600.641 Special Topics in Theoretical Cryptography March 26th, 2007

Lecture 15: Zero-Knowledge Databases

Instructor: Susan Hohenberger *Scribe: Joshua Mason*

Today's lecture was a JHUISI Seminar given by Dr. Moses Liskov from The College of William and Mary.

1 Introduction

Today's topic is zero-knowledge databases (and zero-knowledge sets). In a ZK set, the querier submits x and should only learn if x is in a set D or not, in a provable way. But in a ZK database, the querier submits x and should only learn the corresponding database value $D(x)$ (which might be the empty symbol), in a provable way. We'll define ZK databases and ZK sets in a moment, but first let's discuss a building block that we'll need called mercurial commitments.

2 Commitment Schemes

2.1 Review

As we've seen in this class previously, commitment schemes bind the crafter to a given value without revealing the value. The commitment can be revealed at some later date by releasing a decommitment. Commitments each have a commitment phase where $C(m) = c$ produces a commitment on the value m. The decommitment is accomplished by releasing the decommitment value, d. Then, the receiver of d can verify that the message m matches the commitment c with a verification process, $V(m, c, d) = yes/no$.

Each commitment scheme's efficacy is measured in two properties: `hiding` and `binding`. A hiding commitment scheme releases no information on the value being committed to, m. A binding commitment scheme can only be decomitted to one value, m.

2.2 A new scheme ... Mercurial Commitments

Commitment schemes will facilitate the zero-knowledge database construction, so it is important to consider how to commit to an entire database of values. One option would be to form one m value for an entire database (by concatenating entries or some other similar method) and committing to m, but opening this commitment would reveal the entire database and thus violate the soon to be discussed zero-knowledge goal. Another possibility would be to generate one commitment for each value in the database. However, this would reveal how many elements exist in the database. As well, it would not yield a method for proving an element's non-membership.

The commitment scheme that will be employed in Liskov's construction is called a mercurial commitment, introduced by Chase et al [CHL$^+$05]. In this commitment scheme, there are two types of commitments and two ways to decommit. A "soft commitment" is a

commitment to no value. It cannot be hard-decommitted, but can be soft-decomitted to any value. A "hard commitment" is a commitment to a particular value, it can decommitted with both methods to only that value. A user can tell the difference between a soft-decommitment and a hard-decommitment.

Their security is still measured in terms of the binding and hiding properties discussed earlier. They have non-interactive commitments and decommitments, but use a public random string, σ. If the person crafting the commitment is allowed to choose σ, the "binding" property can be avoided. The mathematical innerworkings of mercurial commitments are beyond the scope of this lecture and as such are left as an exercise to the reader.

3 Zero-knowledge database

Intuitively, a zero-knowledge database is a database that may be queried by a querier without the querier learning any information aside from the queried value (or a symbol $\perp$ indicating that the value is not in the database). Moreover, the database owner *proves* to the querier, in a zero-knowledge fashion, that the query value is correct. Zero-knowledge sets (a specific form of databases) were first constructed by Micali, Rabin, and Kilian [MRK03] in the common reference string (CRS) model with the discrete log assumption. It was later generalized by Chase et al [CHL+05] using mercurial commitments.

For a more formal description, consider the following simple protocol where P is the prover/maintainer of the database and V is the querier. Let x be an index being queried, $D(x)$ the value in the database for index x, and π_x the proof that this value is correct. For example, x might be a person's name and D might be a database of phone numbers, making $D(x)$ person x's phone number. Let PK be the public commitment to the database D.

- $P \xleftarrow{x} V$
- $P \xrightarrow{D(x),\pi_x} V$

We demand that this protocol has two properties. First, it must be hiding. PK and π_x for any x should reveal nothing about the database other than the value stored at index x. It must also be binding so that once PK is published, for each x, only one $D(x)$ can be proved.

4 Secure updates in zero-knowledge databases

4.1 Definitions/Properties

The current formalization of zero-knowledge databases assume a fixed D to which a prover commits. The commitment on the database is then published before any protocol interaction takes place. The database cannot be updated once PK is published. So, Moses Liskov [Lis05] aims to form an efficiently updatable zero-knowledge database.

For our purposes there are two forms of secure updates: `opaque updates` and `transparent updates`. In an opaque update, the querier, V, will not know whether previously queried

values are still valid after an update. Transparent updates allow a querier to update previously received proofs for a given x if x still exists in the database.

A naive approach for opaque updates is to simply craft a fresh commitment to the new database. This is non-optimal because its efficiency would depend greatly on the size of the database. Liskov chooses to solve the transparent update problem. It should be noted, though, a more efficient way of accomplishing opaque updates is still unknown.

Updates should reveal as little as possible, should not increase the size of PK, and should be efficient. Transparent updates necessitate the following 5 operations:

1. $Commit(D, \sigma)$ produces PK and SK where PK is the published commitment and SK contains the decommitment values. σ is a public random string (not chosen by the prover) on which commitments are based.

