Unlinkable Updatable Databases and Oblivious Transfer with Access Control*

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

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

Abstract. An oblivious transfer with access control protocol (OTAC) allows us to protect privacy of accesses to a database while enforcing access control policies. Existing OTAC have several shortcomings. First, their design is not modular. Typically, to create an OTAC, an adaptive oblivious transfer protocol (OT) is extended ad-hoc. Consequently, the security of the OT is reanalyzed when proving security of the OTAC, and it is not possible to instantiate the OTAC with any secure OT. Second, existing OTAC do not allow for policy updates. Finally, in practical applications, many messages share the same policy. However, existing OTAC cannot take advantage of that to improve storage efficiency.
We propose an UC-secure OTAC that addresses the aforementioned shortcomings. Our OTAC uses as building blocks the ideal functionalities for OT, for zero-knowledge (ZK) and for an *unlinkable updatable database* (UUD), which we define and construct. UUD is a protocol between an updater $\mathcal{U}$ and multiple readers $\mathcal{R}_k$. $\mathcal{U}$ sets up a database and updates it. $\mathcal{R}_k$ can read the database by computing UC ZK proofs of an entry in the database, without disclosing what entry is read. In our OTAC, UUD is used to store and read the policies.
We construct an UUD based on subvector commitments (SVC). We extend the definition of SVC with update algorithms for commitments and openings, and we provide an UC ZK proof of a subvector. Our efficiency analysis shows that our UUD is practical.

Keywords: Vector commitments, bilinear maps, universal composability

1 Introduction

Oblivious transfer with access control protocols [17,8] (OTAC) run between a sender $\mathcal{U}$ and receivers $\mathcal{R}_k$. $\mathcal{U}$ receives as input a tuple $(m_i, \mathsf{ACP}_i)_{\forall i \in [1,N]}$ of messages and their associated access control policies. In a transfer phase, a receiver $\mathcal{R}_k$ chooses an index $i \in [1, N]$ and obtains the message m_i if $\mathcal{R}_k$ satisfies the policy ACP_i. $\mathcal{U}$ does not learn i, whereas $\mathcal{R}_k$ does not learn any information about other messages.

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

In the following, we only consider OTAC in which the receivers learn all the policies $(\mathsf{ACP}_i)_{\forall i \in [1,N]}$, that are stateless, i.e. fulfilment of a policy by $\mathcal{R}_k$ does not depend on the history of messages received by $\mathcal{R}_k$, and that are adaptive, i.e. there are several transfers and $\mathcal{R}_k$ can choose i after receiving messages in previous transfers. In §7, we discuss stateful and adaptive OTAC and OTAC with hidden policies. Additionally, we focus on OTAC that provide anonymity and unlinkability, i.e., OTAC where $\mathcal{U}$ cannot link a transfer to a receiver identity $\mathcal{R}_k$ and where transfers to $\mathcal{R}_k$ are unlinkable with respect to each other.

Existing adaptive and stateless OTAC follow a common pattern in their design. In the initialization phase, $\mathcal{U}$ computes N ciphertexts c_i that encrypt m_i. Some OTAC [8,1,23] use a signature that binds ACP_i to c_i, while others [30,31,32,27] use fuzzy identity-based encryption (IBE) or ciphertext-policy attribute-based encryption (CP-ABE) to encrypt m_i under ACP_i. The receivers obtain $(c_i, \mathsf{ACP}_i)_{\forall i \in [1,N]}$. To prove fulfilment of policies, $\mathcal{R}_k$ proves to an authority that she possesses some attributes and obtains a credential or secret key for her attributes. In the transfer phase, $\mathcal{R}_k$ interacts with $\mathcal{U}$ in such a way that $\mathcal{R}_k$ can decrypt c_i for her choice i only if her certified attributes satisfy ACP_i. Those OTAC have several design shortcomings.

Modularity. Although some OTAC are extensions of adaptive oblivious transfer protocols (OT), they do not use OT as building block. Instead, the OT is modified ad-hoc to create the OTAC, blurring which elements were part of the OT and which ones were added to provide access control. The lack of modularity has two disadvantages. First, when the security of the OTAC is analyzed, the security of the underlying OT needs to be reanalyzed. Second, the OTAC cannot be instantiated with any secure adaptive OT, and consequently, whenever more efficient OT schemes are proposed, the OTAC cannot use them and would need to be redesigned.

Policy Updates. All the existing OTAC do not allow for policy updates, i.e., if a policy ACP_i needs to be updated, the initialization phase needs to be rerun. In practical applications of OTAC (e.g. medical or financial databases), it would be desirable to update policies dynamically throughout the protocol execution without needing to re-encrypt messages. To enable policy updates, we would need to separate the encryptions c_i of m_i from the method used to encode policies ACP_i. As explained above, OTAC use signatures schemes or CP-ABE to bind policies to ciphertexts. It would be possible to separate, e.g., a signature on the policy ACP_i from the encryption c_i of m_i, while still allowing $\mathcal{R}_k$ to prove the association between c_i and ACP_i in the transfer phase. However, a revocation mechanism to revoke the outdated signatures would also need to be implemented, which would decrease efficiency.

Storage cost. All the existing OTAC associate each encryption c_i with a policy ACP_i. However, in practical applications, multiple database records are associated with a single policy. Therefore, if we separate the ciphertexts c_i from the method used to encode policies ACP_i, it would be possible to improve efficiency by associating a policy to multiple ciphertexts.

1.1 Our Contribution

We define and construct an unlinkable updatable database (UUD), a novel building block that may be of independent interest, and we use UUD to construct modularly OTAC that enable dynamic policy updates without the need of a revocation mechanism, and that can associate a policy to multiple messages.

Functionality $\mathcal{F}_{\text{UUD}}$. We use the universal composability (UC) framework [15] and define an ideal functionality $\mathcal{F}_{\text{UUD}}$ in §3. We define UUD as a task between multiple readers $\mathcal{R}_k$ and an updater $\mathcal{U}$. $\mathcal{U}$ sets a database DB and updates it at any time throughout the protocol execution. DB consists of N entries of the form $[i, v_{i,1}, \ldots, v_{i,L}]$, where i identifies the database entry and $(v_{i,1}, \ldots, v_{i,L})$ are the values stored in that entry. Any $\mathcal{R}_k$ and $\mathcal{U}$ know the content of DB. A reader $\mathcal{R}_k$ can read DB by computing a zero-knowledge (ZK) proof of knowledge of an entry $[i, v_{i,1}, \ldots, v_{i,L}]$. $\mathcal{F}_{\text{UUD}}$ hides from $\mathcal{U}$ which entry was read but ensures that it is not possible to prove that an entry is stored in DB if that is not the case. $\mathcal{F}_{\text{UUD}}$ allows $\mathcal{R}_k$ to remain anonymous and unlinkable when reading DB.

OTAC. In §6, we propose a functionality $\mathcal{F}_{\text{OTAC}}$. $\mathcal{F}_{\text{OTAC}}$ follows previous OTAC functionalities [8] but introduces two main modifications. First, it splits the initialization interface into two interfaces: $\mathsf{otac.init}$, in which the sender $\mathcal{U}$ receives $(m_i)_{\forall i \in [1,N]}$, and $\mathsf{otac.policy}$, in which $\mathcal{U}$ receives $(\mathsf{ACP}_i)_{\forall i \in [1,N]}$. This enables $\mathcal{U}$ to make policy updates via $\mathsf{otac.policy}$ throughout the protocol execution. Second, previous functionalities include an issuance phase where an issuer certifies $\mathcal{R}_k$ attributes. Instead, $\mathcal{F}_{\text{OTAC}}$ leaves more open and flexible how access control is proven. $\mathcal{U}$ sets and updates a relation R_{ACP} that specifies what $\mathcal{R}_k$ must prove to obtain access to messages. Each policy ACP_i is an instance *ins* for R_{ACP} and, in the transfer phase, $\mathcal{R}_k$ must provide a witness *wit* such that $(wit, ins) \in R_{\mathsf{ACP}}$. *wit* could contain, e.g., signatures from an issuer on $\mathcal{R}_k$ attributes, but in general any data required by R_{ACP}.

We also describe a modular construction Π_{OTAC}. In the UC framework, modularity is achieved by describing hybrid protocols. In a hybrid protocol, the building blocks are described by their ideal functionalities, and parties in the real world invoke those ideal functionalities. Π_{OTAC} uses as building block $\mathcal{F}_{\text{OT}}$, and thus Π_{OTAC} can be instantiated by any secure adaptive OT. To implement access control, Π_{OTAC} uses $\mathcal{F}_{\text{UUD}}$ and $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$. $\mathcal{U}$ stores $(\mathsf{ACP}_i)_{\forall i \in [1,N]}$ in DB in $\mathcal{F}_{\text{UUD}}$. Each entry $[i, v_{i,1}, \ldots, v_{i,L}]$ stores the index i and the representation $\mathsf{ACP}_i = (v_{i,1}, \ldots, v_{i,L})$ of a policy. In a transfer phase, $\mathcal{R}_k$ uses $\mathcal{F}_{\text{UUD}}$ to read ACP_i for her choice i and then $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$ to prove fulfilment of ACP_i. One challenge when defining a hybrid protocol is to ensure that two functionalities receive the same input. For example, in the transfer interface of Π_{OTAC}, we need to ensure that the choice i sent to $\mathcal{F}_{\text{OT}}$ (to obtain m_i) and to $\mathcal{F}_{\text{UUD}}$ (to read ACP_i) are equal. To this end, we use the method in [11], in which functionalities receive committed inputs produced by a functionality $\mathcal{F}_{\text{NIC}}$ for non-interactive commitments.

Our modular design has the following advantages. First, it simplifies the security analysis because security proofs in the hybrid model are simpler and

because, by splitting the protocol into smaller building blocks, security analysis of constructions for those building blocks are also simpler. Second, it allows multiple instantiations by replacing each of the functionalities by any protocols that realize them. Third, it allows the study of the UUD task in isolation, which eases the comparison of different constructions for it.

Construction Π_{UUD}. In §4, we propose a construction Π_{UUD} for $\mathcal{F}_{\mathrm{UUD}}$. Π_{UUD} is based on subvector commitments (SVC) [22], which we extend with a UC ZK proof of knowledge of a subvector. A SVC scheme allows us to compute a commitment *com* to a vector $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[N])$. *com* can be opened to a subvector $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$, where $I = \{i_1, \ldots, i_n\} \subseteq [1, N]$. The size of the opening w_I is independent of N and of $|I|$. SVC were recently proposed as an improvement of vector commitments [25,16], where the size of w_I is independent of N but dependent on $|I|$. We extend the definition of SVC to include algorithms to update commitments and openings when part of the vector is updated.

Π_{UUD} works as follows. $\mathcal{U}$ uses a bulletin board BB to publish the database DB and any $\mathcal{R}_k$ obtains DB from BB. A BB ensures that all readers obtain the same version of DB, which we need to guarantee unlinkability. Both $\mathcal{U}$ and any $\mathcal{R}_k$ map a DB with N entries of the form $[i, v_{i,1}, \ldots, v_{i,L}]$ to a vector $\mathbf{x}$ of length $N \times L$ such that $\mathbf{x}[(i-1)L + j] = v_{i,j}$ for all $i \in [1, N]$ and $j \in [1, L]$, and they compute a commitment *com* to $\mathbf{x}$. To update a database entry, $\mathcal{U}$ updates BB, and $\mathcal{U}$ and any $\mathcal{R}_k$ update *com*. Therefore, updates do not need any revocation mechanism. To prove in ZK that an entry $[i, v_{i,1}, \ldots, v_{i,L}]$ is in DB, $\mathcal{R}_k$ computes an opening w_I for $I = \{(i-1)L + 1, \ldots, (i-1)L + L\}$ and uses it to compute a ZK proof of knowledge of the subvector $(\mathbf{x}[(i-1)L + 1], \ldots, \mathbf{x}[(i-1)L + L])$. This proof guarantees that I is the correct set for index i.

We describe an efficient instantiation of Π_{UUD} in §5 that uses a SVC scheme based on the Cube Diffie-Hellman assumption [22]. In terms of efficiency, the storage cost grows quadratically with the vector length $N \times L$. However, after initializing *com* and the openings w_I to the initial DB, the communication and computation costs of the update and read operations are independent of N. Therefore, our instantiation allows for an OTAC where the database of policies can be updated and read efficiently. We have implemented our instantiation. Our efficiency measurements in §5 show that it is practical.

We describe a variant of our instantiation where each database entry is $[i_{min}, i_{max}, v_{i,1}, \ldots, v_{i,L}]$, where $[i_{min}, i_{max}] \in [1, N]$ is a range of indices. This allows for an OTAC with reduced storage cost. If the messages $(m_{i_{min}}, \ldots, m_{i_{max}})$ are associated with a single policy ACP, only one database entry is needed to store ACP. In contrast, previous OTAC that use signatures or CP-ABE need to embed a policy in every ciphertext.

Π_{UUD} can be regarded as an efficient way of implementing a ZK proof for a disjunction of statements. Namely, proving that an entry $[i, v_{i,1}, \ldots, v_{i,L}]$ is in DB is equivalent to computing an OR proof where the prover proves that he knows at least one of the entries. The proof in Π_{UUD} is of size independent of N. We compare our construction with related work in §7.

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

We summarize the UC framework in §A. An ideal functionality can be invoked by using one or more interfaces. In the notation in [11], the name of a message in an interface consists of three fields separated by dots, e.g., uud.read.ini in $\mathcal{F}_{\text{UUD}}$ in §3. The first field indicates the name of $\mathcal{F}_{\text{UUD}}$ 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 $\mathcal{F}_{\text{UUD}}$ and is the same in all messages that $\mathcal{F}_{\text{UUD}}$ exchanges within the same interface. The third field distinguishes between the messages that belong to the same interface. A message uud.read.ini is the incoming message received by $\mathcal{F}_{\text{UUD}}$, i.e., the message through which the interface is invoked. uud.read.end is the outgoing message sent by $\mathcal{F}_{\text{UUD}}$, i.e., the message that ends the execution of the interface. uud.read.sim is used by $\mathcal{F}_{\text{UUD}}$ to send a message to the simulator $\mathcal{S}$, and uud.read.rep is used to receive a message from $\mathcal{S}$.

