Concise UC Zero-Knowledge Proofs for Oblivious Updatable Databases

Jan Camenisch*, Maria Dubovitskaya*, Alfredo Rial†
*Dfinity, Zurich, Switzerland
Email: {jan,maria}@dfinity.org
†SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
Email: alfredo.rial@uni.lu

2021 IEEE 34th Computer Security Foundations Symposium (CSF) | 978-1-7281-7607-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CSF51468.2021.00008

***Abstract*—We propose an ideal functionality $\mathcal{F}_{\mathrm{CD}}$ and a construction Π_{CD} for oblivious and updatable committed databases. $\mathcal{F}_{\mathrm{CD}}$ allows a prover $\mathcal{P}$ to read, write, and update values in a database and to prove to a verifier $\mathcal{V}$ in zero-knowledge (ZK) that a value is read from or written into a certain position. The following properties must hold: (1) values stored in the database remain hidden from $\mathcal{V}$; (2) a value read from a certain position is equal to the value previously written into that position; (3) (obliviousness) both the value read or written and its position remain hidden from $\mathcal{V}$.**

Π_{CD} is based on vector commitments. After the initialization phase, the cost of read and write operations is independent of the database size, outperforming other techniques that achieve cost sublinear in the dataset size for prover and/or verifier. Therefore, our construction is especially appealing for large datasets.

In existing "commit-and-prove" two-party protocols, the task of maintaining a committed database between $\mathcal{P}$ and $\mathcal{V}$ and reading and writing values into it is not separated from the task of proving statements about the values read or written. $\mathcal{F}_{\mathrm{CD}}$ allows us to improve modularity in protocol design by separating those tasks. In comparison to simply using a commitment scheme to maintain a committed database, $\mathcal{F}_{\mathrm{CD}}$ allows $\mathcal{P}$ to hide efficiently the positions read or written from $\mathcal{V}$. Thanks to this property, we design protocols for e.g. privacy-preserving e-commerce and location-based services where $\mathcal{V}$ gathers aggregate statistics about the statements that $\mathcal{P}$ proves in ZK.

***Index Terms*—Vector commitments, ZK proofs of knowledge, universal composability**

I. Introduction

In protocols that use a commit-and-prove methodology (e.g. [1]), the prover $\mathcal{P}$ commits to her input and then proves in zero-knowledge (ZK) to a verifier $\mathcal{V}$ statements about the committed values. These steps are repeated and intertwined, i.e., commitments are updated, new ones formed, and additional proofs executed.

We regard commitments as a tool used to maintain a database of values between $\mathcal{P}$ and $\mathcal{V}$. When $\mathcal{P}$ commits to a value, the value is *written* into the database. When $\mathcal{P}$ proves a statement about a value committed previously, $\mathcal{P}$ *reads* a value from the database. Commitments guarantee the following properties: (1) values stored in the database are hidden from $\mathcal{V}$ (hiding property); (2) once a value is written into the database at a certain position (i.e, commitment), $\mathcal{P}$ cannot read a different value (binding property); (3) ZK proofs for reading or writing a value ensure that the value remains hidden from $\mathcal{V}$.

The use of commitments to maintain a database between $\mathcal{P}$ and $\mathcal{V}$ is not adequate in protocols where it is necessary to hide not only the value read or written into the database, but also the position where a value is stored. Our main example are protocols that allow $\mathcal{V}$ to gather statistics about what $\mathcal{P}$ proves in ZK, which are of interest for e.g. privacy-preserving e-commerce [2], billing [3] or location sharing services [4]. For instance, in [4], the positions represent locations, and the values are counters on the number of times $\mathcal{P}$ visited a location. When $\mathcal{P}$ visits a location, $\mathcal{P}$ updates the database to increment the counter for that location. To protect $\mathcal{P}$'s privacy, $\mathcal{V}$ must learn neither the location nor the counter value.

Unfortunately, if commitments were used to maintain the database, the cost of a ZK proof that meets those properties grows linearly with the database size. We would like that this cost be independent of the database size.

On the other hand, in commit-and-prove protocols, the task of maintaining a database between $\mathcal{P}$ and $\mathcal{V}$ and reading and writing values into it is not separated from the task of proving statements about the values read or written. I.e., typically $\mathcal{P}$ computes a ZK proof to prove a statement about a committed value. Such a proof involves both reading a value from the database and proving a statement about it.

We propose to separate the task of maintaining a database between $\mathcal{P}$ and $\mathcal{V}$ from the task of proving statements about the values read or written or about the positions where the values are stored. This will improve modularity in protocol design and will lead to simpler and more structured security proofs that are easier to verify. Additionally, it will enable the study of the task of maintaining a database between $\mathcal{P}$ and $\mathcal{V}$ in isolation, which allows an easy comparison of different techniques to maintain a database.

A. Our Contribution

UC functionality. In Section III, we define an ideal functionality $\mathcal{F}_{\mathrm{CD}}$ for an oblivious and updatable committed database (CD) in the universal composability (UC) framework. The database consists of a table Tbl_{cd} with N_{max} entries of the form $[i, v]$, where i is a position in $[1, N_{max}]$ and v is the value stored at that position. $\mathcal{F}_{\mathrm{CD}}$ ensures the following:

1) The values in the database remain hidden from $\mathcal{V}$.
2) $\mathcal{V}$ is guaranteed that a value read from Tbl_{cd} at position i is equal to the value previously written into i.

3) (Obliviousness) In read and write operations, both the value and the position read or written are hidden from $\mathcal{V}$.

The database is updatable because $\mathcal{F}_{\mathrm{CD}}$ allows $\mathcal{P}$ to overwrite values into the database at any time.

$\mathcal{F}_{\mathrm{CD}}$ allows $\mathcal{P}$ to read and write values into the database. To prove statements about a value, or about the position where the value is read or written, $\mathcal{P}$ must use an ideal functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$ for zero-knowledge parameterized by the appropriate statement R. This effectively separates the task of maintaining a database from the task of proving statements about the positions and values read or written. Consequently, we are able to design constructions for the committed database task in isolation and to compare them.

In a hybrid protocol that uses $\mathcal{F}_{\mathrm{CD}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R}$ as building blocks, we need to guarantee that the value and the position read or written into $\mathcal{F}_{\mathrm{CD}}$ are equal to the value and position sent to $\mathcal{F}_{\mathrm{ZK}}^{R}$. For this purpose, we use a method proposed in [5], which consists in sending committed inputs to $\mathcal{F}_{\mathrm{CD}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R}$ (see Section II).

Construction Π_{CD}. In Section V, we provide an efficient construction Π_{CD} for $\mathcal{F}_{\mathrm{CD}}$ that uses vector commitments (VC) [6], [7], which are suitable to implement a database Tbl_{cd} that consists of a one-dimensional array. A VC is a type of commitment that allows committing to a vector of values. The committer can open single positions of the committed vector to $\mathcal{V}$ with communication cost constant and independent of the vector size. VCs can also be updated. The computation cost of updating a commitment vc or a witness w to open a position grows linearly only with the number of updates and does not depend on the size of the committed vector.

Π_{CD} works as follows. At setup, $\mathcal{P}$ and $\mathcal{V}$ map the initial values of Tbl_{cd} to a vector of length N_{max} and compute a vector commitment vc to that vector. When $\mathcal{P}$ wants to read the value v at position i, $\mathcal{P}$ computes a VC witness w for position i and proves in ZK that v is committed at position i. To write a value v at a position i into the database, $\mathcal{P}$ updates vc to vc'. vc' commits to the same vector as vc, except that v is now at position i. Then $\mathcal{P}$ sends vc' to $\mathcal{V}$ and proves in ZK that vc' is an update of vc.

The communication cost of read and write operations is independent of the vector length N_{max}. The cost of computing vc and w grows linearly with N_{max}. However, we note that vc and w only need to be computed once. After that, they can be reused for multiple read operations and, when the database is updated, vc and w can be updated with cost that grows with the number of updates, but that is independent of N_{max}. Therefore Π_{CD} is suitable for large databases. In Section IX, we show that our construction improves in terms of efficiency over the existing ZK protocols for proving statements that involve large witnesses, such as ZK proofs for relations described as ORAM programs [8], [9]. In Section VI, we propose an efficient instantiation of Π_{CD} that uses a VC scheme secure under the DHE assumption.

Modular design and applications of $\mathcal{F}_{\mathrm{CD}}$. In Section VII, we describe how to design a hybrid protocol that uses $\mathcal{F}_{\mathrm{CD}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R}$ following the method in [5]. $\mathcal{F}_{\mathrm{CD}}$ is particularly useful for protocols where $\mathcal{P}$ needs to hide from $\mathcal{V}$ the positions read or written into the database. (Otherwise, a simple commitment scheme could be used to store the database.) In Section VIII, we describe some applications where this is the case. First, we describe how to use our committed database to compute OR proofs, i.e. ZK proofs of a disjunction of statements, with amortized cost independent of the size of the witness. Second, we show how $\mathcal{F}_{\mathrm{CD}}$ can be used to design protocols for privacy-preserving e-commerce and location-based services where service providers can gather aggregate statistics about users. In those protocols, $\mathcal{F}_{\mathrm{CD}}$ is used to store counters on the number of times a user buys a type of item or checks in at a certain location. When a user purchases an item or checks in at a location, the user computes a ZK proof that uses the item or the location as witness. Then the corresponding counter in $\mathcal{F}_{\mathrm{CD}}$ is incremented. At a later stage, the user can read counters from $\mathcal{F}_{\mathrm{CD}}$ to prove that they satisfy a statistic of interest to the service provider. We formalize this use of $\mathcal{F}_{\mathrm{CD}}$ as "zero-knowledge counting", i.e., counting the number of times a witness value is used by a prover in different ZK proofs. Beyond "zero-knowledge counting", we discuss how $\mathcal{F}_{\mathrm{CD}}$ is useful for gathering statistics that are more involved than counting, such as consumption statistics in privacy-preserving billing protocols [3].

B. Previous Work

The work in [5] increased the capabilities of modular protocol design in the UC framework by introducing a new functionality for non-interactive commitments $\mathcal{F}_{\mathrm{NIC}}$. $\mathcal{F}_{\mathrm{NIC}}$ is used to guarantee that two or more functionalities receive the same input. Having $\mathcal{F}_{\mathrm{NIC}}$ as a separate functionality simplifies the composition of building blocks and the security analysis of higher-level protocols.

Many protocols (e.g. [10], [11]) use commitment schemes to maintain a database between $\mathcal{P}$ and $\mathcal{V}$. However, commitments are not adequate to realize $\mathcal{F}_{\mathrm{CD}}$ because they do not allow $\mathcal{P}$ to hide efficiently the positions read from or written into the database. This is necessary in privacy-preserving protocols that allow $\mathcal{V}$ to gather statistics about what $\mathcal{P}$ proves in ZK, which are of interest in e-commerce [2], billing [3] or location sharing services [4].

The literature on commit-and-prove protocols [1] is very large. In the following, we restrict ourselves to discussing primitives that can be applied to construct a committed database that allows $\mathcal{P}$ to hide efficiently the positions read or written and to prove statements about those positions.

Vector commitments (VCs) [6], [7], [12], a type of functional commitment [13], are the mechanism implicitly used in [4] to maintain a database in such a way that $\mathcal{P}$ can hide efficiently the positions read or written and prove statements about them. Unlike commitments, VCs allow us to open a vector component with communication cost independent of the vector length.

ZK proofs of shuffles [14] can be used as a way of hiding positions read or written by shuffling data in the database at each read or write operation. A construction using commitments along with proofs of shuffles could realize the same functionality offered by VCs, but less efficiently.

Several data structures that offer ZK properties exist. Examples include ZK sets [6], [15]–[17], updatable ZK databases [7], [18], ZK lists [19], [20] and trees [20]. These constructions typically require the size of the data structure to be hidden, which is a property we usually do not need for the applications of committed databases (CD) that we consider in this paper. (Some works relax this property [12], [21], [22].) On the other hand, for CD it is fundamental that the data structure can be modified dynamically, which is provided only in [7], [18], [20]. We provide a detailed comparison between those constructions and our committed database in Section IX.

However, security for those ZK data structures has not been defined in a composable security framework. To design modularly protocols that need to maintain a database between $\mathcal{P}$ and $\mathcal{V}$, we could only use the ideal protocol for ZK proofs parameterized with different relations. These relations would describe both the way data is kept, read from and written into the database, as well as the statements proven about those data. Consequently, the modularity of the design is limited because complex ZK proofs for those relations cannot be decomposed into simpler ones and thus require a monolithic security analysis. The concept of CD will allow the security analysis of similar ZK data structures in a composable framework, which will facilitate the modular design and analysis of protocols that use them as a building block.