2. $Prove(D, \sigma, PK, SK, x)$ produces the proof (a set of decommitments), π_x, for any x even if it does not exist in the database

3. $DBUpdate(D, \sigma, PK, SK, x, y)$ produces $(PK^{'}, SK^{'}, U)$ updated such that $D^{'}(x) = y$. $PK^{'}$ is the new commitment for the database, D, and U is a value that allows the verifier to update previously received proofs.

4. $Verify(\sigma, PK, x, \pi_x) = yes/no$ verifies a proof, π_x

5. $PUpdate(\sigma, PK, PK', U, x, \pi_x)$ produces the updated proof, $\pi_x^{'}$

The basic protocol will work as follows:

1. $V \longrightarrow P$: D - which database is being discussed

2. $P \longrightarrow V$: PK - the commitments for D

3. $V \longrightarrow P$: $Query(x)$ - a query asking for the value stored at index $x \in D$

4. $P \longrightarrow V$: $D(x), \pi_x$ - the value stored at index x, $D(x)$, and corresponding decommitment values for x, π_x

5. $V \longrightarrow P$: $Update(x, y)$ - a query requesting the prover change the value stored at index x to y, or $D(x) = y$

6. $P \longrightarrow V$: PK', U - the new commitment values and a value U which allows the verifier to update π_x

To prove soundness, the simulator definition necessitates a trusted third party, T, whose function is primarily to produce decommitments when the querier queries a value.

1. $Sim \longrightarrow V$: σ

2. $V \longrightarrow T$: D

3. $T \longrightarrow Sim$: $Begin!$
4. $Sim \longrightarrow V$: PK
5. $V \longrightarrow P$: $Query(x)$
6. $T \longrightarrow Sim$: $D(x)$
7. $P \longrightarrow V$: $D(x), \pi_x$
8. $V \longrightarrow T$: $Update(x, y)$
9. $T \longrightarrow Sim$: $Update!$
10. $Sim \longrightarrow V$: PK', U

This simulator cannot function as is, though. T would have to reveal whether x was previously queried to know whether to produce a U for V. One method for doing this would be to simply reveal x to the simulator when an update operation is performed, but doing so reveals too much information to the simulator.

Ideally, the trusted party would only reveal whether or not the value being updated has ever been queried before. However, Liskov could not obtain such a construction. Instead, he assigns a psuedonym to each queried value. Assume the verifier makes the following requests:

1. $V \longrightarrow P$: $Query(A)$
2. $V \longrightarrow P$: $Update(A)$
3. $V \longrightarrow P$: $Update(B)$
4. $V \longrightarrow P$: $Update(C)$
5. $V \longrightarrow P$: $Query(B)$
6. $V \longrightarrow P$: $Update(C)$

In step 1, T reveals to the simulator $D(A)$ and #1. #1 acts as the psuedonym for A. Then, when the verifier attempts to update A in step 2, T reveals to the simulator the psuedonym, #1. This informs the simulator that A has been queried before so the simulator knows to produce a U value. When the update of B is performed in step 3, the trusted party sends B's psuedonym to the simulator. Since B has not been queried before, its psuedonym is #2. The simulator now knows that B has never been queried before without learning the value B.

By way of fixing the previous simulator, when the simulator asks the trusted party for the value stored at index x in step 6, the trusted party sends both the value and $N(x)$ or the psuedonym for x. And similarly, where T previously sent "Update", it will now send $N(x)$. The psuedonym function, $N(x)$, is defined such that $N(x) = n$ where n is the n^{th} distinct item seen.

4.2 Construction

The variables shown in figures 1, 2, and 3 are defined as follows:

- v_ω: If $\omega = H(x)$ for some $x \in D$ then $v_\omega = H(D(x))$. If ω is an internal node, the value v_ω is defined recursively as $H(c_{\omega 0} c_{\omega 1})$. If $\omega \neq H(x)$ then $v_\omega = H(\perp)$.
- c_ω: A commitment which is either a soft commitment or a hard commitment to v_ω.
- $D(x)$: The value stored at index x.
- $c_\in$: This value is used as PK

Using this construction, proving a value is in the database can be done by revealing the highlighted nodes in figure 2. The verifier then uses the provided values to construct every v_ω that is a prefix of $H(x)$. To prove non-membership, the values highlighted in figure 3 are sent to the verifier. The verifier then does the same operation, reconstructing the v_ω values that are a prefix of $H(x)$ using $D(x) = \perp$.

To update a value in the database, a new commitment is produced for the new value. This commitment is then propegated upward so the verification process still works. The new $PK = c_\in$ is published as the new commitment to the database. This is not quite enough, though. In order for the verifier to be able to update the value the verifier already has, the mercurial commitments are crafted specially such that given a value U, the verifier may update his proof and thus know the new value has been inserted. Because the details of the mercurial commitment scheme are beyond the scope of this talk, the details of the creation of U are omitted. Please refer to the mercurial commitments paper [CHL+05] and Liskov's paper [Lis05] for an understanding of how U is created.