In our OTAC, to ensure, when needed, that $\mathcal{F}_{\text{UUD}}$ and other functionalities receive the same input, we use the method in [11]. In [11], a functionality $\mathcal{F}_{\text{NIC}}$ for non-interactive commitments is proposed. $\mathcal{F}_{\text{NIC}}$ consists of four interfaces:

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

$\mathcal{F}_{\text{NIC}}$ can be realized by a perfectly hiding commitment scheme, such as Pedersen commitments [11]. To ensure that a party $\mathcal{P}_i$ sends the same input m to several ideal functionalities, $\mathcal{P}_i$ first uses com.commit to get a commitment *com* to m with opening *open*. Then $\mathcal{P}_i$ sends $(com, m, open)$ 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 $\mathcal{F}_{\text{NIC}}$ ensures that all the functionalities received the same input m. Our functionality $\mathcal{F}_{\text{UUD}}$ receives committed inputs as described in [11].

3 Functionality $\mathcal{F}_{\text{UUD}}$

$\mathcal{F}_{\text{UUD}}$ interacts with readers $\mathcal{R}_k$ and an updater $\mathcal{U}$. $\mathcal{F}_{\text{UUD}}$ maintains a database DB. DB consists of N entries of the form $[i, v_{i,1}, \ldots, v_{i,L}]$. $\mathcal{F}_{\text{UUD}}$ has three interfaces uud.update, uud.getdb and uud.read:

1. $\mathcal{U}$ sends the uud.update.ini message on input $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$. For all $i \in [1, N]$, $\mathcal{F}_{\text{UUD}}$ updates DB to contain value $v_{i,j}$ at position $j \in [1, L]$ of entry i. If $v_{i,j} = \perp$, no update at position j of entry i takes place.
2. $\mathcal{R}_k$ sends the uud.getdb.ini message to $\mathcal{F}_{\text{UUD}}$. $\mathcal{F}_{\text{UUD}}$ sends DB to $\mathcal{R}_k$.
3. $\mathcal{R}_k$ sends the uud.read.ini message on input a pseudonym P and a tuple $(i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]})$, where $[i, v_{i,1}, \ldots, v_{i,L}]$ is a database entry and $(com_i, open_i)$ and $(com_{i,j}, open_{i,j})_{\forall j \in [1,L]}$ are commitments and openings to i and to the values $(v_{i,1}, \ldots, v_{i,L})$. $\mathcal{F}_{\text{UUD}}$ verifies the commitments and checks that there is an entry $[i, v_{i,1}, \ldots, v_{i,L}]$ in DB. $\mathcal{F}_{\text{UUD}}$ sends $(com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]})$ to $\mathcal{U}$.

$\mathcal{F}_{\text{UUD}}$ stores counters cr_k for $\mathcal{R}_k$ and a counter cu for $\mathcal{U}$. These counters are used to check that $\mathcal{R}_k$ has the last version of DB. When $\mathcal{U}$ sends an update, cu is incremented. When $\mathcal{R}_k$ receives DB, $\mathcal{F}_{\text{UUD}}$ sets $cr_k \leftarrow cu$. When $\mathcal{R}_k$ reads DB, $\mathcal{F}_{\text{UUD}}$ checks that $cr_k = cu$, which ensures that $\mathcal{R}_k$ and $\mathcal{U}$ have the same DB.

When invoked by $\mathcal{U}$ or $\mathcal{R}_k$, $\mathcal{F}_{\text{UUD}}$ first checks the correctness of the input and aborts if it does not belong to the correct domain. $\mathcal{F}_{\text{UUD}}$ also aborts if an interface is invoked at an incorrect moment in the protocol. For example, $\mathcal{R}_k$ cannot invoke uud.read if uud.update was never invoked.

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

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

1. On input $(\text{uud.update.ini}, sid, (i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]})$ from $\mathcal{U}$:
   - Abort if $sid \notin (\mathcal{U}, sid')$.
   - For all $i \in [1, N]$ and $j \in [1, L]$, abort if $v_{i,j} \notin \mathbb{U}_v$.
   - If (sid, DB, cu) is not stored:
     - For all $i \in [1, N]$ and $j \in [1, L]$, abort if $v_{i,j} = \perp$.
     - Set $\text{DB} \leftarrow (i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ and $cu \leftarrow 0$ and store (sid, DB, cu).
   - Else:
     - For all $i \in [1, N]$ and $j \in [1, L]$, if $v_{i,j} \neq \perp$, update DB by storing $v_{i,j}$ at position j of entry i.
     - Increment cu and update DB and cu in (sid, DB, cu).
   - Create a fresh qid and store qid.
   - Send $(\text{uud.update.sim}, sid, qid, (i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]})$ to $\mathcal{S}$.

S. On input $(\text{uud.update.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if qid is not stored.
   - Delete qid.
   - Send $(\text{uud.update.end}, sid)$ to $\mathcal{U}$.

2. On input $(\text{uud.getdb.ini}, sid)$ from $\mathcal{R}_k$:

- Create a fresh qid and store $(qid, \mathcal{R}_k)$.
- Send $(\mathsf{uud.getdb.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{uud.getdb.rep}, sid, qid)$ from $\mathcal{S}$:
- Abort if $(qid', \mathcal{R}_k)$ such that $qid' = qid$ is not stored.
- If (sid, DB, cu) is not stored, set $\mathsf{DB} \leftarrow \perp$.
- Else, set $cr_k \leftarrow cu$, store $(\mathcal{R}_k, \mathsf{DB}, cr_k)$ and delete any previous tuple $(\mathcal{R}_k, \mathsf{DB}', cr'_k)$.
- Delete $(qid, \mathcal{R}_k)$.
- Send $(\mathsf{uud.getdb.end}, sid, \mathsf{DB})$ to $\mathcal{R}_k$.

3. On input $(\mathsf{uud.read.ini}, sid, P, (i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j}\rangle_{\forall j\in[1,L]}))$ from $\mathcal{R}_k$:
- Abort if $P \notin \mathbb{U}_p$, or if $[i, v_{i,1}, \ldots, v_{i,L}] \notin \mathsf{DB}$, or if $(\mathcal{R}_k, \mathsf{DB}, cr_k)$ is not stored.
- Parse the commitment com_i as $(com'_i, parcom, \mathsf{COM.Verify})$.
- Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_i, i, open_i)$.
- For all $j \in [1, L]$:
  - Parse the commitment $com_{i,j}$ as $(com'_{i,j}, parcom, \mathsf{COM.Verify})$.
  - Abort if $1 \neq \mathsf{COM.Verify}(parcom, com'_{i,j}, v_{i,j}, open_{i,j})$.
- Create a fresh qid and store $(qid, P, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}), cr_k)$.
- Send $(\mathsf{uud.read.sim}, sid, qid, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$ to $\mathcal{S}$.

S. On input $(\mathsf{uud.read.rep}, sid, qid)$ from $\mathcal{S}$:
- Abort if $(qid', P, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}), cr'_k)$ such that $qid' = qid$ is not stored or if $cr'_k \neq cu$, where cu is in (sid, DB, cu).
- Delete the record $(qid, P, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}), cr'_k)$.
- Send $(\mathsf{uud.read.end}, sid, P, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$ to $\mathcal{U}$.

$\mathcal{F}_{\mathrm{UUD}}$ guarantees anonymity and unlinkability. Namely, $\mathcal{F}_{\mathrm{UUD}}$ reveals to $\mathcal{U}$ a pseudonym P rather than the identifier $\mathcal{R}_k$. $\mathcal{R}_k$ can choose different random pseudonyms so that read operations are unlinkable. $\mathcal{F}_{\mathrm{UUD}}$ also ensures zero-knowledge, i.e. a read operation does not reveal the database entry read to $\mathcal{U}$. Additionally, $\mathcal{F}_{\mathrm{UUD}}$ guarantees unforgeability, i.e. $\mathcal{R}_k$ cannot read an entry if that entry was not stored in DB by $\mathcal{U}$.

It is straightforward to modify the $\mathsf{uud.read}$ interface to allow $\mathcal{R}_k$ to read several database entries simultaneously. This variant allows us to reduce communication rounds when $\mathcal{R}_k$ needs to read more than one entry simultaneously. $\mathcal{F}_{\mathrm{UUD}}$ can also be modified to interact with two parties such that both of them can read and update the database, or such that a party reads and updates and the other party receives read and update operations. Π_{UUD} can be easily adapted to realize the variants of $\mathcal{F}_{\mathrm{UUD}}$ discussed here.

4 Construction Π_{UUD}

4.1 Building Blocks

Subvector Commitments. A subvector commitment (SVC) scheme allows us to succinctly compute a commitment com to a vector $\mathbf{x} = (\mathbf{x}[1], \ldots, \mathbf{x}[\ell]) \in \mathcal{M}^\ell$.

A commitment com to $\mathbf{x}$ can be opened to a subvector $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$, where $I = \{i_1, \ldots, i_n\} \subseteq [1, \ell]$ is the set of indices that determine the positions of the committed vector $\mathbf{x}$ that are opened. The size of an opening w_I for $\mathbf{x}_I$ is independent of both the size of I and of the length ℓ of the committed vector. We extend the definition of SVC in [22] with algorithms to update commitments and openings.

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

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

$\mathsf{SVC.Open}(par, I, \mathbf{x})$**.** On input a vector $\mathbf{x}$ and a set $I = \{i_1, \ldots, i_n\} \subseteq [1, \ell]$, compute an opening w_I for the subvector $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$.

$\mathsf{SVC.Verify}(par, com, \mathbf{x}_I, I, w_I)$**.** Output 1 if w_I is a valid opening for the set of positions $I = \{i_1, \ldots, i_n\} \subseteq [1, \ell]$ such that $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$, where $\mathbf{x}$ is the vector committed in com. Otherwise output 0.

$\mathsf{SVC.ComUpd}(par, com, \mathbf{x}, i, x)$**.** On input a commitment com to a vector $\mathbf{x}$, output a commitment com' to a vector $\mathbf{x}'$ such that $\mathbf{x}'[i] = x$ and, for all $j \in [1, \ell] \setminus \{i\}$, $\mathbf{x}'[j] = \mathbf{x}[j]$.

$\mathsf{SVC.OpenUpd}(par, w_I, \mathbf{x}, I, i, x)$**.** On input an opening w_I for a set I valid for a commitment to a vector $\mathbf{x}$, output an opening w'_I valid for a commitment to a vector $\mathbf{x}'$ such that $\mathbf{x}'[i] = x$ and, for all $j \in [1, \ell] \setminus \{i\}$, $\mathbf{x}'[j] = \mathbf{x}[j]$.

A SVC scheme must be correct and binding [22]. In §B, we recall those properties and define correctness for the update algorithms. In §B, we also depict $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$ and $\mathcal{F}_{\mathrm{BB}}$, which we describe briefly below.

Ideal Functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$*.* Π_{UUD} uses the functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ for common reference string generation in [15]. $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CRS.Setup}}$ 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.

$\mathcal{F}_{\mathrm{ZK}}^{R}$**.** Let R be a polynomial time computable binary relation. For tuples $(wit, ins) \in R$ we call wit the witness and ins the instance. Π_{UUD} uses a functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$ for zero-knowledge. $\mathcal{F}_{\mathrm{ZK}}^{R}$ runs with multiple provers $\mathcal{P}_k$ and a verifier $\mathcal{V}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ follows the functionality for zero-knowledge in [15], except that a prover $\mathcal{P}_k$ is identified by a pseudonym P towards $\mathcal{V}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ consists of one interface $\mathsf{zk.prove}$. $\mathcal{P}_k$ uses $\mathsf{zk.prove}$ to send a witness wit, an instance ins and a pseudonym P to $\mathcal{F}_{\mathrm{ZK}}^{R}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ checks whether $(wit, ins) \in R$, and, in that case, sends ins and P to $\mathcal{V}$.

Ideal Functionality $\mathcal{F}_{\mathrm{BB}}$*.* Π_{UUD} uses the functionality $\mathcal{F}_{\mathrm{BB}}$ for a public bulletin board BB [29]. A BB is used to ensure that all the readers receive the same

version of the database, which is needed to provide unlinkability. $\mathcal{F}_{\text{BB}}$ interacts with a writer $\mathcal{W}$ and readers $\mathcal{R}_k$. $\mathcal{W}$ uses the bb.write interface to send a message m to $\mathcal{F}_{\text{BB}}$. $\mathcal{F}_{\text{BB}}$ increments a counter ct of the number of messages stored in BB and appends $[ct, m]$ to BB. $\mathcal{R}_k$ uses the bb.getbb interface on input an index i. If $i \in [1, ct]$, $\mathcal{F}_{\text{BB}}$ takes the message m stored in $[i, m]$ in BB and sends m to $\mathcal{R}_k$.

4.2 Description of Π_{UUD}

In Π_{UUD}, a SVC com is used to commit to the database DB with N entries of the form $[i, v_{i,1}, \ldots, v_{i,L}]$. To this end, com commits to a vector $\mathbf{x}$ of length $N \times L$ such that $\mathbf{x}[(i-1)L + j] = v_{i,j}$ for all $i \in [1, N]$ and $j \in [1, L]$.

In the uud.update interface, $\mathcal{U}$ uses $\mathcal{F}_{\text{BB}}$ to publish the DB and to update it. In the uud.getdb interface, any $\mathcal{R}_k$ retrieves DB and its subsequent updates through $\mathcal{F}_{\text{BB}}$. When DB is published for the first time, $\mathcal{U}$ and $\mathcal{R}_k$ run SVC.Commit to commit to DB. When DB is updated, $\mathcal{U}$ and $\mathcal{R}_k$ update com by using SVC.ComUpd. If $\mathcal{R}_k$ already stores openings w_i, $\mathcal{R}_k$ runs SVC.OpenUpd to update them.

