UC Updatable Non-Hiding Committed Database with Efficient Zero-Knowledge Proofs

Alfredo Rial

SnT, University of Luxembourg
alfredo.rial@uni.lu

Abstract. We define an ideal functionality $\mathcal{F}_{\mathrm{NHCD}}$ and a protocol Π_{NHCD} for an updatable non-hiding committed database (NHCD). NHCD is described as the task of storing a database into a suitable data structure that allows you to efficiently prove in zero-knowledge (ZK) that a value is stored in the database at a certain position. The database is *non-hiding* because both prover and verifier know its content. It is *committed* in the sense that only ZK proofs about position-value pairs that are actually stored are possible. It is *updatable* because its contents can be modified dynamically throughout the protocol execution.
The NHCD task is used implicitly as building block of privacy-preserving protocols for e-commerce, smart billing and access control. In those protocols, this task is intertwined with others. Our functionality $\mathcal{F}_{\mathrm{NHCD}}$ allows us to study constructions for this task in isolation. Furthermore, it allows us to improve modularity in protocol design, by using $\mathcal{F}_{\mathrm{NHCD}}$ as building block of those protocols along with functionalities for other tasks.
Our construction Π_{NHCD} uses a non-hiding vector commitment (VC) scheme as building block. Thanks to the efficiency properties of non-hiding VC schemes, Π_{NHCD} provides ZK proofs whose computation cost (after initialization) and whose size are both independent of the database size. Therefore, Π_{NHCD} is suitable for large databases. Moreover, the database can be updated dynamically and very efficiently.

1 Introduction

Let us consider the following example of a two-party protocol between $\mathcal{P}_0$ and $\mathcal{P}_1$. $\mathcal{P}_1$ chooses a table Tbl of values and sends it to $\mathcal{P}_0$. This table consists of N entries of the form $[i, v]$, where $i \in [1, N]$ is the position and v is the value stored at that position. $\mathcal{P}_0$ receives as input a tuple (x_i, x_v) and a function f. Then $\mathcal{P}_0$ picks the table entry $[i, v]$ such that $i = x_i$ and computes $y = f(v, x_v)$. $\mathcal{P}_0$ sends a zero-knowledge (ZK) proof to prove that y is correctly computed. This ZK proof must prove that $\mathcal{P}_0$ picks the entry $[i, v]$ from Tbl, along with proving the statements $i = x_i$ and $y = f(v, x_v)$.

This protocol, or slight variations of it, is a building block of several privacy-preserving protocols, such as priced oblivious transfer (POT) [2,6,33] and protocols for smart billing [20,32] and privacy-preserving access control [5,23]. For instance, in POT, a seller sells N messages to a buyer, where each message m_i is associated with a price p_i, for $i \in [1, N]$. We can view the list of prices as a table Tbl of entries $[i, p_i]$ that the seller sends to the buyer. The buyer has a deposit dep with the seller and

receives as input a message choice $\sigma \in [1, N]$. We can view dep as x_v and σ as x_i. To purchase the message m_σ, the buyer must subtract p_σ from her deposit dep. We can view the function f as $y = f(x_v, v) = dep - p_\sigma$. The buyer must provide a ZK proof that $y = dep - p_\sigma$, where p_σ is stored in the entry $[\sigma, p_\sigma]$ of Tbl.

In order to allow $\mathcal{P}_0$ to prove that she uses an entry $[i, v]$ from Tbl, $\mathcal{P}_1$ has to store Tbl into some data structure that allows for efficient ZK proofs. Several cryptographic schemes can be used to implement this data structure. For instance, POT protocols [6,33] frequently use a signature scheme with efficient ZK proofs of signature possession. The seller computes signatures s_i on tuples (i, p_i) and sends them to the buyer. In the ZK proof needed to purchase m_σ, the buyer proves possession of a signature s_σ on $[\sigma, p_\sigma]$ in order to show that it is a valid table entry.

We note that the ZK proof computed by $\mathcal{P}_0$ contains two types of statements. On the one hand, $\mathcal{P}_0$ must prove that she uses an entry $[i, v]$ from a table Tbl provided by $\mathcal{P}_1$. On the other hand, $\mathcal{P}_0$ must prove statements about both the position i ($i = x_i$) and the value v ($y = f(v, x_v)$) that are taken from the table. The former is related to proving that the witness of the ZK proof is stored in the data structure. The latter is related to proving statements about the position and the value taken from the data structure.

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.

An important feature that a cryptographic scheme used to implement the data structure must provide is that the data structure be efficiently updateable. For instance, in POT, the seller would like to be able to update the price of a message at any time. However, some schemes do allow for efficient updates. For instance, if signature schemes are used, each time prices are updated, a signature revocation mechanism would be needed to revoke the signatures that sign old prices, which is inefficient.

Our Contribution: functionality $\mathcal{F}_{\mathrm{NHCD}}$. We give a security definition for an updatable non-hiding committed database (NHCD) in Section 3. We define NHCD as the two-party task of storing a table Tbl and proving that an entry $[i, v]$ is stored in Tbl. The database is *non-hiding* in the sense that both $\mathcal{P}_1$ and $\mathcal{P}_0$ know its content. It is *committed* in the sense that it is not possible to prove that $[i, v]$ is stored in Tbl if that is not the case. It is *updatable* because the entries of Tbl can be modified dynamically throughout the protocol execution. We define security in the universal composability (UC) framework [11]. Therefore, we describe an ideal functionality $\mathcal{F}_{\mathrm{NHCD}}$.

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 $\mathcal{F}_{\mathrm{NHCD}}$ as building block in a protocol where $\mathcal{F}_{\mathrm{NHCD}}$ handles the tasks of storing a table Tbl and proving that an entry $[i, v]$ is stored in Tbl, while the ideal functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$ for zero-knowledge is used to prove further statements about the position i and the value v. One challenge when defining a hybrid protocol in the UC model is to ensure that two functionalities receive the same input. To this end, $\mathcal{F}_{\mathrm{NHCD}}$ uses the method proposed in [7], which consists in receiving committed inputs produce

by a functionality $\mathcal{F}_{\text{NIC}}$ for non-interactive commitments. We show how to use $\mathcal{F}_{\text{NHCD}}$ as building block in a protocol designed modularly in Section 5.

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 non-hiding committed database in isolation, which eases the comparison of different constructions for it.

Our Contribution: construction Π_{NHCD}. In Section 4, we propose Π_{NHCD}, a construction that securely realizes $\mathcal{F}_{\text{NHCD}}$. Π_{NHCD} is based on non-hiding vector commitments (VC) [26,12]. A non-hiding VC 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, a witness w_i is computed.

Π_{NHCD} works as follows. $\mathcal{P}_1$ sends a table Tbl to $\mathcal{P}_0$, and both $\mathcal{P}_1$ and $\mathcal{P}_0$ map Tbl to a vector $\mathbf{x}$ and compute a commitment com to $\mathbf{x}$. To update an entry $[i, v]$ to $[i, v']$, $\mathcal{P}_1$ sends $[i, v']$ to $\mathcal{P}_0$, and both $\mathcal{P}_1$ and $\mathcal{P}_0$ run the update algorithm of the non-hiding VC scheme on input com to obtain a commitment com' to a vector $\mathbf{x}'$ such that $\mathbf{x}'[i] = v'$, while the other positions remain unchanged. To prove in zero-knowledge that an entry $[i, v]$ is in Tbl, a party computes a witness w_i and computes a ZK proof of knowledge of (w_i, i, v) that satisfy the verification algorithm of the non-hiding VC scheme.

We describe an efficient instantiation of Π_{NHCD} that uses a non-hiding VC scheme based on the DHE assumption, similar to the mercurial VC scheme in [26]. 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 Section 4.3, we show that the size of a ZK proof that $[i, v] \in \mathsf{Tbl}$ is independent of the size 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.

Our construction 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 an entry $[i, v]$ is in Tbl is equivalent to computing an OR proof where the prover proves that he knows at least one of the table entries. Typically, the size of an OR proof would grow with N, while our proof is of size independent of N. In fact, our construction is suitable for tables Tbl of large sizes N and outperforms other existing ZK proofs for large datasets. We compare our construction with related work more in detail in Section 6.

Outline of the Paper. We summarize the UC framework in Section 2.1, and we depict $\mathcal{F}_{\text{NHCD}}$ in Section 3. In Section 4, we describe Π_{NHCD} and its instantiation. We show how to use $\mathcal{F}_{\text{NHCD}}$ as building block in a protocol designed modularly in Section 5. We analyze related work in Section 6.

2 Universally Composable Security

The universal composability framework [11] is a framework for defining and analyzing the security of cryptographic protocols so that security is retained under 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. The ideal functionality 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 $\mathcal{F}_\varphi$. The environment $\mathcal{Z}$ 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 $\mathcal{G}$. For any protocol ψ that securely realizes the functionality $\mathcal{G}$, 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 [7].

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., nhcd.setup.ini in the functionality $\mathcal{F}_{\mathrm{NHCD}}$ in Section 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. A message nhcd.setup.ini is the incoming message received by the functionality, i.e., the message through which the interface is invoked. A message nhcd.setup.end is the outgoing message sent by the functionality, i.e., the message that ends the execution of the interface. The message nhcd.setup.sim is used by the functionality to send a message to the simulator, and the message nhcd.setup.rep is used to receive a message from the simulator. A message $*.*.$req is used by the functionality to send a message to the simulator to request the description of algorithms from the simulator, and the message $*.*.$alg is used by the simulator to send the description of those algorithms to the functionality.

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 the adversary, 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 $*.*.\mathsf{sim}$ to the simulator in such an interface, a query identifier qid is included in the message. The query identifier must also be included in the response $*.*.\mathsf{rep}$ sent by the simulator. The query identifier is used to identify the message $*.*.\mathsf{sim}$ to which the simulator replies with a message $*.*.\mathsf{rep}$. We note that, typically, the simulator in the security proof may not be able to provide an immediate answer to the functionality after receiving a message $*.*.\mathsf{sim}$. The reason is that the simulator typically needs to interact with the copy of the real adversary it runs in order to produce the message $*.*.\mathsf{rep}$, but the real adversary may not provide the desired answer or may provide a delayed answer. In such cases, when the functionality sends more than one message $*.*.\mathsf{sim}$ to the simulator, the simulator 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 the simulator, 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 the simulator.

