

## 4 Construction $\Pi_{\text{UD}}$

### 4.1 Building Blocks

*Non-Hiding Vector Commitments.* A non-hiding vector commitment (NHVC) scheme allows one to succinctly commit to a vector $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[n]) \in \mathcal{M}^n$ such that it is possible to compute an opening $w$ to $\mathbf{x}[i]$, with the size of $w$ independent of $i$ and $n$. The scheme consists of the following algorithms.

VC.Setup($1^k, \ell$). On input the security parameter $1^k$ and an upper bound $\ell$ on the size of the vector, generate the parameters of the vector commitment scheme *par*, which include a description of the message space $\mathcal{M}$.

VC.Commit($par, \mathbf{x}$). On input a vector $\mathbf{x} \in \mathcal{M}^n$ ($n \leq \ell$), output a commitment *com* to $\mathbf{x}$.

VC.Prove($par, i, \mathbf{x}$). Compute an opening $w$ for $\mathbf{x}[i]$.

VC.Verify($par, com, x, i, w$). Output 1 if $w$ is a valid opening for $x$ being at position $i$ and 0 otherwise.

VC.ComUpd($par, com, j, x, x'$). On input a commitment *com* with value $x$ at position $j$, output a commitment $com'$ with value $x'$ at position $j$. The other positions remain unchanged.

VC.WitUpd($par, w, i, j, x, x'$). On input an opening $w$ for position $i$ valid for a commitment *com* with value $x$ at position $j$, output an opening $w'$ for position $i$ valid for a commitment $com'$ with value $x'$ at position $j$.

A non-hiding VC scheme must be correct and binding [15].

*Ideal Functionality* $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$. Our protocol uses the functionality $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ for common reference string generation in [14]. $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ interacts with any parties $\mathcal{P}$ that obtain the common reference string, and consists of one interface crs.get. A party $\mathcal{P}$ uses the crs.get interface to request and receive the common reference string *crs* from $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$. In the first invocation, $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$ generates *crs* by running algorithm CRS.Setup. The simulator $\mathcal{S}$ also receives *crs*.

*Ideal Functionality* $\mathcal{F}_{\text{AUT}}$. Our protocol uses the functionality $\mathcal{F}_{\text{AUT}}$ for an authenticated channel in [14]. $\mathcal{F}_{\text{AUT}}$ interacts with a sender $\mathcal{T}$ and a receiver $\mathcal{R}$, and consists of one interface aut.send. $\mathcal{T}$ uses the aut.send interface to send a message $m$ to $\mathcal{F}_{\text{AUT}}$. $\mathcal{F}_{\text{AUT}}$ leaks $m$ to the simulator $\mathcal{S}$ and, after receiving a response from $\mathcal{S}$, $\mathcal{F}_{\text{AUT}}$ sends $m$ to $\mathcal{R}$. $\mathcal{S}$ cannot modify $m$. The session identifier *sid* contains the identities of $\mathcal{T}$ and $\mathcal{R}$.

*Ideal Functionality* $\mathcal{F}_{\text{ZK}}^{R}$. Let $R$ be a polynomial time computable binary relation. For tuples $(wit, ins) \in R$ we call *wit* the witness and *ins* the instance. Our protocol uses the ideal functionality $\mathcal{F}_{\text{ZK}}^{R}$ for zero-knowledge in [14]. $\mathcal{F}_{\text{ZK}}^{R}$ is parameterized by a description of a relation $R$, runs with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, and consists of one interface zk.prove. $\mathcal{P}$ uses zk.prove to send a witness *wit* and an instance *ins* to $\mathcal{F}_{\text{ZK}}^{R}$. $\mathcal{F}_{\text{ZK}}^{R}$ checks whether $(wit, ins) \in R$, and, in that case, sends the instance *ins* to $\mathcal{V}$. The simulator $\mathcal{S}$ learns *ins* but not *wit*.

We give the security definitions for non-hiding VC schemes and depict $\mathcal{F}_{\text{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\text{AUT}}$ and $\mathcal{F}_{\text{ZK}}^{R}$ in §B.

### 4.2 Description of $\Pi_{\text{UD}}$

In $\Pi_{\text{UD}}$, an NHVC *com* is used to commit to the database DB. To this end, *com* commits to a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vr_i$ for all $i \in [1, N]$. $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ is parameterized by VC.Setup and generates the parameters *par*.

In the ud.update interface, $\mathcal{U}$ uses $\mathcal{F}_{\text{AUT}}$ to send to $\mathcal{R}$ the update $(i, vu_i)_{\forall i \in [1,N]}$. In the first execution of this interface, $\mathcal{U}$ and $\mathcal{R}$ run VC.Commit to commit to $(i, vu_i)_{\forall i \in [1,N]}$. In the following executions, $\mathcal{U}$ and $\mathcal{R}$ update *com* by using VC.ComUpd. If $\mathcal{R}$ already stores openings $w_i$, $\mathcal{R}$ runs VC.WitUpd to update them.

In the ud.read interface, $\mathcal{R}$ uses $\mathcal{F}_{\text{ZK}}^{R}$ to prove that $com_i$ and $comr_i$ commit to a position $i$ and a value $vr_i$ such that $\mathbf{x}[i] = vr_i$, where $\mathbf{x}$ is the vector committed in *com*. The witness of $R$ includes an opening $w_i$. $\mathcal{R}$ runs VC.Prove to compute it if it is not stored.

*Description of* $\Pi_{\text{UD}}$. $N$ denotes the database size. The universe of values $\mathbb{U}_v$ is given by the message space of the NHVC scheme.

1. On input $(\mathsf{ud.update.ini}, sid, (i, vu_i)_{\forall i \in [1,N]})$:
   - If $(sid, par, com, \mathbf{x}, cu)$ is not stored:
     - $\mathcal{U}$ uses crs.get to obtain the parameters *par* from $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$. To compute *par*, $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ runs $\mathsf{VC.Setup}(1^k, N)$.

- $\mathcal{U}$ initializes a counter $cu \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vu_i$ for all $i \in [1, N]$. $\mathcal{U}$ runs $com \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$ and stores $(sid, par, com, \mathbf{x}, cu)$.

– Else:
  - $\mathcal{U}$ sets $cu' \leftarrow cu + 1$, $\mathbf{x}' \leftarrow \mathbf{x}$ and $com' \leftarrow com$. For all $i \in [1, N]$ such that $vu_i \neq \perp$, $\mathcal{U}$ computes $com' \leftarrow \mathsf{VC.ComUpd}(par, com', i, \mathbf{x}'[i], vu_i)$ and $\mathbf{x}'[i] \leftarrow vu_i$.
  - $\mathcal{U}$ replaces the stored tuple $(sid, par, com, \mathbf{x}, cu)$ by $(sid, par, com', \mathbf{x}', cu')$.

– $\mathcal{U}$ uses $\mathsf{aut.send}$ to send the message $\langle (i, vu_i)_{\forall i \in [1,N]}, cu' \rangle$ to $\mathcal{R}$.
– If $(sid, par, com, \mathbf{x}, cr)$ is stored and $cu' \neq cr + 1$, $\mathcal{R}$ aborts.
– For $j = 1$ to $N$, if $(sid, j, w_j)$ is stored, $\mathcal{R}$ sets $w'_j \leftarrow w_j$ and, for all $i \in [1, N]$ such that $vu_i \neq \perp$, $w'_j \leftarrow \mathsf{VC.WitUpd}(par, w'_j, j, i, \mathbf{x}[i], vu_i)$. $\mathcal{R}$ replaces $(sid, j, w_j)$ by $(sid, j, w'_j)$.
– $\mathcal{R}$ performs the same operations as $\mathcal{U}$ to set or update a tuple $(sid, par, com, \mathbf{x}, cr)$.
– $\mathcal{R}$ outputs $(\mathsf{ud.update.end}, sid, (i, vu_i)_{\forall i \in [1,N]})$.