In the uud.read interface, $\mathcal{R}_k$ uses $\mathcal{F}_{\text{ZK}}^{R}$ to prove that $(com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]})$ commit to an entry i and values $v_{i,1}, \ldots, v_{i,L}$ such that $\mathbf{x}[(i-1)L + j] = v_{i,j}$ for all $j \in [1, L]$, where $\mathbf{x}$ is the vector committed in com. R requires proving knowledge of an opening w_I for the set $I = \{(i-1)L + 1, \ldots, (i-1)L + L\}$ of positions where the values for the database entry i are stored. $\mathcal{R}_k$ runs SVC.Open to compute w_I if it is not stored. R also requires a proof to associate i with I, which we denote by $I = f(i)$, where f is a function that on input i outputs the indices $I = \{(i-1)L + 1, \ldots, (i-1)L + L\}$. In §5, we show a concrete UC ZK proof for R for the SVC scheme in [22].

Description of Π_{UUD}. N denotes the database size and L the size of any entry. The function $f(i) = ((i-1)L + 1, \ldots, (i-1)L + L)$ maps $i \in [1, N]$ to a set of indices where the database entry i is stored. The universe of values $\mathbb{U}_v$ is given by the message space of the SVC scheme.

1. On input $(\text{uud.update.ini}, sid, (i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]})$, $\mathcal{U}$ does the following:
   - If $(sid, par, com, \mathbf{x}, cu)$ is not stored:
     - $\mathcal{U}$ uses crs.get to obtain the parameters par from $\mathcal{F}_{\text{CRS}}^{\text{SVC.Setup}}$. To compute par, $\mathcal{F}_{\text{CRS}}^{\text{SVC.Setup}}$ runs $\text{SVC.Setup}(1^k, N \times L)$.
     - $\mathcal{U}$ initializes a counter $cu \leftarrow 0$ and a vector $\mathbf{x}$ such that $\mathbf{x}[(i-1)L+j] = v_{i,j}$ for all $i \in [1, N]$ and $j \in [1, L]$. $\mathcal{U}$ runs $com \leftarrow \text{SVC.Commit}(par, \mathbf{x})$ and stores $(sid, par, com, \mathbf{x}, cu)$.
   - Else:
     - $\mathcal{U}$ sets $cu' \leftarrow cu + 1$, $\mathbf{x}' \leftarrow \mathbf{x}$ and $com' \leftarrow com$. For all $i \in [1, N]$ and $j \in [1, L]$ such that $v_{i,j} \neq \perp$, $\mathcal{U}$ computes $com' \leftarrow \text{SVC.ComUpd}(par, com', \mathbf{x}', (i-1)L + j, v_{i,j})$ and sets $\mathbf{x}'[(i-1)L + j] \leftarrow v_{i,j}$.
     - $\mathcal{U}$ replaces the stored tuple $(sid, par, com, \mathbf{x}, cu)$ by $(sid, par, com', \mathbf{x}', cu')$.
   - $\mathcal{U}$ uses the bb.write interface to append $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ to the bulletin board.

– $\mathcal{U}$ outputs (uud.update.end, sid).

2. On input (uud.getdb.ini, sid), $\mathcal{R}_k$ does the following:
   – If $(sid, par, com, \mathbf{x}, cr_k)$ is not stored, $\mathcal{R}_k$ obtains par from $\mathcal{F}_{\text{CRS}}^{\mathsf{SVC.Setup}}$ and initializes a counter $cr_k \leftarrow 0$.
   – $\mathcal{R}_k$ increments cr_k and uses the bb.getbb interface to read the message $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ stored at position cr_k in the bulletin board. $\mathcal{R}_k$ continues incrementing the counter and reading the bulletin board until the returned message is $\perp$.
   – $\mathcal{R}_k$ sets a tuple $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$, such that $v_{i,j}$ (for $i \in [1, N]$ and $j \in [1, L]$) is the most recent update for position j of the database entry i received from the bulletin board. If $(sid, par, com, \mathbf{x}, cr_k)$ is not stored, $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ contains the current database to be used to set $\mathbf{x}$, else it contains the update that needs to be performed on $\mathbf{x}$.
   – For $i = 1$ to N, if (sid, i, w_I) is stored, $\mathcal{R}_k$ sets $\mathbf{x}' \leftarrow \mathbf{x}$ and $w_I' \leftarrow w_I$ and, for all $i \in [1, N]$ and $j \in [1, L]$ such that $v_{i,j} \neq \perp$, $w_I' \leftarrow \mathsf{SVC.OpenUpd}(par, w_I', \mathbf{x}', I, (i-1)L + j, v_{i,j})$ and $\mathbf{x}'[(i-1)L + j] = v_{i,j}$. $\mathcal{R}_k$ replaces (sid, i, w_I) by (sid, i, w_I').
   – $\mathcal{R}_k$ performs the same operations as $\mathcal{U}$ to set or update com and $\mathbf{x}$, and stores a tuple $(sid, par, com, \mathbf{x}, cr_k)$.
   – $\mathcal{R}$ outputs (uud.getdb.end, $sid, \mathbf{x}$).

3. On input (uud.read.ini, $sid, P, (i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j}\rangle_{\forall j \in [1,L]})$):
   – $\mathcal{R}_k$ parses com_i as $(com_i', parcom, \mathsf{COM.Verify})$.
   – $\mathcal{R}_k$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, com_i', i, open_i)$.
   – For all $j \in [1, L]$:
     • $\mathcal{R}_k$ parses the commitment $com_{i,j}$ as $(com_{i,j}', parcom, \mathsf{COM.Verify})$.
     • $\mathcal{R}_k$ aborts if $1 \neq \mathsf{COM.Verify}(parcom, com_{i,j}', v_{i,j}, open_{i,j})$.
   – $\mathcal{R}_k$ takes the stored tuple $(sid, par, com, \mathbf{x}, cr_k)$ and aborts if, for any $j \in [1, L]$, $\mathbf{x}[(i-1)L + j] \neq v_{i,j}$.
   – If (sid, i, w_I) is not stored, $\mathcal{R}_k$ computes $I \leftarrow f(i)$, executes the algorithm $w_I \leftarrow \mathsf{SVC.Open}(par, I, \mathbf{x})$ and stores (sid, i, w_I).
   – $\mathcal{R}_k$ sets the witness $wit \leftarrow (w_I, I, i, open_i, \langle v_{i,j}, open_{i,j}\rangle_{\forall j \in [1,L]})$ and the instance $ins \leftarrow (par, com, parcom, com_i', \langle com_{i,j}'\rangle_{\forall j \in [1,L]}, cr_k)$. $\mathcal{R}_k$ uses zk.prove to send wit, ins and P to $\mathcal{F}_{\text{ZK}}^R$. The relation R is

$$
\begin{aligned}
R = \{&(wit, ins) : \\
&1 = \mathsf{COM.Verify}(parcom, com_i', i, open_i) \wedge \\
&\langle 1 = \mathsf{COM.Verify}(parcom, com_{i,j}', v_{i,j}, open_{i,j})\rangle_{\forall j \in [1,L]} \wedge \\
&1 = \mathsf{SVC.Verify}(par, com, \langle v_{i,j}\rangle_{\forall j \in [1,L]}, I, w_I) \ \wedge \ I = f(i)\}
\end{aligned}
$$

   – $\mathcal{U}$ receives P and $ins = (par', com', parcom, com_i', \langle com_{i,j}'\rangle_{\forall j \in [1,L]}, cr_k)$ from $\mathcal{F}_{\text{ZK}}^R$.
   – $\mathcal{U}$ takes the stored tuple $(sid, par, com, \mathbf{x}, cu)$ and aborts if $cr_k \neq cu$, or if $par' \neq par$, or if $com' \neq com$.
   – $\mathcal{U}$ sets $com_i \leftarrow (com_i', parcom, \mathsf{COM.Verify})$ and $\langle com_{i,j} \leftarrow (com_{i,j}', parcom, \mathsf{COM.Verify})\rangle_{\forall j \in [1,L]}$. ($\mathsf{COM.Verify}$ is in the description of R.)

– $\mathcal{U}$ outputs $(\mathsf{uud.read.end}, sid, P, (com_i, \langle com_{i,j}\rangle_{\forall j\in[1,L]}))$.

Theorem 1. Π_{UUD} *securely realizes* $\mathcal{F}_{\mathrm{UUD}}$ *in the* $(\mathcal{F}_{\mathrm{CRS}}^{\mathsf{SVC.Setup}}, \mathcal{F}_{\mathrm{BB}}, \mathcal{F}_{\mathrm{ZK}}^{R})$*-hybrid model if the SVC scheme is binding.*

When $\mathcal{R}_k$ is corrupt, the binding property of the SVC scheme guarantees that the adversary is not able to open the VC *com* to a value $v_{i,j}$ if that value was not previously committed by $\mathcal{U}$ at position $(i-1)L+j$. We analyze in detail the security of Π_{UUD} in §C.

5 Instantiation and Efficiency Analysis

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

Cube Diffie-Hellman (CubeDH) assumption. Let $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$ and $x \leftarrow \mathbb{Z}_p$. Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g^x, \tilde{g}^x)$, for any p.p.t. adversary $\mathcal{A}$, $\Pr[e(g, \tilde{g})^{x^3} \leftarrow \mathcal{A}(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g^x, \tilde{g}^x)] \leq \epsilon(k)$.

SVC scheme. We use a SVC scheme secure under the CubeDH assumption [22], which we extend with update algorithms for commitments and openings.

$\mathsf{SVC.Setup}(1^k, \ell)$**.** Generate $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$. For all $i \in [1, \ell]$, pick $z_i \leftarrow \mathbb{Z}_p$ and compute $g_i \leftarrow g^{z_i}$ and $\tilde{g}_i \leftarrow \tilde{g}^{z_i}$. For all $i \in [1, \ell]$ and $i' \in [1, \ell]$ such that $i \neq i'$, compute $h_{i,i'} \leftarrow g^{z_i z_{i'}}$. Output $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \{g_i, \tilde{g}_i\}_{\forall i\in[1,\ell]}, \{h_{i,i'}\}_{\forall i,i'\in[1,\ell], i\neq i'})$.

$\mathsf{SVC.Commit}(par, \mathbf{x})$**.** Output $com = \prod_{i=1}^{\ell} g_i^{\mathbf{x}[i]}$.

$\mathsf{SVC.Open}(par, I, \mathbf{x})$**.** Output $w_I = \prod_{i\in I}\prod_{i'\notin I} h_{i,i'}^{\mathbf{x}[i']}$.

$\mathsf{SVC.Verify}(par, com, \mathbf{x}_I, I, w_I)$**.** Parse I as $\{i_1, \ldots, i_n\} \subseteq [1, \ell]$ and $\mathbf{x}_I$ as $(\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$. Output 1 if

$$e\left(\frac{com}{\prod_{i\in I} g_i^{\mathbf{x}[i]}}, \prod_{i\in I}\tilde{g}_i\right) = e(w_I, \tilde{g})$$

$\mathsf{SVC.ComUpd}(par, com, \mathbf{x}, i, x)$**.** Output $com' = com \cdot g_i^{x-\mathbf{x}[i]}$.

$\mathsf{SVC.OpenUpd}(par, w_I, \mathbf{x}, I, i, x)$**.** If $i \in I$, output w_I, else $w_I' = w_I \cdot \prod_{j\in I} h_{j,i}^{x-\mathbf{x}[i]}$.

Commitments. We use Pedersen commitments [26], which we recall in §D.

Signatures. We use the structure-preserving signature (SPS) scheme in [2]. In SPSs, the public key, the messages, and the signatures are group elements in $\mathbb{G}$ and $\tilde{\mathbb{G}}$, and verification must consist purely in the checking of pairing product equations. We employ SPSs to sign group elements, while still supporting efficient ZK proofs of signature possession. In this SPS scheme, a elements in $\mathbb{G}$ and b elements in $\tilde{\mathbb{G}}$ are signed.

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

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

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

UC ZK proof. To instantiate $\mathcal{F}_{\mathrm{ZK}}^{R}$, we use the scheme in [12]. In [12], a UC ZK protocol proving knowledge of exponents $(w_1, \ldots, w_n)$ that satisfy the formula $\phi(w_1, \ldots, w_n)$ is described as

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

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

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

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

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

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

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

In R, we need to prove that $I = f(i) = \{(i-1)L+1, \ldots, (i-1)L+L\}$, i.e., we need to prove that the set I of positions opened contains the positions where the database entry i is stored. To prove this statement, the public parameters of the SVC scheme are extended with SPSs that bind g^i with $(g_{(i-1)L+1}, \tilde{g}_{(i-1)L+1}, \ldots, g_{(i-1)L+L}, \tilde{g}_{(i-1)L+L})$, i.e., i is bound with the bases of the positions in I. Given the parameters $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \{g_i, \tilde{g}_i\}_{\forall i \in [1,\ell]}, \{h_{i,i'}\}_{\forall i,i' \in [1,\ell], i \neq i'})$, we create the key pair $(sk, pk) \leftarrow \mathsf{KeyGen}(\langle p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}\rangle, L+1, L+1)$ and, for $i \in [1, \ell]$, we compute $s_i \leftarrow \mathsf{Sign}(sk, \langle g_{(i-1)L+1}, \ldots, g_{(i-1)L+L}, g^i, \tilde{g}_{(i-1)L+1}, \ldots, \tilde{g}_{(i-1)L+L}, \tilde{g}^{sid}\rangle)$, where sid is the session identifier. We remark that these signatures do not need to be updated when the database is updated.