Delayed outputs. We say that an ideal functionality $\mathcal{F}$ *sends a public delayed output* v to a party $\mathcal{P}$ if it engages in the following interaction. $\mathcal{F}$ sends to the simulator $\mathcal{S}$ a note that it is ready to generate an output to $\mathcal{P}$. The note includes the value v, the identity $\mathcal{P}$, and a unique identifier for this output. When $\mathcal{S}$ replies to the note by echoing the unique identifier, $\mathcal{F}$ outputs the value v to $\mathcal{P}$. A *private delayed output* is similar, but the value v is not included in the note.

2.1 Modular Design and Ideal Functionality $\mathcal{F}_{\mathrm{NIC}}$

In the UC framework, protocols can be described modularly by using a hybrid model where parties invoke the ideal functionalities of the building blocks of a protocol. For example, consider a protocol that uses as building blocks a zero-knowledge proof of knowledge and a signature scheme. In a modular description of this protocol in the hybrid model, parties in the real world invoke the ideal functionalities for zero-knowledge proofs and for signatures.

One challenge when describing a UC protocol in the hybrid model is to ensure, when needed, that two or more ideal functionalities receive the same input. To address this issue, we use the method proposed in [7]. In [7], a functionality $\mathcal{F}_{\mathrm{NIC}}$ for non-interactive commitments is proposed. $\mathcal{F}_{\mathrm{NIC}}$ interacts with parties $\mathcal{P}_i$ and consists of four interfaces com.setup, com.validate, com.commit and com.verify:

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 cm and obtain a commitment $ccom$ and an opening $copen$. A commitment $ccom$ consists of $(ccom', cparcom, \mathsf{COM.Verify})$, where $ccom'$ is the commitment, $cparcom$ 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 $ccom$ in order to check that $ccom$ contains the correct public parameters and verification algorithm.
4. Any party $\mathcal{P}_i$ uses the com.verify interface to send $(ccom, cm, copen)$ in order to verify that $ccom$ is a commitment to the message cm with the opening $copen$.

$\mathcal{F}_{\mathrm{NIC}}$ can be realized by a perfectly hiding commitment scheme, such as Pedersen commitments [7]. In [7], a method is described to use $\mathcal{F}_{\mathrm{NIC}}$ in order to ensure that a party sends the same input cm to several ideal functionalities. For this purpose, the party first uses com.commit to get a commitment $ccom$ to cm with opening $copen$. Then the party sends $(ccom, cm, copen)$ as input to each of the functionalities, and each functionality runs COM.Verify to verify the commitment. Finally, other parties in the protocol receive the commitment $ccom$ from each of the functionalities and use the com.validate interface to validate $ccom$. Then, if $ccom$ 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 cm. When using $\mathcal{F}_{\mathrm{NIC}}$, it is needed to work in the $\mathcal{F}_{\mathrm{NIC}}||\mathcal{S}_{\mathrm{NIC}}$-hybrid model, where $\mathcal{S}_{\mathrm{NIC}}$ is any simulator for a construction that realizes $\mathcal{F}_{\mathrm{NIC}}$.

Our functionality $\mathcal{F}_{\mathrm{NHCD}}$ receives committed inputs as described in [7]. We depict $\mathcal{F}_{\mathrm{NIC}}$ below.

Description of $\mathcal{F}_{\mathrm{NIC}}$. COM.TrapCom, COM.TrapOpen and COM.Verify are ppt algorithms.

1. On input (com.setup.ini, sid) from a party $\mathcal{P}_i$:
   - If $(sid, cparcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, ctdcom)$ is already stored, include $\mathcal{P}_i$ in the set $\mathbb{P}$, and send (com.setup.end, sid, OK) as a public delayed output to $\mathcal{P}_i$.
   - Otherwise proceed to generate a random qid, store $(qid, \mathcal{P}_i)$ and send the message (com.setup.req, sid, qid) to $\mathcal{S}$.

S. On input (com.setup.alg, sid, qid, m) from $\mathcal{S}$:
   - Abort if no pair $(qid, \mathcal{P}_i)$ for some $\mathcal{P}_i$ is stored.
   - Delete record $(qid, \mathcal{P}_i)$.
   - If $(sid, cparcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, ctdcom)$ is already stored, include $\mathcal{P}_i$ in the set $\mathbb{P}$ and send (com.setup.end, sid, OK) to $\mathcal{P}_i$.

- Otherwise proceed as follows.
  - m is $(cparcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, ctdcom)$.
  - Initialize both an empty table Tbl_{com} and an empty set $\mathbb{P}$, and store $(sid, cparcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, ctdcom)$.
  - Include $\mathcal{P}_i$ in the set $\mathbb{P}$ and send $(\mathsf{com.setup.end}, sid, OK)$ to $\mathcal{P}_i$.

2. On input $(\mathsf{com.validate.ini}, sid, ccom)$ from any party $\mathcal{P}_i$:
   - Abort if $\mathcal{P}_i \notin \mathbb{P}$.
   - Parse $ccom$ as $(ccom', cparcom', \mathsf{COM.Verify}')$.
   - Set $v \leftarrow 1$ if $cparcom' = cparcom$ and $\mathsf{COM.Verify}' = \mathsf{COM.Verify}$. Otherwise, set $v \leftarrow 0$.
   - Send $(\mathsf{com.validate.end}, sid, v)$ to $\mathcal{P}_i$.
3. On input $(\mathsf{com.commit.ini}, sid, cm)$ from any party $\mathcal{P}_i$:
   - Abort if $\mathcal{P}_i \notin \mathbb{P}$ or if $cm \notin \mathcal{M}$, where $\mathcal{M}$ is defined in $cparcom$.
   - Compute $(ccom, cinfo) \leftarrow \mathsf{COM.TrapCom}(sid, cparcom, ctdcom)$.
   - Abort if there is an entry $[ccom, cm', copen', 1]$ in Tbl_{com} such that $cm \neq cm'$.
   - Run $copen \leftarrow \mathsf{COM.TrapOpen}(sid, cm, cinfo)$.
   - Abort if $1 \neq \mathsf{COM.Verify}(sid, cparcom, ccom, cm, copen)$.
   - Append $[ccom, cm, copen, 1]$ to Tbl_{com}.
   - Set $ccom \leftarrow (ccom, cparcom, \mathsf{COM.Verify})$.
   - Send $(\mathsf{com.commit.end}, sid, ccom, copen)$ to $\mathcal{P}_i$.
4. On input $(\mathsf{com.verify.ini}, sid, ccom, cm, copen)$ from any party $\mathcal{P}_i$:
   - Abort if $\mathcal{P}_i \notin \mathbb{P}$ or if $cm \notin \mathcal{M}$ or if $copen \notin \mathcal{R}$, where $\mathcal{M}$ and $\mathcal{R}$ are defined in $cparcom$.
   - Parse $ccom$ as the tuple $(ccom', cparcom', \mathsf{COM.Verify}')$. Abort if the parameters $cparcom' \neq cparcom$ or $\mathsf{COM.Verify}' \neq \mathsf{COM.Verify}$.
   - If there is an entry $[ccom', cm, copen, u]$ in Tbl_{com}, set $v \leftarrow u$.
   - Else, proceed as follows:
     - If there is an entry $[ccom', cm', copen', 1]$ in Tbl_{com} such that $cm \neq cm'$, set $v \leftarrow 0$.
     - Else, proceed as follows:
       - Set $v \leftarrow \mathsf{COM.Verify}(sid, cparcom, ccom', cm, copen)$.
       - Append $[ccom', cm, copen, v]$ to Tbl_{com}.
   - Send $(\mathsf{com.verify.end}, sid, v)$ to $\mathcal{P}_i$.

3 Ideal Functionality for a Non-Hiding Committed Database

We depict the functionality $\mathcal{F}_{\mathrm{NHCD}}$ for a non-hiding committed database. $\mathcal{F}_{\mathrm{NHCD}}$ interacts with two parties $\mathcal{P}_0$ and $\mathcal{P}_1$ and consists of the interfaces nhcd.setup, nhcd.write and nhcd.prove.

1. $\mathcal{P}_1$ uses nhcd.setup to send a table Tbl of N entries of the form $[i, v]$ to $\mathcal{F}_{\mathrm{NHCD}}$. $\mathcal{F}_{\mathrm{NHCD}}$ stores Tbl and sends Tbl to $\mathcal{P}_0$. The simulator $\mathcal{S}$ also learns Tbl.
2. $\mathcal{P}_1$ uses nhcd.write to send a position i and a value v_w to $\mathcal{F}_{\mathrm{NHCD}}$. $\mathcal{F}_{\mathrm{NHCD}}$ updates Tbl to contain value v_w at position i and sends i and v_w to $\mathcal{P}_0$. The simulator $\mathcal{S}$ also learns i and v_w.

3. $\mathcal{P}_b$ ($b \in [0,1]$) uses nhcd.prove to send a position i and a value v_r to $\mathcal{F}_{\text{NHCD}}$, along with commitments and openings $(ccom_i, copen_i)$ and $(ccom_r, copen_r)$ to the position and value respectively. $\mathcal{F}_{\text{NHCD}}$ verifies the commitments and checks that there is an entry $[i, v_r]$ in the table Tbl. In that case, $\mathcal{F}_{\text{NHCD}}$ sends $ccom_i$ and $ccom_r$ to $\mathcal{P}_{1-b}$. The simulator $\mathcal{S}$ also learns $ccom_i$ and $ccom_r$. Neither $\mathcal{P}_{1-b}$ nor $\mathcal{S}$ learn $(i, copen_i)$ or $(v_r, copen_r)$.

$\mathcal{F}_{\text{NHCD}}$ stores a database in the form of a table Tbl of entries $[i, v]$, where i is the position and v the value stored at that position. $\mathcal{P}_1$ sets up the table and can update any table entries. $\mathcal{F}_{\text{NHCD}}$ sends Tbl and its updates to $\mathcal{P}_0$, i.e. the table is *not hidden* from $\mathcal{P}_0$. It would be possible to define a functionality $\mathcal{F}_{\text{NHCD}}$ where both $\mathcal{P}_1$ and $\mathcal{P}_0$ can set up and update the table, but we restrict them to one party for simplicity and because our applications do not require it.