C. Outline of the Paper

We describe how to design UC protocols modularly in Section II. In Section III, we define our ideal functionality $\mathcal{F}_{\mathrm{CD}}$. In Section IV, we define the building blocks of our construction Π_{CD} and, in Section V, we describe Π_{CD}. We describe an efficient instantiation of our construction in Section VI. In Section VII, we show how to use our functionality as a building block in the modular design of cryptographic protocols. In Section VIII, we describe applications for $\mathcal{F}_{\mathrm{CD}}$. We describe related work in Section IX and conclude and hint some future work in Section X.

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

We summarize the UC framework in the full version [23]. An ideal functionality can be invoked by using one or more interfaces. In the notation in [5], the name of a message in an interface consists of three fields separated by dots, e.g., cd.setup.ini in $\mathcal{F}_{\mathrm{CD}}$ in Section III. The first field indicates the name of $\mathcal{F}_{\mathrm{CD}}$ and is the same for all interfaces. This field is useful for distinguishing between invocations of different functionalities in a hybrid protocol. The second field indicates the kind of action performed by $\mathcal{F}_{\mathrm{CD}}$ and is the same in all messages that $\mathcal{F}_{\mathrm{CD}}$ exchanges within the same interface. The third field distinguishes between the messages that belong to the same interface, and can take the following values. A message cd.setup.ini is the incoming message received by $\mathcal{F}_{\mathrm{CD}}$, i.e., the message through which the interface is invoked. cd.setup.end is the outgoing message sent by $\mathcal{F}_{\mathrm{CD}}$, i.e., the message that ends the execution of the interface. cd.setup.sim is used by $\mathcal{F}_{\mathrm{CD}}$ to send a message to the simulator $\mathcal{S}$, and cd.setup.rep is used to receive a message from $\mathcal{S}$.

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. 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 [5]. In [5], 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 m and obtain a commitment com and an opening $open$. A commitment com consists of $(com', parcom,$ COM.Verify), where com' is the commitment, $parcom$ are the public parameters, and COM.Verify is the verification algorithm.
3) Any party $\mathcal{P}_i$ uses the com.validate interface to send a commitment com in order to check that com contains the correct public parameters and verification algorithm.
4) Any party $\mathcal{P}_i$ uses the com.verify interface to send $(com,$ $m, open)$ in order to verify that com is a commitment to the message m with the opening $open$.

$\mathcal{F}_{\mathrm{NIC}}$ can be realized by a perfectly hiding commitment scheme, such as Pedersen commitments [5]. In [5], a method is described to use $\mathcal{F}_{\mathrm{NIC}}$ in order to ensure that a party sends the same input m to several ideal functionalities. For this purpose, the party first uses com.commit to get a commitment com to m with opening $open$. Then the party sends $(com, m, open)$ 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 com from each of the functionalities and use the com.validate interface to validate com. Then, if com received from all the functionalities is the same, the binding property provided by $\mathcal{F}_{\mathrm{NIC}}$ ensures that all the functionalities received the same input m. 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 $\mathcal{F}_{\mathrm{NIC}}$. The reason is that we need to ensure that the output of COM.Verify is indistinguishable from the output of the com.verify interface of $\mathcal{F}_{\mathrm{NIC}}$. Our functionality $\mathcal{F}_{\mathrm{CD}}$ receives committed inputs as described in [5]. We depict $\mathcal{F}_{\mathrm{NIC}}$ in the full version [23].

III. Ideal Functionality $\mathcal{F}_{\mathrm{CD}}$

Intuition. Our functionality $\mathcal{F}_{\mathrm{CD}}$ interacts with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ and has three interfaces: "setup", "read" and "write". $\mathcal{F}_{\mathrm{CD}}$ maintains a committed database where every entry is stored as a tuple [position, value]. The "setup" interface initializes the database to values known to both $\mathcal{P}$ and $\mathcal{V}$. The "write" interface allows $\mathcal{P}$ to update an entry of the committed database. $\mathcal{V}$ learns neither the position nor the value of the entry being written. The "read" interface allows $\mathcal{P}$ to prove to $\mathcal{V}$ knowledge of an entry in the database without revealing neither its position nor the value to $\mathcal{V}$. To ensure that both the prover and the verifier use the same version of the database, $\mathcal{F}_{\mathrm{CD}}$ maintains counters for the number of writing operations sent by $\mathcal{P}$ and the number of writing operations received by $\mathcal{V}$. The counters are checked for consistency for both read and write queries.

Both "read" and "write" interfaces require commitments to the position and to the value to be provided as input. This allows $\mathcal{F}_{\mathrm{CD}}$ to be used in conjunction with other functionalities (e.g. $\mathcal{F}_{\mathrm{ZK}}^{R}$ for ZK proofs of knowledge) in a modular way to build high-level protocols. That is, once the prover proves to the verifier that the commitment values commit to a database entry by using $\mathcal{F}_{\mathrm{CD}}$, those commitments can be used as input to other functionalities of the high-level (hybrid) protocol. For example, the commitment to the position and to the value can be input to $\mathcal{F}_{\mathrm{ZK}}^{R}$ to prove in ZK statements about the position and the value read from or written into the database. This allows us to prove different statements about the values from the database without revealing neither the value itself nor its position to the verifier. The binding property guaranteed by $\mathcal{F}_{\mathrm{NIC}}$ ensures that the committed values given as input to $\mathcal{F}_{\mathrm{CD}}$ and to $\mathcal{F}_{\mathrm{ZK}}^{R}$ are equal.

Notation. $\mathcal{F}_{\mathrm{CD}}$ is parameterized by a universe of state values U_v and by a state size N_{max}. $\mathcal{F}_{\mathrm{CD}}$ maintains a table Tbl_{cd} that stores the database. Tbl_{cd} contains N_{max} entries of the form $[i, v]$, where $i \in [1, N_{max}]$ is the position in the table and $v \in U_v$ is the value stored at that position. (N_{max} needs to be fixed so that $\mathcal{F}_{\mathrm{CD}}$ can be realized by a construction based on VCs, whose set up algorithm needs knowledge of N_{max}.) $\mathcal{F}_{\mathrm{CD}}$ maintains a counter cp for the number of writing operations sent by $\mathcal{P}$ and a counter cv for the number of writing operations received by $\mathcal{V}$. The interaction between the functionality $\mathcal{F}_{\mathrm{CD}}$, $\mathcal{P}$ and $\mathcal{V}$ takes place through the following interfaces:

- $\mathcal{V}$ uses the $\mathsf{cd.setup}$ interface to initialize Tbl_{cd}. $\mathcal{F}_{\mathrm{CD}}$ stores Tbl_{cd} and sends Tbl_{cd} to $\mathcal{P}$ and to the simulator $\mathcal{S}$.
- $\mathcal{P}$ uses $\mathsf{cd.read}$ to send a position i and a value v_r to $\mathcal{F}_{\mathrm{CD}}$, along with commitments and openings $(com_i, open_i)$ and $(com_r, open_r)$ to the position and value respectively. $\mathcal{F}_{\mathrm{CD}}$ verifies the commitments and checks that there is an entry $[i, v_r]$ in the table Tbl_{cd}. In that case, $\mathcal{F}_{\mathrm{CD}}$ sends com_i and com_r to $\mathcal{V}$. $\mathcal{S}$ also learns com_i and com_r.
- $\mathcal{P}$ uses $\mathsf{cd.write}$ to send a position i and a value v_w to $\mathcal{F}_{\mathrm{CD}}$, along with commitments and openings $(com_i, open_i)$ and $(com_w, open_w)$ to the position and value respectively. $\mathcal{F}_{\mathrm{CD}}$ verifies the commitments and then updates Tbl_{cd} to store v_w at position i. $\mathcal{F}_{\mathrm{CD}}$ sends com_i and com_w to $\mathcal{V}$. $\mathcal{S}$ also learns com_i and com_w.

The commitment parameters $parcom$ and the commitment verification algorithm $\mathsf{COM.Verify}$ are included in the commitment values (as in functionality $\mathcal{F}_{\mathrm{NIC}}$).

We describe $\mathcal{F}_{\mathrm{CD}}$ in Figure 1. We consider static corruptions, i.e. parties can only be corrupted by the adversary at the beginning of the protocol execution. In this description, we list all the abortion conditions and describe how $\mathcal{F}_{\mathrm{CD}}$ saves its state before querying the simulator $\mathcal{S}$ and recovers it after receiving a response from $\mathcal{S}$. These steps are often omitted in the description of ideal functionalities. In the "read" and "write" interfaces, $\mathcal{F}_{\mathrm{CD}}$ creates a query identifier qid to link the replies from $\mathcal{S}$ to the corresponding query to $\mathcal{S}$. In the "setup" interface, this is not needed because that interface can only be invoked once.

When invoked by $\mathcal{V}$ or $\mathcal{P}$, $\mathcal{F}_{\mathrm{CD}}$ first checks the correctness of the input and aborts if it does not belong to the correct domain. $\mathcal{F}_{\mathrm{CD}}$ also aborts if an interface is invoked at an incorrect moment in the protocol. For example, $\mathcal{P}$ cannot invoke $\mathsf{cd.read}$ before $\mathcal{V}$ invokes $\mathsf{cd.setup}$.

$\mathcal{F}_{\mathrm{CD}}$ also aborts if the commitment verification algorithms received as input are not ppt. This check is performed in the manner described in Section 7.2 paragraph "running arbitrary code" in [24].

Discussion of $\mathcal{F}_{\mathrm{CD}}$. The restriction that the identities $\mathcal{P}$ and $\mathcal{V}$ must be included in the session identifier $sid = (\mathcal{P}, \mathcal{V}, sid')$ guarantees that every verifier can create an instance of $\mathcal{F}_{\mathrm{CD}}$ with every prover. $\mathcal{F}_{\mathrm{CD}}$ implicitly checks that sid in a message equals the one received in the first invocation. We note that it is easy to modify $\mathcal{F}_{\mathrm{CD}}$ and our construction for $\mathcal{F}_{\mathrm{CD}}$ so that the setup phase is started by $\mathcal{P}$.

In the "read" interface, $\mathcal{F}_{\mathrm{CD}}$ aborts if $[i, v_r]$ received as input are not stored in Tbl_{cd}. This guarantees to $\mathcal{V}$ that the position and the value committed to in com_i and in com_r respectively correspond to an entry in Tbl_{cd}. After being triggered by $\mathcal{S}$, $\mathcal{F}_{\mathrm{CD}}$ aborts if the query identifier is not stored, or if the number of writing operations received by $\mathcal{V}$ does not equal the number of writing operations sent by $\mathcal{P}$ when the read operation was started. This guarantees that the table used by $\mathcal{P}$ when computing the read operation equals the table used by $\mathcal{V}$ to verify the read operation. A similar check is performed by $\mathcal{F}_{\mathrm{CD}}$ in the "write" interface.

$\mathcal{F}_{\mathrm{CD}}$ sends commitments to the position and to the value to $\mathcal{V}$. To hide the position and the value from $\mathcal{V}$, the hiding property of the commitment is required to hold. This is achieved by using $\mathcal{F}_{\mathrm{NIC}}$ to compute commitments.

IV. Building Blocks

Ideal Functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$. Our protocol uses the functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ for common reference string generation in [25]. $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ interacts with any parties $\mathcal{P}$ that obtain the common reference string, and consists of one interface $\mathsf{crs.get}$. A party $\mathcal{P}$ uses the $\mathsf{crs.get}$ 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 $\mathsf{CRS.Setup}$. The simulator $\mathcal{S}$ also receives crs. We depict $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ in the full version [23].

Ideal Functionality $\mathcal{F}_{\mathrm{AUT}}$. Our protocol uses the functionality $\mathcal{F}_{\mathrm{AUT}}$ for an authenticated channel in [25]. $\mathcal{F}_{\mathrm{AUT}}$ interacts with a sender $\mathcal{T}$ and a receiver $\mathcal{R}$, and consists of one interface $\mathsf{aut.send}$. $\mathcal{T}$ uses the $\mathsf{aut.send}$ 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.

