Zero-Knowledge Proofs and Sets

William Pentney

December 8, 2003

Overview

Zero-Knowledge Proofs: intro

Zero-Knowledge Sets: intro

Implementation of a ZK set

Extensions/Open Questions

Zero-knowledge Proofs

Abstractly, a ZK proof involves:

- a prover and a verifier
- prover wants to convince verifier of statement X (with high probability)
- but prover does not want to reveal how to actually prove statement X

Huh?

Zero-Knowledge Proofs

Where is this useful?

Generally: in showing knowledge of info without revealing it

- authentication over net
- cryptography
- remote maintenance of information
- etc.

Zero-Knowledge Proofs

Note: typically, ZK proofs don't involve "perfect" proof

Basic idea is to prove X with arbitrarily high probability

- e.g. have a test that imposter can pass with probability $\frac{1}{2}$
- after n distinct tests, probability of success for imposter is $\frac{1}{2n}$
- can thus prove with whatever confidence verifier wants

Zero-Knowledge Proofs

Example: finding square roots of numbers mod p

Take some number r. Say $n = r^2$ mod p.

e.g. $r = 5, p = 6, r^2 = 25, n = r^2$ mod $p = 1$.

r is the square root mod p of n.

Zero-Knowledge Proofs

Interesting feature of r and $n = r^2 \bmod p$:

- given r, n is easy to compute
- but ... given only n, r is non-trivial to compute
  - no efficient (poly time) algorithm known

If p,r are very large, finding r from n is practically impossible

This is an example of a *one-way function*

Zero-Knowledge Proofs

Using the square roots mod p problem for fun and profit:

We can use this for basic authentication of identity

- prover P wants to prove he is P to verifier V
- initially, P gives V a large number n
- n has a square root mod p, r that only P knows
- by receiving r, V can check $n = r^2 \bmod p$
- V knows P is P - impostor couldn't have guessed r

However: we'd prefer not to transmit r publicly ...

Zero-Knowledge Proofs

P can prove with high probability that he knows r - without giving it away!

First P gives V the value n.

Then:

- P chooses random number m, sends V the value $x = m^2 \bmod p$
- V sends P random bit $b \in \{0, 1\}$
- P sends V the value $y = mr^b$
- V tests if $y^2 = xn^b \bmod p$
  - since $y^2 = m^2(r^b)^2 = xn^b$

Zero-Knowledge Proofs

We would like our proof system to be:

1. complete - P can give correct answer every time
2. sound - P cannot lie, or an imposter can be caught
3. zero-knowledge - an eavesdropper can't find "secret" info from public info

Zero-Knowledge Proofs

Completeness: P can successfully complete this for both $b = 0$ and 1

- need to know both m and mr, thus know r

Soundness: Imposter P' can give the right answer w/prob. $\frac{1}{2}$

- guess if $b = 0$ or $b = 1$
- if $b = 0$ P' can succeed - just choose m, send x,$y = m$
- if $b = 1$ P' chooses m, sends $x = \frac{m^2}{n}$ and $y = m$

Correct answer is reliable with probability $\frac{1}{2}$

Zero-Knowledge Proofs

Zero-knowledge: can eavesdropper figure out r?

Answer: no

- eavesdropper sees either m and x, or mr and x
- neither is enough to find out r

By repeating test n times, chance of impostor's success becomes about $\frac{1}{2n}$

Overview

"Zero Knowledge Sets" - S. Micali, M. Rabin, J. Killian, FOCS 2003

Goal: an efficient representation of *zero-knowledge (ZK) sets*

What characterizes ZK sets?

- membership can be proven/disproven without revealing other evidence about set:
  - cardinality,
  - other members/nonmembers of set, etc.
- Ideally, we'd like to do this efficiently (poly time)

Note: seems tough to prove non-membership without giving this away ...