$\mathcal{F}_{\text{NHCD}}$ allows both $\mathcal{P}_0$ and $\mathcal{P}_1$ to send to the other party two commitments $ccom_i$ and $ccom_r$ that commit to a position i and a value v such that $[i, v] \in$ Tbl. $\mathcal{F}_{\text{NHCD}}$ checks that $ccom_i$ and $ccom_r$ indeed commit to a table entry. Committed inputs to $\mathcal{F}_{\text{NHCD}}$ are needed to use $\mathcal{F}_{\text{NHCD}}$ as building block of a hybrid protocol along with $\mathcal{F}_{\text{NIC}}$, as described in Section 5. For simplicity, $\mathcal{F}_{\text{NHCD}}$ restricts $ccom_i$ and $ccom_r$ to contain the same parameters and verification algorithm, which means that they must be produced by the same instance of $\mathcal{F}_{\text{NIC}}$.

$\mathcal{F}_{\text{NHCD}}$ stores a counter c_0 for $\mathcal{P}_0$ and a counter c_1 for $\mathcal{P}_1$. These counters are used to check that $\mathcal{P}_0$ and $\mathcal{P}_1$ have the same version of Tbl. When $\mathcal{P}_1$ initiates the nhcd.write interface, c_1 is incremented. When $\mathcal{F}_{\text{NHCD}}$ sends the table update to $\mathcal{P}_0$, $\mathcal{F}_{\text{NHCD}}$ checks that $c_1 = c_0 + 1$ and then increments c_0. In the nhcd.prove interface, $\mathcal{F}_{\text{NHCD}}$ checks that the counters of both parties are equal, which ensures that they have the same table.

The session identifier sid is of the form $(\mathcal{P}_0, \mathcal{P}_1, sid')$. Including the identities in sid ensures that every party $\mathcal{P}_1$ can initiate an instance of $\mathcal{F}_{\text{NHCD}}$ with any other party $\mathcal{P}_0$. Query identifiers qid are created by $\mathcal{F}_{\text{NHCD}}$ to communicate with the simulator $\mathcal{S}$ in the interfaces nhcd.write and nhcd.prove. In the nhcd.setup interface it is not needed because this interface can only be invoked once.

When receiving a message, $\mathcal{F}_{\text{NHCD}}$ checks that each input belongs to its correct domain and aborts if that is not case. $\mathcal{F}_{\text{NHCD}}$ also aborts if a message is received at an incorrect moment in the protocol. For example, $\mathcal{F}_{\text{NHCD}}$ aborts if $\mathcal{P}_1$ invokes nhcd.write before invoking nhcd.setup.

Description of $\mathcal{F}_{\text{NHCD}}$. $\mathcal{F}_{\text{NHCD}}$ is parameterised by a universe of values U_v and by a maximum table size N. $\mathcal{F}_{\text{NHCD}}$ interacts with a party $\mathcal{P}_0$ and a party $\mathcal{P}_1$. In the following, $b \in [0,1]$.

1. On input (nhcd.setup.ini, sid, Tbl) from $\mathcal{P}_1$:
   - Abort if $sid \notin (\mathcal{P}_0, \mathcal{P}_1, sid')$ or if (sid, Tbl', c_1) is already stored.
   - Abort if Tbl does not consist of N entries of the form $[i, v]$.
   - Abort if for $i = 1$ to N, $v \notin U_v$ for any entry $[i, v]$ in Tbl.
   - Initialize a counter $c_1 \leftarrow 0$ for $\mathcal{P}_1$ and store (sid, Tbl, c_1).
   - Send (nhcd.setup.sim, sid, Tbl) to $\mathcal{S}$.

S. On input (nhcd.setup.rep, sid) from $\mathcal{S}$:

- Abort if (sid, Tbl, c_1) is not stored, or if (sid, Tbl, c_0) is already stored.
- Initialize a counter $c_0 \leftarrow 0$ for $\mathcal{P}_0$ and store (sid, Tbl, c_0).
- Send $(\mathsf{nhcd.setup.end}, sid, \mathsf{Tbl})$ to $\mathcal{P}_0$.

2. On input $(\mathsf{nhcd.write.ini}, sid, i, v_w)$ from $\mathcal{P}_1$:
   - Abort if (sid, Tbl, c_1) is not stored.
   - Abort if $i \notin [1, N]$, or if $v_w \notin U_v$.
   - Increment c_1 and update c_1 and the table entry $[i, v_w]$ in (sid, Tbl, c_1).
   - Create a fresh qid and store (qid, i, v_w, c_1).
   - Send $(\mathsf{nhcd.write.sim}, sid, qid, i, v_w)$ to $\mathcal{S}$.

S. On input $(\mathsf{nhcd.write.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if (qid, i, v_w, c_1') or (sid, Tbl, c_0) are not stored, or if $c_1' \neq c_0 + 1$.
   - Increment c_0 and update c_0 and the table entry $[i, v_w]$ in (sid, Tbl, c_0).
   - Delete the record (qid, i, v_w, c_1').
   - Send $(\mathsf{nhcd.write.end}, sid, i, v_w)$ to $\mathcal{P}_0$.

3. On input $(\mathsf{nhcd.prove.ini}, sid, ccom_i, i, copen_i, ccom_r, v_r, copen_r)$ from $\mathcal{P}_b$:
   - Abort if (sid, Tbl, c_b) is not stored.
   - Abort if $i \notin [1, N]$, or if $v_r \notin U_v$, or if $[i, v_r]$ is not stored in Tbl.
   - Parse $ccom_i$ as $(ccom_i', cparcom_i, \mathsf{COM.Verify}_i)$.
   - Parse $ccom_r$ as $(ccom_r', cparcom_r, \mathsf{COM.Verify}_r)$.
   - Abort if $cparcom_i \neq cparcom_r$, or if $\mathsf{COM.Verify}_i \neq \mathsf{COM.Verify}_r$, or if $\mathsf{COM.Verify}_i$ is not a ppt algorithm.
   - Abort if $1 \neq \mathsf{COM.Verify}_i(cparcom_i, ccom_i, i, copen_i)$.
   - Abort if $1 \neq \mathsf{COM.Verify}_r(cparcom_r, ccom_r, v_r, copen_r)$.
   - Create a fresh qid and store $(qid, ccom_i, ccom_r, \mathcal{P}_b, c_b)$.
   - Send $(\mathsf{nhcd.prove.sim}, sid, qid, ccom_i, ccom_r)$ to $\mathcal{S}$.

S. On input $(\mathsf{nhcd.prove.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid, ccom_i, ccom_r, \mathcal{P}_b, c_b')$ or $(sid, \mathsf{Tbl}, c_{1-b})$ are not stored, or if $c_b' \neq c_{1-b}$.
   - Delete the record $(qid, ccom_i, ccom_r, \mathcal{P}_b, c_b')$.
   - Send $(\mathsf{nhcd.prove.end}, sid, ccom_i, ccom_r)$ to $\mathcal{P}_{1-b}$.

4 Construction Π_{NHCD} for a Non-Hiding Committed Database

4.1 Building Blocks of Construction Π_{NHCD}

Non-Hiding Vector Commitments A non-hiding vector commitment (VC) scheme [12] 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 a witness w to $\mathbf{x}[i]$, with the size of w independent of i and n. The scheme consists of the following algorithms.

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

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

$\mathsf{VC.Prove}(par, i, \mathbf{x})$**.** Compute a witness w for $\mathbf{x}[i]$.

$\mathsf{VC.Verify}(par, com, x, i, w)$**.** Output 1 if w is a valid witness for x being at position i and 0 otherwise.

$\mathsf{VC.ComUpd}(par, com, j, x, x')$**.** 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.

$\mathsf{VC.WitUpd}(par, w, i, j, x, x')$**.** On input a witness w for position i valid for a commitment com with value x at position j, output a witness 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 [12]. We give the definition of those properties in Section B.1.

Ideal Functionality $\mathcal{F}_{\mathbf{CRS}}^{\mathsf{CRS.Setup}}$ Our protocol uses the functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ for common reference string generation in [11]. $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ interacts with any parties $\mathcal{P}$ that obtain the common reference string, and consists of one interface crs.get. A party $\mathcal{P}$ uses the crs.get interface to request and receive the common reference string 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. The simulator $\mathcal{S}$ also receives crs.

Description of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$*.* $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ 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 $(\mathsf{crs.get.ini}, sid)$ 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 $(\mathsf{crs.get.sim}, sid, qid, crs)$ 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 $\mathcal{P}$.

Ideal Functionality $\mathcal{F}_{\mathbf{AUT}}$ Our protocol uses the functionality $\mathcal{F}_{\mathrm{AUT}}$ for an authenticated channel in [11]. $\mathcal{F}_{\mathrm{AUT}}$ 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 $\mathcal{F}_{\mathrm{AUT}}$. $\mathcal{F}_{\mathrm{AUT}}$ leaks m to the simulator $\mathcal{S}$ and, after receiving a response from $\mathcal{S}$, $\mathcal{F}_{\mathrm{AUT}}$ sends m to $\mathcal{R}$. $\mathcal{S}$ cannot modify m. The session identifier sid contains the identities of $\mathcal{T}$ and $\mathcal{R}$.

Description of $\mathcal{F}_{\mathrm{AUT}}$*.* $\mathcal{F}_{\mathrm{AUT}}$ 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 $\mathcal{R}$.

Ideal Functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$ for Zero-Knowledge 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 $\mathcal{F}_{\mathrm{ZK}}^{R}$ for zero-knowledge in [11]. $\mathcal{F}_{\mathrm{ZK}}^{R}$ 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 $\mathcal{F}_{\mathrm{ZK}}^{R}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ 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.

Description of $\mathcal{F}_{\mathrm{ZK}}^{R}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ is parameterized by a description of a relation R. $\mathcal{F}_{\mathrm{ZK}}^{R}$ 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 $\mathcal{V}$.

4.2 Construction Π_{NHCD}

We describe a construction Π_{NHCD} that securely realizes the functionality $\mathcal{F}_{\mathrm{NHCD}}$. Our construction uses the non-hiding VC scheme (VC.Setup, VC.Commit, VC.Prove, VC.Verify, VC.ComUpd, VC.WitUpd) and the functionalities $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\mathrm{AUT}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R}$ described in Section 4.1. $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ is parameterized by the algorithm VC.Setup of the non-hiding VC scheme.