2. On input $(\mathsf{ud.read.ini}, sid, i, vr_i, com_i, open_i, comr_i, openr_i)$:
   – $\mathcal{R}$ parses $com_i$ as $(com'_i, parcom, \mathsf{COM.Verify})$.
   – $\mathcal{R}$ parses $comr_i$ as $(comr'_i, parcom, \mathsf{COM.Verify})$.
   – $\mathcal{R}$ aborts if $\mathsf{COM.Verify}$ is not a ppt algorithm.
   – $\mathcal{R}$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, com'_i, i, open_i)$.
   – $\mathcal{R}$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, comr'_i, vr_i, openr_i)$.
   – $\mathcal{R}$ takes the stored tuple $(sid, par, com, \mathbf{x}, cr)$ and aborts if $\mathbf{x}[i] \neq vr_i$.
   – If $(sid, i, w_i)$ is not stored, $\mathcal{R}$ runs $w_i \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x})$ and stores $(sid, i, w_i)$.
   – $\mathcal{R}$ sets the witness $wit \leftarrow (w_i, i, open_i, vr_i, openr_i)$ and the instance $ins \leftarrow (par, com, parcom, com'_i, comr'_i, cr)$. $\mathcal{R}$ uses $\mathsf{zk.prove}$ to send $wit$ and $ins$ to $\mathcal{F}_{\mathrm{ZK}}^{R}$. The relation $R$ is

$$\begin{aligned} R = \{ & (wit, ins) : \\ & 1 = \mathsf{COM.Verify}(parcom, com'_i, i, open_i) \wedge \\ & 1 = \mathsf{COM.Verify}(parcom, comr'_i, vr_i, openr_i) \wedge \\ & 1 = \mathsf{VC.Verify}(par, com, vr_i, i, w_i) \} \end{aligned}$$

   – $\mathcal{U}$ receives $ins = (par', com', parcom, com'_i, comr'_i, cr)$ from $\mathcal{F}_{\mathrm{ZK}}^{R}$.
   – $\mathcal{U}$ takes the stored tuple $(sid, par, com, \mathbf{x}, cu)$ and aborts if $cr \neq cu$, or if $par' \neq par$, or if $com' \neq com$.
   – $\mathcal{U}$ sets $com_i \leftarrow (com'_i, parcom, \mathsf{COM.Verify})$ and $comr_i \leftarrow (comr'_i, parcom, \mathsf{COM.Verify})$. ($\mathsf{COM.Verify}$ is part of the description of $R$.)
   – $\mathcal{U}$ outputs $(\mathsf{ud.read.end}, sid, com_i, comr_i)$.

**Theorem 1.** $\Pi_{\mathrm{UD}}$ *securely realizes* $\mathcal{F}_{\mathrm{UD}}$ *in the* $(\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}, \mathcal{F}_{\mathrm{AUT}}, \mathcal{F}_{\mathrm{ZK}}^{R})$*-hybrid model if the NHVC scheme is binding.*

We analyze in detail the security of $\Pi_{\mathrm{UD}}$ in §C.

*Variants of* $\Pi_{\mathrm{UD}}$. In §3, we describe a variant of $\mathcal{F}_{\mathrm{UD}}$ where $\mathcal{R}$ reads several database entries simultaneously, and another variant where the database entries are of the form $[i, vr_{i,1}, \ldots, vr_{i,m}]$. To construct the former, in the read phase, $\mathcal{R}$ simply needs to compute openings $w_i$ for each entry read. Relation $R$ replicates the equations described above for each entry read.

For the latter, *com* 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 $i$, $\mathcal{R}$ 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 $\mathbf{x}$. $\mathcal{R}$ must also prove that those positions belong to the database entry $i$. To this end, the relation $R$ is modified to involve a witness $wit \leftarrow (i, open_i, \{w_{(i-1)m+j}, vr_{i,j}, openr_{i,j}\}_{\forall j \in [1,m]})$ and an instance $ins \leftarrow (par, com, parcom, com'_i, \{comr'_{i,j}\}_{\forall j \in [1,m]}, cr)$

$$
\begin{aligned}
R = \{&(wit, ins) : \\
&1 = \mathsf{COM.Verify}(parcom, com'_i, i, open_i) \wedge \\
&\{1 = \mathsf{COM.Verify}(parcom, comr'_{i,j}, vr_{i,j}, openr_{i,j}) \wedge \\
&1 = \mathsf{VC.Verify}(par, com, vr_{i,j}, (i-1)m+j, w_{(i-1)m+j})\}_{\forall j \in [1,m]}\}
\end{aligned}
$$

## 5 Instantiation and Efficiency Analysis

*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 $a \in \mathbb{G}$, $b \in \tilde{\mathbb{G}}$.

*$\ell$-Diffie-Hellman 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 $\mathcal{A}$, $\Pr[g^{(\alpha^{\ell+1})} \leftarrow \mathcal{A}(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})] \leq \epsilon(k)$.

*NHVC scheme.* We use a NHVC scheme secure under the $\ell$-DHE assumption [32].

$\mathsf{VC.Setup}(1^k, \ell)$**.** 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 $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell}, \mathcal{M} = \mathbb{Z}_p)$.

$\mathsf{VC.Commit}(par, \mathbf{x})$**.** Let $|\mathbf{x}| = n \leq \ell$. Output $com = \prod_{j=1}^{n} g_{\ell+1-j}^{\mathbf{x}[j]}$ .

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

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

$\mathsf{VC.ComUpd}(par, com, j, x, x')$**.** Output $com' = com \cdot g_{\ell+1-j}^{x'-x}$ .

$\mathsf{VC.WitUpd}(par, w, i, j, x, x')$**.** If $i = j$, output $w$, else $w' = w \cdot g_{\ell+1-j+i}^{x'-x}$ .

This NHVC scheme is correct and binding under the $\ell$-DHE assumption. This theorem is proven in §D.

*Commitment scheme for $\mathcal{F}_{\text{NIC}}$.* A commitment scheme consists of algorithms $\mathsf{CSetup}$, $\mathsf{Com}$ and $\mathsf{VfCom}$. $\mathsf{CSetup}(1^k)$ generates the parameters $par_c$, which include a description of the message space $\mathcal{M}$. $\mathsf{Com}(par_c, x)$ outputs a commitment $com$ to $x \in \mathcal{M}$ and an opening $open$. $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 if $com$ is a commitment to $x$ with opening $open$ or 0 otherwise.

We use the Pedersen commitment scheme [37]. $\mathsf{CSetup}(1^k)$ takes a group $\mathbb{G}$ of prime order $p$ with generator $g$, picks random $\alpha$, computes $h \leftarrow g^\alpha$ and sets the parameters $par_c \leftarrow (\mathbb{G}, g, h)$, which include a description of the message space $\mathcal{M} \leftarrow \mathbb{Z}_p$. $\mathsf{Com}(par_c, x)$ picks random $open \leftarrow \mathbb{Z}_p$ and outputs a commitment $com \leftarrow g^x h^{open}$ to $x \in \mathcal{M}$ and an opening $open$. $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 if $com = g^x h^{open}$. In [10], it is shown that any trapdoor commitment scheme, such as Pedersen commitments, realizes $\mathcal{F}_{\text{NIC}}$.

*ZK proof for $\mathcal{F}_{\text{ZK}}^R$.* To instantiate $\mathcal{F}_{\text{ZK}}^R$, we use the scheme in [12]. In [12], 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

$$\text{ℵ}\, w_1, \ldots, w_n : \phi(w_1, \ldots, w_n) \tag{1}$$

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

$$\text{ℵ}\, sexps, sbases : \phi(sexps, bases \cup sbases) \tag{2}$$

The transformation adds an additional base $h$ to the public bases. For each $g_j \in sbases$, the transformation picks a random exponent $\rho_j$ and computes a blinded base $g'_j = g_j h^{\rho_j}$. The transformation adds $g'_j$ to the public bases *bases*, $\rho_j$ 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 $e(g'_j, \tilde{g}'_j)^{\mathcal{F}_j} e(g'_j, \tilde{h})^{-\mathcal{F}_j \tilde{\rho}_j} e(h, \tilde{g}'_j)^{-\mathcal{F}_j \rho_j} e(h, \tilde{h})^{\mathcal{F}_j \rho_j \tilde{\rho}_j}$.

