Updatable Zero-Knowledge Sets

Moses Liskov Silvio Micali

Abstract

We build on the work of Micali, Rabin, and Kilian [4] to introduce zero-knowledge sets and databases that may be updated in a desirable way. In particular, in order to make an update the owner of the set must publish a commitment to the update, and update the commitment to the set. The update should take time independent of the size of the set. In addition, the update should not leak which key was added (or removed), or what data is associated with that key. Furthermore, our update will be transparent in that those already possessing a proof of a particular key being present or absent should be able to update their proofs to obtain a valid proof relative to the updated set, except if their proof is relative to the element that was changed.

1 Introduction

Recently, the two primitives zero-knowledge sets and zero-knowledge databases were introduced by Micali, Rabin, and Kilian [4]. This is a data structure which contains a set of elements from some universe, published by the set owner in the form of a commitment. The commitment specifies the set but also leaks no information about what elements are in the set, or even the size of the set. Once the set is committed, the set owner can prove, for any given element of the universe x, that x is in the set, or that x is not in the set, while still not revealing any further information.

This is a valuable primitive in itself, but it has a major disadvantage in that it is *static*. Once a set is committed, it cannot be changed, because the commitment would then have to change. Naively, the only way to change the set would be to create a new commitment on the updated set. However, this has two significant drawbacks. First of all, the running time of this update depends on the size of the set – not only is this undesirable from an efficiency standpoint, but it could also reveal information about the size of the set to an attacker. Second, once the new set is constructed, proofs relative to the old set are completely valueless.

Instead, we should set the bar higher. We ask that an update take time proportional only to the security parameter, not to the size of the set. Furthermore, instead of an "opaque" update, we seek a "transparent" one, in that an update should allow the holder of a proof relative to the old set to know whether their proof is valid or not with respect to the new set (except if their proof is relative to the element that was added or removed), and indeed, to be able to update their proof. Except for this, no information should be leaked by the update.

We show how to make efficient transparent updates based on the Micali-Rabin-Kilian construction of zero-knowledge sets, based only on two simple assumptions: (1) that the size of the set be polynomial in the security parameter, and that (2) verifiable random functions [5] exist.

2 Prior Work

Zero knowledge sets were introduced in [4]. The following is a summary of their construction.

2.1 Notation

S will denote a set of *keys* $\{x_1, \ldots, x_n\}$. Each x_i is a string of unbounded size. A database D is a set of pairs $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ such that for any x there is at most one y such that $(x, y) \in D$. We denote by $[D]$ the support of D, that is, the set $\{x_1, \ldots, x_n\}$. To indicate that $x \notin [D]$ we write $D(x) = \perp$. If $x \in [D]$ we write $D(x) = y$ to indicate the unique string y such that $(x, y) \in D$.

2.2 Micali-Rabin-Kilian construction

The construction works in the public random string model. That is, there is a common random reference string σ. From σ, the values p, q, g, and h are derived for use in the Pedersen commitment scheme.

In order to force every key to be of length k, we first hash them to obtain the database $H(D)$. We will construct a key of depth k. Every node in the tree can be labelled by a string $\sigma \in \{0,1\}^{\leq k}$. At each node σ there will be the following values associated:

- a generator h_σ,
- a value v_σ,
- a commitment c_σ,
- an exponent e_σ, and
- a random value r_σ.

The actual tree that is computed will only consist of the part that either has a key as a descendent, or has no keys as descendents, but has a sibling that does. That is to say, if a node has no descendents (that are keys), it will (1) have no children computed, and (2) its parent must have at least one descendent.

Any node that has descendents will be such that $h_\sigma = h^{e_\sigma}$, while any other node will be such that $h_\sigma = g^{e_\sigma}$.

The value v_σ is determined as follows. If $\sigma = H(x)$ for some $x \in [D]$ then $v_\sigma = H(D(x))$. If $|\sigma| = k$ but $\sigma \neq H(x)$ for any $x \in [D]$ then $v_\sigma = H(\perp)$. Otherwise, the value $v_\sigma = H(c_{\sigma 0} h_{\sigma 0} c_{\sigma 1} h_{\sigma 1})$.

The value c_σ is a commitment to v_σ. Namely, $c_\sigma = g^{v_\sigma} h_\sigma^{r_\sigma}$.

The commitment to the set is $c_\varepsilon, h_\varepsilon$

In order to prove that an element x is in the database, the set owner gives a proof consisting of:

- $D(x)$, so that $H(D(x))$ is the value $v_{H(x)}$.
- For every σ that is a prefix of $H(x)$, the values $e_\sigma, r_\sigma, c_\sigma$.
- For every σ that is a prefix of $H(x)$ followed by the opposite bit of $H(x)$, the values c_σ, h_σ.