In the nhcd.setup interface, $\mathcal{P}_1$ receives a table Tbl and sends it to $\mathcal{P}_0$ by using $\mathcal{F}_{\mathrm{AUT}}$. Both $\mathcal{P}_1$ and $\mathcal{P}_0$ map Tbl to a vector $\mathbf{x}$ and run VC.Commit to get a commitment com to $\mathbf{x}$. com is used to store Tbl. A position in the vector commitment corresponds to a position in Tbl, and the value committed to in that position corresponds to the value stored in Tbl in that position.

In the nhcd.write interface, $\mathcal{P}_1$ receives a position i and a value v_w and sends them to $\mathcal{P}_0$ by using $\mathcal{F}_{\mathrm{AUT}}$. Both $\mathcal{P}_1$ and $\mathcal{P}_0$ update $\mathbf{x}$ and run VC.ComUpd to update the vector commitment com. If a witness w_i is already stored, $\mathcal{P}_1$ and $\mathcal{P}_0$ run VC.WitUpd to update it.

In the nhcd.prove interface, $\mathcal{P}_0$ (or $\mathcal{P}_1$) receives a position i and a value v_r, along with their respective commitments and openings $(ccom_i, copen_i)$ and $(ccom_r, copen_r)$. If a witness w_i for i is not stored, $\mathcal{P}_0$ (or $\mathcal{P}_1$) run VC.Prove to compute it. Then $\mathcal{P}_0$ (or $\mathcal{P}_1$) proves in zero-knowledge that the commitments $ccom_i$ and $ccom_r$ commit to a position i and a value v_r such that $\mathbf{x}[i] = v_r$, where $\mathbf{x}$ is the vector committed in com.

Description of Π_{NHCD}. Construction Π_{NHCD} uses a non-hiding VC scheme with algorithms $(\mathsf{VC.Setup}, \mathsf{VC.Commit}, \mathsf{VC.Prove}, \mathsf{VC.Verify}, \mathsf{VC.ComUpd}, \mathsf{VC.WitUpd})$ and the ideal functionalities $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\mathrm{AUT}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R}$. The constant N denotes the size of the array, and the universe of state values U_v is given by the message space of the vector commitment scheme. In the following, $b \in [0, 1]$.

1. On input $(\mathsf{nhcd.setup.ini}, sid, \mathsf{Tbl})$, $\mathcal{P}_1$ and $\mathcal{P}_0$ do the following:
   - $\mathcal{P}_1$ aborts if $sid \notin (\mathcal{P}_0, \mathcal{P}_1, sid')$ or if $(sid, par, com, \mathbf{x}, c_1)$ is already stored.
   - $\mathcal{P}_1$ aborts if Tbl does not contain N entries of the form $[i, v]$ or if, for $i = 1$ to N, $v \notin U_v$.
   - $\mathcal{P}_1$ sends $(\mathsf{crs.get.ini}, sid)$ to $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ and receives $(\mathsf{crs.get.end}, sid, par)$ from $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$. To compute par, $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ runs $\mathsf{VC.Setup}(1^k, N)$.
   - $\mathcal{P}_1$ initializes a counter $c_1 \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[i] = v$ for $i = 1$ to N, where $[i, v] \in \mathsf{Tbl}$. $\mathcal{P}_1$ runs $com \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$.
   - $\mathcal{P}_1$ stores $(sid, par, com, \mathbf{x}, c_1)$.
   - $\mathcal{P}_1$ sets $sid_{\mathrm{AUT}} \leftarrow (\mathcal{P}_1, \mathcal{P}_0, sid')$ and sends $(\mathsf{aut.send.ini}, sid_{\mathrm{AUT}}, \mathsf{Tbl})$ to $\mathcal{F}_{\mathrm{AUT}}$.
   - $\mathcal{P}_0$ receives $(\mathsf{aut.send.end}, sid_{\mathrm{AUT}}, \mathsf{Tbl})$ from $\mathcal{F}_{\mathrm{AUT}}$.
   - $\mathcal{P}_0$ aborts if $(sid, par, com, \mathbf{x}, c_0)$ is already stored.
   - $\mathcal{P}_0$ sends $(\mathsf{crs.get.ini}, sid)$ to $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ and receives $(\mathsf{crs.get.end}, sid, par)$ from $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$.
   - $\mathcal{P}_0$ initializes a counter $c_0 \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[i] = v$ for $i = 1$ to N, where $[i, v] \in \mathsf{Tbl}$. $\mathcal{P}_b$ runs $com \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$.
   - $\mathcal{P}_0$ stores $(sid, par, com, \mathbf{x}, c_0)$.
   - $\mathcal{P}_0$ outputs $(\mathsf{nhcd.setup.end}, sid, \mathsf{Tbl})$.
2. On input $(\mathsf{nhcd.write.ini}, sid, i, v_w)$, $\mathcal{P}_1$ and $\mathcal{P}_0$ do the following:
   - $\mathcal{P}_1$ aborts if $(sid, par, com, \mathbf{x}, c_1)$ is not stored.
   - $\mathcal{P}_1$ aborts if $i \notin [1, N]$ or if $v_w \notin U_v$.
   - $\mathcal{P}_1$ sets $c_1' \leftarrow c_1 + 1$.
   - $\mathcal{P}_1$ sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $\mathbf{x}'[i] \leftarrow v_w$.
   - $\mathcal{P}_1$ computes $com' \leftarrow \mathsf{VC.ComUpd}(par, com, i, \mathbf{x}, \mathbf{x}')$.
   - $\mathcal{P}_1$ replaces the stored tuple $(sid, par, com, \mathbf{x}, c_1)$ by $(sid, par, com', \mathbf{x}', c_1')$.
   - For $j = 1$ to N, if (sid, j, w_j) is stored, $\mathcal{P}_1$ computes $w_j' \leftarrow \mathsf{VC.WitUpd}(par, w_j, j, i, \mathbf{x}[i], \mathbf{x}'[i])$ and replaces (sid, j, w_j) by (sid, j, w_j').
   - $\mathcal{P}_1$ sets $sid_{\mathrm{AUT}} \leftarrow (\mathcal{P}_1, \mathcal{P}_0, sid')$ and sends $(\mathsf{aut.send.ini}, sid_{\mathrm{AUT}}, \langle i, v_w, c_1' \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
   - $\mathcal{P}_0$ receives $(\mathsf{aut.send.end}, sid_{\mathrm{AUT}}, \langle i, v_w, c_1' \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
   - $\mathcal{P}_0$ aborts if $(sid, par, com, \mathbf{x}, c_0)$ is not stored.
   - $\mathcal{P}_0$ aborts if $c_1' \neq c_0 + 1$.
   - $\mathcal{P}_0$ aborts if $i \notin [1, N]$ or if $v_w \notin U_v$.
   - $\mathcal{P}_0$ sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $\mathbf{x}'[i] \leftarrow v_w$.
   - $\mathcal{P}_0$ computes $com' \leftarrow \mathsf{VC.ComUpd}(par, com, i, \mathbf{x}, \mathbf{x}')$.
   - $\mathcal{P}_0$ replaces the stored tuple $(sid, par, com, \mathbf{x}, c_0)$ by $(sid, par, com', \mathbf{x}', c_1')$.
   - For $j = 1$ to N, if (sid, j, w_j) is stored, $\mathcal{P}_0$ computes $w_j' \leftarrow \mathsf{VC.WitUpd}(par, w_j, j, i, \mathbf{x}[i], \mathbf{x}'[i])$ and replaces (sid, j, w_j) by (sid, j, w_j').

- $\mathcal{P}_0$ outputs $(\mathsf{nhcd.write.end}, sid, i, v_w)$.

3. On input $(\mathsf{nhcd.prove.ini}, sid, ccom_i, i, copen_i, ccom_r, v_r, copen_r)$, $\mathcal{P}_b$ and $\mathcal{P}_{1-b}$ do the following:
   - $\mathcal{P}_b$ aborts if $(sid, par, com, \mathbf{x}, c_b)$ is not stored.
   - $\mathcal{P}_b$ parses $ccom_i$ as $(ccom'_i, cparcom_i, \mathsf{COM.Verify}_i)$ and $ccom_r$ as $(ccom'_r, cparcom_r, \mathsf{COM.Verify}_r)$.
   - $\mathcal{P}_b$ aborts if $cparcom_i \neq cparcom_r$, or if $\mathsf{COM.Verify}_i \neq \mathsf{COM.Verify}_r$, or if $\mathsf{COM.Verify}_i$ is not a ppt algorithm, or if $1 \neq \mathsf{COM.Verify}_i(cparcom_i, ccom_i, i, copen_i)$, or if $1 \neq \mathsf{COM.Verify}_r(cparcom_r, ccom_r, v_r, copen_r)$.
   - $\mathcal{P}_b$ aborts if $i \notin [1, N]$, or if $v_r \notin U_v$, or if $\mathbf{x}[i] \neq v_r$.
   - If (sid, i, w_i) is not stored, $\mathcal{P}_b$ runs $w_i \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x})$ and stores (sid, i, w_i).
   - $\mathcal{P}_b$ sets $wit \leftarrow (w_i, i, copen_i, v_r, copen_r)$.
   - $\mathcal{P}_b$ sets $ins \leftarrow (par, com, cparcom_i, ccom'_i, ccom'_r, c_b)$.
   - $\mathcal{P}_b$ sends $(\mathsf{zk.prove.ini}, sid, wit, ins)$ to $\mathcal{F}_{\mathrm{ZK}}^R$.
   - $\mathcal{P}_{1-b}$ receives $(\mathsf{zk.prove.end}, sid, ins)$ from $\mathcal{F}_{\mathrm{ZK}}^R$.
   - $\mathcal{P}_{1-b}$ aborts if $(sid, par, com, \mathbf{x}, c_{1-b})$ is not stored.
   - $\mathcal{P}_{1-b}$ parses ins as $(par', com', cparcom_i, ccom'_i, ccom'_r, c_b)$.
   - $\mathcal{P}_{1-b}$ aborts if $c_b \neq c_{1-b}$, or if $par' \neq par$, or if $com' \neq com$.
   - $\mathcal{P}_{1-b}$ sets $ccom_i \leftarrow (ccom'_i, cparcom_i, \mathsf{COM.Verify}_i)$ and $ccom_r \leftarrow (ccom'_r, cparcom_i, \mathsf{COM.Verify}_i)$. ($\mathsf{COM.Verify}_i$ is part of the description of the relation R.)
   - $\mathcal{P}_{1-b}$ outputs $(\mathsf{nhcd.prove.end}, sid, ccom_i, ccom_r)$.