*Signature schemes.* We use a signature scheme for the ZK proof for relation $R$ in §5.1. A signature scheme consists of the algorithms $\mathsf{KeyGen}$, $\mathsf{Sign}$ and $\mathsf{VfSig}$. $\mathsf{KeyGen}(1^k)$ outputs a secret key $sk$ and a public key $pk$, which include a description of the message space $\mathcal{M}$. $\mathsf{Sign}(sk, m)$ outputs a signature $s$ on the message $m \in \mathcal{M}$. $\mathsf{VfSig}(pk, s, m)$ outputs 1 if $s$ 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, $\mathsf{KeyGen}(1^k, n)$ receives the maximum number $n$ of messages as input. A signature scheme must be existentially unforgeable [22].

We use the structure-preserving signature (SPS) scheme in [2]. 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.

$\mathsf{KeyGen}(grp, a, b)$**.** 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 $sk \leftarrow (pk, u_1, \ldots, u_b, v, w_1, \ldots, w_a, z)$.

$\mathsf{Sign}(sk, \langle m_1, \ldots, m_{a+b}\rangle)$**.** 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 $s \leftarrow (R, S, T)$.

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

### 5.1 UC ZK Proof for Relation $R$

To instantiate $\mathcal{F}_{\mathrm{ZK}}^{R}$ with the protocol in [12], we need to instantiate $R$ with our chosen NHVC and commitment schemes. Then we need to express $R$ following the notation for UC ZK proofs described above.

In $R$, we need to prove that the position $i$ committed in $com_i'$ equals the position opened in the NHVC $com$ thorough the verification equation $e(com, \tilde{g}_i) = e(w, \tilde{g}) \cdot e(g_1, \tilde{g}_\ell)^x$. In our NHVC scheme, $\alpha$ is secret, which makes the relation between $\tilde{g}_i = \tilde{g}^{\alpha^i}$ and $i$ not efficiently provable. To solve this problem, the public parameters are extended with 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 $(sk, pk)$, for $i \in [1, \ell]$, $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ computes $s_i \leftarrow \mathsf{Sign}(sk, \langle g^{sid}, \tilde{g}_i, \tilde{g}^i\rangle)$, where $sid$ is the session identifier. (We note that, in many practical settings, $\mathcal{U}$ can compute the parameters and signatures.) We remark that these signatures do not need to be updated when the database is updated.

Let $(U_1, U_2, V, W_1, Z)$ be the public key of the signature scheme. Let $(R, S, T)$ be a signature on $(g^{sid}, \tilde{g}_i, \tilde{g}^i)$. Let $(g, h)$ be the parameters of the Pedersen commitment scheme. $R$ 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 another base $\tilde{h} \leftarrow \tilde{\mathbb{G}}$ is added to randomize bases in $\tilde{\mathbb{G}}$. Following the notation in [12], we describe the proof as follows.

$$\nexists\, i, open_i, v, openr_i, \tilde{g}_i, w, R, S, T :$$

$$com_i = g^i h^{open_i} \ \wedge \ comr_i = g^v h^{openr_i} \ \wedge \tag{3}$$

$$e(R, V)e(S, \tilde{g})e(g^{sid}, W_1)e(g, Z)^{-1} = 1 \ \wedge \tag{4}$$

$$e(R, T)e(U_1, \tilde{g}_i)e(U_2, \tilde{g})^i e(g, \tilde{g})^{-1} = 1 \ \wedge \tag{5}$$

$$e(com, \tilde{g}_i)^{-1} e(w, \tilde{g})e(g_1, \tilde{g}_\ell)^v = 1 \tag{6}$$

Equation 3 proves knowledge of the openings of the Pedersen commitments $com_i$ and $comr_i$. Equation 4 and Equation 5 prove knowledge of a signature $(R, S, T)$ on a message $\langle g^{sid}, \tilde{g}_i, \tilde{g}^i \rangle$. Equation 6 proves that the value $v$ in $comr_i$ is equal to the value committed in the position $i$ of the vector commitment $com$.

*Instantiations of variants of* $\Pi_{\text{UD}}$. To instantiate the variant of $\Pi_{\text{UD}}$ where several database entries are read simultaneously, we replicate the ZK proof described above for each entry read. To instantiate the variant with database entries $[i, vr_{i,1}, \ldots, vr_{i,m}]$, we compute signatures $s_i \leftarrow \mathsf{Sign}(sk, \langle g^i, g^{sid}, \tilde{g}_{(i-1)m+1}, \ldots, \tilde{g}_{im} \rangle)$ 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:

$$
\begin{aligned}
&\aleph\, i, open_i, \{vr_{i,j}, openr_{i,j}, \tilde{g}_{(i-1)m+j}, w_{(i-1)m+j}\}_{\forall j \in [1,m]}, R, S, T : \\
&\quad com_i = g^i h^{open_i} \ \wedge\ \{comr_{i,j} = g^{vr_{i,j}} h^{openr_{i,j}}\}_{\forall j \in [1,m]} \ \wedge \\
&\quad e(R,V)e(S,\tilde{g})e(g,W_1)^i e(g^{sid},W_2)e(g,Z)^{-1} = 1 \ \wedge \\
&\quad e(R,T)e(U_1,\tilde{g}_{(i-1)m+1}) \cdots e(U_m,\tilde{g}_{im})e(g,\tilde{g})^{-1} = 1 \ \wedge \\
&\quad \{e(com,\tilde{g}_{(i-1)m+j})^{-1} e(w_{(i-1)m+j},\tilde{g})e(g_1,\tilde{g}_\ell)^{vr_{i,j}} = 1\}_{\forall j \in [1,m]}
\end{aligned}
$$

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.

### 5.2 Efficiency Analysis

We analyze the storage, communication, and computation costs of our instantiation of $\Pi_{\text{UD}}$.

**Storage Cost.** $\mathcal{R}$ and $\mathcal{U}$ store the common reference string, whose size grows linearly with $N$. Throughout the protocol execution, $\mathcal{R}$ and $\mathcal{U}$ also store the last update of $com$ and the committed vector. $\mathcal{R}$ stores the openings $w_i$. In conclusion, the storage cost is linear in $N$.

**Communication Cost.** In the $\mathsf{ud.update}$ interface, $\mathcal{U}$ sends $(i, vu_i)_{\forall i \in [1,N]}$ to $\mathcal{R}$. The communication cost is linear in the number of entries updated, except for the first update in which all entries must be initialized. In the $\mathsf{ud.read}$ interface, $\mathcal{R}$ sends an instance and a ZK proof to $\mathcal{U}$. The size of the witness and of the instance is constant and independent of $N$. Therefore, the communication cost of the proof is constant. In conclusion, after the first update phase, the communication cost does not depend on $N$.

**Computation Cost.** In the $\mathsf{ud.update}$ interface, $\mathcal{U}$ and $\mathcal{R}$ update $com$ with cost linear in the number of updates (except for the first update where all the positions are initialized). $\mathcal{R}$ also updates the stored openings $w_i$ with cost linear in the number of updates. In the $\mathsf{ud.read}$ interface, if $w_i$ is not stored, $\mathcal{R}$ computes it with cost that grows linearly with $N$. However, if $w_i$ is stored, the computation cost of the proof is constant and independent of $N$.

We note that it is possible to defer opening updates to the ud.read interface, so as to only update openings that are actually needed to compute ZK proofs. Thanks to that, the computation cost in the ud.update interface is constant. In the ud.read interface, if $w_i$ is stored but needs to be updated, the computation cost grows linearly with the number of updates but it is independent of $N$. The only overhead introduced by deferring opening updates is the need to store the tuples $(i, vu_i)_{\forall i \in [1,N]}$ sent by $\mathcal{U}$.