Let $(U_1, \ldots, U_{L+1}, V, W_1, \ldots, W_{L+1}, Z)$ be the public key of the signature scheme. Let (R, S, T) be a signature on $(g_{(i-1)L+1}, \ldots, g_{(i-1)L+L}, g^i, \tilde{g}_{(i-1)L+1}, \ldots, \tilde{g}_{(i-1)L+L}, \tilde{g}^{sid})$. Let (g, h) be the parameters of the Pedersen commitment scheme. R involves proofs about secret bases and we use the transformation described above for those proofs. The base h is also used to randomize secret bases in $\mathbb{G}$, and another base $\tilde{h} \leftarrow \tilde{\mathbb{G}}$ is added to randomize bases in $\tilde{\mathbb{G}}$. Following the notation in [12], we describe the proof as follows.

$$
\begin{aligned}
&\nvDash i, open_i, \langle v_{i,j}, open_{i,j}, g_{(i-1)L+j}, \tilde{g}_{(i-1)L+j}\rangle_{\forall j \in [1,L]}, w_I, R, S, T : \\
&\quad com'_i = g^i h^{open_i} \ \wedge \ \langle com'_{i,j} = g^{v_{i,j}} h^{open_{i,j}}\rangle_{\forall j \in [1,L]} \ \wedge && (3)\\
&\quad e(R,V)e(S,\tilde{g})(\prod_{j \in [1,L]} e(g_{(i-1)L+j}, W_j))e(g, W_{L+1})^i e(g,Z)^{-1} = 1 \ \wedge && (4)\\
&\quad e(R,T)(\prod_{j \in [1,L]} e(U_j, \tilde{g}_{(i-1)L+j}))e(U_{L+1}, \tilde{g}^{sid})e(g,\tilde{g})^{-1} = 1 \ \wedge && (5)\\
&\quad e\left(\frac{com}{\prod_{j \in [1,L]} g_{(i-1)L+j}^{v_{i,j}}}, \prod_{j \in [1,L]} \tilde{g}_{(i-1)L+j}\right) = e(w_I, \tilde{g}) && (6)
\end{aligned}
$$

Equation 3 proves knowledge of the openings of the Pedersen commitments com'_i and $\langle com'_{i,j}\rangle_{\forall j \in [1,L]}$. Equation 4 and Equation 5 prove knowledge of a signature (R, S, T) on a message $(g_{(i-1)L+1}, \ldots, g_{(i-1)L+L}, g^i, \tilde{g}_{(i-1)L+1}, \ldots, \tilde{g}_{(i-1)L+L}, \tilde{g}^{sid})$. Equation 6 proves that the values $\langle v_{i,j}\rangle_{\forall j \in [1,L]}$ in $\langle com'_{i,j}\rangle_{\forall j \in [1,L]}$ are equal to the values committed in the positions $I = f(i) = \{(i-1)L+1, \ldots, (i-1)L+L\}$ of the vector commitment com. We remark that, in comparison to the relation R in §4.2, in the witness we replace I by the secret bases $\langle g_{(i-1)L+j}, \tilde{g}_{(i-1)L+j}\rangle_{\forall j \in [1,L]}$, from which I can be derived. Like in R, the positions $j \in [1, L]$ inside the database entry i of the values $v_{i,j}$ committed in $com'_{i,j}$ are revealed to the verifier.

When a range of indices $[i_{min}, i_{max}]$ stores always (i.e., even after database updates) the same tuple $[v_{i,1}, \ldots, v_{i,L}]$, we can improve storage efficiency as follows. We compute signatures on tuples $(g_{(i'-1)L+1}, \ldots, g_{(i'-1)L+L}, g^{i_{min}}, g^{i_{max}}, \tilde{g}_{(i'-1)L+1}, \ldots, \tilde{g}_{(i'-1)L+L}, \tilde{g}^{sid})$ that bind all the indices in $[i_{min}, i_{max}]$ with the bases for the positions where the tuple is stored. (i' is used to denote the position in the SVC where the tuple is stored.) Then, in the UC ZK proof for R, we add

a range proof to prove that $i \in [i_{min}, i_{max}]$, where i is committed in com'_i, to prove that we are opening the correct subvector for i.

Efficiency Analysis. We analyze the storage, communication, and computation costs of our instantiation of Π_{UUD}.

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

Communication Cost. In the uud.update interface, $\mathcal{U}$ sends the tuples $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$, which are retrieved by $\mathcal{R}_k$ in the uud.getdb interface. The communication cost is linear in the number of entries updated, except for the first update in which all entries must be initialized. In the uud.read interface, $\mathcal{R}_k$ sends an instance and a ZK proof to $\mathcal{U}$. The size of the witness and of the instance grows linearly with L but is independent of N. In conclusion, after the first update phase, the communication cost does not depend on N.

Computation Cost. In the uud.update and uud.getdb interfaces, $\mathcal{U}$ and $\mathcal{R}_k$ update *com* with cost linear in the number t of updates, except for the first update where all the positions are initialized. $\mathcal{R}_k$ also updates the stored openings w_I with cost linear in $t \times L$. In the uud.read interface, if w_I is not stored, $\mathcal{R}_k$ computes it with cost that grows linearly with $N \times L$. However, if w_I is stored, the computation cost of the proof grows linearly with L but is independent of N.

It is possible to defer opening updates to the uud.read interface, so as to only update openings that are actually needed to compute ZK proofs. Thanks to that, the computation cost in the uud.getdb interface is independent of N. In the uud.read interface, if w_I is stored but needs to be updated, the computation cost grows linearly with $t \times L$ but it is independent of N. The only overhead introduced by deferring opening updates is the need to store the tuples $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$.

In summary, after initialization of *com* and the openings w_I, the communication and computation costs are independent of N, so in terms of communication and computation our instantiation of Π_{UUD} is practical for large databases.

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

To compute UC ZK proofs for $\mathcal{R}_k$, we use the compiler in [12]. The public parameters of the proof system contain a public key of the Paillier encryption scheme, the parameters for a multi-integer commitment scheme, and the specification of a DSA group. (We refer to [12] for a description of how those primitives are used in the compiler.) The cost of a proof depends on the number of elements

Table 1. Π_{UUD} execution times in seconds

Interface	$N = 50$ $L = 10$	$N = 100$ $L = 5$	$N = 100$ $L = 15$	$N = 150$ $L = 10$	$N = 200$ $L = 5$	$N = 200$ $L = 15$
Setup	61.35	61.37	553.71	554.52	249.61	2205.72
Update	0.0001	0.0002	0.0001	0.0001	0.0002	0.0001
Getdb	0.0004	0.0004	0.0006	0.0004	0.0003	0.0004
Computation of com	0.0371	0.0350	0.1093	0.1035	0.0707	0.2145
One value update of com	1.59e-05	1.59e-05	1.71e-05	1.99e-05	1.69e-05	1.59e-05
Computation of w_I	0.3491	0.1753	1.5659	1.0485	0.3513	3.1330
One value update of w_I	0.0002	0.0001	0.0003	0.0002	0.0001	0.0003
Read proof (1024 bit key)	3.6737	2.1903	4.9621	3.6811	2.1164	5.0268
Read proof (2048 bit key)	16.6220	10.6786	25.2909	16.8730	9.8916	23.4896

in the witness and on the number of equations composed by Boolean ANDs. The computation cost for the prover of a Σ-protocol for $\mathcal{R}_k$ involves one evaluation of each of the equations and one multiplication per value in the witness. The compiler in [12] extends a Σ-protocol and requires, additionally, a computation of a multi-integer commitment that commits to the values in the witness, an evaluation of a Paillier encryption for each of the values in the witness, a Σ-protocol to prove that the commitment and the encryptions are correctly generated, and 3 exponentiations in the DSA group. The computation cost for the verifier, as well as the communication cost, also depends on the number of values in the witness, and on the number of equations. Therefore, as the number of values in the witness and the number of equations is independent of N in our proof for relation R, the computation and communication costs of our proof do not depend on N.

Table 1 lists the execution times of the uud.update and uud.getdb interfaces, the computation costs for read proofs, and the costs for computing and updating w_I and com, in our implementation, in seconds. The execution times of the interfaces of the protocol have been evaluated against the size N of the database, and the size of each entry L of the database. In the setup phase, the public parameters of all the building blocks are computed, and the database is set up by computing com. In the second and third rows of Table 1, we depict the execution times for the uud.update and uud.getdb interfaces for the updater $\mathcal{U}$, and a reader $\mathcal{R}_k$ respectively, after the update of a single value in an entry of the database. In the fourth row of Table 1, we show the cost of computing com, and as can be seen from these values, the computation times for com depend on the total number of values $N \times L$ in the database. However, the cost of updating com is very small, and linear in the number t of updates, and this in turn results in small computation costs for the uud.update interface, independent of N. (As required by our applications in §6, the committed vector that we use consists of small numbers rather than random values in $\mathbb{Z}_p$.) The cost of computation of w_I also depends on the total number of values $N \times L$ in the database, while

the cost of updating w_I is linear in $t \times L$, and thus the execution times for the uud.getdb interface (which involves the updates of stored witnesses, in addition to the update of *com* as in the case of the uud.update interface) are also small.

In the last two rows of Table 1, we show the computation costs for a read proof. These values have been evaluated against varying key lengths for the Paillier encryption scheme used in the proof system in our instantiation of Π_{UUD}. The execution times for the read interface depend greatly upon the security parameters of the Paillier encryption scheme, and increase linearly with the entry size of the database L. However, the execution times are independent of the database size N.

6 Modular Design with $\mathcal{F}_{\mathrm{UUD}}$ and Application to OTAC

First, we show how to describe a protocol modularly by using $\mathcal{F}_{\mathrm{UUD}}$ as building block. As an example, consider the following relation R':

$$R' = \{(wit, ins) : [i, v_{i,1}, \ldots, v_{i,L}] \in \mathsf{DB} \ \wedge \ 1 = \mathrm{pred}_i(i) \ \wedge \ \langle 1 = \mathrm{pred}_j(v_{i,j})\rangle_{\forall j \in [1,L]}\}$$

where the witness is $wit = (i, \langle v_{i,j}\rangle_{\forall j \in [1,L]})$ and the instance is $ins = \mathsf{DB}$. pred_i and pred_j represent predicates that i and $\langle v_{i,j}\rangle_{\forall j \in [1,L]}$ must fulfill, e.g., predicates that require i and $\langle v_{i,j}\rangle_{\forall j \in [1,L]}$ to belong to a range or set of values.

We would like to construct a ZK protocol for R' between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ that uses different functionalities $\mathcal{F}_{\mathrm{ZK}}^{R_i}$ and $\langle \mathcal{F}_{\mathrm{ZK}}^{R_j}\rangle_{\forall j \in [1,L]}$ to prove each of the statements in R'. We show how this protocol is constructed by using $\mathcal{F}_{\mathrm{UUD}}$ and $\mathcal{F}_{\mathrm{NIC}}$ as building blocks.

1. On input DB, $\mathcal{V}$ uses the uud.update interface to send DB to $\mathcal{F}_{\mathrm{UUD}}$.
2. $\mathcal{P}$ uses the uud.getdb interface to retrieve DB.
3. On input $(i, v_{i,1}, \ldots, v_{i,L})$ and P, $\mathcal{P}$ checks that $[i, v_{i,1}, \ldots, v_{i,L}] \in \mathsf{DB}$.
4. $\mathcal{P}$ runs the com.setup interface of $\mathcal{F}_{\mathrm{NIC}}$. $\mathcal{P}$ uses the com.commit interface of $\mathcal{F}_{\mathrm{NIC}}$ on input i to obtain a commitment com_i with opening $open_i$. Similarly, from $j = 1$ to L, $\mathcal{P}$ obtains from $\mathcal{F}_{\mathrm{NIC}}$ commitments $com_{i,j}$ to $v_{i,j}$ with opening $open_{i,j}$.
5. $\mathcal{P}$ uses the uud.read interface to send the tuple $(P, i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j}\rangle_{\forall j \in [1,L]})$ to $\mathcal{F}_{\mathrm{UUD}}$, which sends $(P, com_i, \langle com_{i,j}\rangle_{\forall j \in [1,L]})$ to $\mathcal{V}$.
6. $\mathcal{V}$ runs the com.setup interface of $\mathcal{F}_{\mathrm{NIC}}$. $\mathcal{V}$ uses the com.validate interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate the commitments com_i and $\langle com_{i,j}\rangle_{\forall j \in [1,L]}$. Then $\mathcal{V}$ stores P, com_i and $\langle com_{i,j}\rangle_{\forall j \in [1,L]}$ and sends a message to $\mathcal{P}$ to acknowledge the receipt of the commitments.
7. $\mathcal{P}$ parses the commitment com_i as $(com'_i, parcom, \mathsf{COM.Verify})$. $\mathcal{P}$ sets the witness $wit \leftarrow (i, open_i)$ and the instance $ins \leftarrow (parcom, com'_i)$. $\mathcal{P}$ uses the zk.prove interface to send wit, ins and P to $\mathcal{F}_{\mathrm{ZK}}^{R_i}$, where R_i is

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

8. $\mathcal{V}$ receives *ins* from $\mathcal{F}_{\text{ZK}}^{R_i}$. $\mathcal{V}$ checks that pseudonym and the commitment in *ins* are equal to the stored pseudonym and commitment com_i. If the commitments are equal, the binding property guaranteed by $\mathcal{F}_{\text{NIC}}$ ensures that $\mathcal{F}_{\text{UUD}}$ and $\mathcal{F}_{\text{ZK}}^{R_i}$ received as input the same position i.
9. The last two steps are replicated to prove, for $j = 1$ to L, that $v_{i,j}$ fulfills $1 = \text{pred}_j(v_{i,j})$ by using $\mathcal{F}_{\text{ZK}}^{R_j}$.

Application to OTAC. In §E, we depict our functionality $\mathcal{F}_{\text{OTAC}}$ and our construction Π_{OTAC}. $\mathcal{F}_{\text{OTAC}}$ consists of the following interfaces:

1. The sender $\mathcal{U}$ uses the otac.init interface to send the messages $\langle m_n \rangle_{n=1}^{N}$.
2. The receiver $\mathcal{R}_k$ uses the otac.retrieve interface to retrieve N.
3. $\mathcal{U}$ uses the otac.policy interface to send (or update) the policies $\langle \mathsf{ACP}_n \rangle_{n=1}^{N}$ and the relation R_{ACP} to $\mathcal{F}_{\text{OTAC}}$.
4. $\mathcal{R}_k$ uses the otac.getpol interface to obtain $\langle \mathsf{ACP}_n \rangle_{n=1}^{N}$ and R_{ACP}.
5. $\mathcal{R}_k$ uses the otac.transfer to send a choice i and a witness *wit* to $\mathcal{F}_{\text{OTAC}}$. If $(wit, \mathsf{ACP}_i) \in R_{\mathsf{ACP}}$, $\mathcal{F}_{\text{OTAC}}$ sends m_i to $\mathcal{R}_k$.