ZK EDBs

We will to implement ZK *elementary databases* (EDBs)

Say we have:

- prover P
- verifier V
- database, represented as function $D : \{0,1\}^* \rightarrow \{0,1\}^*$
- (we will say $D(x) = \perp$ if a key x is not in database)

Two phases:

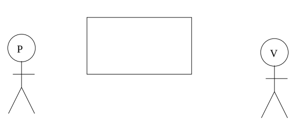
- commitment - P and V share commitment information
- verification - P is asked question, gives answer, proof to V, checked using commitment

EDB Verification Phase

P provides commitment

Basic procedure:

- P receives D and public random string T
- P computes two keys:
  - PK (public)
  - SK (private)

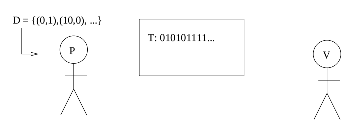


EDB Verification Phase

P provides commitment

Basic procedure:

- P receives D and public random string T
- P computes two keys:
  - PK (public)
  - SK (private)

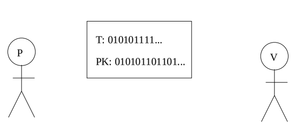


EDB Verification Phase

P provides commitment

Basic procedure:

- P receives D and public random string T
- P computes two keys:
  - PK (public)
  - SK (private)

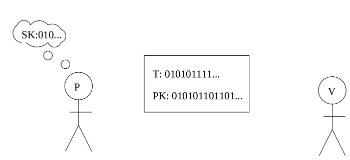


EDB Verification Phase

P provides commitment

Basic procedure:

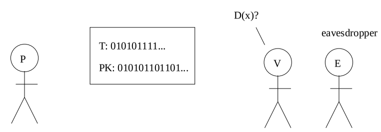
- P receives D and public random string T
- P computes two keys:
  - PK (public)
  - SK (private)



EDB Proof Phase

Say V gives P a string x

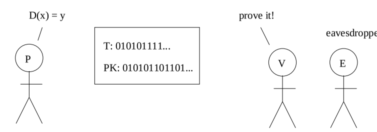
- P finds y = $D(x)$ (may be $\perp$)
- Using SK, P produces a proof π_x of $D(x) = y$
- V runs algorithm on π_x, PK, and T
- V concludes proof is valid or invalid



EDB Proof Phase

Say V gives P a string x

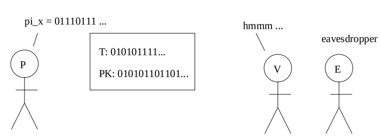
- P finds y = $D(x)$ (may be $\perp$)
- Using SK, P produces a proof π_x of $D(x) = y$
- V runs algorithm on π_x, PK, and T
- V concludes proof is valid or invalid



EDB Proof Phase

Say V gives P a string x

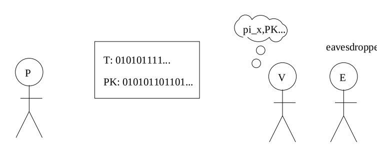
- P finds y = $D(x)$ (may be $\perp$)
- Using SK, P produces a proof π_x of $D(x) = y$
- V runs algorithm on π_x, PK, and T
- V concludes proof is valid or invalid



EDB Proof Phase

Say V gives P a string x

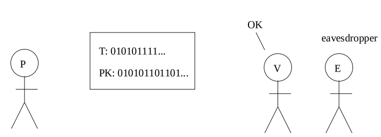
- P finds y = $D(x)$ (may be $\perp$)
- Using SK, P produces a proof π_x of $D(x) = y$
- V runs algorithm on π_x, PK, and T
- V concludes proof is valid or invalid



EDB Proof Phase

Say V gives P a string x

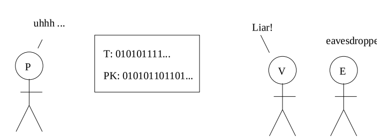
- P finds y = $D(x)$ (may be $\perp$)
- Using SK, P produces a proof π_x of $D(x) = y$
- V runs algorithm on π_x, PK, and T
- V concludes proof is valid or invalid



