UC Updatable Databases and Applications

AFRICACRYPT 2020

Aditya Damodaran and Alfredo Rial
SnT, University of Luxembourg

Luxembourg National Research Fund

Supported by the Luxembourg National Research Fund (FNR) CORE project C17/11650748

securityandtrust.lu

Index

Signature Schemes in PPB and POT protocols

- In a Priced Oblivious Transfer (POT) protocol, a user purchases a message m_i from a set of N messages held by a provider. Each message is associated with a price p_i, and *both i and p_i are hidden from the provider* during each purchase.

- In a Privacy Preserving Billing (PPB) protocol, a meter measures the consumption of a service *c*, and a user must pay a price for c to a provider, who defines a tariff policy, consisting of several functions. For instance, one such policy could apply a rate r_i to a time interval i of consumption, and the resulting price would be $p = r_i c$. The user proves to the provider that p has been correctly computed, *without revealing r_i or c*.

Signature Schemes in PPB and POT protocols

- In both cases, the provider typically computes signatures s_i on $\langle i, p_i \rangle$ in POT protocols; or on $\langle i, r_i \rangle$ in the case of PPB protocols, using a signature scheme with efficient ZK proofs of signature possession.

- The user can then prove knowledge of a signature s_i, to prove to the provider that prices have been computed correctly.

- This is prone to be problematic because a signature revocation mechanism must be used in order to revoke signatures to old entries when they are replaced by new entries.

Shortcomings of using signature schemes in PPB and POT protocols

1. Updates require signature revocation
   - An updater would need to *revoke signatures for old database entries*, while updating them.

2. They do not support modularity in protocol design
   - Most protocols use ZK proofs that involve statements that prove that a witness is stored in a data structure, in addition to statements that prove something else about the witness; i.e., *these two tasks are not separated*.

   - Security analysis tends to be complex because the tasks of maintaining the database and for proving statements about entries in the database are also intertwined. Security proofs in the hybrid model are simpler because it is simpler to analyse the security of each building block in isolation of the others.

Updatable Databases

- An Updatable Database (UD) is a two party protocol between an *updater* and a *reader*.
- The updater sets up a database, and *may update this database at any time* during protocol execution.
- The contents of the database are *visible by both parties*.
- The reader computes zero knowledge proofs of knowledge of database entries to prove that a certain value is stored at a specific position *without revealing the position or the value to the updater*.

Related Work (I)

- Accumulators
  - They allow us to represent a set X as a single accumulator value A, and some schemes are equipped with efficient ZK proofs to prove knowledge of W_x such that $x \in X$. However, NHVC schemes allow us to commit to a vector of messages and prove statements about a message m_i and additionally, its position i in the vector.

- Zero Knowledge Sets and Databases
  - Zero knowledge Sets allow a prover to commit to a set X. The prover can then prove membership or non membership of an element x in X to a verifier.
  - In a Zero Knowledge Database, each element x is associated with a value v, and a proof for x reveals v to the verifier.
  - However, these datastructures hide the contents of X from the verifier. In our work, the database is public. Existing ZK data structures also reveal i and v to the prover, but our database hides this information.

Related Work (II)

- ZK proofs for large datasets
  - Computation and communication costs grow linearly with witness size in most ZK proofs.
  - Although some techniques which enable costs sublinear in the size of the dataset N exist, the cost for the prover is linear in N. On the other hand, in our construction, the verification cost of a proof is constant and independent of N.
  - Only the cost of computing an opening is linear in N, but this opening can be reused and updated with cost independent of N. Thus, the computation of ZK proofs in our construction has an amortized cost independent of N, which makes it practical for large databases.

Our UD scheme

We propose an UD scheme which solves the aforementioned shortcomings.

1. The scheme allows for efficient updates:
   - Updates do not require a revocation mechanism, and the cost of updates is independent of database size.

2. The scheme has been designed modularly:
   - Security analysis is simpler because the scheme is composed of several building blocks whose security can be analysed in isolation.
   - The ideal functionalities used can be replaced by protocols that realise them, and thus multiple instantiations of the protocol are possible.
   - The study of the task of creating an updatable database can be carried out in isolation, which eases the comparison of different constructions for it.

Our contribution:

- *Definition of a new Ideal Functionality for UD.*
- *A new UC-secure UD scheme.*

Our Ideal Functionality F_{UD} for Updatable Databases

The ideal functionality makes use of two interfaces:

1. Update

   Allows an updater to update positions in the Updatable Database.