Theorem 1. Π_{NHCD} *securely realizes* $\mathcal{F}_{\mathrm{NHCD}}$ *in the* $(\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}, \mathcal{F}_{\mathrm{AUT}}, \mathcal{F}_{\mathrm{ZK}}^R)$*-hybrid model if the non-hiding VC scheme (*VC.Setup*,* VC.Commit*,* VC.Prove*,* VC.Verify*,* VC.ComUpd*,* VC.WitUpd*) is binding.*

When $\mathcal{P}_0$ or $\mathcal{P}_1$ are corrupt, the binding property of the vector commitment scheme guarantees that the adversary is not able to open the vector commitment to a position and a value if that value was not previously committed at that position. We analyze in detail the security of Π_{NHCD} in Section A.

4.3 Instantiation of Construction Π_{NHCD} and Efficiency Analysis

We show an instantiation of Π_{NHCD} based on a non-hiding VC scheme secure under the DHE assumption, similar to the scheme in [26]. 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 $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and an efficient algorithm to compute $e(a, b)$ for any $a \in \mathbb{G}$, $b \in \tilde{\mathbb{G}}$. We recall the Diffie-Hellman exponent (t-DHE) assumption.

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

Let $k \in \mathbb{N}$ denote the security parameter. The scheme works as follows:

$\mathsf{VC.Setup}(1^k, \ell)$**.** 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 $w' = w \cdot g_{\ell+1-j+i}^{x'-x}$.

Theorem 2. *This non-hiding VC scheme is correct and binding under the ℓ-DHE assumption. This theorem is proven in Section B.2.*

ZK proof for R. We show a ZK proof for the relation R required in Π_{NHCD}. R involves proving knowledge of a position i, a value v and a witness w such that the verification equation $e(com, \tilde{g}_i) = e(w, g)e(g_1, \tilde{g}_\ell)^v$ holds. Additionally, it involves proving that the position i is committed in a commitment $ccom_i$ with opening $copen_i$, and the value v is committed in a commitment $ccom_r$ with opening $copen_r$. Because α is secret, the relation between $\tilde{g}_i = \tilde{g}^{\alpha^i}$ and i is not efficiently provable. For this reason, we extend the parameters of the vector commitment scheme with structure preserving signatures (SPS) that bind i with $\tilde{g}_i$. Concretely, we use the SPS scheme in [1] with algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VfSig})$ to sign tuples $\langle g^i, g^{sid}, \tilde{g}_i \rangle$, where sid is the session identifier.

$\mathsf{VC.Setup}(1^k, \ell)$**.** 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 $\tilde{g}_i = \tilde{g}^{(\alpha^i)}$. Compute $(sk, pk) \leftarrow \mathsf{KeyGen}(grp, 2, 1)$. For $i \in [1, \ell]$, run $s_i \leftarrow \mathsf{Sign}(sk, \langle g^i, g_{\ell+1-i}, \tilde{g}_i \rangle)$. Compute additional bases $h \leftarrow \mathbb{G}$ and $\tilde{h} \leftarrow \tilde{\mathbb{G}}$. Output the parameters $par = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell}, pk, s_1, \ldots, s_\ell, h, \tilde{h}, \mathcal{M} = \mathbb{Z}_p, \mathcal{R} = \mathbb{Z}_p)$.

Let (g, h) be the parameters of the Pedersen commitment scheme. Let (U_1, V, W_1, W_2, Z) be the public key of the signature scheme. Let (R, S, T) be a signature on $(g^i, g^{sid}, \tilde{g}_i)$. Following the notation in [9], we describe the proof as follows.

$$
\begin{aligned}
&\text{ZK}\, i, copen_i, v, copen_r, \tilde{g}_i, w, R, S, T : \\
&\quad ccom_i = g^i h^{copen} \ \wedge \ ccom_r = g^v h^{copen_r} \wedge && (1)\\
&\quad e(R, V)e(S, \tilde{g})e(g, W_1)^i e(g^{sid}, W_2)e(g, Z)^{-1} = 1 \wedge && (2)\\
&\quad e(R, T)e(U_1, \tilde{g}_i)e(g, \tilde{g})^{-1} = 1 \wedge && (3)\\
&\quad e(com, \tilde{g}_i)^{-1} e(w, \tilde{g})e(g_1, \tilde{g}_\ell)^v = 1 && (4)
\end{aligned}
$$

Equation 1 proves knowledge of the openings of the Pedersen commitments $ccom_i$ and $ccom_r$. Equation 2 and Equation 3 prove knowledge of a signature (R, S, T) on a message $\langle g^i, g^{sid}, \tilde{g}_i \rangle$. Equation 4 proves that the value v in $ccom_r$ is equal to the value committed in the position i of the vector commitment com.

Efficiency Analysis. We provide below the efficiency analysis of our construction. We can conclude that, after setup, the computation and communication costs of the nhcd.write and nhcd.prove interfaces are constant and independent of the size N of Tbl, except for the computation cost of a witness w_i, which is linear in N. When w_i is stored, it can be updated with constant cost.

Storage Cost. $\mathcal{P}_0$ and $\mathcal{P}_1$ store the common reference string par, whose size grows linearly with N. Throughout the protocol execution, in addition to par, $\mathcal{P}_0$ and $\mathcal{P}_1$ also store the last update of the vector commitment com, the committed vector, and the witnesses w_i for $i \in [1, N]$. In conclusion, the storage cost is linear in N.

Communication Cost. In the nhcd.setup interface, $\mathcal{P}_1$ sends Tbl to $\mathcal{P}_0$. The communication cost grows linearly with N. In the nhcd.write interface, $\mathcal{P}_1$ sends a position and a value to $\mathcal{P}_0$. The communication cost is constant. In the nhcd.prove interface, $\mathcal{P}_0$ (resp. $\mathcal{P}_1$) sends an instance and a ZK proof to $\mathcal{P}_1$ (resp. $\mathcal{P}_0$). 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 setup phase, the communication cost is constant.

Computation Cost. In the nhcd.setup interface, $\mathcal{P}_0$ and $\mathcal{P}_1$ compute the vector commitment com with cost linear with N. In the nhcd.write interface, $\mathcal{P}_0$ and $\mathcal{P}_1$ update com with constant computation cost. They also update the stored witnesses w_i. Each witness update requires constant computation cost. In the nhcd.prove interface, if a witness w_i is not stored, $\mathcal{P}_0$ (resp. $\mathcal{P}_1$) 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.

5 Modular Protocol Design with $\mathcal{F}_{\mathrm{NHCD}}$

We show how to use $\mathcal{F}_{\mathrm{NHCD}}$ as building block of a protocol described in the hybrid model. As a simple example, we describe a two-party protocol between $\mathcal{P}_0$ and $\mathcal{P}_1$ for the following task. $\mathcal{P}_1$ receives as input a table Tbl and sends it to $\mathcal{P}_0$. The table Tbl consists of N entries of the form $[i, v]$, where $i \in [1, N]$ is the position and v is the value. $\mathcal{P}_0$ receives an input (x_i, x_v). $\mathcal{P}_0$ then has to prove in zero-knowledge that he evaluates correctly a function f on input (x_i, x_v) and one of the table entries. Consider for example the relation R:

$$R = \{(wit, ins) : [i, v] \in \mathsf{Tbl} \ \wedge \ i = x_i \ \wedge \ y = f(x_v, v)\}$$

where the witness is $wit = (x_i, x_v, i, v)$ and the instance is $ins = (y, \mathsf{Tbl})$. I.e, $\mathcal{P}_0$ must prove that y is the result of evaluating f on input x_v and v, where v is the value stored in the entry $[i, v] \in \mathsf{Tbl}$ such that $i = x_i$. For simplicity, R does not specify where x_i and x_v are stored, i.e. they could be messages in a commitment, ciphertext or signature, and R could be extended accordingly to show that.

We describe a protocol for this task that uses $\mathcal{F}_{\mathrm{NHCD}}$, $\mathcal{F}_{\mathrm{NIC}}$ and a functionality $\mathcal{F}_{\mathrm{ZK}}^{R'}$, where R' is described below.

1. On input Tbl, $\mathcal{P}_1$ uses the nhcd.setup interface to send Tbl to $\mathcal{F}_{\mathrm{NHCD}}$, which sends Tbl to $\mathcal{P}_0$.

2. On input (x_i, x_v), $\mathcal{P}_0$ takes the entry $[i, v] \in \mathsf{Tbl}$ such that $i = x_i$ and computes $y = f(x_v, v)$.
3. $\mathcal{P}_0$ uses the com.commit interface of $\mathcal{F}_{\mathrm{NIC}}$ on input i to obtain a commitment $ccom_i$ with opening $copen_i$. Similarly, $\mathcal{P}_0$ obtains from $\mathcal{F}_{\mathrm{NIC}}$ a commitment $ccom_r$ to v with opening $copen_r$.
4. $\mathcal{P}_0$ uses the nhcd.prove interface to send $(ccom_i, i, copen_i, ccom_r, v, copen_r)$ to $\mathcal{F}_{\mathrm{NHCD}}$. $\mathcal{F}_{\mathrm{NHCD}}$ sends $ccom_i$ and $ccom_r$ to $\mathcal{P}_1$.
5. $\mathcal{P}_1$ uses the com.validate interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate the commitments $ccom_i$ and $ccom_r$. Then $\mathcal{P}_1$ stores $ccom_i$ and $ccom_r$ and sends a message to $\mathcal{P}_0$ to acknowledge the receipt of the commitments.
6. $\mathcal{P}_0$ parses the commitments $ccom_i$ and $ccom_r$ as $(ccom'_i, cparcom, \mathsf{COM.Verify})$ and $(ccom'_r, cparcom, \mathsf{COM.Verify})$. $\mathcal{P}_0$ sets the witness $wit \leftarrow (i, copen_i, v, copen_r, x_i, x_v)$ and the instance $ins \leftarrow (cparcom, ccom'_i, ccom'_r, y)$. $\mathcal{P}_0$ uses the zk.prove interface to send wit and ins to $\mathcal{F}_{\mathrm{ZK}}^{R'}$, where R' is

$$\begin{aligned} R' = \{(wit, ins) : i = x_i \ \wedge \ y = f(x_v, v) \wedge \\ 1 = \mathsf{COM.Verify}(cparcom, ccom_i, i, copen_i) \wedge \\ 1 = \mathsf{COM.Verify}(cparcom, ccom_r, v, copen_r) \ \} \end{aligned}$$

7. $\mathcal{P}_1$ receives ins from $\mathcal{F}_{\mathrm{ZK}}^{R'}$. $\mathcal{P}_1$ checks that the commitments in ins are equal to the stored commitments $ccom_i$ and $ccom_r$. If they are equal, the binding property guaranteed by $\mathcal{F}_{\mathrm{NIC}}$ ensures that $\mathcal{F}_{\mathrm{NHCD}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R'}$ received as input the same position i and value v.