$\mathcal{F}_{\text{OTAC}}$ follows previous OTAC functionalities [8] but introduces two main modifications. First, it splits the initialization interface into two interfaces: otac.init and otac.policy, to enable $\mathcal{U}$ to make policy updates. Second, previous functionalities include an issuance phase where an issuer certifies $\mathcal{R}_k$ attributes, whereas $\mathcal{F}_{\text{OTAC}}$ does not have it. Instead, in the transfer phase of $\mathcal{F}_{\text{OTAC}}$, $\mathcal{R}_k$ must provide a witness *wit* such that $(wit, \mathsf{ACP}_i) \in R_{\mathsf{ACP}}$. *wit* could contain, e.g., signatures from an issuer on $\mathcal{R}_k$ attributes, but in general any data required by R_{ACP}.

Π_{OTAC} uses $\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{UUD}}$, $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$, $\mathcal{F}_{\text{BB}}$ and a functionality $\mathcal{F}_{\text{NYM}}$ for a secure pseudonymous channel. $\mathcal{F}_{\text{OT}}$ and $\mathcal{F}_{\text{NYM}}$ are depicted in §E.2. $\mathcal{F}_{\text{OT}}$ is used to implement the otac.init and otac.retrieve interfaces, as well as to allow $\mathcal{R}_k$ to obtain messages obliviously in the otac.transfer interface. $\mathcal{F}_{\text{OT}}$ receives a committed input to the choice i. It is generally straightforward to adapt existing UC OTs to realize our $\mathcal{F}_{\text{OT}}$ with committed inputs.

To implement access control, Π_{OTAC} uses $\mathcal{F}_{\text{UUD}}$, $\mathcal{F}_{\text{BB}}$ and $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$. In the otac.policy interface, $\mathcal{U}$ uses $\mathcal{F}_{\text{UUD}}$ to store the policies, and $\mathcal{U}$ uses $\mathcal{F}_{\text{BB}}$ to store the relation R_{ACP}. In the otac.getpol interface, $\mathcal{R}_k$ retrieves the policies and the relation from $\mathcal{F}_{\text{UUD}}$ and $\mathcal{F}_{\text{BB}}$.

In the otac.transfer interface, $\mathcal{R}_k$ reads the policy $\mathsf{ACP}_i = \langle v_{i,j} \rangle_{\forall j \in [1,L]}$ for her choice i by using $\mathcal{F}_{\text{UUD}}$. To do so, $\mathcal{R}_k$ obtains commitments com_i and $\langle com_{i,j} \rangle_{\forall j \in [1,L]}$ to i and to the values $\langle v_{i,j} \rangle_{\forall j \in [1,L]}$ that represent the policy from $\mathcal{F}_{\text{NIC}}$. $\langle com_{i,j} \rangle_{\forall j \in [1,L]}$ are sent as input to $\mathcal{F}_{\text{ZK}}^{R_{\mathsf{ACP}'}}$ so that $\mathcal{R}_k$ proves fulfilment of the policy. com_i is sent as input to $\mathcal{F}_{\text{OT}}$ to obtain the message m_i.

$R_{\mathsf{ACP}'}$ is a modification of R_{ACP}. In $R_{\mathsf{ACP}'}$, the instance $\mathsf{ACP}_i = \langle v_{i,j} \rangle_{\forall j \in [1,L]}$ of R_{ACP} is replaced by $\langle com_{i,j} \rangle_{\forall j \in [1,L]}$, while the witness is extended to contain $wit' \leftarrow (wit, \langle v_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]})$. I.e., the instance in $R_{\mathsf{ACP}'}$ contains commitments to the policy rather than the policy itself, which allows $\mathcal{R}_k$ to hide what policy is being used from $\mathcal{U}$.

Π_{OTAC} supports any policies that can be represented by tuples of values. In [23], policies are represented by branching programs. If the ZK proof for a

policy committed in $\langle com_{i,j}\rangle_{\forall j\in[1,L]}$ requires $\mathcal{R}_k$ to hide the indices j that are used from the policy, the proof for $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$ can follow an approach similar to Π_{UUD} to compute an OR proof. I.e., the values committed in $\langle com_{i,j}\rangle_{\forall j\in[1,L]}$ can be committed in a vector commitment, and then a position of the vector commitment can be opened, without disclosing what position is opened.

Π_{OTAC} uses $\mathcal{F}_{\text{OT}}$ as building block. Thanks to that, it can be instantiated with multiple OT schemes and their security does not need to be reanalyzed. Moreover, $\mathcal{U}$ can update the access control policies at any time without restarting or modifying the OT used as building block, and without using a revocation mechanism to disallow old policies. Additionally, when many messages are associated with the same policy, we can use our optimization in §5 so that the policies in the database do not need to be replicated.

7 Related Work

Vector Commitments (VC). SVC schemes are an extension of VC schemes [25,16]. While an opening in SVC allows us to open a subset of positions, in VC it allows us to open one position. Our construction could be based on a VC scheme. In that case, the efficiency of the UC ZK proof for the uud.read interface would decrease because we would need to prove knowledge of L openings. However, storage cost would improve because the public parameter size of some VC schemes grows linearly with the vector length. We note that [22,6] propose SVC with short parameters based on hidden order groups, but those constructions are better suited for bit vectors.

Polynomial commitments (PC) allow a committer to commit to a polynomial and open the commitment to an evaluation of the polynomial. PC can be used as vector commitments by committing to a polynomial that interpolates the vector to be committed. The PC construction in [21] has the disadvantage that efficient updates cannot be computed without knowledge of the trapdoor. A further generalization of vector commitments and polynomial commitments are functional commitments [24,22].

OTAC. Our OTAC is adaptive, i.e., $\mathcal{R}_k$ can choose an index i after receiving other messages previously. In [5], an oblivious language-based envelope protocol (OLBE) is proposed based on smooth projective hash functions. OLBE can be viewed as a non-adaptive OTAC.

Our OTAC is stateless, i.e. fulfilment of a policy by $\mathcal{R}_k$ does not depend in the history of messages accessed by $\mathcal{R}_k$. In [17], a stateful OTAC is proposed where policies are defined by a directed graph that determines the possible states of $\mathcal{R}_k$, the transitions between states and the messages that can be accessed at each stage. Price oblivious transfer protocols (POT) [3,28,9] require the user to pay a price for each message. Typically, they involve a prepaid method, where $\mathcal{R}_k$ makes a deposit and later subtracts the prices paid from it without revealing the current funds or the prices paid. Those stateful OTAC where not designed modularly. Recently, a modular POT protocol was proposed [18] based on an

updatable database without unlinkability [19]. Our OTAC differs from it in that it provides unlinkability to $\mathcal{R}_k$ and in that it considers more complex policies expressed by tuples of values, while in POT the policy is simply the message price. Additionally, our OTAC can improve storage efficiency when the same policy is applied to several messages.

Our OTAC reveals the policies to $\mathcal{R}_k$. In [10,7], OTAC with hidden policies are proposed. Our approach based on SVC cannot be followed to design modularly OTAC with hidden policies that allow for policy updates.

8 Conclusion and Future Work

We propose an OTAC protocol that can be instantiated with any secure OT scheme, that allows for policy updates and that can reduce storage cost when a policy is associated to a group of messages. As building block, we define and construct an unlinkable updatable database. Our construction based on subvector commitments allows efficient policy updates. As future work, we plan to extend our OTAC protocol to consider stateful policies.

References

1. Abe, M., Camenisch, J., Dubovitskaya, M., Nishimaki, R.: Universally composable adaptive oblivious transfer (with access control) from standard assumptions. In: DIM'13, Proceedings of the 2013 ACM Workshop on Digital Identity Management. pp. 1–12
2. Abe, M., Groth, J., Haralambiev, K., Ohkubo, M.: Optimal structure-preserving signatures in asymmetric bilinear groups. In: CRYPTO 2011. pp. 649–666
3. Aiello, W., Ishai, Y., Reingold, O.: Priced oblivious transfer: How to sell digital goods. In: EUROCRYPT 2001. pp. 119–135
4. Akinyele, J.A., Garman, C., Miers, I., Pagano, M.W., Rushanan, M., Green, M., Rubin, A.D.: Charm: a framework for rapidly prototyping cryptosystems. J. Cryptographic Engineering **3**(2), 111–128 (2013)
5. Blazy, O., Chevalier, C., Germouty, P.: Adaptive oblivious transfer and generalization. In: ASIACRYPT 2016. pp. 217–247
6. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to iops and stateless blockchains. In: CRYPTO
7. Camenisch, J., Dubovitskaya, M., Enderlein, R.R., Neven, G.: Oblivious transfer with hidden access control from attribute-based encryption. In: Security and Cryptography for Networks - 8th International Conference, SCN 2012. pp. 559–579
8. Camenisch, J., Dubovitskaya, M., Neven, G.: Oblivious transfer with access control. In: ACM Conference on Computer and Communications Security, CCS 2009. pp. 131–140
9. Camenisch, J., Dubovitskaya, M., Neven, G.: Unlinkable priced oblivious transfer with rechargeable wallets. In: Financial Cryptography and Data Security, FC 2010. pp. 66–81
10. Camenisch, J., Dubovitskaya, M., Neven, G., Zaverucha, G.M.: Oblivious transfer with hidden access control policies. In: PKC 2011. pp. 192–209

11. Camenisch, J., Dubovitskaya, M., Rial, A.: UC commitments for modular protocol design and applications to revocation and attribute tokens. In: CRYPTO 2016. pp. 208–239 (2016)
12. Camenisch, J., Krenn, S., Shoup, V.: A framework for practical universally composable zero-knowledge protocols. In: ASIACRYPT 2011. pp. 449–467
13. Camenisch, J., Lehmann, A., Neven, G., Rial, A.: Privacy-preserving auditing for attribute-based credentials. In: ESORICS 2014. pp. 109–127
14. Camenisch, J., Neven, G., Shelat, A.: Simulatable adaptive oblivious transfer. In: EUROCRYPT 2007. pp. 573–590
15. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001. pp. 136–145 (2001)
16. Catalano, D., Fiore, D.: Vector commitments and their applications. In: PKC 2013. pp. 55–72
17. Coull, S.E., Green, M., Hohenberger, S.: Controlling access to an oblivious database using stateful anonymous credentials. In: Public Key Cryptography - PKC 2009. pp. 501–520
18. Damodaran, A., Dubovitskaya, M., Rial, A.: UC priced oblivious transfer with purchase statistics and dynamic pricing. In: Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings. pp. 273–296
19. Damodaran, A., Rial, A.: UC updatable databases and applications. In: Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology
20. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM J. Comput. **17**(2), 281–308 (1988)
21. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: ASIACRYPT 2010. pp. 177–194
22. Lai, R.W.F., Malavolta, G.: Subvector commitments with application to succinct arguments. In: CRYPTO 2019. pp. 530–560
23. Libert, B., Ling, S., Mouhartem, F., Nguyen, K., Wang, H.: Adaptive oblivious transfer with access control from lattice assumptions. In: ASIACRYPT 2017. pp. 533–563
24. Libert, B., Ramanna, S.C., Yung, M.: Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In: ICALP 2016. pp. 30:1–30:14
25. Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In: TCC 2010. pp. 499–517
26. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO '91. pp. 129–140
27. Rial, A.: Blind attribute-based encryption and oblivious transfer with fine-grained access control. Des. Codes Cryptogr. **81**(2), 179–223 (2016)
28. Rial, A., Kohlweiss, M., Preneel, B.: Universally composable adaptive priced oblivious transfer. In: Pairing-Based Cryptography - Pairing 2009. pp. 231–247
29. Wikström, D.: A universally composable mix-net. In: Naor, M. (ed.) Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2951, pp. 317–335. Springer
30. Xu, L., Zhang, F.: Oblivious transfer with threshold access control. J. Inf. Sci. Eng. **28**(3), 555–570
31. Xu, L., Zhang, F.: Oblivious transfer with complex attribute-based access control. In: Information Security and Cryptology - ICISC 2010. pp. 370–395

32. Zhang, Y., Au, M.H., Wong, D.S., Huang, Q., Mamoulis, N., Cheung, D.W., Yiu, S.: Oblivious transfer with access control : Realizing disjunction without duplication. In: Pairing-Based Cryptography - Pairing 2010. pp. 96–115

A Universally Composable Security

We prove our protocol secure in the universal composability framework [15]. The UC framework allows one to define and analyze the security of cryptographic protocols so that security is retained under an arbitrary composition with other protocols. The security of a protocol is defined by means of an ideal protocol that carries out the desired task. In the ideal protocol, all parties send their inputs to an ideal functionality $\mathcal{F}$ for the task. $\mathcal{F}$ locally computes the outputs of the parties and provides each party with its prescribed output.

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

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

In the ideal functionalities described in this paper, we consider static corruptions. When describing ideal functionalities, we use the following conventions as in [11].

Interface Naming Convention. An ideal functionality can be invoked by using one or more interfaces. The name of a message in an interface consists of three fields separated by dots, e.g., uud.read.ini in $\mathcal{F}_{\mathrm{UUD}}$ in §3. The first field indicates the name of the functionality and is the same in all interfaces of the functionality. This field is useful for distinguishing between invocations of different functionalities in a hybrid protocol that uses two or more different functionalities. The second field indicates the kind of action performed by the functionality and is the same in all messages that the functionality exchanges within the same interface. The third field distinguishes between the messages that belong to the same interface, and can take the following different values. A message uud.read.ini is the incoming message received by the functionality, i.e., the message through which the interface is invoked. A message uud.read.end is the outgoing message sent by the functionality, i.e., the message that ends the execution of the interface. The message

uud.read.sim is used by the functionality to send a message to $\mathcal{S}$, and the message uud.read.rep is used to receive a message from $\mathcal{S}$.