In summary, after initializing *com* and the openings $w_i$, the communication and computation costs are independent of $N$, which makes our instantiation of $\Pi_{\text{UD}}$ practical for large databases.

### 5.3 Implementation and Efficiency Measurements

We have implemented our instantiation of $\Pi_{\text{UD}}$ in the Python programming language, using the Charm cryptographic framework [4], on a computer equipped with an Intel Core i5-7300U CPU clocked at 2.60 GHz, and 8 gigabytes of RAM. The BN256 curve was used for the pairing group setup.

To compute the UC ZK proofs for $R$, we use the compiler in [12]. The public parameters of the proof system contain a public key of the Paillier encryption scheme, the parameters for a multi-integer commitment scheme and the specification of a DSA group. (We refer to [12] for a description of how those primitives are used in the compiler.) The cost of a proof depends on the number of elements in the witness and of the number of equations composed by Boolean ANDs. The computation cost for the prover of a $\Sigma$-protocol for $R$ involves one evaluation of each of the equations and one multiplication per value in the witness. The compiler in [12] extends a $\Sigma$-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 $\Sigma$-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 $R$, the computation and communication costs of our proof do not depend on $N$.

Table 1 lists the execution times of the update and read interfaces of the protocol, in seconds. The execution times of the interfaces of the protocol have been evaluated against the size $N$ of the database, and against the security parameter of the Paillier encryption algorithm.

In the first update, the public parameters of all the building blocks are computed, and the database is set up by computing *com*. In the second row of Table 1, we show the cost of just computing *com*, which is virtually the same as that of computing an opening $w_i$. The computation time of *com* and $w_i$ is very small. (As required by our applications in §6, the committed vector that we use consists of small numbers rather than random values in $\mathbb{Z}_p$.) In the 1-entry update, one database entry is modified and *com* is updated. The cost of updating

**Table 1.** $\Pi_{\text{UD}}$ execution times in seconds

| | 1024 bit key | | 2048 bit key | |
|---|---|---|---|---|
| Interface | $N = 100$ | $N = 1000$ | $N = 100$ | $N = 1000$ |
| First update | 0.6844 | 5.9952 | 0.7940 | 6.0822 |
| Computation of *com* or $w_i$ | 0.0032 | 0.03787 | 0.0032 | 0.03787 |
| 1-entry update of *com* or $w_i$ | 0.0001 | 0.0001 | 0.0001 | 0.0001 |
| Read | 0.7496 | 0.7545 | 3.8945 | 3.5911 |

an opening $w_i$ is virtually the same. As can be seen, the cost of the first update grows linearly with the size $N$ of the database, as does the cost of setting up *com* or $w_i$, whereas the cost of updating *com* or $w_i$ is very small and independent of $N$. The execution times for the read interface depend greatly upon the security parameters for the Paillier encryption scheme. However, the execution time is independent of the database size $N$.

## 6 Modular Design with $\mathcal{F}_{\text{UD}}$ and Applications

Consider the following relation $R'$:

$$R' = \{(wit, ins) : [i, vr_i] \in \mathsf{DB} \ \wedge \ 1 = \text{pred}_i(i) \ \wedge \ 1 = \text{pred}_v(vr_i)\}$$

where the witness is $wit = (i, vr_i)$ and the instance is $ins = \mathsf{DB}$. $\text{pred}_i$ and $\text{pred}_v$ represent predicates that $i$ and $vr_i$ must fulfill, e.g., predicates that require $i$ and $vr_i$ to belong to a range or set of values.

We would like to construct a ZK protocol for $R'$ that separates each of the equations of $R'$. We show how this protocol is constructed by using $\mathcal{F}_{\text{UD}}$ and $\mathcal{F}_{\text{NIC}}$ as building blocks, along with the functionalities $\mathcal{F}_{\text{ZK}}^{R_i}$ and $\mathcal{F}_{\text{ZK}}^{R_v}$.

1. On input $\mathsf{DB}$, the verifier uses the $\mathsf{ud.update}$ interface to send $\mathsf{DB}$ to $\mathcal{F}_{\text{UD}}$, which sends $\mathsf{DB}$ to the prover.
2. On input $(i, vr_i)$, the prover checks that $[i, vr_i] \in \mathsf{DB}$.
3. The prover runs the $\mathsf{com.setup}$ interface of $\mathcal{F}_{\text{NIC}}$. The prover uses the $\mathsf{com.commit}$ interface of $\mathcal{F}_{\text{NIC}}$ on input $i$ to obtain a commitment $com_i$ with opening $open_i$. Similarly, the prover obtains from $\mathcal{F}_{\text{NIC}}$ a commitment $comr_i$ to $vr_i$ with opening $openr_i$.
4. The prover uses $\mathsf{ud.read}$ to send $(i, vr_i, com_i, open_i, comr_i, openr_i)$ to $\mathcal{F}_{\text{UD}}$. $\mathcal{F}_{\text{UD}}$ sends $com_i$ and $comr_i$ to the verifier.
5. The verifier runs the $\mathsf{com.setup}$ interface of $\mathcal{F}_{\text{NIC}}$. The verifier uses the $\mathsf{com.validate}$ interface of $\mathcal{F}_{\text{NIC}}$ to validate the commitments $com_i$ and $comr_i$. Then the verifier stores $com_i$ and $comr_i$ and sends a message to the prover to acknowledge the receipt of the commitments.
6. The prover parses the commitment $com_i$ as $(com_i', parcom, \mathsf{COM.Verify})$. The prover sets the witness $wit \leftarrow (i, open_i)$ and the instance $ins \leftarrow (parcom,$

$com'_i$). The prover uses the zk.prove interface to send *wit* and *ins* to $\mathcal{F}_{\text{ZK}}^{R_i}$, where $R_i$ is

$$R_i = \{(wit, ins) : 1 = \mathsf{COM.Verify}(parcom, com'_i, i, open_i) \ \wedge \ 1 = \text{pred}_i(i)\}$$

7. The verifier receives *ins* from $\mathcal{F}_{\text{ZK}}^{R_i}$. The verifier checks that the commitment in *ins* is equal to the stored commitment $com_i$. If it is equal, the binding property guaranteed by $\mathcal{F}_{\text{NIC}}$ ensures that $\mathcal{F}_{\text{UD}}$ and $\mathcal{F}_{\text{ZK}}^{R_i}$ received as input the same position $i$.
8. The last two steps are replicated to prove that $vr_i$ fulfills $1 = \text{pred}_v(vr_i)$ by using $\mathcal{F}_{\text{ZK}}^{R_v}$.

We think that a modular design has two advantages. First, it allows for a simple security analysis. A security proof of a protocol described in the hybrid model is much simpler than a proof that requires reductions to the security properties of different cryptographic primitives. Moreover, each of the building blocks realizes a simpler task and thus requires a simpler protocol with a less involved security analysis. Second, it facilitates the study in isolation of how to create efficient and secure ZK data structures. Namely, different constructions for $\mathcal{F}_{\text{UD}}$ can easily be compared in terms of security and efficiency.

*Application to POT.* The POT protocols in [40,9] are based on previously proposed oblivious transfer (OT) protocols. However, they do not use OT as a building block. Instead, the OT protocol is modified ad-hoc to create the POT protocol, and its security has to be reanalyzed when analyzing the security of the POT protocol.

$\mathcal{F}_{\text{UD}}$ can be used to design a POT protocol modularly. The database DB consists of entries $[i, p_i]$, where $p_i$ is the price to be paid for message $m_i$. To purchase $m_i$, the buyer uses the ud.read interface of $\mathcal{F}_{\text{UD}}$ to read the entry $[i, p_i]$. The provider receives the commitments $com_i$ to $i$ and $comr_i$ to $p_i$. $comr_i$ is used as input to a functionality $\mathcal{F}_{\text{ZK}}^{R_v}$ where the buyer proves that he subtracts the price $p_i$ from his account. $com_i$ is used as input to a functionality for oblivious transfer (modified to receive committed inputs as described in [10]) to allow the buyer to retrieve $m_i$.