Functionality $\mathcal{F}_{\mathrm{CD}}$ is parameterized by a universe of values U_v and by a maximum table size N_{max}. $\mathcal{F}_{\mathrm{CD}}$ interacts with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

1) On input $(\mathsf{cd.setup.ini}, sid, \mathsf{Tbl}_{cd})$ from $\mathcal{V}$:
   - Abort if $sid \notin (\mathcal{P}, \mathcal{V}, sid')$ or if (sid, Tbl_{cd}) is already stored.
   - Abort if Tbl_{cd} does not consist of entries of the form $[i, v]$, or if the number of entries in Tbl_{cd} is not N_{max}.
   - Abort if for $i = 1$ to N_{max}, $v \notin U_v$ for any entry $[i, v]$ in Tbl_{cd}.
   - Initialize a counter $cv \leftarrow 0$ for the verifier and store (sid, cv) and (sid, Tbl_{cd}).
   - Send $(\mathsf{cd.setup.sim}, sid, \mathsf{Tbl}_{cd})$ to $\mathcal{S}$.

S. On input $(\mathsf{cd.setup.rep}, sid)$ from $\mathcal{S}$:
   - Abort if (sid, Tbl_{cd}) is not stored, or if $(sid, \mathsf{Tbl}_{cd}, cp)$ is already stored.
   - Initialize a counter $cp \leftarrow 0$ for the prover and store $(sid, \mathsf{Tbl}_{cd}, cp)$.
   - Send $(\mathsf{cd.setup.end}, sid, \mathsf{Tbl}_{cd})$ to $\mathcal{P}$.

2) On input $(\mathsf{cd.read.ini}, sid, com_i, i, open_i, com_r, v_r, open_r)$ from $\mathcal{P}$:
   - Abort if $(sid, \mathsf{Tbl}_{cd}, cp)$ is not stored.
   - Abort if $i \notin [1, N_{max}]$, or if $v_r \notin U_v$, or if $[i, v_r]$ is not stored in Tbl_{cd}.
   - Parse the commitment com_i as $(com'_i, parcom_i, \mathsf{COM.Verify}_i)$.
   - Parse the commitment com_r as $(com'_r, parcom_r, \mathsf{COM.Verify}_r)$.
   - Abort if $\mathsf{COM.Verify}_i$ or $\mathsf{COM.Verify}_r$ are not ppt algorithms.
   - Abort if $1 \neq \mathsf{COM.Verify}_i(parcom_i, com'_i, i, open_i)$.
   - Abort if $1 \neq \mathsf{COM.Verify}_r(parcom_r, com'_r, v_r, open_r)$.
   - Create a fresh qid and store (qid, com_i, com_r, cp).
   - Send $(\mathsf{cd.read.sim}, sid, qid, com_i, com_r)$ to $\mathcal{S}$.

S. On input $(\mathsf{cd.read.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if (qid, com_i, com_r, cp') is not stored.
   - Abort if $cp' \neq cv$, where cv is stored in (sid, cv).
   - Delete the record (qid, com_i, com_r, cp').
   - Send $(\mathsf{cd.read.end}, sid, com_i, com_r)$ to $\mathcal{V}$.

3) On input $(\mathsf{cd.write.ini}, sid, com_i, i, open_i, com_w, v_w, open_w)$ from $\mathcal{P}$:
   - Abort if $(sid, \mathsf{Tbl}_{cd}, cp)$ is not stored.
   - Abort if $i \notin [1, N_{max}]$, or if $v_w \notin U_v$.
   - Parse the commitment com_i as $(com'_i, parcom_i, \mathsf{COM.Verify}_i)$.
   - Parse the commitment com_w as $(com'_w, parcom_w, \mathsf{COM.Verify}_w)$.
   - Abort if $\mathsf{COM.Verify}_i$ or $\mathsf{COM.Verify}_w$ are not ppt algorithms.
   - Abort if $1 \neq \mathsf{COM.Verify}_i(parcom_i, com'_i, i, open_i)$.
   - Abort if $1 \neq \mathsf{COM.Verify}_w(parcom_w, com'_w, v_w, open_w)$.
   - Increment the counter cp in $(sid, \mathsf{Tbl}_{cd}, cp)$ and store $[i, v_w]$ in Tbl_{cd}.
   - Create a fresh qid and store (qid, com_i, com_w, cp).
   - Send $(\mathsf{cd.write.sim}, sid, qid, com_i, com_w)$ to $\mathcal{S}$.

S. On input $(\mathsf{cd.write.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if (qid, com_i, com_w, cp') is not stored.
   - Abort if $cp' \neq cv + 1$, where cv is stored in (sid, cv).
   - Increment the counter cv in (sid, cv).
   - Delete the record (qid, com_i, com_w, cp').
   - Send $(\mathsf{cd.write.end}, sid, com_i, com_w)$ to $\mathcal{V}$.

Fig. 1. Functionality $\mathcal{F}_{\mathrm{CD}}$

The session identifier sid contains the identities of $\mathcal{T}$ and $\mathcal{R}$. We depict $\mathcal{F}_{\mathrm{AUT}}$ in the full version [23].

Ideal Functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$. Let R be a polynomial time computable binary relation. For tuples $(wit, ins) \in R$ we call wit the witness and ins the instance. Our protocol uses the ideal functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$ for zero-knowledge in [25]. $\mathcal{F}_{\mathrm{ZK}}^{R}$ 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 $\mathsf{zk.prove}$. $\mathcal{P}$ uses $\mathsf{zk.prove}$ 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. We depict $\mathcal{F}_{\mathrm{ZK}}^{R}$ in the full version [23].

Vector Commitments. Vector commitments (VC) [6], [7] allow us to commit to a vector of messages and to open the commitment to one of the messages in such a way that the size of the witness is independent of the length of the vector. A VC 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 commitment scheme par, which include a description of the message space $\mathcal{M}$ and a description of the randomness space $\mathcal{R}$.

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

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

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

$\mathsf{VC.ComUpd}(par, vc, j, x, r, x', r')$**.** On input a commitment vc with value x at position j and randomness r, output a commitment vc' with value x' at position j and randomness r'. The other positions remain unchanged.

$\mathsf{VC.VerComUpd}(par, vc, vc', w, j, x, r, x', r')$**.** On input commitments vc and vc', a witness w, a position j, the values x and x' and the randomness r and r', output 1 if w is a valid witness for x being at position j in the commitment vc and if vc' is an update of vc that replaces x by x' at position j and r by r'.

$\mathsf{VC.WitUpd}(par, w, i, j, x, r, x', r')$**.** On input a witness w for a position i valid for a commitment vc with value x at position j and randomness r, output a witness w' for position i valid for a commitment vc' with value x' at position j and randomness r'.

A VC scheme must be correct, hiding, and binding, as defined in the full version [23].

V. Construction Π_{CD} for a Committed Database

Our construction Π_{CD} in Figure 2 and Figure 3 uses a VC scheme. A vector commitment vc is used to store the table Tbl_{cd}. A position in the vector commitment acts as a position in Tbl_{cd}, and the value committed to in that position acts as the value stored in Tbl_{cd} in that position.

In the setup interface, $\mathcal{P}$ and $\mathcal{V}$ obtain the VC parameters par from the functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ for common reference string, which is parameterized by the setup algorithm $\mathsf{VC.Setup} = \mathsf{CRS.Setup}$ of the VC scheme. $\mathcal{V}$ receives as input a table Tbl_{cd} an computes a commitment vc to Tbl_{cd} with 0 randomness. $\mathcal{V}$ sends Tbl_{cd} to $\mathcal{P}$ by using the ideal functionality $\mathcal{F}_{\mathrm{AUT}}$ for an authenticated channel, and $\mathcal{P}$ also computes a commitment vc to Tbl_{cd} with 0 randomness.

In the read interface, $\mathcal{P}$ receives as input a position i and a value v_r, along with commitments and openings $(com_i, open_i)$ and $(com_r, open_r)$. $\mathcal{P}$ uses the ideal functionality $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ for ZK proofs of knowledge to prove to $\mathcal{V}$ that com_i and com_r commit to i and v_r such that v_r is the message committed at position i in the commitment vc. This proves that $[i, v_r] \in \mathsf{Tbl}_{cd}$.

In the write interface, $\mathcal{P}$ receives as input a position i and a value v_w, along with commitments and openings $(com_i, open_i)$ and $(com_w, open_w)$. $\mathcal{P}$ updates the commitment vc to a commitment vc' that commits to v_w at position i, while other positions remain unchanged. $\mathcal{P}$ uses the functionality $\mathcal{F}_{\mathrm{ZK}}^{R_w}$ to prove to $\mathcal{V}$ that com_i and com_w commit to i and v_w, and that vc' is an update of vc where v_w is committed in the position i. We note that vc' contains randomness chosen by $\mathcal{P}$ and thus, after the first execution of the write interface, the committed table will be hidden from $\mathcal{V}$.

We describe Π_{CD} in Figure 2 and Figure 3. For brevity, we omit some abortion conditions or the messages sent to the functionalities used as building blocks. The full description is in the full version [23].

Theorem 5.1: Π_{CD} securely realizes $\mathcal{F}_{\mathrm{CD}}$ in the $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_w}$-hybrid model if the VC scheme is hiding and binding.

When $\mathcal{P}$ is corrupt, the binding property of the VC 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. When $\mathcal{V}$ is corrupt, the hiding property of the VC scheme guarantees that the committed vector remains hidden from $\mathcal{V}$. We analyze in detail the security of Π_{CD} in the full version [23].

VI. Efficient Instantiation of Construction Π_{CD}

Our instantiation of Π_{CD} is based on a VC scheme secure under the DHE assumption and on ZK proofs of knowledge for the relations R_r and R_w for that VC scheme. To compute those proofs, the VC scheme is extended with a structure-preserving signature scheme. We use Pedersen commitments as the commitment scheme used to realize $\mathcal{F}_{\mathrm{NIC}}$.

In Section VI-A, we describe the building blocks of our instantiation. In Section VI-B, we describe the ZK proofs for relations R_r and R_w. We analyze the efficiency of our instantiation in Section VI-C.

A. Building blocks of Our Instantiation

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

Π_{CD} uses a VC scheme and the functionalities $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$, $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_r}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_w}$. The table size N_{max} is the maximum length of a committed vector, and the universe of values U_v is given by the message space of the VC scheme.

1) On input $(\mathsf{cd.setup.ini}, sid, \mathsf{Tbl}_{cd})$, $\mathcal{V}$ and $\mathcal{P}$ do the following:
   - $\mathcal{V}$ uses the crs.get interface of $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{VC.Setup}}$ to obtain the VC parameters par.
   - $\mathcal{V}$ initializes a counter $cv \leftarrow 0$ that counts the write operations received.
   - $\mathcal{V}$ stores Tbl_{cd} in a vector $\mathbf{x}$: for $i = 1$ to N_{max}, $\mathbf{x}[i] = v$, where $[i, v] \in \mathsf{Tbl}_{cd}$.
   - $\mathcal{V}$ commits to $\mathbf{x}$: set $r \leftarrow 0$ and run $vc \leftarrow \mathsf{VC.Commit}(par, \mathbf{x}, r)$.
   - $\mathcal{V}$ stores (sid, cv, par, vc).
   - $\mathcal{V}$ uses the aut.send interface of $\mathcal{F}_{\mathrm{AUT}}$ to send Tbl_{cd} to $\mathcal{P}$.
   - $\mathcal{P}$ follows the same steps as $\mathcal{V}$ to get par, set $\mathbf{x}$ and compute vc.
   - $\mathcal{P}$ initializes a counter $cp \leftarrow 0$ that counts write operations started.
   - $\mathcal{P}$ stores $(sid, cp, par, vc, \mathbf{x}, r)$.
   - $\mathcal{P}$ outputs $(\mathsf{cd.setup.end}, sid, \mathsf{Tbl}_{cd})$.
2) On input $(\mathsf{cd.read.ini}, sid, com_i, i, open_i, com_r, v_r, open_r)$, $\mathcal{P}$ and $\mathcal{V}$ do:
   - $\mathcal{P}$ parses com_i as $(com_i', parcom_i, \mathsf{COM.Verify}_i)$.
   - $\mathcal{P}$ parses com_r as $(com_r', parcom_r, \mathsf{COM.Verify}_r)$.
   - $\mathcal{P}$ takes the stored tuple $(sid, cp, par, vc, \mathbf{x}, r)$.
   - If (sid, i, w) is not stored, $\mathcal{P}$ computes a VC witness w for position i: run $w \leftarrow \mathsf{VC.Wit}(par, i, \mathbf{x}, r)$ and store (sid, i, w).
   - $\mathcal{P}$ sets $wit_r \leftarrow (w, i, open_i, v_r, open_r)$.
   - $\mathcal{P}$ sets $ins_r \leftarrow (par, vc, parcom_i, com_i', parcom_r, com_r', cp)$.
   - $\mathcal{P}$ uses the zk.prove interface to send wit_r and ins_r to $\mathcal{F}_{\mathrm{ZK}}^{R_r}$, where R_r is

$$\begin{aligned} R_r = \{ & (wit_r, ins_r) : \\ & 1 = \mathsf{COM.Verify}_i(parcom_i, com_i', i, open_i) \wedge && (1) \\ & 1 = \mathsf{COM.Verify}_r(parcom_r, com_r', v_r, open_r) \wedge && (2) \\ & 1 = \mathsf{VC.Verify}(par, vc, v_r, i, w) \} && (3) \end{aligned}$$

   In equation 1, $\mathcal{P}$ proves that com_i' is a commitment to i with opening $open_i$. Similarly, in equation 2, $\mathcal{P}$ proves that com_r' is a commitment to v_r with opening $open_r$. In equation 3, $\mathcal{P}$ proves that v_r is stored in the position i of the vector commitment vc.
   - $\mathcal{V}$ receives $ins_r = (par', vc', parcom_i, com_i', parcom_r, com_r', cp)$.
   - $\mathcal{V}$ takes the stored tuple (sid, cv, par, vc).
   - $\mathcal{V}$ aborts if $cp \neq cv$, or if $par' \neq par$, or if $vc' \neq vc$.
   - $\mathcal{V}$ sets $com_i \leftarrow (com_i', parcom_i, \mathsf{COM.Verify}_i)$.
   - $\mathcal{V}$ sets $com_r \leftarrow (com_r', parcom_r, \mathsf{COM.Verify}_r)$.
   - $\mathcal{V}$ outputs $(\mathsf{cd.read.end}, sid, com_i, com_r)$.