In this protocol, we have divided the proof required by relation R into two parts. First, $\mathcal{F}_{\mathrm{NHCD}}$ proves that $[i, v] \in \mathsf{Tbl}$, and then $\mathcal{F}_{\mathrm{ZK}}^{R'}$ proves that $i = x_i$ and $y = f(x_v, v)$. $\mathcal{F}_{\mathrm{NIC}}$ ensures that both functionalities receive the same input $[i, v]$.

Naturally, it would be possible to describe a non-modular protocol for a proof for relation R, which would use a non-hiding VC scheme as building block. However, we think that a modular design has two advantages. First, a modular design 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.

The second advantage applies in particular to the design of ZK proofs. Usually, when a party in a protocol needs to compute a ZK proof, the relation R involves two types of statements: statements about where the witness is stored (e.g. $[i, v] \in \mathsf{Tbl}$), and statements about predicates or conditions that are satisfied by the witness ($e.g., i = x_i \ \wedge \ y = f(x_v, v)$). Very frequently, in existing protocols, both types of statements are intertwined in the same relation. Commitment schemes are implicitly used in many protocols as a zero-knowledge data structure to store witnesses and prove statements about them. By splitting up the two types of statements with a modular design, we facilitate the study in isolation of how to create efficient and secure zero-knowledge data structures. Namely, different constructions for $\mathcal{F}_{\mathrm{NHCD}}$ can easily be compared in terms of security and efficiency.

An important feature of $\mathcal{F}_{\text{NHCD}}$ is that it allows us to prove statements not only about the values stored but also about the positions where they are stored. Thanks to that, it supports more involved data structures. $\mathcal{F}_{\text{NHCD}}$ can be used to store tables where each entry consists of a tuple of values rather than a single value. To prover that a table entry is stored in the table, the prover proves statements about the positions where the values are stored in order to show that they belong to the same entry. Other data structures can also be considered, by requiring the prover to prove the necessary statements about the positions where values are stored.

6 Related Work

Accumulators. A cryptographic accumulator [4] allows one 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 $x \in X$.

Non-hiding VC schemes are similar to accumulator schemes that use a trusted setup and are non-hiding [10,3,30,8], i.e., A does not hide X. (Recently, hiding accumulators [14,17] have been proposed.) The instantiation of non-hiding VC schemes based on the DHE assumption resembles the accumulator scheme in [8]. The main difference between accumulators and non-hiding VC schemes is that, while accumulators allow us to commit to a set, non-hiding VC 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 where a value v is stored, which is needed for $\mathcal{F}_{\text{NHCD}}$.

Vector Commitments. VC schemes [26,13] can be non-hiding and hiding, and can be based on different assumptions such as CDH, RSA and DHE. We use a non-hiding VC scheme based on DHE, which, although based on an assumption that is less standard than CDH and RSA, has efficiency advantages. A mercurial VC scheme based on DHE was proposed in [26], and subsequently non-hiding and hiding DHE VC schemes were used in [24,19,23]. In our instantiation of Π_{NHCD}, we use a non-hiding VC scheme based on DHE that is extended with a ZK proof of knowledge of a witness w_i to prove that a value v is stored at position i. For this proof, a signature scheme is used along with the non-hiding VC scheme.

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 [21], 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 [25].

Zero-Knowledge Data Structures. Zero-Knowledge Sets (ZKS) [28] 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 [26] and an updatable ZKDB with short proofs is proposed in [13]. In [21], constructions for "nearly" ZKS and ZKDB, which do not hide the size of the set or database, are given. In [18], 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 [27,13].

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.

Zero-Knowledge Authenticated Data Structures. Authenticated data structures (ADS) are a three-party protocol between a trusted owner, a trusted client and a server [34]. The owner uploads data to the server. The server receives queries from the client and answers them. A secure ADS protects data authenticity when the server is adversarial.

Zero-knowledge ADS (ZKADS) provide privacy in addition to authenticity, i.e., the client does not learn anything about the data structure besides what can be inferred from the answers to the queries. The data owner is trusted. Most constructions for ZKADS are not updatable (see [18] and references therein). Recently, an updatable ZKADS for sets with proofs of membership [31], and an updatable ZKADS for lists, trees and partially-ordered sets of bounded dimension [16] were proposed.

Our construction differs from ADS and ZKADS in that the prover is not split up into a trusted data owner and a server. Unlike ZKADS, in our work the content of the database is revealed to the verifier. Similarly to the case of ZK data structures, existing ZKADS constructions are not oblivious, i.e., answers to queries are revealed in the clear, and their use as building block of a protocol is not described.

ZK proofs for large datasets. In most ZK proofs, the computation and communication cost grow linearly with the size of the witness, which is inadequate for proofs about datasets of large size N. However, there are techniques that attain costs sublinear in N. Probabilistically checkable proofs [22] achieve verification cost sublinear in N, but the cost for the prover is linear in N. In succinct non-interactive arguments of knowledge [15], verification cost is independent of N, but the cost for the prover is still linear in N. ZK proofs for oblivious RAM programs [29] 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 follows a similar approach to [29], i.e. at setup the table is committed, and then ZK proofs are computed. The cost at setup is linear in N for prover and verifier. However, the verification cost of a ZK proof is constant an independent of N. To compute a ZK proof, only the cost of computing a witness w_i is linear in N, but we note that w_i only needs to be computed once and can after that be reused and updated with cost independent of N. Therefore, computing a ZK proof has amortized constant cost, which makes our construction a better candidate for proofs about large datasets.

Acknowledgements. This research is supported by the Luxembourg National Research Fund (FNR) CORE Programme (Project code: C17/11650748).

References

1. Abe, M., Groth, J., Haralambiev, K., Ohkubo, M.: Optimal structure-preserving signatures in asymmetric bilinear groups. pp. 649–666 (2011)
2. Aiello, W., Ishai, Y., Reingold, O.: Priced oblivious transfer: How to sell digital goods. pp. 119–135 (2001)
3. 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. pp. 295–308 (2009)
4. Benaloh, J.C., de Mare, M.: One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). pp. 274–285 (1994)
5. Camenisch, J., Dubovitskaya, M., Neven, G.: Oblivious transfer with access control. pp. 131–140 (2009)
6. Camenisch, J., Dubovitskaya, M., Neven, G.: Unlinkable priced oblivious transfer with rechargeable wallets. pp. 66–81 (2010)
7. Camenisch, J., Dubovitskaya, M., Rial, A.: UC commitments for modular protocol design and applications to revocation and attribute tokens. pp. 208–239 (2016)
8. Camenisch, J., Kohlweiss, M., Soriente, C.: An accumulator based on bilinear maps and efficient revocation for anonymous credentials. pp. 481–500 (2009)
9. Camenisch, J., Krenn, S., Shoup, V.: A framework for practical universally composable zero-knowledge protocols. pp. 449–467 (2011)
10. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. pp. 61–76 (2002)
11. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. pp. 136–145 (2001)
12. Catalano, D., Fiore, D.: Vector commitments and their applications. In: PKC. pp. 55–72 (2013)
13. Catalano, D., Fiore, D.: Vector commitments and their applications. pp. 55–72 (2013)
14. Derler, D., Hanser, C., Slamanig, D.: Revisiting cryptographic accumulators, additional properties and relations to other primitives. pp. 127–144 (2015)
15. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. pp. 626–645 (2013)
16. Ghosh, E., Goodrich, M.T., Ohrimenko, O., Tamassia, R.: Verifiable zero-knowledge order queries and updates for fully dynamic lists and trees. pp. 216–236 (2016)
17. Ghosh, E., Ohrimenko, O., Papadopoulos, D., Tamassia, R., Triandopoulos, N.: Zero-knowledge accumulators and set algebra. pp. 67–100 (2016)
18. Ghosh, E., Ohrimenko, O., Tamassia, R.: Zero-knowledge authenticated order queries and order statistics on a list. pp. 149–171 (2015)

19. Izabachène, M., Libert, B., Vergnaud, D.: Block-wise P-signatures and non-interactive anonymous credentials with efficient attributes. pp. 431–450 (2011)
20. Jawurek, M., Johns, M., Kerschbaum, F.: Plug-in privacy for smart metering billing. pp. 192–210 (2011)
21. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. pp. 177–194 (2010)
22. Kilian, J.: A note on efficient zero-knowledge proofs and arguments (extended abstract). pp. 723–732 (1992)
23. Kohlweiss, M., Rial, A.: Optimally private access control. In: Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society. pp. 37–48. ACM (2013)
24. Libert, B., Peters, T., Yung, M.: Group signatures with almost-for-free revocation. pp. 571–589 (2012)
25. Libert, B., Ramanna, S.C., Yung, M.: Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. pp. 30:1–30:14 (2016)
26. Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. pp. 499–517 (2010)
27. Liskov, M.: Updatable zero-knowledge databases. pp. 174–198 (2005)
28. Micali, S., Rabin, M.O., Kilian, J.: Zero-knowledge sets. pp. 80–91 (2003)
29. Mohassel, P., Rosulek, M., Scafuro, A.: Sublinear zero-knowledge arguments for RAM programs. pp. 501–531 (2017)
30. Nguyen, L.: Accumulators from bilinear pairings and applications. pp. 275–292 (2005)
31. Pöhls, H.C., Samelin, K.: On updatable redactable signatures. pp. 457–475 (2014)
32. Rial, A., Danezis, G.: Privacy-preserving smart metering. In: Proceedings of the 10th annual ACM workshop on Privacy in the electronic society. pp. 49–60. ACM (2011)
33. Rial, A., Kohlweiss, M., Preneel, B.: Universally composable adaptive priced oblivious transfer. pp. 231–247 (2009)
34. Tamassia, R.: Authenticated data structures. In: European symposium on algorithms. pp. 2–5. Springer (2003)

A Security Analysis of Construction $\Pi_{\mathbf{NHCD}}$

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

$\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\mathrm{AUT}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R}$. 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.