Therefore, $\mathcal{F}_{\text{UD}}$ allows the design of a POT protocol that uses a functionality for OT as building block. Thanks to that, the POT protocol can be instantiated with multiple OT schemes and their security does not need to be reanalyzed. Moreover, $\mathcal{F}_{\text{UD}}$ allows the provider to update prices at any time.

*Application to PPB.* In the PPB protocols in [38,39], a meter reading comprises the consumption $c$ and the time interval $i$ of consumption. The tariff policy associates a different function $p = f_i(c)$ to each time interval (and possibly to each consumption interval). $\mathcal{F}_{\text{UD}}$ can be used to design a PPB protocol modularly, where the database DB consists of entries $[i, f_i]$. The PPB protocol works as follows. First, the meter outputs a signed meter reading $(c, i)$. The user reads $[i, f_i]$ through $\mathcal{F}_{\text{UD}}$, and the provider receives commitments $com_i$ to $i$ and $comr_i$ to $f_i$. $com_i$ is used as input to a functionality $\mathcal{F}_{\text{ZK}}^{R_i}$ to prove that $i$ equals the value

signed in the meter reading. $comr_i$ is used as input to a functionality $\mathcal{F}_{\text{ZK}}^{R_v}$ to prove that $p = f_i(c)$. If $f_i$ is represented by a tuple of values (e.g. the coefficients of a polynomial) the variant of $\mathcal{F}_{\text{UD}}$ for databases of the form $[i, vr_{i,1}, \ldots, vr_{i,m}]$ should be used. If the formula $f_i$ also changes with the consumption interval, the database can also store the minimum and maximum values of the consumption interval to allow the user to prove that he uses the right formula. Using $\mathcal{F}_{\text{UD}}$ allows the design of PPB protocols modularly and allows the provider to modify the pricing policies efficiently and at any time.

## 7 Related Work

*Accumulators.* A cryptographic accumulator [6] allows us to represent a set $X$ succinctly as a single accumulator value $A$. To prove that a value $x \in X$, a party computes a witness $W_x$ whose size is independent of $X$. Some accumulator schemes are equipped with efficient ZK proofs to prove knowledge of $W_x$ such that $x \in X$.

NHVC schemes are similar to accumulator schemes that use a trusted setup and are non-hiding [13,5,36,11], i.e., $A$ does not hide $X$. (Recently, hiding accumulators [18,20] have been proposed.) The instantiation of NHVC schemes based on the DHE assumption resembles the accumulator scheme in [11]. The main difference between accumulators and NHVC schemes is that, while accumulators allow us to commit to a set, NHVC schemes allow us to commit to a vector of messages, where each message is committed at a specific position. This allows parties to prove statements about the position $i$ and about the value $vr_i$ stored at $i$, which is needed for $\mathcal{F}_{\text{UD}}$.

*Vector Commitments.* VC schemes [32,15] can be non-hiding and hiding, and can be based on different assumptions such as CDH, RSA and DHE. It would be possible to instantiate our construction under the more standard CDH or RSA assumptions. However, the instantiation of NHVC schemes based on DHE has efficiency advantages. A mercurial VC scheme based on DHE was proposed in [32], and subsequently non-hiding and hiding DHE VC schemes were used in [30,23,27]. In our instantiation of $\Pi_{\text{UD}}$, we use a NHVC scheme based on DHE that is extended with a ZK proof of knowledge of a witness $w_i$ to prove that a value $vr_i$ is stored at position $i$. For this proof, a signature scheme is used along with the NHVC scheme.

Recently, in [28], subvector commitments (SVC) are proposed. In SVC, a commitment can be opened to a set of positions such that the size of the opening does not depend on the size of the set. A construction for SVC secure under the cube Diffie-Hellman assumption is given, in which the public parameter size grows quadratically with the vector length. Our functionality $\mathcal{F}_{\text{UD}}$ only requires to open one vector component at a time. SVC may be used to construct the variant of $\mathcal{F}_{\text{UD}}$ where several positions are read simultaneously, or the variant where the database entries are of the form $[i, vr_{i,1}, \ldots, vr_{i,m}]$. In the read phase, SVC would yield a ZK proof where one opening can be used to open several positions (at the

expense of increasing the storage cost of the public parameters). Despite that SVC provides openings of size independent of the number of positions open, we note that the entire witness of the ZK proof would still grow with the number of positions opened, and thus the efficiency of those proofs would not be independent of the number of positions opened. In [28,7], constructions for SVC based on groups of hidden order are proposed, which are better suited for bit vectors.

Polynomial commitments allow a committer to commit to a polynomial and open the commitment to an evaluation of the polynomial. Polynomial commitments can be used as vector commitments by committing to a polynomial that interpolates the vector to be committed. In [25], a construction of polynomial commitments from the SDH assumption is proposed. The polynomial commitment scheme from SDH has the disadvantage that efficient updates cannot be computed without knowledge of the trapdoor. A further generalization of vector commitments and polynomial commitments are functional commitments [31,28].

*Zero-Knowledge Data Structures.* Zero-Knowledge Sets (ZKS) [34] allow a prover P to commit to a set $X$ and to subsequently prove to a verifier V (non-)membership of an element $x$ in $X$. Zero-Knowledge Databases (ZKDB) are similar to ZKS but each element $x \in X$ is associated with a value $v$, in such a way that a proof that $x \in X$ reveals $v$ to V. Both ZKS and ZKDB are two-party protocols between a prover and a verifier. Zero-knowledge requires that proofs of (non-)membership reveal nothing else beyond (non-)membership, not even the set size.

A ZKS with short proofs for membership and non-membership is proposed in [32] and an updatable ZKDB with short proofs is proposed in [15]. In [25], constructions for "nearly" ZKS and ZKDB, which do not hide the size of the set or database, are given. In [21], a construction for zero-knowledge lists (ZKL) is proposed, where a list is defined as an ordered set. In contrast to our work, existing constructions for ZKS, ZKDB and ZKL are not updatable, with the exceptions of the ZKDB in [33,15].

The main difference between ZK data structures and our work is that ZK data structures hide the database content from the verifier, while in our work the database is public. Another difference is that our database is *oblivious* in the sense that it provides ZK proofs about a committed position $i$ and value $v$, without revealing $i$ or $v$. In existing ZK data structures, the prover reveals $i$ and $v$ along with the proof to the verifier. This property allows our database to be used as building block in privacy-preserving protocols where $i$ and $v$ must remain hidden from the verifier. As for modular design, in those works a method to integrate modularly the proposed ZK data structures as building blocks of other protocols is not given.

*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 [26] achieve verification cost sublinear in $N$, but the cost for the prover is linear in $N$. In succinct non-interactive arguments of knowledge [19], verification cost is independent of $N$, but the cost for the

prover is still linear in $N$. ZK proofs for oblivious RAM programs [35] 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 [35], 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 and 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.

## 8 Conclusion and Future Work

We have proposed an ideal functionality $\mathcal{F}_{\text{UD}}$ and a construction $\Pi_{\text{UD}}$ for an updatable database. In addition to POT and PPB, (non-)updatable databases are implicitly used as building blocks of other protocols. For example, many oblivious transfer with access control [16,8,1,29] protocols and other privacy preserving access control protocols [27] use a database that associates the index $i$ of messages $m_i$ with an access control policy $\mathsf{ACP}_i$ ($\forall i \in [1, N]$). As another example, privacy-preserving client-side profiling protocols [17] use a database that stores a codification of a profiling algorithm. These protocols also use signatures as a way of implementing the database. In those protocols, the reader needs to remain anonymous and unlinkable towards the updater. Therefore, to be used in those protocols, $\mathcal{F}_{\text{UD}}$ and $\Pi_{\text{UD}}$ need to be modified to interact with multiple readers and to guarantee unlikability of readers towards the updater.