Fig. 2. Construction Π_{CD}: interfaces cd.setup and cd.read

A VC Scheme From the DHE Assumption. We show a VC scheme that is secure under the Diffie-Hellman Exponent (DHE) assumption [6]. Let $k \in \mathbb{N}$ denote the security parameter and let $\epsilon(k)$ denote a negligible function. We recall the ℓ-DHE assumption.

Definition 6.1: [ℓ-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 $(g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})$ such that $g_i = g^{(\alpha^i)}$ and $\tilde{g}_i = \tilde{g}^{(\alpha^i)}$, for any p.p.t. adversary $\mathcal{A}$, $\Pr[g^{(\alpha^{\ell+1})} \leftarrow \mathcal{A}(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell})] \leq \epsilon(k)$.

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

$\mathsf{VC.Commit}(par, \mathbf{x}, r)$**.** Let $|\mathbf{x}| = n \leq \ell$. Output

$$vc = g^r \cdot \prod_{j=1}^{n} g_{\ell+1-j}^{\mathbf{x}[j]} = g^r \cdot g_\ell^{\mathbf{x}[1]} \cdots g_{\ell+1-n}^{\mathbf{x}[n]} \ .$$

$\mathsf{VC.Wit}(par, i, \mathbf{x}, r)$**.** Let $|\mathbf{x}| = n \leq \ell$. Output

$$w = g_i^r \cdot \prod_{j=1, j\neq i}^{n} g_{\ell+1-j+i}^{\mathbf{x}[j]} \ .$$

3. On input $(\mathsf{cd.write.ini}, sid, com_i, i, open_i, com_w, v_w, open_w)$, $\mathcal{P}$ and $\mathcal{V}$ do:
   - $\mathcal{P}$ takes the stored tuple $(sid, cp, par, vc, \mathbf{x}, r)$.
   - $\mathcal{P}$ parses com_i as $(com'_i, parcom_i, \mathsf{COM.Verify}_i)$.
   - $\mathcal{P}$ parses com_w as $(com'_w, parcom_w, \mathsf{COM.Verify}_w)$.
   - If (sid, i, w) is not stored, $\mathcal{P}$ computes a VC witness w for position i: run $w \leftarrow \mathsf{VC.Wit}(par, i, \mathbf{x}, r)$ and store (sid, i, w).
   - $\mathcal{P}$ updates the vector commitment vc to vc': pick random $r' \leftarrow \mathcal{R}$ and run $vc' \leftarrow \mathsf{VC.ComUpd}(par, vc, i, v_r, r, v_w, r')$, where $v_r \leftarrow \mathbf{x}[i]$.
   - $\mathcal{P}$ increments the counter of write operations started $cp' \leftarrow cp + 1$.
   - $\mathcal{P}$ sets $wit_w \leftarrow (w, i, open_i, v_r, v_w, open_w, r, r')$.
   - $\mathcal{P}$ sets $ins_w \leftarrow (par, vc, vc', parcom_i, com'_i, parcom_w, com'_w, cp')$.
   - $\mathcal{P}$ updates the vector $\mathbf{x}$ to $\mathbf{x}'$: set $\mathbf{x}' \leftarrow \mathbf{x}$ and $\mathbf{x}'[i] \leftarrow v_w$.
   - $\mathcal{P}$ updates the stored tuple to $(sid, cp', par, vc', \mathbf{x}', r')$.
   - $\mathcal{P}$ updates the stored witnesses: for $j = 1$ to N_{max}, if (sid, j, w) is stored, run $w' \leftarrow \mathsf{VC.WitUpd}(par, w, j, i, x, r, x', r')$ and update to (sid, j, w').
   - $\mathcal{P}$ uses the zk.prove interface to send wit_w and ins_w to $\mathcal{F}_{\mathrm{ZK}}^{R_w}$, where R_w is

$$\begin{aligned} R_w =&\{(wit_w, ins_w) : \\ &1 = \mathsf{COM.Verify}_i(parcom_i, com'_i, i, open_i) \wedge && (4)\\ &1 = \mathsf{COM.Verify}_w(parcom_w, com'_w, v_w, open_w) \wedge && (5)\\ &1 = \mathsf{VC.VerComUpd}(par, vc, vc', w, i, v_r, r, v_w, r')\} && (6)\end{aligned}$$

   In equation 4 and equation 5, the prover proves that com'_i and com'_w are commitments to i and v_w respectively. In equation 6, the prover proves that v_r is stored in the position i in the vector commitment vc, and that vc' is a vector commitment that stores the same values as vc, except that it stores v_w in the position i and that its random value is r' instead of r.
   - $\mathcal{V}$ gets $ins_w = (\hat{par}, \hat{vc}, vc', parcom_i, com'_i, parcom_w, com'_w, cp)$.
   - $\mathcal{V}$ takes the stored tuple (sid, cv, par, vc).
   - $\mathcal{V}$ aborts if $cp \neq cv + 1$, or if $\hat{par} \neq par$, or if $\hat{vc} \neq vc$.
   - $\mathcal{V}$ sets $cv' \leftarrow cv + 1$ and updates the stored tuple to (sid, cv', par, vc').
   - $\mathcal{V}$ sets $com_i \leftarrow (com'_i, parcom_i, \mathsf{COM.Verify}_i)$.
   - $\mathcal{V}$ sets $com_w \leftarrow (com'_w, parcom_w, \mathsf{COM.Verify}_w)$.
   - $\mathcal{V}$ outputs $(\mathsf{cd.write.end}, sid, com_i, com_w)$.

Fig. 3. Construction Π_{CD}: interface cd.write

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

$\mathsf{VC.ComUpd}(par, vc, j, x, r, x', r')$. Output the commitment

$$vc' = vc \cdot \frac{g^{r'} \cdot g_{\ell+1-j}^{x'}}{g^r \cdot g_{\ell+1-j}^{x}} = vc \cdot g^{r'-r} \cdot g_{\ell+1-j}^{x'-x} \ .$$

$\mathsf{VC.VerComUpd}(par, vc, vc', w, j, x, r, x', r')$. Compute $v \leftarrow \mathsf{VC.Verify}(par, vc, x, j, w)$. If $v \leftarrow 0$, output v, else run $\bar{vc} \leftarrow \mathsf{VC.ComUpd}(par, vc, j, x, r, x', r')$. If $\bar{vc} = vc'$, output 1, else output 0.

$\mathsf{VC.WitUpd}(par, w, i, j, x, r, x', r')$. If $i = j$, output w. Otherwise output the witness

$$w' = w \cdot \frac{g_i^{r'} \cdot g_{\ell+1-j+i}^{x'}}{g_i^{r} \cdot g_{\ell+1-j+i}^{x}} = w \cdot g_i^{r'-r} \cdot g_{\ell+1-j+i}^{x'-x} \ .$$

Theorem 6.2: This VC scheme is correct, hiding, and binding under the ℓ-DHE assumption.

We prove this theorem in the full version [23].

Notation for ZK Proofs of Knowledge. We use classical results for efficient ZK proofs of knowledge for discrete logarithm relations. In the notation of [26], a UC ZK protocol proving knowledge of exponents $(w_1, \ldots, w_n)$ satisfying the formula $\phi(w_1, \ldots, w_n)$ is described as

$$\text{⅄}\, w_1, \ldots, w_n : \phi(w_1, \ldots, w_n) \qquad (7)$$

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

A proof system for (7) can be transformed into a proof system for the following more expressive statements about secret exponents $sexps$ and secret bases $sbases$:

$$\text{⅄}\, sexps, sbases : \phi(sexps, bases \cup sbases) \qquad (8)$$

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

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

Structure-Preserving Signatures. A signature scheme consists of the algorithms KeyGen, Sign and VfSig. Algorithm $\mathsf{KeyGen}(1^k)$ outputs a secret key sk and a public key pk, which include a description of the message space $\mathcal{M}$. $\mathsf{Sign}(sk, m)$ outputs a signature s on the message $m \in \mathcal{M}$. $\mathsf{VfSig}(pk, s, m)$ outputs 1 if s is a valid signature on m and 0 otherwise. This definition can be extended to blocks of messages $\bar{m} = (m_1, \ldots, m_n)$. In this case, $\mathsf{KeyGen}(1^k, n)$ receives the maximum number of messages as input. A signature scheme must fulfill the correctness and existential unforgeability properties [27].

In structure-preserving signatures (SPS), the public key, the messages, and the signatures are group elements in $\mathbb{G}$ and $\tilde{\mathbb{G}}$, and verification must consist purely in the checking of pairing product equations. We employ SPS to sign group elements, while still supporting efficient ZK proofs of signature possession. For concreteness, we recall the scheme proposed by [28] for the case in which a elements in $\mathbb{G}$ and b elements in $\tilde{\mathbb{G}}$ are signed. This scheme is strongly existentially unforgeable against adaptive chosen message attacks in the generic group model [28].

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

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

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

Commitment Schemes. A commitment scheme consists of algorithms CSetup, Com and VfCom. The algorithm $\mathsf{CSetup}(1^k)$ generates the parameters of the commitment scheme par_c, which include a description of the message space $\mathcal{M}$. $\mathsf{Com}(par_c, x)$ outputs a commitment com to $x \in \mathcal{M}$ and some auxiliary information $open$. The verification algorithm $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 if com is a commitment to $x \in \mathcal{M}$ with some auxiliary information $open$ or 0 if that is not the case. A commitment scheme should fulfill the correctness, trapdoor and binding properties as described in [5].

The Pedersen commitment scheme [29] fulfills the correctness, trapdoor and binding properties. In [5], it is proven that any commitment scheme that fulfills those properties fulfills the ideal functionality $\mathcal{F}_{\mathrm{NIC}}$. The Pedersen commitment scheme works as follows. $\mathsf{CSetup}(1^k)$ takes a group $\mathbb{G}$ of prime order p with generator g, picks random α, computes $h \leftarrow g^{\alpha}$ and sets the parameters $par_c \leftarrow (\mathbb{G}, g, h)$, which include a description of the message space $\mathcal{M} \leftarrow \mathbb{Z}_p$. $\mathsf{Com}(par_c, x)$ picks random $open \leftarrow \mathbb{Z}_p$ and outputs a commitment $com \leftarrow g^x h^{open}$ to $x \in \mathcal{M}$ and some auxiliary information $open$. The algorithm $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 if $com = g^x h^{open}$.

B. ZK Proofs for R_r and R_w

In Π_{CD}, we need to compute two ZK proofs of knowledge: a proof for a read operation and a proof for a write operation. We describe these proofs for the ℓ-DHE VC scheme.