The verifier checks that every c_σ where σ is a prefix of $H(x)$ is properly formed by checking that $c_\sigma = g^{v_\sigma} h^{e_\sigma r_\sigma}$, where $v_{H(x)}$ is $H(D(x))$ and otherwise, v_σ is given recursively, where the helper values c_σ and h_σ for the siblings are needed. Finally, the verifier checks that $c_\varepsilon, h_\varepsilon$ is the commitment to the set.

In order to prove that an element x is *not* in the database, the set owner gives a proof consisting of

- The value $e_{H(x)}$.
- For every σ that is a prefix of $H(x)$, the values $h_\sigma, r_\sigma, c_\sigma$,

- For every σ that is a prefix of $H(x)$ followed by the opposite bit of $H(x)$, the values c_σ and h_σ.

The verifier checks as before, except that the verifier checks that $g^{e_{H(x)}} = h_{H(x)}$. The interesting thing here is that these values may not have been computed previously by the set owner; but if that is the case, then the h_σ values are actually known powers of g, so the set owner can simply randomly generate those values on the fly.

2.3 Other work

Other related work on zero-knowledge sets include papers by Kilian [1], Micali and Rabin [3], and Ostrovsky, Rackoff, and Smith [6].

3 Transparent versus Opaque updates

In this section, we describe what we desire for both opaque updates and for transparent updates.

First of all, for either kind of update, we require that an update that ends up putting an element in the set be computationally indistinguishable from an update that removes and element from the set. (This should still be the case even if we "remove" an element that was already absent or add (or change data) for an element that was already present.)

An opaque update should be *adaptive chosen-query secure.* What this means is that the adversary should be empowered, once learning the commitment to the original set, to make repeated queries for proofs about keys of its choosing from that set until making a signal that it is ready for the set to be updated. Then, once given the update (which adds or deletes a key that was queried), the adversary should again be allowed to make repeated queries for proofs about keys of its choosing. Even given all this power, the adversary should not be able to predict whether any given key is present or not present in the database with probability non-negligibly greater than ought to be the case.

A transparent update, on the other hand, is a little more difficult to describe. First of all, given just the update, one should be able to compute $U(x, \pi)$ where $U(x, \pi)$ is either $\perp$ or a new proof π' such that (1) if π is not a proof that x is in S or that x is not in S, then $U(x, \pi) = \perp$, (2) if π is a proof about x, then $U(x, \pi) = \pi'$ if the update did not change x, and π' proves the same thing that π did but relative to the updated set, and (3) if x was changed, then $U(x, \pi) = \perp$. This property will be called *transparency completeness.*

This implies that an adversary will learn, for every x queried before an update, whether x was updated or not. We want to say that the adversary learns nothing more. Suppose we empower the adversary to make adaptive queries to the database before and after the update. We ask that the adversary not be able to predict the membership of any key not queried with probability non-negligibly greater than $1/2$. In the case where the element updated was queried by the adversary before the update but not after, we also ask that the adversary not be able to predict whether that element is in the set or not with probability non-negligibly greater than $1/2$.

Furthermore, we want that the adversary should have no information about the key that was updated unless the adversary happens to query that value before the update.

These definitions are making certain implicit assumptions about how updates are chosen in these experiments, which in this extended abstract we will not discuss. In the full paper we will do away with these assumptions and give rigorous definitions.

Note that an update scheme cannot be both opaque and transparent: if it were transparent then this would give a way to break opaqueness, and if it were opaque then a transparent update procedure would be a contradiction.

4 Our construction

4.1 First attempt

Suppose that we wish assign a particular value y (possibly $\perp$) to $D(x)$, for a given x, in a zero-knowledge database already constructed.

Here is a proposal: we regenerate the values $e_{H(x)}$ and $r_{H(x)}$ and from this recompute the values in the tree going up, leaving everything else the same. Now, for every prefix σ of $H(x)$, the value v_σ may change, so the value c_σ may also change. The set owner then publishes h_ε and c_ε anew.

This is almost correct. In order to make this a zero-knowledge set, we must be careful when adding an element to the set that all its ancestors are such that $h_\sigma = h^{e_\sigma}$ where e_σ is known, so we can prove membership of that element. Thus, when we add an element to the set that was previously not in the set, we must also change h_σ for every prefix σ of $H(x)$ so that $h_\sigma = h^{e_\sigma}$ rather than g^{e_σ}.

Now, the set committed to is a zero-knowledge set in itself, just as was constructed before. However, as it stands, this is neither a transparent nor an opaque update. There is no clear way to update proofs. Furthermore, it is not opaque. If an adversary had a proof of an element x before, it would see a hash path in the tree. If the adversary then made a query after the update about an element y such that $H(x)$ was close enough to $H(y)$ (when x is not updated) then the hash paths for x and y would share several pieces at the top; the adversary would learn how many of these were updated in the update, and if not all were updated, the adversary would be assured that x remained as before.

4.2 Second attempt

Suppose we do as above, but publish not just c_ε and h_ε but instead publish c_σ and h_σ for every prefix σ of $H(x)$. We do not publish σ along with these, just the values, though we publish them in order of the length of the string σ.