EDB Proof Phase

Say V gives P a string x

- P finds y = $D(x)$ (may be $\perp$)
- Using SK, P produces a proof π_x of $D(x) = y$
- V runs algorithm on π_x, PK, and σ
- V concludes proof is valid or invalid



Our Proof System Should Be:

Complete - for any EDB D, correct value of $D(x)$ can be proven

Sound - PK commits prover to partial function D

- no one can, in polytime, find x, y, z, $y \neq z$, and prove $D(x) = y$ and $D(x) = z$
- this ensures prover cannot lie, or imposter cannot forge result

Zero-knowledge -

- say V sees a commitment to EDB D and sequence of proofs for $x_1, x_2, ...$
- then V queries trusted party about x_1, x_2... and only receives values in response
- knowledge obtained by both processes should be identical

ZK Set Preliminaries

Our ZK set construction will make use of:

- Pederson's Commitment Scheme and hash function
- Merkle trees

We will now go over these ...

Commitment Schemes

We will try to calculate a commitment for our ZK set

A proof scenario: parties P and V share random string T

For P to commit:

- P is given input m
- P returns commitment string c and keeps secret *proof* r

Later, for verification:

- P publicizes input c and r
- V checks c,r using m,T

Pedersen's Commitment Scheme

Common approach to commitment: T is public quadruple (p, q, g, h)

- p, q prime, $q | p - 1$
- Z_p is group of integers mod p
- Z_q is a cyclic subgroup of Z_p with q elements
- g, h generators of Z_q

To commit, P picks random r, outputs $c = g^m h^r$ mod p

To verify, V gets c, r checks if $c = g^m h^r$ mod p

It is very, very difficult to find two m that produce same c

- relies on "Discrete Logarithm Assumption"

Pedersen's Hash Function

Pederson's commitment scheme yields a good hash function $H(a, b) = H(a, b)_{pqgh}$:

- $H(a, b)_{pqgh} = g^a h^b$ mod p

It is very difficult to find two (a, b) that hash to the same value with this function

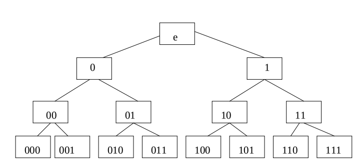
We can thus trust that given $H(a, b)$, it will be tough to find another set of values c, d, such that $H(c, d) = H(a, b)$

Trees

Let T_k = binary tree with 2^k leaves

Label root node with e (empty string)

For a node v with parent u, label u with kb, where k = u's label and b = 0 if v is left child, 1 if v is right



Merkle Trees

We will use *Merkle trees* to store our ZK set

How does a Merkle tree work?

- leaves may store data items (values in database)
- find hash function H mapping two items x, y to value z
- for node with children a and b, store $H(a, b)$

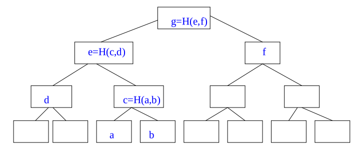
Value at root node is dependent on values in leaves

- represents a commitment to a particular tree

Merkle Trees

To prove that node x stores a:

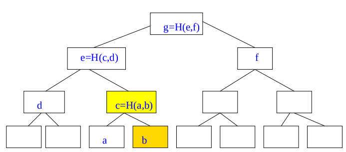
- look at nodes along path from root to x
- using values stored in nodes' siblings, calculate ancestors
- compare result at root to commitment



Merkle Trees

To prove that node x stores a:

- look at nodes along path from root to x
- using values stored in nodes' siblings, calculate ancestors
- compare result at root to commitment

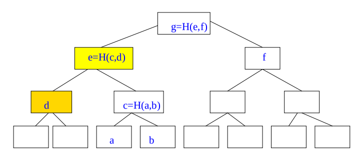


Merkle Trees

Say we have value stored in root, g

To prove that node x stores a:

- look at ancestors of x all the way up to root
- using values stored in nodes' siblings, calculate ancestors
- compare result at root to commitment

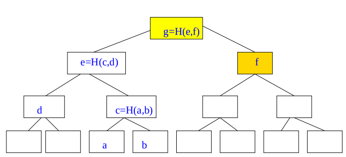


Merkle Trees

Say we have value stored in root, g

To prove that node x stores a:

- look at ancestors of x all the way up to root
- using values stored in nodes' siblings, calculate ancestors
- compare result at root to commitment

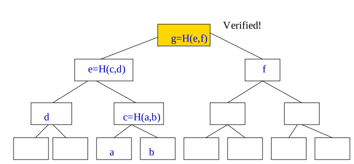


Merkle Trees

Say we have value stored in root, g

To prove that node x stores a:

- look at ancestors of x all the way up to root
- using values stored in nodes' siblings, calculate ancestors
- compare result at root to commitment



Merkle Trees

Path from root to x with siblings represents *authentication path*

Given M's root value, can we compute two different authentication paths to prove y and z both stored in x?

- choose a good hash function (e.g. Pedersen's), and this is infeasible

ZK EDB - Commitment

What we want to do:

- create a commitment using Merkle trees, as described above
- but - don't want to give away too much info about values in tree

ZK EDB - Commitment

Each node in tree has a stored value and a commitment value corresponding to it

Verifier has a hash function H available to it (public info)

ZK EDB - Commitment

What we do:

- start w/hash function H (Pedersen's hash function)
- create Merkle tree for data
- for key/value pair (x, y), store $H(y)$ in tree leaf $H(x)$
- store 0 in empty siblings as needed
- calculate commitment c for each leaf using Pedersen's commitment scheme
- for parent p of nodes a and b, store $H(a, b)$ in p
- calculate commitment c for p using Pedersen's commitment scheme

Do this in recursive, bottom-up fashion through Merkle tree

Final commitment given to verifier by prover is commitment c for root

ZK EDB - Verification

Now, say we want to prove (x, y) is key/value pair in database

- give values and commitments for each node from leaf up to root, along with its sibling
- (actually, we give more than this - details, details)
- verifier checks that root commitment matches original commitment
- verifier confirms, using hash function H, that prover did not "cheat"

ZK EDB - Verification

If we want to prove x is NOT a key in database, it's trickier

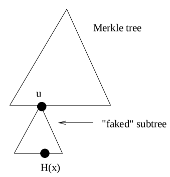
- $G(x)$ may not be in Merkle tree
- we could show that ancestor node of $G(x)$ in binary tree is a leaf, and therefore $G(x)$ is not in tree
- but this would show too much info about tree!
- we know about non-members of set ...

ZK EDB - Verification

We apply clever technique to “fake” nodes in Merkle tree

- set up commitment so that we can change it for empty leaves

To prove $D(x) = \perp$ create a fake subtree containing node $G(x)$ containing 0, fill in values of parents as needed, and give path from $G(x)$ to root



ZK EDB - Verification

To prove $D(x)$ not in database, we “weld” a new subtree to our tree:

- find furthest leaf u in tree on path from root to x
- fill node $G(x)$ with 0, calculate commitment c
- calculate commitments for parents until we reach leaf u
- give u a new “fake” commitment to match new hash values from subtree

Give nodes in path from $G(x)$ to root and their siblings; V will see correct proof but not know that other nodes are really empty

Additional Notes

The construction described can be enhanced so:

- one can not show whether $x \in D$ and exactly what $D(X)$ is (anonymous statistics)
- prove portions of info in $D(x)$
- only certain people may read $D(x)$, or portions of $D(x)$
- database can be distributed in nature

Open Questions

Can a ZK set be updated at low cost?

Can we handle multiple provers?

Can we consider other ZK operations/data structures as well?