For a read operation, we need a ZK proof of knowledge of a witness w that a value v_r was committed to in a vector commitment vc at position i. I.e., we need to compute a proof

$$\begin{aligned} &\text{⋊}\, i, open_i, v_r, open_r, w : \\ &\quad 1 = \mathsf{COM.Verify}_i(parcom_i, com'_i, i, open_i) \wedge \\ &\quad 1 = \mathsf{COM.Verify}_r(parcom_r, com'_r, v_r, open_r) \wedge \\ &\quad 1 = \mathsf{VC.Verify}(par, vc, v_r, i, w).\} \end{aligned}$$

This proof involves proving knowledge of a position i, a value v_r and a witness w such that the verification equation $e(vc, \tilde{g}_i) = e(w, g) e(g_1, \tilde{g}_\ell)^{v_r}$ holds. Additionally, it involves proving that the position i is committed in a commitment com'_i with opening $open_i$, and the value v_r is committed in a commitment com'_r with opening $open_r$.

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 VC scheme with structure preserving signatures (SPS) that bind i with $\tilde{g}_i$. We also need to bind i with $g_{\ell+1-i}$ for a ZK proof of a write operation. (In practice, i does not necessarily need to belong to $[1, N_{max}]$, i.e., other unique identifiers in $\mathbb{Z}_p$ can be assigned to the positions.) To this end, the setup algorithm of the VC scheme is extended 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 $\tilde{g}_i = \tilde{g}^{(\alpha^i)}$. Compute $(sk, pk) \leftarrow \mathsf{KeyGen}(grp, 1, 2)$. For $i \in [1, \ell]$, run $s_i \leftarrow \mathsf{Sign}(sk, \langle g_{\ell+1-i}, \tilde{g}_i, \tilde{g}^i \rangle)$. Compute additional bases $h \leftarrow \mathbb{G}$ and $\tilde{h} \leftarrow \tilde{\mathbb{G}}$. Output the parameters $par = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_\ell, \tilde{g}_\ell, g_{\ell+2}, \ldots, g_{2\ell}, pk, s_1, \ldots, s_\ell, h, \tilde{h}, \mathcal{M} = \mathbb{Z}_p, \mathcal{R} = \mathbb{Z}_p)$.

Therefore, our instantiation of Π_{CD} is secure under the unforgeability property of the SPS scheme in addition to the ℓ-DHE assumption. Let (g, h) be the parameters of the Pedersen commitment scheme for both $parcom_i$ and $parcom_r$. Let (U_1, U_2, V, W_1, Z) be the public key of the signature scheme.

Let (R, S, T) be a signature on $(g_{\ell+1-i}, \tilde{g}_i, \tilde{g}^i)$. We describe the proof as follows.

$$\begin{aligned}
&\text{K}\, i, open_i, v_r, open_r, \tilde{g}_i, g_{\ell+1-i}, w, R, S, T : \\
&\quad com'_i = g^i h^{open_i} \wedge && (9)\\
&\quad com'_r = g^{v_r} h^{open_r} \wedge && (10)\\
&\quad e(R, V) e(S, \tilde{g}) e(g_{\ell+1-i}, W_1) e(g, Z)^{-1} = 1 \wedge && (11)\\
&\quad e(R, T) e(U_1, \tilde{g}_i) e(U_2, \tilde{g})^i e(g, \tilde{g})^{-1} = 1 \wedge && (12)\\
&\quad e(vc, \tilde{g}_i)^{-1} e(w, \tilde{g}) e(g_1, \tilde{g}_\ell)^{v_r} = 1 && (13)
\end{aligned}$$

Equation 9 and Equation 10 prove knowledge of the openings of the Pedersen commitments com'_i and com'_r. Equation 11 and Equation 12 prove knowledge of a signature (R, S, T) on a message $(g_{\ell+1-i}, \tilde{g}_i, \tilde{g}^i)$. Equation 13 proves that the value v_r in com'_r is equal to the value committed in the position i of the vector commitment vc. To prove knowledge of the secret bases $(\tilde{g}_i, g_{\ell+1-i}, w, R, S, T)$, the equations need to be modified as described in Section VI-A.

For a write operation, we need a ZK proof of knowledge that a vector commitment vc' is an update of a vector commitment vc that contains randomness r' instead of r and value v_w instead of v_r at a position i.

$$\begin{aligned}
&\text{K}\, i, open_i, v_w, open_w, v_r, r, r', w : \\
&\quad 1 = \mathsf{VfCom}(parcom_i, com'_i, i, open_i) \wedge \\
&\quad 1 = \mathsf{VfCom}(parcom_w, com'_w, v_w, open_w) \wedge \\
&\quad 1 = \mathsf{VC.VerComUpd}(par, vc, vc', w, i, v_r, r, v_w, r')
\end{aligned}$$

Let (g, h) be the parameters of the Pedersen commitment scheme for both $parcom_i$ and $parcom_w$. Let (U_1, U_2, V, W_1, Z) be the public key of the signature scheme. Let (R, S, T) be a signature on $(g_{\ell+1-i}, \tilde{g}_i, \tilde{g}^i)$. We describe the proof as follows.

$$\begin{aligned}
&\text{K}\, i, open_i, v_w, open_w, v_r, r' - r, \tilde{g}_i, g_{\ell+1-i}, w, R, S, T : \\
&\quad com'_i = g^i h^{open_i} \wedge && (14)\\
&\quad com'_w = g^{v_w} h^{open_w} \wedge && (15)\\
&\quad e(R, V) e(S, \tilde{g}) e(g_{\ell+1-i}, W_1) e(g, Z)^{-1} = 1 \wedge && (16)\\
&\quad e(R, T) e(U_1, \tilde{g}_i) e(U_2, \tilde{g})^i e(g, \tilde{g})^{-1} = 1 \wedge && (17)\\
&\quad e(vc, \tilde{g}_i)^{-1} e(w, \tilde{g}) e(g_1, \tilde{g}_\ell)^{v_r} = 1 \wedge && (18)\\
&\quad vc'/vc = g^{r'-r} \cdot g_{\ell+1-i}^{v_w - v_r} && (19)
\end{aligned}$$

Equation 14 and Equation 15 prove knowledge of the openings of the Pedersen commitments com'_i and com'_w. Equation 16 and Equation 17 prove knowledge of a signature (R, S, T) on a message $(g_{\ell+1-i}, \tilde{g}_i, \tilde{g}^i)$. Equation 18 proves that v_r is the value committed in the position i of the vector commitment vc. Equation 19 proves that vc' is an update of vc that contains v_w instead of v_r at position i. To prove knowledge of the secret bases $(\tilde{g}_i, g_{\ell+1-i}, w, R, S, T)$, the equations need to be modified as described in Section VI-A.

C. *Efficiency Analysis*

We analyze the storage, communication, and computation costs of our instantiation of Π_{CD}. We summarize them in Table I.

Storage Cost. $\mathcal{P}$ stores the common reference string, which consists of the parameters of the VC scheme. Its size grows linearly with the maximum size N_{max} of the database. Throughout the protocol execution, in addition to the common reference string, $\mathcal{P}$ also stores the last update of the vector commitment, the committed vector, the randomness used to compute that commitment, and the witnesses computed so far. In conclusion, the storage cost for $\mathcal{P}$ grows linearly with N_{max}.

$\mathcal{V}$ also stores the common reference string. Although in Π_{CD}, $\mathcal{V}$ stores the whole common reference string, we observe that in our instantiation of Π_{CD} this is not necessary. In practice, $\mathcal{V}$ only needs to store $(g, \tilde{g}, h, \tilde{h}, g_1, \tilde{g}_\ell)$ and the public key (U_1, U_2, V, W_1, Z) of the signature scheme. These values suffice to verify the ZK proofs of knowledge for read and write operations. In addition to the common reference string, $\mathcal{V}$ only needs to store the last update of the vector commitment. Therefore, the storage cost for $\mathcal{V}$ is constant and independent of N_{max}.

Communication Cost. In the setup phase, the communication grows with N_{max} because $\mathcal{V}$ sends Tbl_{cd} to $\mathcal{P}$. This step can be avoided in practice if Tbl_{cd} is initialized to default values known by $\mathcal{P}$ and $\mathcal{V}$. In a read operation, $\mathcal{P}$ and $\mathcal{V}$ run a ZK proof of knowledge for R_r. The size of the witness and of the instance is constant and independent of N_{max}. Therefore, the communication cost of the proof is independent of N_{max}. Similarly, in a write operation, $\mathcal{P}$ sends a ZK proof of knowledge for R_w to $\mathcal{V}$. The size of the witness and of the instance is also independent of N_{max}, and thus the communication cost of the proof is independent of N_{max}.

Computation Cost. To compute a proof for a read operation, $\mathcal{P}$ first needs to compute a VC witness for the position to be read. If a witness w was not computed before, the computation cost of this step grows linearly with N_{max}. When a witness has already been computed, the computation cost is independent of N_{max}. Concretely, if the database was not updated since the moment w was computed, the same witness w can be reused. If the database was updated, w can be updated with a computation cost linear in the number of updates, but independent of N_{max}.

The remaining steps in the computation of the proof for R_r are also independent of N_{max}. Therefore, after computing a witness w for a position, the remaining proofs can be computed with cost independent of N_{max}.

A proof for a write operation also requires the computation of a witness w for the position to be written. The same optimization used for a read proof can be applied here, i.e., if a witness for that position has already been computed, a new witness can be computed with cost independent of N_{max}. In addition to the witness, $\mathcal{P}$ also needs to update the vector commitment. The computation cost of a vector commitment update is also independent of N_{max}. The remaining steps to compute the proof for R_w are also independent of N_{max}.

Worst/Average/Best Case Computation Cost. The computation cost depends on the number of positions that $\mathcal{P}$ needs to read or write throughout the protocol execution, as well as on the number of values in the database that are 0.

In the worst case, $\mathcal{P}$ needs to read and/or write all the positions in the database throughout the protocol execution. In

TABLE I
EFFICIENCY ANALYSIS OF OUR CONSTRUCTION.

	Prover	Verifier
Param Size	$(4\ell+4)\lvert\mathbb{G}\rvert + (2\ell+5)\lvert\tilde{\mathbb{G}}\rvert$	$5\lvert\mathbb{G}\rvert + 6\lvert\tilde{\mathbb{G}}\rvert$
Communication Cost		
Setup (default Tbl_{cd})	const.	const.
Setup ($\mathcal{V}$ sends Tbl_{cd})	N_{max}	N_{max}
Read	const.	const.
Write	const.	const.
Computation Cost		
Setup (vc computation)	N_{max}	N_{max}
R/W: w not computed	N_{max}	const.
R/W: w not updated	n_{upd}	const.
R/W: w updated	const.	const.

this case, $\mathcal{P}$ computes vc with cost linear in $\mathbb{Z}_p$, and computes N_{max} VC witnesses w with a total cost that grows quadratically with N_{max}. Furthermore, if the values stored in the database are big (e.g. random values in $\mathbb{Z}_p$), the computation cost of vc and each w is negatively affected.

In the best case, $\mathcal{P}$ needs to read and/or write a small subset of positions, and the database is initialized to a vector of zeroes. In this case, the computation cost for vc is constant (because the database is initialized to zeroes), and $\mathcal{P}$ only needs to compute VC witnesses w for those positions that are read/written. Moreover, the cost of computing those witnesses only increases linearly with the number of non-zero values stored in the database (instead of linearly with N_{max}). Because only a small subset of positions are written, most values in the database are 0. Furthermore, if the values written into the database are small numbers (rather than big or random numbers in $\mathbb{Z}_p$), the computation of each VC witness w is very efficient.

The average case would be somewhere in between the worst and best cases. I.e., $\mathcal{P}$ reads and/or writes a relatively small number of positions, and most of the values in the database are 0 or relatively small.

Efficiency measurements. Let $|\mathbb{G}|$, $|\tilde{\mathbb{G}}|$, and $|\mathbb{G}_t|$ be the bit length of $\mathbb{G}$, $\tilde{\mathbb{G}}$, and $\mathbb{G}_t$, respectively. In the DHE VC scheme, given the maximum vector length ℓ, the parameters are of size $(4\ell+4)\cdot|\mathbb{G}| + (2\ell+5)\cdot|\tilde{\mathbb{G}}|$. (This includes the signatures needed for the proofs for relations R_r and R_w.) We recall that $\mathcal{V}$ only needs to store a small part of the parameters, whose length is $5\cdot|\mathbb{G}|+6\cdot|\tilde{\mathbb{G}}|$. A vector commitment and a witness are of length $1\cdot|\mathbb{G}|$. The cost of computing a vector commitment or a witness increases with N_{max}, but the cost of updating commitments and witnesses increases only with the number of updated elements.