## References


1. Abe, M., Camenisch, J., Dubovitskaya, M., Nishimaki, R.: Universally composable adaptive oblivious transfer (with access control) from standard assumptions. In: DIM'13, Proceedings of the 2013 ACM Workshop on Digital Identity Management. pp. 1–12
2. Abe, M., Groth, J., Haralambiev, K., Ohkubo, M.: Optimal structure-preserving signatures in asymmetric bilinear groups. In: CRYPTO 2011. pp. 649–666
3. Aiello, W., Ishai, Y., Reingold, O.: Priced oblivious transfer: How to sell digital goods. In: EUROCRYPT 2001. pp. 119–135
4. Akinyele, J.A., Garman, C., Miers, I., Pagano, M.W., Rushanan, M., Green, M., Rubin, A.D.: Charm: a framework for rapidly prototyping cryptosystems. J. Cryptographic Engineering **3**(2), 111–128 (2013)
5. Au, M.H., Tsang, P.P., Susilo, W., Mu, Y.: Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In: CT-RSA 2009. pp. 295–308
6. Benaloh, J.C., de Mare, M.: One-way accumulators: A decentralized alternative to digital signatures (extended abstract). In: EUROCRYPT '93. pp. 274–285

7. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to iops and stateless blockchains. In: CRYPTO
8. Camenisch, J., Dubovitskaya, M., Neven, G.: Oblivious transfer with access control. In: Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009. pp. 131–140
9. Camenisch, J., Dubovitskaya, M., Neven, G.: Unlinkable priced oblivious transfer with rechargeable wallets. In: Financial Cryptography and Data Security, 14th International Conference, FC 2010. pp. 66–81
10. Camenisch, J., Dubovitskaya, M., Rial, A.: UC commitments for modular protocol design and applications to revocation and attribute tokens. In: CRYPTO 2016. pp. 208–239
11. Camenisch, J., Kohlweiss, M., Soriente, C.: An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In: PKC 2009. pp. 481–500
12. Camenisch, J., Krenn, S., Shoup, V.: A framework for practical universally composable zero-knowledge protocols. In: ASIACRYPT 2011. pp. 449–467
13. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: CRYPTO 2002. pp. 61–76
14. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001 (ePrint 2000/067 version 14-Dec-2005). pp. 136–145 (2001)
15. Catalano, D., Fiore, D.: Vector commitments and their applications. In: PKC 2013. pp. 55–72
16. Coull, S.E., Green, M., Hohenberger, S.: Controlling access to an oblivious database using stateful anonymous credentials. In: PKC 2009. pp. 501–520
17. Danezis, G., Kohlweiss, M., Livshits, B., Rial, A.: Private client-side profiling with random forests and hidden markov models. In: PETS 2012. pp. 18–37
18. Derler, D., Hanser, C., Slamanig, D.: Revisiting cryptographic accumulators, additional properties and relations to other primitives. In: CT-RSA 2015. pp. 127–144
19. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcps. In: EUROCRYPT 2013. pp. 626–645
20. Ghosh, E., Ohrimenko, O., Papadopoulos, D., Tamassia, R., Triandopoulos, N.: Zero-knowledge accumulators and set algebra. In: ASIACRYPT 2016. pp. 67–100
21. Ghosh, E., Ohrimenko, O., Tamassia, R.: Zero-knowledge authenticated order queries and order statistics on a list. In: ACNS 2015. pp. 149–171
22. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM J. Comput. **17**(2), 281–308 (1988)
23. Izabachène, M., Libert, B., Vergnaud, D.: Block-wise p-signatures and non-interactive anonymous credentials with efficient attributes. In: Cryptography and Coding - 13th IMA International Conference, IMACC 2011. pp. 431–450
24. Jawurek, M., Johns, M., Kerschbaum, F.: Plug-in privacy for smart metering billing. In: PETS 2011. pp. 192–210
25. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: ASIACRYPT 2010. pp. 177–194
26. Kilian, J.: A note on efficient zero-knowledge proofs and arguments (extended abstract). In: ACM STOC 1992. pp. 723–732
27. Kohlweiss, M., Rial, A.: Optimally private access control. In: WPES 2013. pp. 37–48
28. Lai, R.W.F., Malavolta, G.: Subvector commitments with application to succinct arguments. In: CRYPTO 2019. pp. 530–560

29. Libert, B., Ling, S., Mouhartem, F., Nguyen, K., Wang, H.: Adaptive oblivious transfer with access control from lattice assumptions. In: ASIACRYPT 2017. pp. 533–563
30. Libert, B., Peters, T., Yung, M.: Group signatures with almost-for-free revocation. In: CRYPTO 2012. pp. 571–589
31. Libert, B., Ramanna, S.C., Yung, M.: Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In: ICALP 2016. pp. 30:1–30:14
32. Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In: TCC 2010. pp. 499–517
33. Liskov, M.D.: Updatable zero-knowledge databases. In: ASIACRYPT 2005. pp. 174–198
34. Micali, S., Rabin, M.O., Kilian, J.: Zero-knowledge sets. In: FOCS 2003. pp. 80–91
35. Mohassel, P., Rosulek, M., Scafuro, A.: Sublinear zero-knowledge arguments for RAM programs. In: EUROCRYPT 2017. pp. 501–531
36. Nguyen, L.: Accumulators from bilinear pairings and applications. In: CT-RSA 2005. pp. 275–292
37. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO '91. pp. 129–140
38. Rial, A., Danezis, G.: Privacy-preserving smart metering. In: WPES 2011. pp. 49–60
39. Rial, A., Danezis, G., Kohlweiss, M.: Privacy-preserving smart metering revisited. Int. J. Inf. Sec. **17**(1), 1–31 (2018)
40. Rial, A., Kohlweiss, M., Preneel, B.: Universally composable adaptive priced oblivious transfer. In: Pairing 2009. pp. 231–247

## A Universally Composable Security

We prove our protocol secure in the universal composability framework [14]. 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 $\varphi$ is analyzed by comparing the view of an environment $\mathcal{Z}$ in a real execution of $\varphi$ 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 $\varphi$ 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 $\varphi$ 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 $\varphi$ is allowed to invoke the ideal functionality $\mathcal{G}$. Therefore, for any protocol $\psi$ that securely realizes $\mathcal{G}$, the composed protocol $\varphi^{\psi}$, which is obtained by replacing each

invocation of an instance of $\mathcal{G}$ with an invocation of an instance of $\psi$, 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 [10].

**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., ud.read.ini in $\mathcal{F}_{\text{UD}}$ in §3. 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 ud.read.ini is the incoming message received by the functionality, i.e., the message through which the interface is invoked. A message ud.read.end is the outgoing message sent by the functionality, i.e., the message that ends the execution of the interface. The message ud.read.sim is used by the functionality to send a message to $\mathcal{S}$, and the message ud.read.rep is used to receive a message from $\mathcal{S}$.

**Network vs local communication.** The identity of an interactive Turing machine instance (ITI) consists 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 ud.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 ud.read.rep sent by $\mathcal{S}$. The query identifier is used to identify the message ud.read.sim to which $\mathcal{S}$ replies with a message ud.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 ud.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 ud.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 ud.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 of the Building Blocks

*Security of Non-Hiding Vector Commitments.* A non-hiding VC scheme must be correct and binding [15].

**Correctness.** The correctness property requires that for $par \leftarrow \mathsf{VC.Setup}(1^k, \ell)$, $\mathbf{x} \leftarrow (\mathbf{x}[1], \ldots, \mathbf{x}[n]) \in \mathcal{M}^n$, $com \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$, $i \leftarrow [1, n]$ and $w \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x})$, $\mathsf{VC.Verify}(par, com, \mathbf{x}[i], i, w)$ outputs 1 with probability 1.

**Binding.** The binding property requires that no adversary can output a vector commitment $com$, a position $i \in [1, \ell]$, two values $x$ and $x'$ and two respective witnesses $w$ and $w'$ such that VC.Verify accepts both, i.e., for $\ell$ polynomial in $k$:

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

*Description of* $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. The functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ is parameterized by a ppt algorithm CRS.Setup. $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ interacts with any parties $\mathcal{P}$ that obtain the common reference string:

1. On input (crs.get.ini, $sid$) from any party $\mathcal{P}$:
   - If $(sid, crs)$ is not stored, run $crs \leftarrow \mathsf{CRS.Setup}$ and store $(sid, crs)$.
   - Create a fresh $qid$ and store $(qid, \mathcal{P})$.
   - Send (crs.get.sim, $sid$, $qid$, $crs$) to $\mathcal{S}$.