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

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

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

B Security Definitions of the Building Blocks of Π_{UUD}

Security of Subvector Commitments. We recall the correctness and binding properties [22] and define correctness for the update algorithms.

Correctness. Correctness requires that for any $par \leftarrow \mathsf{SVC.Setup}(1^k, \ell)$, $\mathbf{x} \leftarrow (\mathbf{x}[1], \ldots, \mathbf{x}[\ell]) \in \mathcal{M}^\ell$, $com \leftarrow \mathsf{SVC.Commit}(par, \mathbf{x})$, $I = \{i_1, \ldots, i_n\} \subseteq$

$[1,\ell]$ and $w_I \leftarrow \mathsf{SVC.Open}(par, I, \mathbf{x})$, $\mathsf{SVC.Verify}(par, com, \mathbf{x}_I, I, w_I)$ outputs 1 with probability 1, where $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$.

For the update algorithms, correctness requires that, for any par, $\mathbf{x}$, com, I and w_I computed as shown above, and for any $i \in [1,\ell]$, $x \in \mathcal{M}$, com' $\leftarrow \mathsf{SVC.ComUpd}(par, com, \mathbf{x}, i, x)$ and $w'_I \leftarrow \mathsf{SVC.OpenUpd}(par, w_I, \mathbf{x}, I, i, x)$, $\mathsf{SVC.Verify}(par, com', \mathbf{x}'_I, I, w'_I)$ outputs 1 with probability 1, where $\mathbf{x}'_I = (\mathbf{x}'[i_1], \ldots, \mathbf{x}'[i_n])$ and $\mathbf{x}'$ is such that $\mathbf{x}'[i] = x$ and for all $j \in [1,\ell] \setminus \{i\}$, $\mathbf{x}'[j] = \mathbf{x}[j]$.

Binding. This property requires that no adversary can output a commitment com, two sets of positions $I = \{i_1, \ldots, i_n\} \subseteq [1,\ell]$ and $J = \{j_1, \ldots, j_{n'}\} \subseteq [1,\ell]$, two subvectors $\mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n])$ and $\mathbf{x}_J = (\mathbf{x}'[j_1], \ldots, \mathbf{x}'[j_{n'}])$ and two openings w_I and w_J such that $\mathsf{SVC.Verify}$ accepts both but there exists an index $i \in I \cap J$ such that $\mathbf{x}[i] \neq \mathbf{x}'[i]$, i.e., for ℓ polynomial in k:

$$\Pr\left[\begin{array}{l} par \leftarrow \mathsf{SVC.Setup}(1^k, \ell);\ (com, I, J, \mathbf{x}_I, \mathbf{x}_J, w_I, w_J) \leftarrow \mathcal{A}(par) : \\ 1 = \mathsf{SVC.Verify}(par, com, \mathbf{x}_I, I, w_I) \wedge \\ 1 = \mathsf{SVC.Verify}(par, com, \mathbf{x}_J, J, w_J) \wedge \\ I = \{i_1, \ldots, i_n\} \subseteq [1,\ell] \ \wedge\ J = \{j_1, \ldots, j_{n'}\} \subseteq [1,\ell] \wedge \\ \mathbf{x}_I = (\mathbf{x}[i_1], \ldots, \mathbf{x}[i_n]) \in \mathcal{M}^n \wedge \\ \mathbf{x}_J = (\mathbf{x}'[j_1], \ldots, \mathbf{x}'[j_{n'}]) \in \mathcal{M}^{n'} \wedge \\ \exists\ i \in I \cap J \text{ such that } \mathbf{x}[i] \neq \mathbf{x}'[i] \end{array}\right] \leq \epsilon(k)\ .$$

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

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

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

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

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

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

Description of $\mathcal{F}_{\text{BB}}$. $\mathcal{F}_{\text{BB}}$ is parameterized by a universe of messages $\mathbb{U}_m$. $\mathcal{F}_{\text{BB}}$ interacts with a writer $\mathcal{W}$ and readers $\mathcal{R}_k$.

1. On input $(\mathsf{bb.write.ini}, sid, m)$ from $\mathcal{W}$:
   - Abort if $sid \notin (\mathcal{W}, sid')$.
   - Abort if $m \notin \mathbb{U}_m$.
   - If (sid, BB, ct) is not stored, set $\mathsf{BB} \leftarrow \perp$ and $ct \leftarrow 0$.
   - Increment ct, append $[ct, m]$ to BB and update ct and BB in (sid, BB, ct).
   - Create a fresh qid and store qid.
   - Send $(\mathsf{bb.write.sim}, sid, qid, m)$ to $\mathcal{S}$.

S. On input $(\mathsf{bb.write.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if qid is not stored.
   - Delete qid.
   - Send $(\mathsf{bb.write.end}, sid)$ to $\mathcal{W}$.

2. On input $(\mathsf{bb.getbb.ini}, sid, i)$ from $\mathcal{R}_k$:
   - Create a fresh qid and store $(qid, \mathcal{R}_k, i)$.
   - Send $(\mathsf{bb.getbb.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{bb.getbb.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', \mathcal{R}_k, i)$ such that $qid' = qid$ is not stored.
   - If (sid, BB, ct) is stored and $i \in [1, ct]$, take $[i, m]$ from BB and set $m' \leftarrow m$, else set $m' \leftarrow \perp$.
   - Send $(\mathsf{bb.getbb.end}, sid, m')$ to $\mathcal{R}_k$.

C Security Analysis of Π_{UUD}

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

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

In §C.1, we analyze the security of Π_{UUD} when (a subset of) readers $\mathcal{R}_k$ are corrupt. In §C.2, we analyze the security of Π_{UUD} when $\mathcal{U}$ is corrupt. We do not analyze in detail the security of Π_{UUD} when $\mathcal{U}$ and (a subset of) readers $\mathcal{R}_k$ are corrupt. We note that, in Π_{UUD}, honest readers communicate with $\mathcal{U}$ but not with other readers. Therefore, for this case the simulator and the security proof are similar to the case where only $\mathcal{U}$ is corrupt.

C.1 Security Analysis of Π_{UUD} when $\mathcal{R}_k$ are Corrupt

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

Initialization of $\mathcal{S}$. $\mathcal{S}$ runs the crs.get interface of an instance of $\mathcal{F}_{\text{CRS}}^{\text{SVC.Setup}}$ to get the parameters *par*.

Honest $\mathcal{U}$ sends an update. On input the message (uud.update.sim, *sid*, *qid*, $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$) from functionality $\mathcal{F}_{\text{UUD}}$, $\mathcal{S}$ runs the bb.write interface of a copy of $\mathcal{F}_{\text{BB}}$ on input $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ to write the update into the bulletin board in $\mathcal{F}_{\text{BB}}$. $\mathcal{S}$ follows the same steps described in Π_{UUD} in order to set and update a tuple $(sid, par, com, \mathbf{x}, cu)$. Then $\mathcal{S}$ sends (uud.update.rep, *sid*, *qid*) to $\mathcal{F}_{\text{UUD}}$.

$\mathcal{A}$ requests database. When $\mathcal{A}$ invokes the bb.getbb interface on input an index i, $\mathcal{S}$ sends (uud.getdb.ini, *sid*) to $\mathcal{F}_{\text{UUD}}$. When $\mathcal{F}_{\text{UUD}}$ sends (uud.getdb.sim, *sid*, *qid*), $\mathcal{S}$ sends (uud.getdb.rep, *sid*, *qid*) to $\mathcal{F}_{\text{UUD}}$. When $\mathcal{F}_{\text{UUD}}$ sends the message (uud.getdb.end, *sid*, DB), $\mathcal{S}$ runs the bb.getbb interface of $\mathcal{F}_{\text{BB}}$ on input i to send the update $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$ to $\mathcal{A}$. We recall that the update was already stored in $\mathcal{F}_{\text{BB}}$ when it was sent by the honest $\mathcal{U}$.

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

$\mathcal{A}$ sends a proof. When $\mathcal{A}$ invokes the zk.prove interface on input a pseudonym P, a witness *wit* and an instance *ins*, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{ZK}}^{R}$ on that input. Then $\mathcal{S}$ parses *ins* as $(par', com', parcom, com'_i, \langle com'_{i,j} \rangle_{\forall j \in [1,L]}, cr_k)$ and *wit* as $(w_I, I, i, open_i, \langle v_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]})$. $\mathcal{S}$ sets $com_i \leftarrow (com'_i, parcom,$ COM.Verify) and $\langle com_{i,j} \leftarrow (com'_{i,j}, parcom, \text{COM.Verify}) \rangle_{\forall j \in [1,L]}$. $\mathcal{S}$ sends (uud.read.ini, *sid*, P, $(i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]})$) to the functionality $\mathcal{F}_{\text{UUD}}$. When $\mathcal{F}_{\text{UUD}}$ sends the message (uud.read.sim, *sid*, *qid*, $(com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]})$), $\mathcal{S}$ does the following:

- $\mathcal{S}$ retrieves the stored tuple $(sid, par, com, \mathbf{x}, cu)$. If $cr_k \neq cu$, or if $par' \neq par$, or if $com' \neq com$, $\mathcal{S}$ sends $\mathcal{F}_{\text{UUD}}$ a message that makes $\mathcal{F}_{\text{UUD}}$ abort.
- Else, if for any $j \in [1, L]$, $\mathbf{x}[(i-1)L + j] \neq v_{i,j}$, $\mathcal{S}$ outputs failure.
- Else, $\mathcal{S}$ sends (uud.read.rep, *sid*, *qid*) to $\mathcal{F}_{\text{UUD}}$.

Theorem 2. *When (a subset of) readers $\mathcal{R}_k$ are corrupt, Π_{UUD} securely realizes $\mathcal{F}_{\text{UUD}}$ in the $(\mathcal{F}_{\text{CRS}}^{\text{SVC.Setup}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{ZK}}^{R})$-hybrid model if the SVC scheme is binding.*

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

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

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

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

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

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

- When the challenger sends the parameters *par*, B stores *par* as common reference string in the copy of $\mathcal{F}_{\text{CRS}}^{\mathsf{SVC.Setup}}$.
- When the adversary sends a valid proof with witness $wit = (w_I, I, i, open_i, \langle v_{i,j}, open_{i,j}\rangle_{\forall j \in [1,L]})$ and instance $ins = (par, com, parcom, com'_i, \langle com'_{i,j}\rangle_{\forall j \in [1,L]}, cr_k)$ such that, for any $j \in [1, L]$, $\mathbf{x}[(i-1)L + j] \neq v_{i,j}$, where $\mathbf{x}$ is in the stored tuple $(sid, par, com, \mathbf{x}, cu)$, B runs $w_I \leftarrow \mathsf{SVC.Open}(par, I, \mathbf{x})$ and sends $(com, I, I, \langle v_{i,j}\rangle_{\forall j \in [1,L]}, \langle \mathbf{x}[(i-1)L + j]\rangle_{\forall j \in [1,L]}, w_I, w'_I)$ to the challenger.

This concludes the proof of Theorem 3.

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

C.2 Security Analysis of Π_{UUD} when $\mathcal{U}$ is Corrupt

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

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

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

$\mathcal{A}$ sends update. When $\mathcal{A}$ invokes the bb.write interface on input the message $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$, $\mathcal{S}$ sends $(\mathsf{uud.update.ini}, sid, (i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]})$ to $\mathcal{F}_{\text{UUD}}$. When $\mathcal{F}_{\text{UUD}}$ sends the message $(\mathsf{uud.update.sim}, sid, qid, (i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]})$, $\mathcal{S}$ sends $(\mathsf{uud.update.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUD}}$. When $\mathcal{F}_{\text{UUD}}$ sends $(\mathsf{uud.update.end}, sid)$, $\mathcal{S}$ follows the same steps described in Π_{UUD} in order to set and update a tuple $(sid, par, com, \mathbf{x}, cu)$. Finally, $\mathcal{S}$ runs the bb.write interface of a copy of $\mathcal{F}_{\text{BB}}$ on input $(i, v_{i,1}, \ldots, v_{i,L})_{\forall i \in [1,N]}$.

Honest $\mathcal{R}_k$ requests database. When $\mathcal{F}_{\text{UUD}}$ sends $(\mathsf{uud.getdb.sim}, sid, qid)$, $\mathcal{S}$ runs the bb.getbb interface of $\mathcal{F}_{\text{BB}}$ on input a random index i. Then $\mathcal{S}$ sends $(\mathsf{uud.getdb.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUD}}$.

Honest $\mathcal{R}_k$ sends a proof. On input from $\mathcal{F}_{\text{UUD}}$ the message $(\mathsf{uud.read.sim}, sid, qid, (com_i, \langle com_{i,j}\rangle_{\forall j \in [1,L]}))$, $\mathcal{S}$ sends $(\mathsf{uud.read.rep}, sid, qid)$ to $\mathcal{F}_{\text{UUD}}$ and receives the message $(\mathsf{uud.read.end}, sid, P, (com_i, \langle com_{i,j}\rangle_{\forall j \in [1,L]}))$ from the functionality $\mathcal{F}_{\text{UUD}}$. $\mathcal{S}$ does the following:

– $\mathcal{S}$ retrieves the stored tuple $(sid, par, com, \mathbf{x}, cu)$.
– $\mathcal{S}$ parses com_i as $(com'_i, parcom, \mathsf{COM.Verify})$ and $\langle com_{i,j}\rangle_{\forall j\in[1,L]}$ as the tuples $\langle(com'_{i,j}, parcom, \mathsf{COM.Verify})\rangle_{\forall j\in[1,L]}$.
– $\mathcal{S}$ sets $ins \leftarrow (par, com, parcom, com'_i, \langle com'_{i,j}\rangle_{\forall j\in[1,L]}, cu)$.
– $\mathcal{S}$ sets the message corresponding to the $\mathsf{zk.prove}$ interface of $\mathcal{F}_{\mathrm{ZK}}^{R}$ to send P and ins to $\mathcal{A}$. Note that $\mathcal{S}$ does not know the witness, so it does no run a copy of the functionality. Instead, $\mathcal{S}$ sets the message as if it was sent by a copy of $\mathcal{F}_{\mathrm{ZK}}^{R}$.