To compute the UC ZK proofs of knowledge for R_r and R_w, we use the compiler in [26]. The public parameters of the proof system contain a public key of the Paillier encryption scheme, the parameters for a multi-integer commitment scheme and the specification of a DSA group. The cost of a proof depends on the number of secret elements in the witness, which is 10 in R_r and 12 in R_w, and of the number of equations composed by Boolean ANDs, which is 5 in R_r and 6 in R_w. The computation cost for $\mathcal{P}$ of a Σ-protocol for R_r or for R_w involves one evaluation of each of the equations and one multiplication per secret value in the witness. The compiler in [26] extends a Σ-protocol and requires, additionally, a computation of a multi-integer commitment that commits to the secret values in the witness, an evaluation of a Paillier encryption for each of the secret values in the witness, a Σ-protocol to prove that the commitment and the encryptions are correctly generated, and 3 exponentiations in the DSA group. The computation cost for $\mathcal{V}$, as well as the communication cost, also depends on the number of secret values in the witness and on the number of equations. Therefore, as the number of secret values in the witness and of equations is constant in our proofs for R_r and R_w, the computation and communication cost of our proofs do not depend on N_{max}.

VII. MODULAR DESIGN WITH $\mathcal{F}_{\mathrm{CD}}$

We describe how $\mathcal{F}_{\mathrm{CD}}$ is used as building block together with $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R_i}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_v}$ to describe a hybrid protocol. Consider as a simple example a protocol where the prover $\mathcal{P}$ writes a value v at position i into the database and later on proves statements about i and v to the verifier $\mathcal{V}$. $\mathcal{P}$ needs to hide i and v from $\mathcal{V}$ when v is written into and when it is read from the database. The construction works as follows:

1) $\mathcal{V}$ runs the com.setup interface of $\mathcal{F}_{\mathrm{NIC}}$.
2) $\mathcal{V}$ uses the cd.setup interface of $\mathcal{F}_{\mathrm{CD}}$ to initialize Tbl_{cd}, which is sent to $\mathcal{P}$.
3) $\mathcal{P}$ runs the com.setup interface of $\mathcal{F}_{\mathrm{NIC}}$.
4) $\mathcal{P}$ uses the com.commit interface of $\mathcal{F}_{\mathrm{NIC}}$ to get commitment and opening $(com_i, open_i)$ to the position i.
5) $\mathcal{P}$ uses the com.commit interface of $\mathcal{F}_{\mathrm{NIC}}$ to get commitment and opening $(com_w, open_w)$ to the value v.
6) $\mathcal{P}$ uses the cd.write interface of $\mathcal{F}_{\mathrm{CD}}$ on input the tuple $(com_i, i, open_i, com_w, v, open_w)$ to write the entry $[i, v]$ into Tbl_{cd}. $\mathcal{V}$ receives (com_i, com_w). Thanks to the hiding property of the commitments computed by $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{V}$ is oblivious to the position and the value being written.
7) $\mathcal{V}$ uses the com.validate interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate that com_i contains the parameters and verification algorithm used by $\mathcal{F}_{\mathrm{NIC}}$.
8) $\mathcal{V}$ uses the com.validate interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate com_w.
9) When $\mathcal{P}$ wants to prove a statement about i and v, $\mathcal{P}$ uses the com.commit interface of $\mathcal{F}_{\mathrm{NIC}}$ to get a fresh commitment and opening $(com_i{}', open_i')$ to i.
10) $\mathcal{P}$ uses the com.commit interface of $\mathcal{F}_{\mathrm{NIC}}$ to get a fresh commitment and opening $(com_r, open_r)$ to v.
11) $\mathcal{P}$ uses the cd.read interface of $\mathcal{F}_{\mathrm{CD}}$ on input $(com_i', i, open_i', com_r, v, open_r)$ to read the entry $[i, v]$ in Tbl_{cd}. $\mathcal{V}$ receives (com_i', com_r). Thanks to the hiding property of the commitments computed by $\mathcal{F}_{\mathrm{NIC}}$, the verifier is oblivious to the position and the value being read.
12) $\mathcal{V}$ uses the com.validate interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate com_i'.
13) $\mathcal{V}$ uses the com.validate interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate com_r.

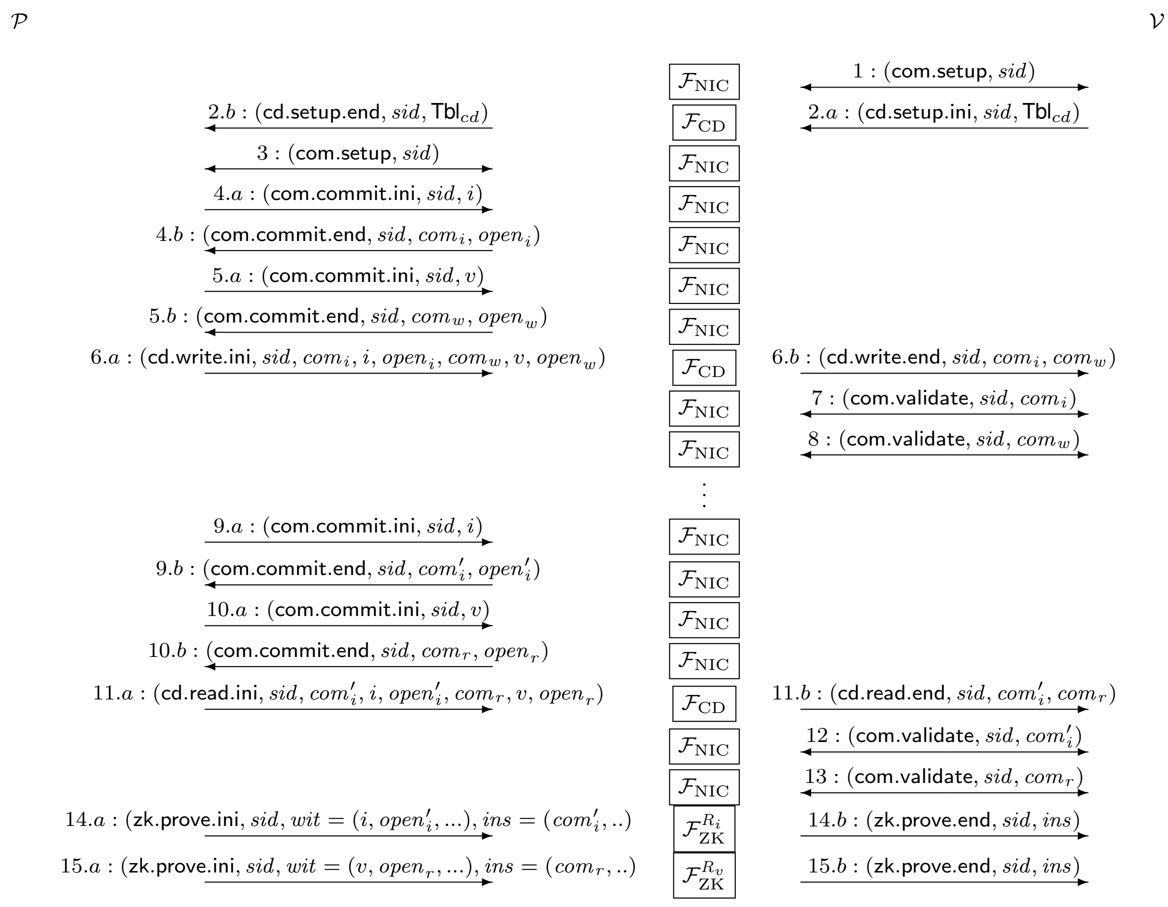
$\mathcal{P}$
$\mathcal{V}$
$\mathcal{F}_{\text{NIC}}$ 1 : (com.setup, sid)
2.b : (cd.setup.end, sid, Tbl_{cd}) $\mathcal{F}_{\text{CD}}$ 2.a : (cd.setup.ini, sid, Tbl_{cd})
3 : (com.setup, sid) $\mathcal{F}_{\text{NIC}}$
4.a : (com.commit.ini, sid, i) $\mathcal{F}_{\text{NIC}}$
4.b : (com.commit.end, sid, com_i, $open_i$) $\mathcal{F}_{\text{NIC}}$
5.a : (com.commit.ini, sid, v) $\mathcal{F}_{\text{NIC}}$
5.b : (com.commit.end, sid, com_w, $open_w$) $\mathcal{F}_{\text{NIC}}$
6.a : (cd.write.ini, sid, com_i, i, $open_i$, com_w, v, $open_w$) $\mathcal{F}_{\text{CD}}$ 6.b : (cd.write.end, sid, com_i, com_w)
$\mathcal{F}_{\text{NIC}}$ 7 : (com.validate, sid, com_i)
$\mathcal{F}_{\text{NIC}}$ 8 : (com.validate, sid, com_w)
$\vdots$
9.a : (com.commit.ini, sid, i) $\mathcal{F}_{\text{NIC}}$
9.b : (com.commit.end, sid, com'_i, $open'_i$) $\mathcal{F}_{\text{NIC}}$
10.a : (com.commit.ini, sid, v) $\mathcal{F}_{\text{NIC}}$
10.b : (com.commit.end, sid, com_r, $open_r$) $\mathcal{F}_{\text{NIC}}$
11.a : (cd.read.ini, sid, com'_i, i, $open'_i$, com_r, v, $open_r$) $\mathcal{F}_{\text{CD}}$ 11.b : (cd.read.end, sid, com'_i, com_r)
$\mathcal{F}_{\text{NIC}}$ 12 : (com.validate, sid, com'_i)
$\mathcal{F}_{\text{NIC}}$ 13 : (com.validate, sid, com_r)
14.a : (zk.prove.ini, sid, $wit = (i, open'_i, ...)$, $ins = (com'_i, ..)$) $\mathcal{F}_{\text{ZK}}^{R_i}$ 14.b : (zk.prove.end, sid, ins)
15.a : (zk.prove.ini, sid, $wit = (v, open_r, ...)$, $ins = (com_r, ..)$) $\mathcal{F}_{\text{ZK}}^{R_v}$ 15.b : (zk.prove.end, sid, ins)

Fig. 4. High-level view on an example construction that uses $\mathcal{F}_{\text{CD}}$ along with $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{ZK}}^{R_i}$ and $\mathcal{F}_{\text{ZK}}^{R_v}$.

14) $\mathcal{P}$ uses the zk.prove interface of $\mathcal{F}_{\text{ZK}}^{R_i}$ to prove statements about i. $\mathcal{P}$ sends an instance and a witness such that com'_i is in the instance, whereas $(i, open'_i)$ are in the witness. $\mathcal{V}$ receives the instance and checks that com'_i in the instance is equal to the one received from $\mathcal{F}_{\text{CD}}$. Then the binding property of the commitments computed by $\mathcal{F}_{\text{NIC}}$ guarantees that the same position i that was input to $\mathcal{F}_{\text{CD}}$ is now input to $\mathcal{F}_{\text{ZK}}^{R_i}$.
15) Similarly, $\mathcal{P}$ uses the zk.prove interface of $\mathcal{F}_{\text{ZK}}^{R_v}$ to prove statements about v.

This protocol, which we depict in Figure 4, hides from $\mathcal{V}$ that the commitments (com_i, com_w) written into the database and the commitments (com'_i, com_r) read from the database commit to the same entry $[i, v]$. If this property is not needed, a simple protocol where $\mathcal{P}$ sends to $\mathcal{V}$ commitments to i and v and later on uses $\mathcal{F}_{\text{ZK}}^{R_i}$ and $\mathcal{F}_{\text{ZK}}^{R_v}$ to prove statements about i and v can be used. Therefore, $\mathcal{F}_{\text{CD}}$ is particularly useful as building block of protocols where $\mathcal{P}$ needs to hide from $\mathcal{V}$ the positions read or written.

VIII. Applications of $\mathcal{F}_{\text{CD}}$

$\mathcal{F}_{\text{CD}}$ can be used as a building block in "commit-and-prove" two-party protocols, where $\mathcal{P}$ commits to her inputs and subsequently proves in ZK statements about the committed values. Using $\mathcal{F}_{\text{CD}}$ allows us to separate the task of storing values to be used in further ZK proofs from the task of proving statements in ZK about those values. Thanks to that, the design of the protocol is more modular, which leads to a simpler security analysis.

$\mathcal{F}_{\text{CD}}$ is particularly appealing for protocols that need to hide the positions read or written from $\mathcal{V}$. A simple example of such a protocol is an OR proof. Our main application of $\mathcal{F}_{\text{CD}}$ is its use as building block in protocols where $\mathcal{V}$ needs to obtain statistics about what $\mathcal{P}$ proves in zero-knowledge. To showcase this application, we describe a novel task we refer to as zero-knowledge counting, and we describe its application to privacy-preserving e-commerce and to privacy-preserving location sharing services. Moreover, we show that $\mathcal{F}_{\text{CD}}$ can be used for gathering other types of statistics beyond zero-knowledge counting. Concretely, we discuss the application of $\mathcal{F}_{\text{CD}}$ to privacy-preserving billing protocols. As described below, our applications of $\mathcal{F}_{\text{CD}}$ to obtaining statistics are very efficient because they mirror the best case for computation cost described in VI-C.