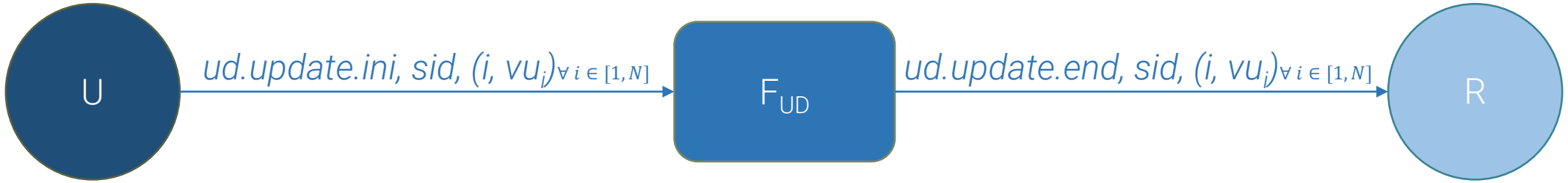
2. Read

   Allows a reader to prove that an entry $[i, vr_i]$ is contained in the Updatable Database.

F_{UD} : Efficient Updates (I)

- F_{UD} maintains a database *DB* that consists of entries of the form $[i, vr_i]$, for i=1 to N.
- In the update phase, an updater U sends one or more entries of the form $[i,vr_i]$ to F_{UD}. F_{UD} updates *DB* accordingly, so that the entries corresponding to all i have been updated.
- F_{UD} also maintains a counter *cu* for the updater and a counter *cr* for the reader, and increments *cu* upon updating the database.
- F_{UD} Aborts if $cr \neq cu + 1$. Otherwise, F_{UD} sends a copy of the updates to the reader.

F_{UD} : Efficient Updates (II)



F_{UD} : Read interface

- The Reader receives a position value *i*, the value stored at that position in *DB* vr_i, and commitments com_i and $comr_i$ to *i* and vr_i respectively, as input.
- F_{UD} checks if the commitments com_i and $comr_i$ are valid, and if $[i,vr_i] \in DB$.
- F_{UD} aborts if $cr \neq cu$.
- F_{UD} sends com_i and $comr_i$ to the Updater.

Our UD protocol for F_{UD}

- We propose a protocol in the UC model that realises F_{UD}.

- We describe how UD is designed modularly, and how it allows for efficient updates.

UD: Modular Design (I)

- We use a hybrid model to design the protocol modularly, as described in the UC framework.
- The hybrid protocol makes use of a Non Hiding Vector Commitment Scheme.
- The hybrid protocol also makes use of the following Ideal Functionalities:
    1. F_{CRS} – Ideal functionality for common reference string generation
    2. F_{AUT} – Ideal functionality for an authenticated channel
    3. F_{NIC} – Ideal functionality for non-interactive commitments
    4. F^{R}_{ZK} – Ideal functionality for zero knowledge

UD : Modular Design (II)

- However, because our hybrid protocol uses ideal functionalities as building blocks, instances arise where we must ensure that two or more functionalities receive the same input.

- We use the functionality F_{NIC} for non-interactive commitments in Camenisch et al CRYPTO 2016, to commit to input values, so that these commitments may also be passed as input to the functionalities.

- The functionalities used in our protocol have been modified to receive committed inputs.

- The functionalities can then verify the commitments they receive as input. The binding property for commitments ensured by F_{NIC} guarantees that these inputs are the same.

UD : Efficient Updates (I)

- In the first execution of the update phase, the updater sets up a counter *cu* = 0 and a vector *x*, such that x[i] = vu_i for all *i*, and computes an NHVC commitment *com* to *x*. The updater then uses F_{AUT} to send (i,vu_i) for all *i* to the reader; and the reader also sets up *cr, x,* and computes a commitment to *x*.

- In subsequent executions of the update phase, the updater updates *x* as required based on the tuples it receives as input, increments *cu*, and updates *com*. The updated values and *cu* are sent to the reader via F_{AUT}, and the reader also updates its copy of the database (if *cu* = *cr* +1), and updates its commitment *com*.

- The costs for updating *com* are linear in the number of updates performed on the database, but independent of the size of the database *N*.

UD : Efficient Updates (II)

- The reader might hold openings for database entries from previous executions of the read interface; these openings are also updated by the reader after *com* has been updated, and the cost of these updates is also independent of *N*.