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

Proof of Theorem 4. There are two differences between the real world protocol and $\mathcal{S}$. First, $\mathcal{S}$ uses a random index i to run the $\mathsf{bb.getbb}$ interface of $\mathcal{F}_{\mathrm{BB}}$. This change does not alter the view of the environment because $\mathcal{F}_{\mathrm{BB}}$ does not disclose i to the adversary. Second, $\mathcal{S}$ does not run $\mathcal{F}_{\mathrm{ZK}}^{R}$ because $\mathcal{S}$ does not know the witness of the proof. Because $\mathcal{F}_{\mathrm{ZK}}^{R}$ does not leak the witness to the adversary and the pseudonym and the instance are not modified, this change does not alter the view of the environment.

D Building Blocks of Our Instantiation of Π_{UUD}

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

We use the Pedersen commitment scheme [26]. $\mathsf{CSetup}(1^k)$ 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 an opening $open$. $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 if $com = g^x h^{open}$. In [11], it is shown that Pedersen commitments realize $\mathcal{F}_{\mathrm{NIC}}$.

Signature schemes. We use a signature scheme for the ZK proof for relation R in §5. A signature scheme consists of the algorithms KeyGen, Sign and VfSig. $\mathsf{KeyGen}(1^k)$ outputs a secret key sk and a public key pk, which include a description of the message space $\mathcal{M}$. $\mathsf{Sign}(sk, m)$ outputs a signature s on the message $m \in \mathcal{M}$. $\mathsf{VfSig}(pk, s, m)$ outputs 1 if s is a valid signature on m and 0 otherwise. This definition can be extended to blocks of messages $\bar{m} = (m_1, \ldots, m_n)$. In this case, $\mathsf{KeyGen}(1^k, n)$ receives the maximum number n of messages as input. A signature scheme must be existentially unforgeable [20].

E Oblivious Transfer with Access Control

E.1 Functionality $\mathcal{F}_{\text{OTAC}}$

$\mathcal{F}_{\text{OTAC}}$ interacts with a sender $\mathcal{U}$ and receivers $\mathcal{R}_k$. $\mathcal{F}_{\text{OTAC}}$ consists of the following interfaces:

1. $\mathcal{U}$ uses the $\mathsf{otac.init}$ interface to send the messages $\langle m_n \rangle_{n=1}^{N}$.
2. The receiver $\mathcal{R}_k$ uses the $\mathsf{otac.retrieve}$ interface to retrieve N.
3. $\mathcal{U}$ uses the $\mathsf{otac.policy}$ interface to send (or update) the policies $\langle \mathsf{ACP}_n \rangle_{n=1}^{N}$ and the relation R_{ACP} to $\mathcal{F}_{\text{OTAC}}$.
4. $\mathcal{R}_k$ uses the $\mathsf{otac.getpol}$ interface to obtain $\langle \mathsf{ACP}_n \rangle_{n=1}^{N}$ and R_{ACP}.
5. $\mathcal{R}_k$ uses the $\mathsf{otac.transfer}$ to send a choice i and a witness wit to $\mathcal{F}_{\text{OTAC}}$. If $(wit, \mathsf{ACP}_i) \in R_{\mathsf{ACP}}$, $\mathcal{F}_{\text{OTAC}}$ sends m_i to $\mathcal{R}_k$.

The relation R_{ACP} is

$$\begin{aligned} R_{\mathsf{ACP}'} = \{ & (wit, ins) : \\ & 1 = f(wit, \langle v_{i,j} \rangle_{\forall j \in [1,L]}) \} \end{aligned}$$

The instance $\mathsf{otac.policy} = \langle v_{i,j} \rangle_{\forall j \in [1,L]}$ of R_{ACP} is a policy represented as a tuple of values. The function f evaluates whether the witness wit satisfies the policy. wit can contain different elements depending on f. It can, e.g., contain signatures of the attributes of $\mathcal{R}_k$ certified by an issuer.

Description of $\mathcal{F}_{\text{OTAC}}$. $\mathcal{F}_{\text{OTAC}}$ is parameterised by universes of messages $\mathbb{U}_m$ and policies $\mathbb{U}_{pol}$.

1. On input $(\mathsf{otac.init.ini}, sid, \langle m_n \rangle_{n=1}^{N})$ from $\mathcal{U}$:
   - Abort if $sid \notin (\mathcal{U}, sid')$, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is already stored.
   - Abort if for $n = 1$ to N, $m_n \notin \mathbb{U}_m$.
   - Store $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$.
   - Send $(\mathsf{otac.init.sim}, sid, N)$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.init.rep}, sid)$ from $\mathcal{S}$:
   - Abort if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is not stored, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is already stored.
   - Store $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$.
   - Send $(\mathsf{otac.init.end}, sid)$ to $\mathcal{U}$.

2. On input $(\mathsf{otac.retrieve.ini}, sid)$ from $\mathcal{R}_k$:
   - Create a fresh qid and store $(qid, \mathcal{R}_k)$.
   - Send $(\mathsf{otac.retrieve.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.retrieve.rep}, sid, qid)$ from $\mathcal{S}$.
   - Abort if $(qid', \mathcal{R}_k)$ such that $qid' = qid$ is not stored.
   - If $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is not stored, set $N \leftarrow \perp$.
   - Else, store $(sid, \mathcal{R}_k, N)$.
   - Delete $(qid, \mathcal{R}_k)$.
   - Send $(\mathsf{otac.retrieve.end}, sid, N)$ to $\mathcal{R}_k$.

3. On input $(\mathsf{otac.policy.ini}, sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ from $\mathcal{U}$:
   - Abort if $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is not stored or if the number of policies is not equal to number of messages received.
   - For all $n \in [1, N]$, abort if $\mathsf{ACP}_n \notin \mathbb{U}_{pol}$.
   - If $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$ is not stored:
     - For all $n \in [1, N]$, abort if $\mathsf{ACP}_n = \perp$.
     - Abort if $R_{\mathsf{ACP}} = \perp$.
     - Store $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.
   - Else:
     - For all $n \in [1, N]$, if $\mathsf{ACP}_n \neq \perp$, update ACP_n in the stored tuple $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.
     - If $R_{\mathsf{ACP}} \neq \perp$, update R_{ACP} in $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.
     - Increment cu and update cu in $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.
   - Create a fresh qid and store qid.
   - Send $(\mathsf{otac.policy.sim}, sid, qid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.policy.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if qid is not stored.
   - Delete qid.
   - Send $(\mathsf{otac.policy.end}, sid)$ to $\mathcal{U}$.

4. On input $(\mathsf{otac.getpol.ini}, sid)$ from $\mathcal{R}_k$:
   - Create a fresh qid and store $(qid, \mathcal{R}_k)$.
   - Send $(\mathsf{otac.getpol.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.getpol.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', \mathcal{R}_k)$ such that $qid' = qid$ is not stored.
   - If $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$ is not stored, set $\langle \mathsf{ACP}_n \rangle_{n=1}^{N} \leftarrow \perp$ and $R_{\mathsf{ACP}} = \perp$.
   - Else, set $cr_k \leftarrow cu$, store $(\mathcal{R}_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cr_k)$ and delete any previous tuple $(\mathcal{R}_k, \langle \mathsf{ACP}'_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cr'_k)$.
   - Delete $(qid, \mathcal{R}_k)$.
   - Send $(\mathsf{otac.getpol.end}, sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ to $\mathcal{R}_k$.

5. On input $(\mathsf{otac.transfer.ini}, sid, i, wit)$ from $\mathcal{R}_k$:
   - Abort if $(sid, \mathcal{R}'_k, N)$ or $(\mathcal{R}'_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cr_k)$ such that $\mathcal{R}'_k = \mathcal{R}_k$ are not stored.
   - Abort if $i \notin [1, N]$, or if $(wit, \mathsf{ACP}_i) \notin R_{\mathsf{ACP}}$.
   - Create a fresh qid and store (qid, m_i, cr_k), where m_i is stored in the tuple $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$.
   - Send $(\mathsf{otac.transfer.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{otac.transfer.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if (qid, m_i, cr_k) is not stored, or if $cr_k \neq cu$, where cu is stored in the tuple $(sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}, cu)$.
   - Delete the record (qid, m_i, cr_k).
   - Send $(\mathsf{otac.transfer.end}, sid, m_i)$ to $\mathcal{R}_k$.

E.2 Building Blocks of $\Pi_{\mathbf{OTAC}}$

Functionality $\mathcal{F}_{\mathbf{OT}}$. $\mathcal{F}_{\mathrm{OT}}$ interacts with a sender $\mathcal{U}$ and receivers $\mathcal{R}_k$, and consists of four interfaces ot.init, ot.retrieve, ot.request and ot.transfer.

1. $\mathcal{U}$ uses the ot.init interface to send the messages $\langle m_n \rangle_{n=1}^{N}$ to $\mathcal{F}_{\mathrm{OT}}$. $\mathcal{F}_{\mathrm{OT}}$ stores $\langle m_n \rangle_{n=1}^{N}$. The simulator $\mathcal{S}$ learns N.
2. $\mathcal{R}_k$ uses the ot.retrieve interface to receive the number N of messages input by $\mathcal{U}$ to $\mathcal{F}_{\mathrm{OT}}$. The simulator $\mathcal{S}$ learns N.
3. $\mathcal{R}_k$ uses the ot.request interface to send a pseudonym P, an index $\sigma \in [1, N]$, a commitment com_σ and an opening $open_\sigma$ to $\mathcal{F}_{\mathrm{OT}}$. $\mathcal{F}_{\mathrm{OT}}$ parses the commitment com_σ as $(com'_\sigma, parcom, \mathsf{COM.Verify})$ and verifies the commitment by running COM.Verify. $\mathcal{F}_{\mathrm{OT}}$ stores $[\mathcal{R}_k, P, \sigma, com_\sigma]$ and sends P and com_σ to $\mathcal{U}$.
4. $\mathcal{U}$ uses the ot.transfer interface to send P and com_σ to $\mathcal{F}_{\mathrm{OT}}$. If a tuple $[\mathcal{R}_k, P, \sigma, com_\sigma]$ is stored, $\mathcal{F}_{\mathrm{OT}}$ sends the message m_σ to $\mathcal{R}_k$.

$\mathcal{F}_{\mathrm{OT}}$ is similar to existing functionalities for OT [14], except that it receives a commitment com_σ to the index σ and an opening $open_\sigma$ for that commitment. In addition, the transfer phase is split up into two interfaces ot.request and ot.transfer, so that $\mathcal{U}$ receives com_σ in the request phase. These changes are needed to use $\mathcal{F}_{\mathrm{OT}}$ in our OTAC protocol in order to ensure that $\mathcal{R}_k$ sends the same index σ to $\mathcal{F}_{\mathrm{OT}}$ and to $\mathcal{F}_{\mathrm{UUD}}$. It is generally easy to modify existing UC OT protocols so that they realize our functionality $\mathcal{F}_{\mathrm{OT}}$.

Description of $\mathcal{F}_{\mathrm{OT}}$. Functionality $\mathcal{F}_{\mathrm{OT}}$ runs with a sender $\mathcal{U}$ and receivers $\mathcal{R}_k$, and is parameterised with a maximum number of messages $\mathcal{N}_{max}$, a universe of pseudonyms $\mathbb{U}_p$ and a message space $\mathcal{M}$.

1. On input (ot.init.ini, sid, $\langle m_n \rangle_{n=1}^{N}$) from $\mathcal{U}$:
   - Abort if $sid \notin (\mathcal{U}, sid')$, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is already stored, or if $N > \mathcal{N}_{max}$.
   - Abort if for $n = 1$ to N, $m_n \notin \mathcal{M}$.
   - Store $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$.
   - Send $(\mathsf{ot.init.sim}, sid, N)$ to S.

S. On input (ot.init.rep, sid) from $\mathcal{S}$:
   - Abort if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is not stored, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is already stored.
   - Store $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ and initialize an empty table Tbl_{ot}.
   - Send $(\mathsf{ot.init.end}, sid)$ to $\mathcal{U}$.

2. On input $(\mathsf{ot.retrieve.ini}, sid)$ from $\mathcal{R}_k$:
   - Create a fresh qid and store $(qid, \mathcal{R}_k)$.
   - Send $(\mathsf{ot.retrieve.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{ot.retrieve.rep}, sid, qid)$ from $\mathcal{S}$.
   - Abort if $(qid', \mathcal{R}_k)$ such that $qid' = qid$ is not stored.
   - If $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is not stored, set $N \leftarrow \perp$.
   - Else, store $(sid, \mathcal{R}_k, N)$.

- Delete $(qid, \mathcal{R}_k)$.
- Send $(\mathsf{ot.retrieve.end}, sid, N)$ to $\mathcal{R}_k$.

3. On input $(\mathsf{ot.request.ini}, sid, P, \sigma, com_\sigma, open_\sigma)$ from $\mathcal{R}_k$:
   - Abort if $P \notin \mathbb{U}_p$
   - Abort if$(sid, \mathcal{R}_k, N)$ is not stored.
   - Abort if $\sigma \notin [1, N]$.
   - Parse com_σ as $(com'_\sigma, parcom, \mathsf{COM.Verify})$.
   - Abort if $\mathsf{COM.Verify}$ is not a ppt algorithm, or if $1 \neq \mathsf{COM.Verify}(parcom, com'_\sigma, open_\sigma, \sigma)$.
   - Create a fresh qid and store $(qid, \mathcal{R}_k, P, \sigma, com_\sigma)$.
   - Send $(\mathsf{ot.request.sim}, sid, qid, com_\sigma)$ to S.