OR proofs. In the full version [23], we describe how $\mathcal{F}_{\text{CD}}$ can be used to compute ZK proofs for OR relations, i.e., relations where $\mathcal{P}$ proves that at least one value in a database fulfills a statement, while hiding from $\mathcal{V}$ which of the values fulfills it. In a nutshell, the database of values for the OR relation is written into Tbl_{cd}. After that, $\mathcal{P}$ simply reads one of the values from Tbl_{cd} and uses $\mathcal{F}_{\rm ZK}^{R}$ to prove in ZK that this value fulfills the statement. $\mathcal{F}_{\rm NIC}$ is used to ensure that $\mathcal{F}_{\rm ZK}^{R}$ receives the input read from $\mathcal{F}_{\rm CD}$. $\mathcal{V}$ learns neither what value is read nor the position where it is stored.

The protocol for an OR proof is more efficient when the values in Tbl_{cd} are known by both $\mathcal{P}$ and $\mathcal{V}$. In this case, $\mathcal{V}$ sets up the database, and then $\mathcal{P}$ performs a read operation.

When Tbl_{cd} needs to be hidden from $\mathcal{V}$, if the database size is N_{max}, $\mathcal{P}$ needs to perform N_{max} write operations to write the database values into Tbl_{cd}. Therefore, this case is similar to the worst case for computation cost described in Section VI-C. $\mathcal{P}$ needs to compute a VC witness for each of the database positions with a computation cost that grows quadratically with N_{max}. Still, this cost can be amortized if the number of read operations performed subsequently is big compared to N_{max}.

Zero-Knowledge counting. In the full version [23], we describe the use of $\mathcal{F}_{\rm CD}$ to construct a protocol for "zero-knowledge counting", which roughly speaking is about counting the number of times each possible witness is used in the computation of different ZK proofs. ZK counting is a task parameterized by a relation R. For the possible witness values wit such that, for any instance ins, $(wit, ins) \in R$, ZK counting consists in counting how many times each witness value wit was used by $\mathcal{P}$. We define an ideal functionality $\mathcal{F}_{\rm ZKC}^{R}$ for ZK counting that stores one counter for each possible witness value. When $\mathcal{P}$ sends $(wit, ins) \in R$, $\mathcal{F}_{\rm ZKC}^{R}$ increments the counter for the witness value wit. At this stage, no information about the witness used is revealed to $\mathcal{V}$, but later $\mathcal{F}_{\rm ZKC}^{R}$ allows $\mathcal{P}$ to disclose to $\mathcal{V}$ the value of the counter for a given witness.

We construct ZK counting by using $\mathcal{F}_{\rm CD}$ as a building block. Basically, the array stored by $\mathcal{F}_{\rm CD}$ stores the counters for each of the witness values. Each position in the array is associated with one of the witness values. $\mathcal{F}_{\rm ZK}^{R_c}$ is used to prove in ZK that a counter is correctly incremented. $\mathcal{F}_{\rm NIC}$ is used to guarantee that equality between the counter read from $\mathcal{F}_{\rm CD}$ and the one input to $\mathcal{F}_{\rm ZK}^{R_c}$, and also to guarantee equality between the updated counter input to $\mathcal{F}_{\rm ZK}^{R_c}$ and the counter written into $\mathcal{F}_{\rm CD}$.

ZK counting is useful for the collection of aggregate statistics about $\mathcal{P}$. We describe two applications of ZK counting to privacy-preserving e-commerce and to privacy-preserving location-sharing services. In this applications, the witnesses represent types of items for sale and locations respectively, and the counters represent the number of times users purchased an item or checked-in at a location. Therefore, for privacy protection, $\mathcal{P}$ needs to hide both witnesses and counters from $\mathcal{V}$. $\mathcal{P}$ also needs to prove in ZK statements about them. For example, $\mathcal{P}$ must prove that she increments the counter for the item that she purchases.

Privacy-Preserving E-Commerce. Priced oblivious transfer (POT) is a protocol between a seller and a buyer that can be applied to e-commerce of digital goods. The seller sells messages $(m_1, \ldots, m_N)$. The prices of the messages are $(p_1, \ldots, p_N)$. At each purchase, the buyer chooses $\sigma \in [1, N]$ and obtains message m_σ. The seller does not learn any information about σ, but is guaranteed that the buyer does not learn any information about messages different from m_σ. Some constructions of POT use zero-knowledge proofs as building block [11], [30]. At each purchase, a buyer uses σ as witness in order to prove in zero-knowledge that she purchases message m_σ and pays the price p_σ.

POT schemes use a prepaid mechanism [2], [11], [30]. The buyer pays an initial deposit to the seller. When the buyer purchases m_σ, the buyer subtracts the price p_σ from the deposit in such a way that the seller learns neither p_σ nor the new value of the deposit.

Recently, in [31], $\mathcal{F}_{\rm CD}$ is used as building block in the construction of a POT scheme that allows the seller to obtain aggregate statistics about the purchases of a buyer. Seller and buyer execute multiple POT protocols where each of the indices $[1, N]$ is associated with a type of digital content. $\mathcal{F}_{\rm CD}$ is used to store a database of N counters, where the counter at position $i \in [1, N]$ counts the number of times the buyer has purchased messages associated with index i. At each purchase, the corresponding counter is incremented. We remark that hiding the position of the counter that is updated is crucial to maintain the privacy properties of POT. $\mathcal{F}_{\rm CD}$ is also used to store the buyer's deposit, which is updated at each purchase.

As described in [31], $\mathcal{F}_{\rm CD}$ allows the design of the first POT protocol where the seller obtains statistics about the buyer's purchases. Aggregate statistics about multiple buyers are also possible by using a secure multiparty computation protocol on input the committed database of each of the buyers.

As explained in [31], the use of $\mathcal{F}_{\rm CD}$ introduces little overhead in comparison to previous POT protocols [11], [30] that do not gather statistics. The reason is that previous protocols already needed a database (in the form of a commitment) to store and update the deposit. With $\Pi_{\rm CD}$ the database also stores the counters N, but the communication cost and amortized computation cost of reading and writing the database is independent of N. Furthermore, this application of $\mathcal{F}_{\rm CD}$ mirrors the best case for computation cost described in Section VI-C. Namely, from the probably large number N of messages sold by the seller, the average buyer is likely to purchase a small subset, so only a small subset of positions is read and written. Additionally, the counters are initialized to zero.

Privacy-Preserving Location-Sharing Services. In location-sharing services, a user checks in at a venue (e.g., a restaurant or shop) and reports her location to the service provider. The service provider sends the user's location to the user's registered friends.

In a privacy-preserving location-sharing service, a user encrypts her location and sends it to the service provider [4]. The service provider forwards the encrypted location to the user's registered friends, but does not learn the location himself. Still, the service provider requires the user to prove in ZK that the encrypted location is a venue registered with the service provider. There are different ways this ZK proof can be computed. For example, the service provider can issue a list of signatures where each signature signs a registered venue. The user then picks the signature that signs the venue where she checks in and proves in ZK that the encrypted message equals a message signed by the service provider.

In this setting, $\mathcal{F}_{\mathrm{CD}}$ can be used to allow the service provider to obtain aggregate information about the venues where the user checks in. Each time a user checks in at a venue, the user uses the identifier of the venue as witness in the ZK proof. By using $\mathcal{F}_{\mathrm{CD}}$, the protocol can be augmented to count the number of times each of the venues' identifiers is used as witness by the user, and to eventually reveal to the service provider aggregate data about the venues where the user checks in. This could be used by users, e.g., to obtain discounts at venues where they have checked-in a certain number of times. We remark that this application of $\mathcal{F}_{\mathrm{CD}}$ is possible thanks to the fact that, when a counter is incremented, $\mathcal{F}_{\mathrm{CD}}$ hides from the provider both the counter value and the witness associated to that counter.

In terms of efficiency, the communication cost and amortized computation cost of our construction for $\mathcal{F}_{\mathrm{CD}}$ are independent of N. Moreover, this application of $\mathcal{F}_{\mathrm{CD}}$ also mirrors the best case for computation cost, i.e., the average user is likely to check-in at a small subset of the probably large number N of venues offered by the provider, and the counters are initialized at 0.

Privacy-Preserving Billing. In privacy-preserving billing [3], a meter measures the consumption of a service by a user and outputs signed meter readings. The service provider establishes a tariff policy. To protect user privacy, the user computes the price to be paid for all the meter readings in a billing period and proves in ZK that the price is correct according to the signed meter readings and the tariff policy.

$\mathcal{F}_{\mathrm{CD}}$ can be used in this setting to allow the provider to get aggregate statistics about user consumption and to enable history-dependent policies. For example, the provider can offer discounts to users whose consumption does not exceed a threshold during e.g. 90 per cent of time on high-demand periods. Using $\mathcal{F}_{\mathrm{CD}}$, the user can store her consumption readings in the database at positions that represent the time of consumption, and then prove in ZK that she fulfills the policy without disclosing the times of consumption used.

IX. Related Work

Several cryptographic primitives allow us to represent a number of values in a much smaller cryptographic artifact and later prove knowledge of a number represented in the artifact. We summarize these primitives and compare them to our solution. In the full version [23], we give a more detailed review of those primitives.

Commitments and ZK proofs of shuffles. Many protocols (e.g. [1], [10], [11]) use commitment schemes to maintain a database between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. However, commitments on their own are not adequate to realize $\mathcal{F}_{\mathrm{CD}}$ because they do not allow $\mathcal{P}$ to hide the positions read from or written into the database. ZK proofs of shuffles [14] can be used to shuffle the commitments in order to hide the positions read or written. A construction using commitments along with proofs of shuffles could realize $\mathcal{F}_{\mathrm{CD}}$, but less efficiently than a construction based on VCs.

Accumulators. A cryptographic accumulator [32] allows one to represent a set X succinctly as a single accumulator value A. It also provides a method to prove succinctly that an element x belongs to X to any party that holds A. This method consists in computing a witness W whose size is independent of the size $|X|$ of the set. Soundness (or collision-freeness) guarantees that it is infeasible to prove that $x \in X$ if $x \notin X$. The main difference between VCs and accumulators is that, while accumulators allow for committing to a set, VCs allow for committing to a vector of messages, where each message is committed at a specific position. This allows the construction of an updatable committed database where it is also possible to prove statements about the position where a message is written or read.

Vector Commitments. In [7], a definition of non-hiding VCs with updates is given. To obtain hiding VCs, it is suggested to compose a non-hiding VC scheme with a standard commitment scheme. Two constructions of non-hiding VCs are given based on the CDH and RSA assumptions. In [6], a construction of mercurial VCs based on the Diffie-Hellman Exponent (DHE) assumption is proposed. This construction leads to constructions of non-hiding and hiding VCs based on DHE, which were used in, e.g., [33]–[35].

Our committed database uses as building block any hiding VC scheme with updates, along with ZK proofs of knowledge that a vector component is being read or written. To instantiate our committed database, we use a construction of hiding VCs with updates based on the DHE assumption along with the corresponding ZK proofs for reading and writing. In this construction, the size of the public parameters is linear in the maximum vector length. In comparison, in a hiding VC construction from CDH, the size of the public parameters would be quadratic (the advantage would be to use a more standard assumption).

Recently, in [36], subvector commitments (SVC) are proposed. In SVC, a commitment can be opened to a set of positions such that the size of the witness does not depend on the size of the set. A construction for SVC secure under the cube Diffie-Hellman assumption is given, in which the public parameter size grows quadratically with the vector length. $\mathcal{F}_{\mathrm{CD}}$ and our applications for it in Section VIII only require to open one vector component at a time. SVC can be used to construct a variant of $\mathcal{F}_{\mathrm{CD}}$ where several positions are read or written simultaneously. Nevertheless, we note that, despite that SVC provides witnesses of size independent of the number of positions open, the entire witness of read or write proofs would still grow with the number of positions, and thus the efficiency of those proofs would not be independent of the set size. In [36], [37], constructions for SVC based on groups of hidden order are proposed that are better suited for bit vectors.