- Thus, the construction allows for efficient updates, and is practical for large databases.

UD : Read Interface

- The reader computes commitments com_i and $comr_i$ to the position i and the value vr_i being read from the database, respectively.
- If the reader has not already computed an opening w_i for this entry, it uses the NHVC scheme to compute an opening; otherwise, w_i is reused.
- The reader then sends w_i, i, vr_i, com_i and $comr_i$, and the current counter value cr to F_{ZK}, in order to prove to the updater that it is reading a valid entry from the database.
- The updater receives com and cr from F_{ZK}, and checks if these values are consistent with its own com and cu. This also ensures that the reader and the updater have been working on the same version of the database.

Variants

1. Reading several entries simultaneously
   - The reader must simply compute openings w_i for each entry read.
   - The relation R used by FZK must also be modified accordingly, to replicate the equations needed for one entry to all entries read.

2. Entries of the form $[i,vr_{i,1},\ldots,vr_{i,m}]$
   - This variant of UD is useful in situations where a party needs to prove that a tuple of values belongs to the same entry, and that each element in the entry is stored at a particular position within this entry.
   - *Com* must commit to a vector of length $N \times m$ such that $x[(i-1)m+j] = vr_{i,j}$ for all $i \in [1,N]$ and $j \in [1,m]$.
   - In the update phase, each element of the vector can be modified independently of the others.
   - In the read phase, the reader must compute openings $(w_{(i-1)m+j},\ldots,w_m)$ to open the positions $[(i-1)m+1,im]$ of the committed vector.
   - The relation R used by FZK must also be modified to prove that these positions belong to the same entry.

Efficiency of our UD scheme (I)

- As the protocol has been designed in the hybrid model, the computation and communication cost of our UD depends on the building blocks used to instantiate each of its functionalities.

- F_{NIC} introduces an overhead as commitments must be computed for every input value that must be sent to multiple functionalities, but this overhead is small.

- As UD must be instantiated with a vector commitment scheme, it also involves the use of a common reference string that grows linearly with the size of the database N.

Efficiency of our UD scheme (II)

- Although the commitments and their openings are of size independent of N, and the vector commitment and its openings can be updated with cost independent of N, the cost of computation of an opening grows with N. However, these openings can be updated and reused throughout the execution of the protocol, yielding amortized computation cost independent of N.

- The ZK proofs of NHVC openings also offer computation and communication cost independent of N.

- Thus, the protocol remains efficient when N is large, under this instantiation.

Efficiency of our UD scheme (III)

- The following table lists execution times for an instantiation of our protocol using a NHVC scheme secure under the l-DHE assumption, and the Pedersen commitment scheme for F_{NIC}, against varying database sizes N, and against the security parameters of the Paillier encryption scheme used for ZK proof computation in the read phase (We use the compiler in Camenisch et al ASIACRYPT 2011).

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

Applications (I)

1. POT protocols:

   - F_{UD} can be used to design a POT protocol modularly, whilst also using a functionality for oblivious transfer as a building block. The provider can store the prices for each message in a database DB in the form [i, p_i], where p_i is the message price for m_i.

   - The buyer can then use the read interface of F_{UD} to prove to the provider that he/she has used the right prices for the message being purchased, as F_{UD} in turn invokes F_{ZK}. Thanks to the fact that F_{UD} also computes commitments com_i and $comr_i$ to i and p_i respectively, com_i can be sent as input to an OT functionality, and the buyer can retrieve m_i.

Applications (II)

2. PPB protocols:

- F_{UD} can be used to design a PPB protocol modularly, and the provider can make use of a database DB consisting of entries of the form $[i, f_i]$, where f_i represents a function corresponding to a time or consumption interval i.

- A meter outputs a signed reading of the form (c,i), and the user computes commitments com_i and $comr_i$ to i and f_i respectively. Com_i can then be sent to F_{ZK} to prove that i is the value signed in the meter reading, and $comr_i$ can be sent to F_{ZK} to prove that $p = f_i(c)$. The provider can also modify tariff policy functions in the database efficiently and at any time during the execution of the protocol.

Conclusion

Our UD scheme provides functionalities, such as those listed below, whilst also allowing for instantiations that are practical for large datasets:

1. The scheme allows for efficient updates.

2. The scheme is UC secure and has been designed modularly.