References

[CHL+05] Melissa Chase, Alexander Healy, Anna Lysyanskaya, Tal Malkin, and Leonid Reyzin. Mercurial commitments with applications to zero-knolwedge sets. In Ronald Cramer, editor, *Advances in Cryptology—EUROCRYPT 2005*, Lecture Notes in Computer Science, pages 422–439. Springer-Verlag, 2005.

[Lis05] Moses Liskov. Updatable zero-knowledge databases. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 174–198. Springer, 2005.

[MRK03] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-knowledge sets. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, page 80, Washington, DC, USA, 2003. IEEE Computer Society.

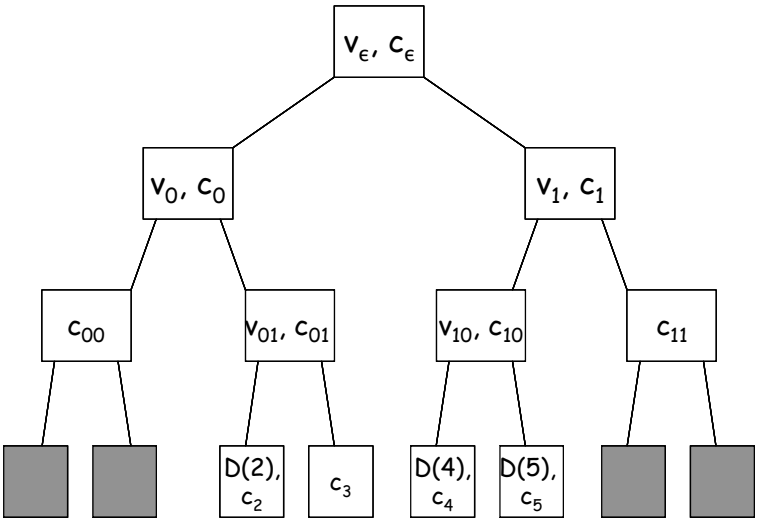

Figure 1: The main updatable zero-knowledge database construction.

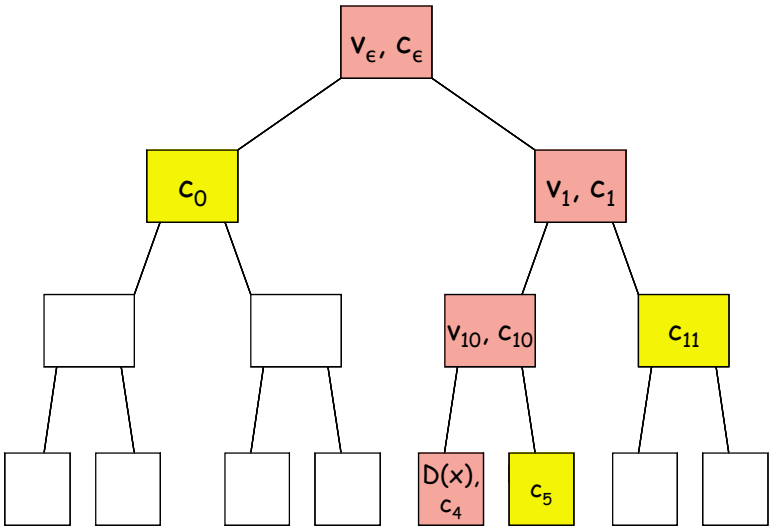

Figure 2: Values revealed during proof of membership of D(x).

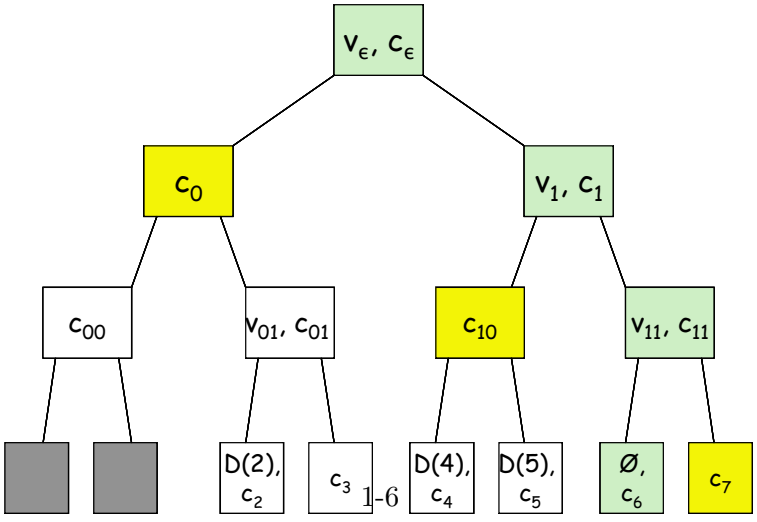

Figure 3: Values revealed during proof of non-membership.