S. On input $(\mathsf{ot.request.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid, \mathcal{R}_k, P, \sigma, com_\sigma)$ is not stored.
   - Append $[\mathcal{R}_k, P, \sigma, com_\sigma]$ to Tbl_{ot}.
   - Delete the record $(qid, \mathcal{R}_k, P, \sigma, com_\sigma)$.
   - Send $(\mathsf{ot.request.end}, sid, P, com_\sigma)$ to $\mathcal{U}$.

4. On input $(\mathsf{ot.transfer.ini}, sid, P, com_\sigma)$ from $\mathcal{U}$:
   - Abort if there is no entry $[\mathcal{R}_k, P, \sigma, com_\sigma]$ in Tbl_{ot}.
   - Create a fresh qid and store (qid, σ).
   - Send $(\mathsf{ot.transfer.sim}, sid, qid)$ to S.

S. On input $(\mathsf{ot.transfer.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if (qid, σ) is not stored.
   - Set $v \leftarrow m_\sigma$.
   - Delete the record (qid, σ).
   - Send $(\mathsf{ot.transfer.end}, sid, v)$ to $\mathcal{R}$.

Functionality $\mathcal{F}_{\mathbf{NYM}}$. $\mathcal{F}_{\mathrm{NYM}}$ models an idealized secure pseudonymous channel. We use $\mathcal{F}_{\mathrm{NYM}}$ to describe our protocol in order to abstract away the details of real-world pseudonymous channels. $\mathcal{F}_{\mathrm{NYM}}$ is similar to the functionality for anonymous secure message transmission in [13]. $\mathcal{F}_{\mathrm{NYM}}$ interacts with senders $\mathcal{T}_k$ and a replier $\mathcal{R}$ and consists of two interfaces $\mathsf{nym.send}$ and $\mathsf{nym.reply}$. $\mathcal{T}_k$ uses $\mathsf{nym.send}$ to send a message m and a pseudonym P to $\mathcal{R}$. $\mathcal{R}$ uses $\mathsf{nym.reply}$ to send a message m and a pseudonym P. $\mathcal{F}_{\mathrm{NYM}}$ checks if there is a party $\mathcal{T}_k$ associated with pseudonym P that is awaiting a reply, and in that case sends m and P to $\mathcal{T}_k$. Therefore, $\mathcal{R}$ replies to messages from $\mathcal{T}_k$ by specifying P.

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

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

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

- Store $(sid, P, \mathcal{T}_k)$.
- Delete the record $(qid, P, \mathcal{T}_k, m)$.
- Parse sid as $(\mathcal{R}, sid')$.
- Send $(\mathsf{nym.send.end}, sid, m, P)$ to $\mathcal{R}$.

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

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

E.3 Construction Π_{OTAC}

Π_{OTAC} uses an ideal functionality $\mathcal{F}_{\mathrm{OT}}$ as building block. $\mathcal{F}_{\mathrm{OT}}$ is used to implement the $\mathsf{otac.init}$ and $\mathsf{otac.retrieve}$ interfaces, as well as to allow $\mathcal{R}_k$ to obtain messages obliviously in the $\mathsf{otac.transfer}$ interface.

To implement access control, Π_{OTAC} uses the functionalities $\mathcal{F}_{\mathrm{UUD}}$, $\mathcal{F}_{\mathrm{BB}}$ and $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ACP}'}}$. In the $\mathsf{otac.policy}$ interface, $\mathcal{U}$ uses $\mathcal{F}_{\mathrm{UUD}}$ to store the policies, and $\mathcal{U}$ uses $\mathcal{F}_{\mathrm{BB}}$ to store the relation R_{ACP}. In the $\mathsf{otac.getpol}$ interface, $\mathcal{R}_k$ retrieves the policies and the relation from $\mathcal{F}_{\mathrm{UUD}}$ and $\mathcal{F}_{\mathrm{BB}}$.

In the $\mathsf{otac.transfer}$ interface, $\mathcal{R}_k$ reads the policy $\mathsf{ACP}_i = \langle v_{i,j} \rangle_{\forall j \in [1,L]}$ for her choice i by using $\mathcal{F}_{\mathrm{UUD}}$. To do show, $\mathcal{R}_k$ obtains commitments com_i and $\langle com_{i,j} \rangle_{\forall j \in [1,L]}$ to i and to the values $\langle v_{i,j} \rangle_{\forall j \in [1,L]}$ that represent the policy. $\langle com_{i,j} \rangle_{\forall j \in [1,L]}$ are sent as input to $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ACP}'}}$ so that $\mathcal{R}_k$ proves fulfilment of the policy. com_i is sent as input to $\mathcal{F}_{\mathrm{OT}}$ to obtain the message m_i.

$R_{\mathsf{ACP}'}$ is a modification of R_{ACP}. In $R_{\mathsf{ACP}'}$, the instance $\langle v_{i,j} \rangle_{\forall j \in [1,L]}$ of R_{ACP} is replaced by $\langle com_{i,j} \rangle_{\forall j \in [1,L]}$, while the witness is extended to contain $wit' \leftarrow (wit, \langle v_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]})$. I.e., the instance in $R_{\mathsf{ACP}'}$ contains commitments to the policy rather than the policy itself, which allows $\mathcal{R}_k$ to hide what instance is being used from $\mathcal{U}$. The relation $R_{\mathsf{ACP}'}$ is

$$
\begin{aligned}
R_{\mathsf{ACP}'} = \{ & (wit', ins') : \\
& \langle 1 = \mathsf{COM.Verify}(parcom, com'_{i,j}, v_{i,j}, open_{i,j}) \rangle_{\forall j \in [1,L]} \wedge \\
& 1 = f(wit, \langle v_{i,j} \rangle_{\forall j \in [1,L]}) \}
\end{aligned}
$$

Description of Π_{OTAC}. Π_{OTAC} is parameterised by universes of pseudonyms $\mathbb{U}_p$, messages $\mathbb{U}_m$ and policies $\mathbb{U}_{pol}$.

1. On input $(\mathsf{otac.init.ini}, sid, \langle m_n \rangle_{n=1}^N)$:
   - $\mathcal{U}$ uses the $\mathsf{ot.init}$ interface to send the messages $\langle m_n \rangle_{n=1}^N$ to $\mathcal{F}_{\mathrm{OT}}$.
   - $\mathcal{U}$ outputs $(\mathsf{otac.init.end}, sid)$.
2. On input $(\mathsf{otac.retrieve.ini}, sid)$:

- $\mathcal{R}_k$ uses the ot.retrieve interface of $\mathcal{F}_{\mathrm{OT}}$ to retrieve the number of messages N.
- $\mathcal{R}_k$ outputs (otac.retrieve.end, sid, N).

3. On input (otac.policy.ini, $sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}$):
   - $\mathcal{U}$ parses $\langle \mathsf{ACP}_n \rangle_{n=1}^{N}$ as $(n, v_{n,1}, \ldots, v_{n,L})_{\forall n \in [1,N]}$. $\mathbb{U}_{pol}$ admits policies that can be represented by tuples of values.
   - $\mathcal{U}$ uses the uud.update interface to send $(n, v_{n,1}, \ldots, v_{n,L})_{\forall n \in [1,N]}$ to $\mathcal{F}_{\mathrm{UUD}}$.
   - $\mathcal{U}$ uses the bb.write interface of $\mathcal{F}_{\mathrm{BB}}$ to write R_{ACP} into the bulletin board.
   - $\mathcal{U}$ outputs (otac.policy.end, sid).
4. On input (otac.getpol.ini, sid):
   - $\mathcal{R}_k$ uses the uud.getdb interface of $\mathcal{F}_{\mathrm{UUD}}$ to retrieve the policies $\mathsf{DB} = (n, v_{n,1}, \ldots, v_{n,L})_{\forall n \in [1,N]} = \langle \mathsf{ACP}_n \rangle_{n=1}^{N}$.
   - If $(sid, cr_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ is not stored, $\mathcal{R}_k$ sets $cr_k \leftarrow 0$.
   - $\mathcal{R}_k$ increments cr_k and uses the bb.getbb interface of $\mathcal{F}_{\mathrm{BB}}$ to receive the description of a relation R_{ACP}. $\mathcal{R}_k$ continues incrementing the counter and reading the bulletin board until the returned message is $\perp$. Then $\mathcal{R}_k$ takes the previous cr_k and the last description of a relation R_{ACP} received from $\mathcal{F}_{\mathrm{BB}}$, stores $(sid, cr_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$ and deletes any previous tuple $(sid, cr_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$.
   - $\mathcal{R}_k$ outputs (otac.getpol.end, $sid, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}}$).
5. On input (otac.transfer.ini, sid, i, wit):
   - In the first execution of this interface, $\mathcal{R}_k$ runs the com.setup interface of $\mathcal{F}_{\mathrm{NIC}}$.
   - $\mathcal{R}_k$ picks the policy $\mathsf{ACP}_i = (i, v_{i,1}, \ldots, v_{i,L})$ and R_{ACP} from the stored tuple $(sid, cr_k, \langle \mathsf{ACP}_n \rangle_{n=1}^{N}, R_{\mathsf{ACP}})$.
   - $\mathcal{R}_k$ aborts if $(wit, \mathsf{ACP}_i) \notin R_{\mathsf{ACP}}$.
   - $\mathcal{R}_k$ uses the com.commit interface of $\mathcal{F}_{\mathrm{NIC}}$ to obtain commitments and openings $(com_i, open_i, \langle com_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]}$ to i and $v_{i,1}, \ldots, v_{i,L}$.
   - $\mathcal{R}_k$ picks a random pseudonym $P \leftarrow \mathbb{U}_p$.
   - $\mathcal{R}_k$ uses the uud.read interface to send $(P, (i, com_i, open_i, \langle v_{i,j}, com_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]}))$ to $\mathcal{F}_{\mathrm{UUD}}$.
   - $\mathcal{U}$ receives $(P, (com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]}))$ from $\mathcal{F}_{\mathrm{UUD}}$.
   - In the first execution of this interface, $\mathcal{U}$ runs the com.setup interface of $\mathcal{F}_{\mathrm{NIC}}$.
   - $\mathcal{U}$ uses the com.validate interface of $\mathcal{F}_{\mathrm{NIC}}$ to validate the commitments $(com_i, \langle com_{i,j} \rangle_{\forall j \in [1,L]})$.
   - $\mathcal{U}$ uses the nym.reply interface of $\mathcal{F}_{\mathrm{NYM}}$ to send to $\mathcal{R}_k$ a message that acknowledges receipt of the commitments. (Here, we assume that $\mathcal{F}_{\mathrm{UUD}}$ uses the nym.send interface of $\mathcal{F}_{\mathrm{NYM}}$ to send a message from $\mathcal{R}_k$ to $\mathcal{U}$.)
   - $\mathcal{R}_k$ sets the instance $ins' \leftarrow (\langle com_{i,j} \rangle_{\forall j \in [1,L]})$ and the witness $wit' \leftarrow (wit, (\langle v_{i,j}, open_{i,j} \rangle_{\forall j \in [1,L]}))$.
   - $\mathcal{R}_k$ uses the zk.prove interface to send wit', ins' and P to $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ACP}'}}$.
   - $\mathcal{U}$ receives ins' and P from $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ACP}'}}$. $\mathcal{U}$ aborts if the commitments in ins' or the pseudonym are not equal to the ones received from $\mathcal{F}_{\mathrm{UUD}}$.

– $\mathcal{U}$ uses the nym.reply interface of $\mathcal{F}_{\text{NYM}}$ to send to $\mathcal{R}_k$ a message that acknowledges receipt of the ZK proof. (Here, we assume that $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$ uses the nym.send interface of $\mathcal{F}_{\text{NYM}}$ to send a message from $\mathcal{R}_k$ to $\mathcal{U}$.)
– $\mathcal{R}_k$ uses the ot.request interface to send P, i, com_i, and $open_i$ to $\mathcal{F}_{\text{OT}}$.
– $\mathcal{U}$ receives P and com_i from $\mathcal{F}_{\text{OT}}$. $\mathcal{U}$ aborts if com_i or the pseudonym is not equal to the commitment received from $\mathcal{F}_{\text{UUD}}$.
– $\mathcal{U}$ uses the ot.transfer interface to send P and com_σ to $\mathcal{F}_{\text{OT}}$.
– $\mathcal{R}_k$ receives m_i from $\mathcal{F}_{\text{OT}}$.
– $\mathcal{R}_k$ outputs (otac.transfer.end, sid, m_i).

E.4 Security Analysis of Π_{OTAC}

Theorem 5. Π_{OTAC} *securely realizes* $\mathcal{F}_{\text{OTAC}}$ *in the* ($\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{UUD}}$, $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$, $\mathcal{F}_{\text{BB}}$, $\mathcal{F}_{\text{NYM}}$)*-hybrid model.*

When $\mathcal{U}$ is corrupt, our simulator $\mathcal{S}$ proceeds by running the receiver part of Π_{OTAC}, with two modifications. First, the choice i is replaced by a random choice. This change does not alter the view of the environment because $\mathcal{F}_{\text{OT}}$ does not leak any information on i to the adversary. Second, because $\mathcal{S}$ does not know wit, $\mathcal{S}$ does not run $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$. Instead, $\mathcal{S}$ creates the message sent by $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$ to the adversary. This change does not alter the view of the environment because $\mathcal{F}_{\text{ZK}}^{R_{\text{ACP}'}}$ does not leak any information on wit to the adversary.

When $\mathcal{R}_k$ is corrupt, $\mathcal{S}$ proceeds by running the sender part of Π_{OTAC}, with the following modifications. First, when the honest $\mathcal{U}$ inputs the messages, $\mathcal{S}$ sends N random messages to $\mathcal{F}_{\text{OT}}$. When the adversary sends its choice i, $\mathcal{S}$ obtains m_i from $\mathcal{F}_{\text{OTAC}}$ and replaces the random message stored in $\mathcal{F}_{\text{OT}}$ by m_i before the ot.transfer interface of $\mathcal{F}_{\text{OT}}$ is run.