S. On input (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 (crs.get.end, $sid$, $crs$) to $\mathcal{P}$.

*Description of* $\mathcal{F}_{\text{AUT}}$. $\mathcal{F}_{\text{AUT}}$ is parameterized by a message space $\mathcal{M}$.

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

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

*Description of* $\mathcal{F}_{\text{ZK}}^{R}$. $\mathcal{F}_{\text{ZK}}^{R}$ is parameterized by a description of a relation $R$. $\mathcal{F}_{\text{ZK}}^{R}$ interacts with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

1. On input $(\mathsf{zk.prove.ini}, sid, wit, ins)$ from $\mathcal{P}$:
   - Abort if $sid \neq (\mathcal{P}, \mathcal{V}, sid')$ or if $(wit, ins) \notin R$.
   - Create a fresh $qid$ and store $(qid, ins)$.
   - 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 $(qid, ins)$ is not stored.
   - Parse $sid$ as $(\mathcal{P}, \mathcal{V}, sid')$.
   - Delete the record $(qid, ins)$.
   - Send $(\mathsf{zk.prove.end}, sid, ins)$ to $\mathcal{V}$.

## C Security Analysis of Construction $\Pi_{\text{UD}}$

To prove that $\Pi_{\text{UD}}$ securely realizes $\mathcal{F}_{\text{UD}}$, 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 $\mathcal{F}_{\text{UD}}$. $\mathcal{S}$ thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\text{UD}}$ for all corrupt parties in the ideal world.

$\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\text{AUT}}$ and $\mathcal{F}_{\text{ZK}}^{R}$. In the descriptions of our simulators below, for brevity, we omit part of the communication between $\mathcal{S}$ and $\mathcal{A}$. Whenever a copy of any of those functionalities sends a message $(*.*.\mathsf{sim})$ to $\mathcal{S}$, $\mathcal{S}$ implicitly forwards that message to $\mathcal{A}$ and runs again a copy of that functionality on input the response provided by $\mathcal{A}$. 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.

In Section C.1, we analyze the security of $\Pi_{\text{UD}}$ when $\mathcal{R}$ is corrupt. In Section C.2, we analyze the security of $\Pi_{\text{UD}}$ when $\mathcal{U}$ is corrupt.

### C.1 Security Analysis of $\Pi_{\text{UD}}$ when $\mathcal{R}$ is Corrupt

We describe the simulator $\mathcal{S}$ for the case in which $\mathcal{R}$ is corrupt.

**Initialization of $\mathcal{S}$.** $\mathcal{S}$ sets $cu \leftarrow 0$. $\mathcal{S}$ runs the $\mathsf{crs.get}$ interface of an instance of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ to get the parameters $par$.

**Honest $\mathcal{U}$ sends an update.** On input from functionality $\mathcal{F}_{\text{UD}}$ the message $(\mathsf{ud.update.sim}, sid, qid, (i, vu_i)_{\forall i \in [1,N]})$, $\mathcal{S}$ sends the message $(\mathsf{ud.update.rep}, sid, qid)$ to $\mathcal{F}_{\text{UD}}$. When $\mathcal{F}_{\text{UD}}$ sends $(\mathsf{ud.update.end}, sid, (i, vu_i)_{\forall i \in [1,N]})$, if $(sid, par, com, \mathbf{x}, cu)$ is not stored, $\mathcal{S}$ does the following:

- $\mathcal{S}$ initializes a counter $cu \leftarrow 0$.
- $\mathcal{S}$ initializes a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vu_i$ for $i \in [1, N]$.
- $\mathcal{S}$ runs $com \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$ and stores $(sid, par, com, \mathbf{x}, cu)$.

Else:

- $\mathcal{S}$ sets $cu' \leftarrow cu + 1$, sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $com' \leftarrow com$.
- For all $i \in [1, N]$ such that $vu_i \neq \perp$, $\mathcal{S}$ computes $\mathbf{x}'[i] \leftarrow vu_i$ and $com' \leftarrow \mathsf{VC.ComUpd}(par, com', i, \mathbf{x}[i], vu_i)$.
- $\mathcal{S}$ replaces the stored tuple $(sid, par, com, \mathbf{x}, cu)$ by $(sid, par, com', \mathbf{x}', cu')$.

$\mathcal{S}$ uses the $\mathsf{aut.send}$ interface of $\mathcal{F}_{\text{AUT}}$ to send $\langle (i, vu_i)_{\forall i \in [1,N]}, cu' \rangle$ to $\mathcal{A}$.

**$\mathcal{A}$ requests and receives $par$.** When $\mathcal{A}$ invokes the $\mathsf{crs.get}$ interface, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ on that input to send $par$ to $\mathcal{A}$.

**$\mathcal{A}$ sends a proof.** When $\mathcal{A}$ invokes the $\mathsf{zk.prove}$ interface on input the witness $wit$ and the instance $ins$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{ZK}}^{R}$ on that input. Then $\mathcal{S}$ parses $ins$ as $(par', com', parcom, com'_i, comr'_i, cr)$ and $wit$ as $(w_i, i, open_i, vr_i, openr_i)$. $\mathcal{S}$ sets $com_i \leftarrow (com'_i, parcom, \mathsf{COM.Verify})$ and $comr_i \leftarrow (comr'_i, parcom, \mathsf{COM.Verify})$. The simulator $\mathcal{S}$ sends the message $(\mathsf{ud.read.ini}, sid, i, vr_i, com_i, open_i, comr_i, openr_i)$ to the functionality $\mathcal{F}_{\text{UD}}$. When the functionality $\mathcal{F}_{\text{UD}}$ sends the message $(\mathsf{ud.read.sim}, sid, qid, com_i, comr_i)$, $\mathcal{S}$ does the following:

- $\mathcal{S}$ retrieves the stored tuple $(sid, par, com, \mathbf{x}, cu)$. If $cr \neq cu$, or if $par' \neq par$, or if $com' \neq com$, $\mathcal{S}$ sends $\mathcal{F}_{\text{UD}}$ a message that makes $\mathcal{F}_{\text{UD}}$ abort.
- Else, if $\mathbf{x}[i] \neq vr_i$, $\mathcal{S}$ outputs failure.
- Else, $\mathcal{S}$ sends $(\mathsf{ud.read.rep}, sid, qid)$ to $\mathcal{F}_{\text{UD}}$.

**Theorem 2.** *When $\mathcal{R}$ is corrupt, $\Pi_{\text{UD}}$ securely realizes $\mathcal{F}_{\text{UD}}$ in the $(\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}, \mathcal{F}_{\text{AUT}}, \mathcal{F}_{\text{ZK}}^{R})$-hybrid model if the NHVC scheme is binding.*

*Proof of Theorem 2.* We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish the real-world protocol from the ideal-world protocol with non-negligible probability. We denote by Pr [**Game** $i$] the probability that the environment distinguishes **Game** $i$ from the real-world protocol.

**Game** 0**:** This game corresponds to the execution of the real-world protocol. Therefore, Pr [**Game** 0] = 0.

**Game** 1**:** **Game** 1 follows **Game** 0, except that **Game** 1 runs an initialization phase to set a counter $cu$ and the parameters $par$. **Game** 1 stores and updates a tuple $(sid, par, com, \mathbf{x}, cu)$. These changes do not alter the view of the environment. Therefore, $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| = 0$.

**Game** 2**:** **Game** 2 follows **Game** 1, except that, when the adversary sends a valid proof with witness $wit$ and instance $ins$, **Game** 2 outputs failure if the values $i$ and $vr_i$ in the witness are such that $\mathbf{x}[i] \neq vr_i$, where $\mathbf{x}[i]$ is in the stored tuple $(sid, par, com, \mathbf{x}, cu)$. The probability that **Game** 2 outputs failure is bound by the following claim.