In Section A.1, we analyze the security of construction Π_{NHCD} when $\mathcal{P}_0$ is corrupt. In Section A.2, we analyze the security of construction Π_{NHCD} when $\mathcal{P}_1$ is corrupt.

A.1 Security Analysis of Construction $\Pi_{\mathbf{NHCD}}$ when $\mathcal{P}_0$ is Corrupt

We first describe the simulator $\mathcal{S}$ for the case in which $\mathcal{P}_0$ is corrupt.

Initialization of $\mathcal{S}$. $\mathcal{S}$ does the following:

- $\mathcal{S}$ sets $c_1 \leftarrow 0$.
- $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ on input $(\mathsf{crs.get.ini}, sid)$. When $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$ sends $(\mathsf{crs.get.sim}, sid, qid, par)$, $\mathcal{S}$ sends $(\mathsf{crs.get.rep}, sid, qid)$ to $\mathcal{F}_{\text{CRS}}^{\mathsf{VC.Setup}}$, which sends a message $(\mathsf{crs.get.end}, sid, par)$.

Honest $\mathcal{P}_1$ starts setup. On input the message $(\mathsf{nhcd.setup.sim}, sid, \mathsf{Tbl})$ from functionality $\mathcal{F}_{\text{NHCD}}$, $\mathcal{S}$ sets $sid_{\text{AUT}} \leftarrow (\mathcal{P}_1, \mathcal{P}_0, sid')$ and runs a copy of $\mathcal{F}_{\text{AUT}}$ on input $(\mathsf{aut.send.ini}, sid_{\text{AUT}}, \mathsf{Tbl})$. When $\mathcal{F}_{\text{AUT}}$ sends $(\mathsf{aut.send.sim}, sid_{\text{AUT}}, qid, \mathsf{Tbl})$, $\mathcal{S}$ forwards it to $\mathcal{A}$.

$\mathcal{A}$ concludes setup. On input $(\mathsf{aut.send.rep}, sid_{\text{AUT}}, qid)$ from $\mathcal{A}$, $\mathcal{S}$ sends the message $(\mathsf{nhcd.setup.rep}, sid)$ to $\mathcal{F}_{\text{NHCD}}$. When $\mathcal{F}_{\text{NHCD}}$ sends $(\mathsf{nhcd.setup.end}, sid, \mathsf{Tbl})$, $\mathcal{S}$ runs the copy of $\mathcal{F}_{\text{AUT}}$ on input $(\mathsf{aut.send.rep}, sid_{\text{AUT}}, qid)$. When the copy of $\mathcal{F}_{\text{AUT}}$ sends $(\mathsf{aut.send.end}, sid_{\text{AUT}}, \mathsf{Tbl})$, $\mathcal{S}$ initializes a vector $\mathbf{x}$ such that $\mathbf{x}[i] = v$ for $i = 1$ to N, where $[i, v] \in \mathsf{Tbl}$. $\mathcal{S}$ runs $com \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$ and stores $(sid, par, com, \mathbf{x}, c_1)$. $\mathcal{S}$ sends $(\mathsf{aut.send.end}, sid, \mathsf{Tbl})$ to $\mathcal{A}$.

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

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

Honest $\mathcal{P}_1$ starts a writing operation. On input from functionality $\mathcal{F}_{\text{NHCD}}$ the message $(\mathsf{nhcd.write.sim}, sid, qid, i, v_w)$, $\mathcal{S}$ sets $sid_{\text{AUT}} \leftarrow (\mathcal{P}_1, \mathcal{P}_0, sid')$, takes the stored tuple $(sid, par, com, \mathbf{x}, c_1)$, sets $c_1' \leftarrow c_1 + 1$ and runs a copy of $\mathcal{F}_{\text{AUT}}$ on input $(\mathsf{aut.send.ini}, sid_{\text{AUT}}, \langle i, v_w, c_1' \rangle)$. When $\mathcal{F}_{\text{AUT}}$ sends $(\mathsf{aut.send.sim}, sid_{\text{AUT}}, qid', \langle i, v_w, c_1' \rangle)$, $\mathcal{S}$ stores $(sid, qid, i, v_w, c_1', qid')$ and forwards that message to $\mathcal{A}$.

$\mathcal{A}$ receives a writing operation. On input the message $(\mathsf{aut.send.rep}, sid_{\text{AUT}}, qid')$ from $\mathcal{A}$, if $(sid, qid, i, v_w, c_1', qid')$ is stored, $\mathcal{S}$ sends $(\mathsf{nhcd.write.rep}, sid, qid)$ to $\mathcal{F}_{\text{NHCD}}$. When $\mathcal{F}_{\text{NHCD}}$ sends $(\mathsf{nhcd.write.end}, sid, i, v_w)$, $\mathcal{S}$ runs the copy of $\mathcal{F}_{\text{AUT}}$ on input the message $(\mathsf{aut.send.rep}, sid_{\text{AUT}}, qid')$. When the copy of $\mathcal{F}_{\text{AUT}}$ sends $(\mathsf{aut.send.end}, sid, \langle i, v_w, c_1' \rangle)$, $\mathcal{S}$ takes the stored tuple $(sid, par, com, \mathbf{x}, c_1)$, sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $\mathbf{x}'[i] \leftarrow v_w$, computes $com' \leftarrow \mathsf{VC.ComUpd}(par, com, i, \mathbf{x}, \mathbf{x}')$, and replaces the stored tuple $(sid, par, com, \mathbf{x}, c_1)$ by $(sid, par, com', \mathbf{x}', c_1')$. $\mathcal{S}$ sends $(\mathsf{aut.send.end}, sid, \langle i, v_w, c_1' \rangle)$ to $\mathcal{A}$.

Honest $\mathcal{P}_1$ starts a proof. On input from $\mathcal{F}_{\text{NHCD}}$ the message$(\mathsf{nhcd.prove.sim}, sid, qid, ccom_i, ccom_r)$, $\mathcal{S}$ does the following:

- $\mathcal{S}$ retrieves the stored tuple $(sid, par, com, \mathbf{x}, c_1)$.
- $\mathcal{S}$ parses $ccom_i$ as $(ccom_i', cparcom_i, \mathsf{COM.Verify}_i)$ and $ccom_r$ as $(ccom_r', cparcom_r, \mathsf{COM.Verify}_r)$.
- $\mathcal{S}$ sets $ins \leftarrow (par, com, cparcom_i, ccom_i', ccom_r', c_1)$ and stores (sid, qid, ins).
- $\mathcal{S}$ sends $(\mathsf{zk.prove.sim}, sid, qid, ins)$ to $\mathcal{A}$.

$\mathcal{A}$ receives a 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 (sid, qid, ins) is not stored.

- $\mathcal{S}$ sends the message $(\mathsf{nhcd.prove.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{NHCD}}$ and receives the message $(\mathsf{nhcd.prove.end}, sid, ccom_i, ccom_r)$ from $\mathcal{F}_{\mathrm{NHCD}}$.
- $\mathcal{S}$ deletes (sid, qid, ins).
- $\mathcal{S}$ sends $(\mathsf{zk.prove.end}, sid, ins)$ to $\mathcal{A}$.

$\mathcal{A}$ starts a proof. On input $(\mathsf{zk.prove.ini}, sid, wit, ins)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{ZK}}^{R}$ on that input. When the copy of $\mathcal{F}_{\mathrm{ZK}}^{R}$ sends $(\mathsf{zk.prove.sim}, sid, qid, ins)$, $\mathcal{S}$ stores (sid, qid, wit, ins) and sends $(\mathsf{zk.prove.sim}, sid, qid, ins)$ to $\mathcal{A}$.

Honest $\mathcal{P}_1$ receives a proof. On input $(\mathsf{zk.prove.rep}, sid, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{ZK}}^{R}$ on that input. When the copy of $\mathcal{F}_{\mathrm{ZK}}^{R}$ outputs $(\mathsf{zk.prove.end}, sid, ins)$, $\mathcal{S}$ retrieves the stored tuple (sid, qid, wit, ins) and parses the instance ins as $(par', com', cparcom_i, ccom'_i, ccom'_r, c_0)$. $\mathcal{S}$ sets the tuples $ccom_i \leftarrow (ccom'_i, cparcom_i, \mathsf{COM.Verify}_i)$ and $ccom_r \leftarrow (ccom'_r, cparcom_i, \mathsf{COM.Verify}_i)$. $\mathcal{S}$ sends the message $(\mathsf{nhcd.prove.ini}, sid, ccom_i, i, copen_i, ccom_r, v_r, copen_r)$ to the functionality $\mathcal{F}_{\mathrm{NHCD}}$. When $\mathcal{F}_{\mathrm{NHCD}}$ sends $(\mathsf{nhcd.prove.sim}, sid, qid, ccom_i, ccom_r)$, $\mathcal{S}$ does the following:

- $\mathcal{S}$ retrieves the stored tuple $(sid, par, com, \mathbf{x}, c_1)$. If $c_0 \neq c_1$, or if $par' \neq par$, or if $com' \neq com$, $\mathcal{S}$ sends $\mathcal{F}_{\mathrm{NHCD}}$ a message that makes $\mathcal{F}_{\mathrm{NHCD}}$ abort.
- Else, $\mathcal{S}$ retrieves the stored tuple (sid, qid, wit, ins) and parses wit as $(w_i, i, copen_i, v_r, copen_r)$. $\mathcal{S}$ retrieves the stored tuple $(sid, par, com, \mathbf{x}, c_1)$. If $\mathbf{x}[i] \neq v_r$, $\mathcal{S}$ outputs failure.
- Else, $\mathcal{S}$ sends $(\mathsf{nhcd.prove.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{NHCD}}$.