Polynomial and functional commitments. Polynomial commitments allow a committer to commit to a polynomial and open the commitment to an evaluation of the polynomial. They can be used as VCs by committing to a polynomial that interpolates the vector to be committed. In [12], a construction of polynomial commitments from the SDH assumption is proposed, which has the disadvantage that efficient updates cannot be computed without knowledge of the trapdoor. A further generalization of VCs and polynomial commitments are functional commitments [13], [36].

Zero-Knowledge Data Structures. Zero-Knowledge Sets (ZKS) [15] allow a prover $\mathcal{P}$ to commit to a set X and to subsequently prove to a verifier $\mathcal{V}$ (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 $\mathcal{V}$. Both ZKS and ZKDB are two-party protocols between $\mathcal{P}$ and $\mathcal{V}$. Security for $\mathcal{V}$ requires that an adversarial $\mathcal{P}$ is not able to prove $x \in X$ if $x \notin X$ (and vice versa), while zero-knowledge requires that proofs of (non-)membership reveal nothing else beyond (non-)membership, not even the size of the set. In [19], a construction for zero-knowledge lists (ZKL) is proposed, where a list is defined as an ordered set. Updatable ZKDB were first proposed in [18]. In [7], a construction for updatable ZKDB based on updatable VCs and trapdoor mercurial commitments is proposed.

There are several differences between our committed database and previous work. First, our committed database is updatable, which was only considered in [7], [18]. Second, our committed database is oblivious. $\mathcal{P}$ proves in ZK that a pair of commitments commit to a position and value that are stored in the database. In contrast, in previous constructions, $\mathcal{P}$ reveals a position and a value along with a proof that the position and the value are stored in the database. The obliviousness property allows our committed database to be used as building block in applications that protect the privacy of the $\mathcal{P}$, because $\mathcal{P}$ could choose to open the commitments, but could also prove statements in ZK about the committed position and value without revealing them.

From a definitional point of view, security definitions given in previous works are not in the UC model and a mechanism to integrate modularly ZKS or ZKDB as building blocks of other protocols is not given. Our functionality for a committed database allows the security analysis of ZK data structures in a composable framework, which will facilitate the modular design and analysis of protocols that use them as a building block.

Another key difference is that we do not require the size of the database to be hidden. Thanks to that, our construction for a committed database is more efficient than existing constructions for ZKS or ZKDB. This relaxation of the ZK property is not relevant for our applications for a committed database. In this respect, our construction for a committed database is similar to the constructions for "nearly" ZKS and ZKDB given in [12]. However, this "nearly" ZKS and ZKDB constructions based on the SDH assumption are not updatable and, moreover, extending them with efficient updates is not possible without knowledge of the trapdoor. In fact, as pointed out in [38], when hiding the size of the database is required, for any construction that uses a non-interactive commitment phase (as is the case in the ZKS and ZKDB constructions cited above), black-box extraction of the database by the simulator in the security proof is not possible. In [38], a secure committed database where the database size is hidden is defined in the UC model, and a construction that uses an interactive commitment phase is proposed. As a consequence of needing to hide the database size, the construction in [38] is also less efficient than ours. Also, their ideal functionality does not facilitate modular design, and it outputs position-value pairs instead of commitments to a position and to a value, which hinders its use as building block in protocols that need to protect the privacy of the prover.

In [39], a functionality for a database such that both prover and verifier know the database contents is proposed. The verifier can write information into the database, while the prover performs a read operation similar to the one of $\mathcal{F}_{\mathrm{CD}}$. Non-hiding VCs are used to construct the functionality. In [40], a variant of the functionality in [39] that interacts with multiple provers and provides unlinkable read operations is defined and constructed by using SVCs.

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 M of large size $|M|$. However, there are techniques that attain costs sublinear in $|M|$. Probabilistically checkable proofs [41] achieve verification cost sublinear in $|M|$, but the cost for the prover is linear in $|M|$. In succinct non-interactive arguments of knowledge [42], verification cost is independent of $|M|$, but the cost for the prover is still linear in $|M|$.

ZK proofs for relations described as ORAM programs [8], [9] involve an initialization phase in which the prover commits to M. In [8], the cost for the prover is linear in $|M|$, whereas the cost for the verifier is independent of $|M|$. After the initialization phase, many proofs can be computed about M whose cost is sublinear (proportional to the runtime of the ORAM program) both for the prover and for the verifier. The protocol in [8] uses a non-programmable random oracle, which is key for achieving constant cost for the verifier in the initialization phase. Any protocol in the standard model would involve communication cost linear in M to allow knowledge extraction.

To compare our protocol with [8], we consider the setting of an OR proof when the database must be hidden from $\mathcal{V}$, i.e., $\mathcal{P}$ writes M into Tbl_{cd}, and after that reads values in M from Tbl_{cd}. Assuming the worst case of all non-zero values in M, the cost of writing M into Tbl_{cd} is quadratic in $|M|$ for Tbl_{cd} and linear in $|M|$ for $\mathcal{V}$. However, after that the cost of each read operation is independent of $|M|$ for both $\mathcal{P}$ and $\mathcal{V}$. Our protocol provides thus better asymptotic amortized cost than the state of the art protocol in [8] when the number of read operations is big compared to $|M|$. (In the initialization phase, the cost for the verifier is linear in $|M|$ which is unavoidable when aiming for security in the standard CRS-hybrid model.) We note that [8] does not provide a concrete instantiation or efficiency analysis of their protocol, so we do not compare it with the instantiation of our protocol in Section VI.

X. Conclusion and Future Work

We have made a couple of design choices in $\mathcal{F}_{\mathrm{CD}}$. For instance, all parties are authenticated and the functionality implies interaction between $\mathcal{P}$ and $\mathcal{V}$. Our functionality could be extended, depending on the requirements of potential applications. For instance, a non-interactive functionality would allow for applications with multiple verifiers. Another extension of $\mathcal{F}_{\mathrm{CD}}$ could provide pseudonymity or anonymity, which would be suitable for applications such as attribute-based credentials [10]. $\mathcal{F}_{\mathrm{CD}}$ is suitable for protocols in which a one-dimensional array is adequate to implement the database. $\mathcal{F}_{\mathrm{CD}}$ could be extended to more complex updatable data structures, such as multi-dimensional arrays, lists, trees, and graphs. Some of these would require operations beyond read and write.

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

References

[1] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai, "Universally composable two-party and multi-party secure computation," in *STOC 2002*, pp. 494–503.

[2] W. Aiello, Y. Ishai, and O. Reingold, "Priced oblivious transfer: How to sell digital goods," in *EUROCRYPT 2001*, pp. 119–135.

[3] A. Rial and G. Danezis, "Privacy-preserving smart metering," in *WPES 2011*, pp. 49–60.

[4] M. Herrmann, A. Rial, C. Díaz, and B. Preneel, "Practical privacy-preserving location-sharing based services with aggregate statistics," in *WiSec'14*, pp. 87–98.

[5] J. Camenisch, M. Dubovitskaya, and A. Rial, "UC commitments for modular protocol design and applications to revocation and attribute tokens," in *CRYPTO 2016*, pp. 208–239.

[6] B. Libert and M. Yung, "Concise mercurial vector commitments and independent zero-knowledge sets with short proofs," in *TCC 2010*, pp. 499–517.

[7] D. Catalano and D. Fiore, "Vector commitments and their applications," in *PKC 2013*, pp. 55–72.

[8] P. Mohassel, M. Rosulek, and A. Scafuro, "Sublinear zero-knowledge arguments for RAM programs," in *EUROCRYPT 2017*, pp. 501–531.

[9] Z. Hu, P. Mohassel, and M. Rosulek, "Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost," in *CRYPTO 2015*, pp. 150–169.

[10] J. Camenisch and A. Lysyanskaya, "An efficient system for non-transferable anonymous credentials with optional anonymity revocation," in *EUROCRYPT 2001*, pp. 93–118.

[11] A. Rial, M. Kohlweiss, and B. Preneel, "Universally composable adaptive priced oblivious transfer," in *Pairing 2009*, pp. 231–247.

[12] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT 2010*, pp. 177–194.

[13] B. Libert, S. C. Ramanna, and M. Yung, "Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions," in *ICALP 2016*, pp. 30:1–30:14.

[14] M. Chase, M. Kohlweiss, A. Lysyanskaya, and S. Meiklejohn, "Malleable proof systems and applications," in *EUROCRYPT 2012*, pp. 281–300.

[15] S. Micali, M. O. Rabin, and J. Kilian, "Zero-knowledge sets," in *FOCS 2003*, pp. 80–91.

[16] M. Chase, A. Healy, A. Lysyanskaya, T. Malkin, and L. Reyzin, "Mercurial commitments with applications to zero-knowledge sets," *J. Cryptology*, vol. 26, no. 2, pp. 251–279, 2013.

[17] D. Catalano, D. Fiore, and M. Messina, "Zero-knowledge sets with short proofs," in *EUROCRYPT 2008*, pp. 433–450.

[18] M. Liskov, "Updatable zero-knowledge databases," in *ASIACRYPT 2005*, 2005, pp. 174–198.

[19] E. Ghosh, O. Ohrimenko, and R. Tamassia, "Zero-knowledge authenticated order queries and order statistics on a list," in *ACNS 2015*, pp. 149–171.

[20] E. Ghosh, M. T. Goodrich, O. Ohrimenko, and R. Tamassia, "Verifiable zero-knowledge order queries and updates for fully dynamic lists and trees," in *SCN 2016*, pp. 216–236.

[21] R. Ostrovsky, C. Rackoff, and A. D. Smith, "Efficient consistency proofs for generalized queries on a committed database," in *ICALP 2004*, pp. 1041–1053.

[22] S. Goldberg, M. Naor, D. Papadopoulos, L. Reyzin, S. Vasant, and A. Ziv, "NSEC5: provably preventing DNSSEC zone enumeration," in *NDSS 2015*, 2015.

[23] J. Camenisch, M. Dubovitskaya, and A. Rial, "Concise uc zero-knowledge proofs for oblivious updatable databases," 2019. [Online]. Available: http://hdl.handle.net/10993/39423

[24] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," Cryptology ePrint Archive, Report 2000/067, 2000, https://eprint.iacr.org/2000/067.

[25] ——, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS 2001*, pp. 136–145.

[26] J. Camenisch, S. Krenn, and V. Shoup, "A framework for practical universally composable zero-knowledge protocols," in *ASIACRYPT 2011*, pp. 449–467.

[27] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM J. Comput.*, vol. 17, no. 2, pp. 281–308, 1988.

[28] M. Abe, J. Groth, K. Haralambiev, and M. Ohkubo, "Optimal structure-preserving signatures in asymmetric bilinear groups," in *CRYPTO 2011*, pp. 649–666.

[29] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO '91*, pp. 129–140.

[30] J. Camenisch, M. Dubovitskaya, and G. Neven, "Unlinkable priced oblivious transfer with rechargeable wallets," in *FC 2010*, pp. 66–81.

[31] A. Damodaran, M. Dubovitskaya, and A. Rial, "UC priced oblivious transfer with purchase statistics and dynamic pricing," in *INDOCRYPT 2019*, pp. 273–296.

[32] J. C. Benaloh and M. de Mare, "One-way accumulators: A decentralized alternative to digital sinatures (extended abstract)," in *EUROCRYPT '93*, pp. 274–285.

[33] B. Libert, T. Peters, and M. Yung, "Group signatures with almost-for-free revocation," in *CRYPTO 2012*, pp. 571–589.

[34] M. Izabachène, B. Libert, and D. Vergnaud, "Block-wise p-signatures and non-interactive anonymous credentials with efficient attributes," in *Cryptography and Coding - 13th IMA International Conference, IMACC 2011*, pp. 431–450.

[35] M. Kohlweiss and A. Rial, "Optimally private access control," in *WPES 2013*, pp. 37–48.

[36] R. W. F. Lai and G. Malavolta, "Subvector commitments with application to succinct arguments," in *CRYPTO 2019*, pp. 530–560.

[37] D. Boneh, B. Bünz, and B. Fisch, "Batching techniques for accumulators with applications to iops and stateless blockchains," in *CRYPTO 2019*, pp. 561–586.

[38] M. Chase and I. Visconti, "Secure database commitments and universal arguments of quasi knowledge," in *CRYPTO 2012*, pp. 236–254.

[39] A. Damodaran and A. Rial, "UC updatable databases and applications," in *AFRICACRYPT 2020*, pp. 66–87.

[40] ——, "Unlinkable updatable databases and oblivious transfer with access control," in *ACISP 2020*, pp. 584–604.

[41] J. Kilian, "A note on efficient zero-knowledge proofs and arguments (extended abstract)," in *STOC 1992*, pp. 723–732.

[42] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *EUROCRYPT 2013*, pp. 626–645.