**Theorem 3.** *Under the binding property of the NHVC scheme, we have that* $|Pr[\textbf{\textit{Game } 2}] - Pr[\textbf{\textit{Game } 1}]| \leq Adv_{\mathcal{A}}^{\mathsf{bin-nhvc}}$.

*Proof of Theorem 3.* We construct an algorithm $B$ that, given an adversary that makes **Game** 2 fail with non-negligible probability, breaks the binding property of the NHVC scheme with non-negligible probability. $B$ behaves as **Game** 2 with the following modifications:

- When the challenger sends the parameters $par$, $B$ stores $par$ as common reference string in the copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$.
- When the adversary sends a valid proof with witness $wit = (w_i, i, open_i, vr_i, openr_i)$ and instance $ins = (par, com, parcom, com_i', comr_i', cr)$ such that the values $i$ and $vr_i$ in the witness fulfill $\mathbf{x}[i] \neq vr_i$, where $\mathbf{x}[i]$ is in the stored tuple $(sid, par, com, \mathbf{x}, cu)$, $B$ runs $w_i' \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x})$ and sends $(com, i, vr_i, \mathbf{x}[i], w_i, w_i')$ to the challenger.

This concludes the proof of Theorem 3.

The distribution of **Game** 2 is identical to our simulation. This concludes the proof of Theorem 2.

### C.2 Security Analysis of Construction $\Pi_{\text{UD}}$ when $\mathcal{U}$ is Corrupt

We describe the simulator $\mathcal{S}$ for the case in which $\mathcal{U}$ is corrupt.

**Initialization of** $\mathcal{S}$**.** $\mathcal{S}$ sets $cr \leftarrow 0$. $\mathcal{S}$ runs the $\mathsf{crs.get}$ interface of an instance of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ to get the parameters $par$.

$\mathcal{A}$ **requests and receives** $par$**.** $\mathcal{S}$ proceeds as in the case where $\mathcal{R}$ is corrupt.

$\mathcal{A}$ **sends update.** When $\mathcal{A}$ invokes the $\mathsf{aut.send}$ interface on input the message $\langle (i, vu_i)_{\forall i \in [1,N]}, cu' \rangle$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{AUT}}$ on that input. If $(sid, par, com, \mathbf{x}, cr)$ is not stored.

- $\mathcal{S}$ initializes a counter $cr \leftarrow 0$.
- $\mathcal{S}$ initializes a vector $\mathbf{x}$ such that $\mathbf{x}[i] = vu_i$ for $i = [1, N]$.
- $\mathcal{S}$ runs $com \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$, and stores $(sid, par, com, \mathbf{x}, cr)$.

Else:

- $\mathcal{S}$ sends an abortion message if $cu' \neq cr + 1$, or if, for all $i \notin [1, N]$, $vu_i \notin \mathbb{U}_v$.
- Otherwise $\mathcal{S}$ sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $com' \leftarrow com$. For all $i \in [1, N]$ such that $vu_i \neq \perp$, $\mathcal{S}$ computes $\mathbf{x}'[i] \leftarrow vu_i$ and $com' \leftarrow \mathsf{VC.ComUpd}(par, com', i, \mathbf{x}[i], vu_i)$.

– $\mathcal{S}$ replaces the stored tuple $(sid, par, com, \mathbf{x}, cr)$ by $(sid, par, com', \mathbf{x}', cu')$.

$\mathcal{S}$ sends $(\mathsf{ud.update.ini}, sid, (i, vu_i)_{\forall i \in [1,N]})$ to $\mathcal{F}_{\mathrm{UD}}$. When $\mathcal{F}_{\mathrm{UD}}$ sends the message $(\mathsf{ud.update.sim}, sid, qid, (i, vu_i)_{\forall i \in [1,N]})$, $\mathcal{S}$ sends $(\mathsf{ud.update.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{UD}}$.

**Honest $\mathcal{R}$ sends a proof.** On input from $\mathcal{F}_{\mathrm{UD}}$ the message $(\mathsf{ud.read.sim}, sid, qid, com_i, comr_i)$, $\mathcal{S}$ sends $(\mathsf{ud.read.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{UD}}$ and receives the message $(\mathsf{ud.read.end}, sid, com_i, comr_i)$ from $\mathcal{F}_{\mathrm{UD}}$. $\mathcal{S}$ does the following:

– $\mathcal{S}$ retrieves the stored tuple $(sid, par, com, \mathbf{x}, cr)$.
– $\mathcal{S}$ parses $com_i$ as $(com_i', parcom, \mathsf{COM.Verify})$.
– $\mathcal{S}$ parses $comr_i$ as $(comr_i', parcom, \mathsf{COM.Verify})$.
– $\mathcal{S}$ sets $ins \leftarrow (par, com, parcom, com_i', comr_i', cr)$.
– $\mathcal{S}$ sets the message corresponding to the $\mathsf{zk.prove}$ interface of $\mathcal{F}_{\mathrm{ZK}}^{R}$ to send $ins$ to $\mathcal{A}$. Note that $\mathcal{S}$ does not know the witness, so it does no run a copy of the functionality. Instead, $\mathcal{S}$ sets the message as if it was sent by a copy of $\mathcal{F}_{\mathrm{ZK}}^{R}$.

**Theorem 4.** *When $\mathcal{U}$ is corrupt, $\Pi_{\mathrm{UD}}$ securely realizes $\mathcal{F}_{\mathrm{UD}}$ in the $(\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}, \mathcal{F}_{\mathrm{AUT}}, \mathcal{F}_{\mathrm{ZK}}^{R})$-hybrid model.*

*Proof of Theorem 4.* The only difference between the real world protocol and $\mathcal{S}$ is that $\mathcal{S}$ does not run $\mathcal{F}_{\mathrm{ZK}}^{R}$ because $\mathcal{S}$ does not know the witness of the proof. Because $\mathcal{F}_{\mathrm{ZK}}^{R}$ does not leak the witness to the adversary, this change does not alter the view of the environment.

## D Security Analysis of the DHE Non-Hiding VC Scheme

**Theorem 5.** *The NHVC scheme is correct and binding under the $\ell$-DHE assumption.*

*Proof.* Correctness can be checked as follows:

$$
\begin{aligned}
& e(com, \tilde{g}_i)/e(w, \tilde{g}) = \\
& \quad = \frac{e(g^{\sum_{j=1}^{n} \mathbf{x}[j](\alpha^{\ell+1-j})}, \tilde{g}^{(\alpha^i)})}{e(g^{(\sum_{j=1, j\neq i}^{n} \mathbf{x}[j](\alpha^{\ell+1-j+i}))}, \tilde{g})} \\
& \quad = \frac{e(g^{(\sum_{j=1}^{n} \mathbf{x}[j](\alpha^{\ell+1-j+i})}, \tilde{g})}{e(g^{(\sum_{j=1, j\neq i}^{n} \mathbf{x}[j](\alpha^{\ell+1-j+i}))}, \tilde{g})} \\
& \quad = e(g, \tilde{g})^{\mathbf{x}[i](\alpha^{\ell+1})} \\
& \quad = e(g_1, \tilde{g}_\ell)^{\mathbf{x}[i]} \ .
\end{aligned}
$$

We show that this NHVC scheme fulfills the binding property under the $\ell$-DHE assumption. Given an adversary $\mathcal{A}$ that breaks the binding property with non-negligible probability $\nu$, we construct an algorithm $\mathcal{T}$ that breaks the $\ell$-DHE assumption with non-negligible probability $\nu$. 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 $\ell$-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 $(com, i, x, x', w, w')$ such that $\mathsf{VC.Verify}(par, com, x, i, w) = 1$, $\mathsf{VC.Verify}(par, com, x', i, w') = 1$, $i \in [1, \ell]$, $x, x' \in \mathcal{M}$, 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 $\ell$-DHE problem.