Now, this becomes a candidate for a transparent update. The key observation is that if one knows a hash path in the original tree corresponding to a string τ one can figure out the longest prefix that both σ and τ share. Say this longest shared prefix has length m. Once we do this, we can update the proof: the top m nodes in the hash path become the updated ones published; since the r values have not changed, we still have a decommitment. The top $m-1$ siblings remain the same as before, while the mth one becomes the $m+1$st published values. If the proof was a proof of membership, all the e values remain the same, as long as $m \neq k$ where k is the depth of the tree: in that case, e changes at the bottom, but this is fine since that corresponds to when the element was actually the one updated.

In order to figure out m given τ, the published values along σ, and the hash path along τ, we do the following.

We can compute the value v_ε in two different ways: either assuming the published value is on the left or is on the right. Since the hash path along τ includes both of the old values, we have all the values we need. Furthermore, we have r_ε so we can check which value is the correct one. This gives us the first bit of σ. If the first bit σ_0 of σ is the same as the first bit of τ, we will be in the same position: we will be able to compute v_{σ_0} since we have all the values $c_{\sigma_0 0}, h_{\sigma_0 0}, c_{\sigma_0 1}, h_{\sigma_0 1}$ in our hash path along τ. Thus, if $\sigma_0 = \tau_0$ we can find σ_1, and so on. Thus, we can keep finding σ_i as long as $\sigma_{i-1} = \tau_{i-1}$. When this stops working, we will know m such that $\sigma_i = \tau_i$ for all $i \leq m$ but $\sigma_{m+1} \neq \tau_{m+1}$.

Is this secure? Not quite; there are two things to notice. First of all, notice that if σ and τ coincide enough that an adversary gets to see an h_x value change, then the adversary knows that all points below the point where τ diverges from σ correspond to elements not in the set. Also, more directly, the adversary learns some several bits that begin $H(x)$ and this might give information about x, since hash functions do not necessarily obscure their inputs substantially.

4.3 Third attempt

One final modification is needed to deal with both the problems of the second construction. In order to make this modification we introduce a new primitive which may be interesting independently. Verifiable random functions were described by Micali, Rabin, and Vadhan [5]; they consist of a method of producing a public key PK such that to each PK is an SK and the SK allows one to compute a pseudorandom function and to give a proof relative to the public key PK that this particular output is correctly computed. From this we define the notion of a *verifiable pseudorandom permutation* which is just a VRF that is also a permutation. This can be easily constructed given a VRF: in order to make a VRP, we just choose three public keys for the VRF and then use each of those as individual seeds in the Luby-Rackoff [2] construction of a pseudorandom permutation from a pseudorandom function. A proof consists of each of the pseudorandom function values on the appropriate inputs along with proofs of those relative to the appropriate public key.

Now to return to updatable zero-knowledge sets, instead of storing a key x under $H(x)$, we will store x under $P(H(x))$ where P is a verifiable random permutation. We make the key for the VRP part of the commitment to the set. A proof of an elements membership or non-membership is as before, plus a proof that $P(H(x))$ is properly computed from PK and $H(x)$, as given by the pseudorandom permutation.

What's the point? Now, both concerns are dealt with. As for the first, though a range of values of $P(H(\cdot))$ are known not to correspond to elements in the set, it is impossible to be certain that any particular value is in that range. As to the second, even revealing $P(H(x))$ entirely should give no information about $H(x)$ or, by extension, x.

A final possible objection is the following: in the first concern, we still are left with a range of values known not to be in the set; while none of the values are known, this does make an implication about the size of the set. However, since the size of the set is assumed to be polynomial in k, and the tree is perfectly balanced, we expect this range to be small enough that it doesn't matter. This is a highly technical point we will expand upon in the full version of this work.

5 Conclusion and open problems

We have given a construction for an efficiently transparent-updatable zero-knowledge set, assuming the set is of size polynomial in k. Other open problems we will touch on in the full version include:

- Efficiently opaque-updatable zero-knowledge sets
- Other update operations than merely adding or deleting an element
- The issue of known distribution of updates in our definitions.

References

[1] J. Kilian. Efficiently committing to databases. TR 97-040, NEC Research Institute, 1997.

[2] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, Berkeley, California, 28–30 May 1986.

[3] Silvio Micali and Michael Rabin. Hashing on strings, cryptography, and protection of privacy. In *Proceedings of Compression and Complexity of Sequences*, page 1, Los Alamitos, California, 1997. IEEE Computer Society.

[4] Silvio Micali, Michael Rabin, and Joseph Kilian. Zero-knowledge sets. In *44th Annual Symposium on Foundations of Computer Science*, Cambridge, October 1999. IEEE.

[5] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science*, pages 120–130, New York, October 1999. IEEE.

[6] R. Ostrovsky, C. Rackoff, and A. Smith. Personal communication. Personal communication.