Theorem 3. *When $\mathcal{P}_0$ is corrupt, Π_{NHCD} securely realizes $\mathcal{F}_{\mathrm{NHCD}}$ in the $(\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}, \mathcal{F}_{\mathrm{AUT}}, \mathcal{F}_{\mathrm{ZK}}^{R})$-hybrid model if the non-hiding VC scheme with algorithms $(\mathsf{VC.Setup}, \mathsf{VC.Commit}, \mathsf{VC.Prove}, \mathsf{VC.Verify}, \mathsf{VC.ComUpd}, \mathsf{VC.WitUpd})$ is binding.*

Proof of Theorem 3. 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 c_1 and the parameters par. **Game** 1 stores a tuple $(sid, par, com, \mathbf{x}, c_1)$ at setup and updates that tuple each time $\mathcal{P}_1$ sends a valid write operation. 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 $\mathcal{P}_1$ sends a proof, **Game** 2 creates the messages $(\mathsf{zk.prove.sim}, sid, qid, ins)$ and $(\mathsf{zk.prove.end}, sid, ins)$ without invoking $\mathcal{F}_{\mathrm{ZK}}^{R}$. These changes do not alter the view of the environment. Therefore, $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]| = 0$.

Game 3**:** **Game** 3 follows **Game** 2, except that, when the adversary sends a valid proof with witness wit and instance ins, **Game** 3 outputs failure if the values i and v_r in the witness are such that $\mathbf{x}[i] \neq v_r$, where $\mathbf{x}[i]$ is in the stored tuple $(sid, par, com, \mathbf{x}, c_1)$. The probability that **Game** 3 outputs failure is bound by the following claim.

Theorem 4. *Under the binding property of the non-hiding VC scheme, we have that* $|Pr\,[\textbf{\textit{Game}}\ 3] - Pr\,[\textbf{\textit{Game}}\ 2]| \leq Adv_{\mathcal{A}}^{\mathsf{bin-vc}}$.

Proof of Theorem 4. We construct an algorithm B that, given an adversary that makes **Game** 3 fail with non-negligible probability, breaks the binding property of the vector commitment scheme with non-negligible probability. B behaves as **Game** 3 with the following three modifications:

- When the challenger sends the parameters par, B stores par as common reference string in the copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$.
- When the adversary sends a valid proof with witness $wit = (w_i, i, copen_i, v_r, copen_r)$ and instance $ins = (par', com', cparcom_i, ccom'_i, ccom'_r, c_0)$ such that the values i and v_r in the witness fulfill $\mathbf{x}[i] \neq v_r$, where $\mathbf{x}[i]$ is in the stored tuple $(sid, par, com, \mathbf{x}, c_1)$, B runs $w'_i \leftarrow \mathsf{VC.Prove}(par, i, \mathbf{x})$ and sends $(com, i, v_r, \mathbf{x}[i], w_i, w'_i)$ to the challenger.

This concludes the proof of Theorem 4.

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

A.2 Security Analysis of Construction Π_{NHCD} when $\mathcal{P}_1$ is Corrupt

We describe the simulator $\mathcal{S}$ for the case in which $\mathcal{P}_1$ is corrupt.

Initialization of $\mathcal{S}$. $\mathcal{S}$ does the following:

- $\mathcal{S}$ sets $c_0 \leftarrow 0$.
- $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ on input $(\mathsf{crs.get.ini}, sid)$. When $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ sends the message $(\mathsf{crs.get.sim}, sid, qid, par)$, $\mathcal{S}$ sends the message $(\mathsf{crs.get.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$, which sends a message $(\mathsf{crs.get.end}, sid, par)$.

$\mathcal{A}$ **requests** par**.** $\mathcal{S}$ proceeds as in the case where $\mathcal{P}_0$ is corrupt.

$\mathcal{A}$ **receives** par**.** $\mathcal{S}$ proceeds as in the case where $\mathcal{P}_0$ is corrupt.

$\mathcal{A}$ **starts setup.** On input $(\mathsf{aut.send.ini}, sid_{\mathrm{AUT}}, \mathsf{Tbl})$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{AUT}}$ on input that message. When the copy of $\mathcal{F}_{\mathrm{AUT}}$ sends $(\mathsf{aut.send.sim}, sid_{\mathrm{AUT}}, qid, \mathsf{Tbl})$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest $\mathcal{P}_0$ **receives setup.** On input the message $(\mathsf{aut.send.rep}, sid_{\mathrm{AUT}}, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{AUT}}$ on input that message. When the copy of $\mathcal{F}_{\mathrm{AUT}}$ sends $(\mathsf{aut.send.end}, sid, \mathsf{Tbl})$, $\mathcal{S}$ sends an abortion message if $(sid, par, com, \mathbf{x}, c_0)$ is already stored. Otherwise $\mathcal{S}$ initializes a vector $\mathbf{x}$ such that $\mathbf{x}[i] = v$ for $i = 1$ to N, where $[i, v] \in \mathsf{Tbl}$, runs $com \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$, and stores $(sid, par, com, \mathbf{x}, c_0)$. $\mathcal{S}$ sends $(\mathsf{nhcd.setup.ini}, sid, \mathsf{Tbl})$ to $\mathcal{F}_{\mathrm{NHCD}}$. When $\mathcal{F}_{\mathrm{NHCD}}$ sends $(\mathsf{nhcd.setup.sim}, sid, \mathsf{Tbl})$, $\mathcal{S}$ sends $(\mathsf{nhcd.setup.rep}, sid)$ to $\mathcal{F}_{\mathrm{NHCD}}$.

$\mathcal{A}$ **starts writing operation.** On input $(\mathsf{aut.send.ini}, sid_{\mathrm{AUT}}, \langle i, v_w, c'_1 \rangle)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{AUT}}$ on input that message. When the copy of $\mathcal{F}_{\mathrm{AUT}}$ sends the message $(\mathsf{aut.send.sim}, sid_{\mathrm{AUT}}, qid, \langle i, v_w, c'_1 \rangle)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ receives a writing operation. On input $(\mathsf{aut.send.rep}, sid_{\mathrm{AUT}}, qid)$ from $\mathcal{A}$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{AUT}}$ on input that message. When the copy of $\mathcal{F}_{\mathrm{AUT}}$ sends the message $(\mathsf{aut.send.end}, sid, \langle i, v_w, c_1' \rangle)$, $\mathcal{S}$ sends an abortion message if $(sid, par, com, \mathbf{x}, c_0)$ is not stored, or if $c_1' \neq c_0 + 1$, or if $i \notin [1, N]$, or if $v_w \notin U_v$. Otherwise $\mathcal{S}$ sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $\mathbf{x}'[i] \leftarrow v_w$, computes $com' \leftarrow \mathsf{VC.ComUpd}(par, com, i, \mathbf{x}, \mathbf{x}')$, and replaces the stored tuple $(sid, par, com, \mathbf{x}, c_0)$ by $(sid, par, com', \mathbf{x}', c_1')$. $\mathcal{S}$ sends the message $(\mathsf{nhcd.write.ini}, sid, i, v_w)$ to $\mathcal{F}_{\mathrm{NHCD}}$. When $\mathcal{F}_{\mathrm{NHCD}}$ sends the message $(\mathsf{nhcd.write.sim}, sid, qid, i, v_w)$, $\mathcal{S}$ sends the message $(\mathsf{nhcd.write.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{NHCD}}$.

Honest $\mathcal{P}_0$ starts a proof. $\mathcal{S}$ proceeds as in the case where $\mathcal{P}_0$ is corrupt, with the roles of $\mathcal{P}_0$ and $\mathcal{P}_1$ exchanged.

$\mathcal{A}$ receives a proof. $\mathcal{S}$ proceeds as in the case where $\mathcal{P}_0$ is corrupt, with the roles of $\mathcal{P}_0$ and $\mathcal{P}_1$ exchanged.

$\mathcal{A}$ starts a proof. $\mathcal{S}$ proceeds as in the case where $\mathcal{P}_0$ is corrupt, with the roles of $\mathcal{P}_0$ and $\mathcal{P}_1$ exchanged.

Honest $\mathcal{P}_0$ receives a proof. $\mathcal{S}$ proceeds as in the case where $\mathcal{P}_0$ is corrupt, with the roles of $\mathcal{P}_0$ and $\mathcal{P}_1$ exchanged.

Theorem 5. *When $\mathcal{P}_1$ is corrupt, Π_{NHCD} securely realizes $\mathcal{F}_{\mathrm{NHCD}}$ in the $(\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}, \mathcal{F}_{\mathrm{AUT}}, \mathcal{F}_{\mathrm{ZK}}^{R})$-hybrid model if the non-hiding VC scheme with algorithms $(\mathsf{VC.Setup}, \mathsf{VC.Commit}, \mathsf{VC.Prove}, \mathsf{VC.Verify}, \mathsf{VC.ComUpd}, \mathsf{VC.WitUpd})$ is binding.*

The proof of Theorem 5 is very similar to the proof of Theorem 3 in Section A.1. When honest $\mathcal{P}_0$ computes a proof, $\mathcal{S}$ sets the messages $(\mathsf{zk.prove.sim}, sid, qid, ins)$ and $(\mathsf{zk.prove.end}, sid, ins)$ without invoking $\mathcal{F}_{\mathrm{ZK}}^{R}$, but these changes do not alter the view of the environment. When the adversary computes a proof, the binding property of the vector commitment scheme ensures that the position and the value committed in $ccom_i'$ and $ccom_r'$ are in the vector committed in the vector commitment com.

B Security for Non-Hiding Vector Commimtments

B.1 Security Definition for Non-Hiding Vector Commitments

Definition 2. *A non-hiding vector commitment scheme must be correct and binding [12].*

Correctness. Correctness requires that for $par \xleftarrow{\$} \mathsf{VC.Setup}(1^k, \ell)$, $\mathbf{x} \xleftarrow{\$} (\mathbf{x}[1], \ldots, \mathbf{x}[n]) \in \mathcal{M}^n$, $com \leftarrow \mathsf{VC.Commit}(par, \mathbf{x})$, $i \xleftarrow{\$} [1, n]$ 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 ℓ polynomial in k:

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

B.2 Security Analysis of the DHE Non-Hiding Vector Commitment Scheme

Theorem 6. *The non-hiding vector commitment scheme is correct and binding under the ℓ-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 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 (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 ℓ-DHE problem.