Updatable Zero-Knowledge Databases

Moses Liskov

Computer Science Department
The College of William and Mary
Williamsburg, Virginia, USA
mliskov@cs.wm.edu

Abstract. Micali, Rabin, and Kilian [9] recently introduced zero-knowledge sets and databases, in which a prover sets up a database by publishing a commitment, and then gives proofs about particular values. While an elegant and useful primitive, zero-knowledge databases do not offer any good way to perform updates. We explore the issue of updating zero-knowledge databases. We define and discuss *transparent* updates, which (1) allow holders of proofs that are still valid to update their proofs, but (2) otherwise maintain secrecy about the update.
We give rigorous definitions for transparently updatable zero-knowledge databases, and give a practical construction based on the Chase et al [2] construction, assuming that verifiable random functions exist and that mercurial commitments exist, in the random oracle model. We also investigate the idea of *updatable commitments*, an attempt to make simple commitments transparently updatable. We define this new primitive and give a simple secure construction.

Keywords: zero-knowledge databases, zero-knowledge sets, transparent updates, zero-knowledge, protocols, commitments, updatable commitments

1 Introduction

Recently, zero-knowledge databases were introduced by Micali, Rabin, and Kilian [9]. A zero-knowledge database is a finite partial function D mapping binary strings to binary strings (i.e., a set of pairs of strings (x, y) such that no two pairs have equal first entries but different second entries).[1] The *database owner* chooses D and "publishes" the zero-knowledge database in the form of a commitment that pins down the database but leaks nothing, not even its size. Once the database is committed, the set owner acts as a *prover*: on a query x, the prover gives a proof that either x lies outside D or $D(x) = y$, while still not revealing any further information about D. Commitments and proofs in a zero-knowledge database are *non-interactive* and done in the common random string model. Zero-knowledgeness is shown by exhibiting a polynomial-time simulator that produces a full transcript distribution (i.e., the commitment and the proofs to all query strings) identical to that of the real prover, knowing only "$D(x) = y$" or "x is not in D" for each query and at the last possible moment. While it is conceptually simpler to deal with computational zero-knowledge (and in fact computationally zero-knowledge databases were provided in earlier versions of their paper [5, 8]), the Micali-Rabin-Kilian solution is more desirable because it is perfect zero-knowledge. Further, it is much more efficient as it does not involve complex general purpose non-interactive zero-knowledge proofs.

Zero-knowledge databases are a powerful primitive, but they have a major disadvantage in that they are static. This seems like an undesirable property in most applications. For example, if the database were a list of people under investigation for criminal activities, updates would be a critical part of the system. Naively, the only way to update a zero-knowledge database would be to commit to its new version from scratch. However, this is undesirable in two significant ways.

[1] Micali, Rabin, and Kilian call these simple databases "elementary" databases. All databases in this paper are of this simple type.

– First, the running time of such an update depends on the size of D, which may be huge, even though the newest version may differ only on a single pair (x, y).
– Second, it may be that those who have seen proofs of membership or non-membership in the original set may be entitled to, or may request again, the same proofs in the new set (for example, if proofs are given due to subscription to some service). If this is the case, the owner would have to reissue old proofs, which could be a huge additional expense.

The second of these points brings up a question that is of interest: when updating such a database, should the proofs be updated as well, or should the new set be private even against those with old proofs?[2] Depending on the application in which the zero-knowledge set is used, either one may be the desirable kind of update. We distinguish these two types of updates by giving them different names:

– *opaque* updates make the updated commitment indistinguishable from a new commitment (hence, the database becomes “opaque” to the users after the update);
– *transparent* updates allow the users to determine whether their proofs are still valid, and provide a mechanism to update proofs (hence, “transparent” to proof holders).

We focus on the problem of transparent updates for two reasons: first, we believe it is the more desirable of the two, as the idea of a subscription service of some type seems to naturally fit the idea of a zero knowledge database, and second, an inefficient but adequate method exists for opaquely updatable zero-knowledge sets, namely, reconstructing the updated commitment from scratch, while no method exists for transparently updatable zero-knowledge sets.

In this paper, we define the notion of transparently updatable zero-knowledge databases, and show how to construct efficient transparently updatable zero-knowledge databases both based specifically on the Micali-Rabin-Kilian construction and on the more general construction of Chase et al [2], under the additional assumption that verifiable random functions exist in the random oracle model. We also define the notion of an updatable commitment and give a computationally hiding, perfectly binding secure updatable commitment scheme.

In appendix B, we discuss the problem of opaquely updatable zero-knowledge databases.

1.1 Related work

Zero knowledge sets were introduced in the work of Micali, Rabin, and Kilian [9]. Important precursors to zero knowledge sets appeared in earlier papers by those authors [5, 8]. Chase, Healy, Lysyanskaya, Malkin, and Reyzin [2] describe the notion of *mercurial commitments*, that is, commitments that can be “hard” or “soft,” an abstraction of the type of commitments used in the Micali-Rabin-Kilian construction, and show that any mercurial commitment scheme can be used to construct zero-knowledge databases. Recent work by Ostrovsky, Rackoff, and Smith [11] greatly enlarges the functionality of zero-knowledge databases by allowing more complex queries (e.g., “does the database’s support intersect a given string interval?”). They first design a data structure that, without any privacy concerns, efficiently handles complex queries, and then augment it with zero-knowledge proofs so as to provide privacy, constructing zero-knowledge sets under general assumptions.

1.2 Structure of the paper

In section 2, we give notation to be used in the rest of the paper. In section 3, we define the security properties needed for updatable zero-knowledge databases. In section 4, we summarize various primitives and previous work, and introduce the notion of updating commitments. In section 5, we give a construction for transparently updatable zero-knowledge databases. In section 6, we discuss the efficiency of our construction. We conclude and discuss open problems in section 7.

[2] It is possible that neither will hold, but it seems natural that we should want one of these.

2 Notation

We shall follow in our notation from many previous papers, particularly from [9, 1].

Probabilistic assignments and experiments. By $x \leftarrow M$ we indicate that the variable x is assigned according to M. If M is a finite set, we assume x is drawn from the uniform distribution on M. The notation $x_1 \leftarrow M_1; x_2 \leftarrow M_2; \ldots$ denotes the probability distribution that arises when we first assign x_1 from distribution M_1, then x_2, et cetera. If p is a predicate, then the notation $Pr[x_1 \leftarrow M_1; x_2 \leftarrow M_2; \ldots : p(x_1, x_2, \ldots)]$ denotes the probability that p is true given that distribution.

Databases. A database D is a set of pairs $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ such that for any database key x there is at most one y such that $(x, y) \in D$. Each x_i and each y_i is a string of unbounded size. 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 $(x, y) \in D$. By $D(x) \leftarrow y$ we mean that D shall be changed so that $D(x) = y$. This may involve exchanging one pair (x, y') for (x, y), or adding (x, y) to the set, or if $y = \perp$, removing the pair (x, y') if any such pair is present.

Polynomial-time adversaries. For the purposes of our definitions, adversaries are specified as Turing machines that repeatedly make outputs of the form (w_i, s_i), where w_i is some query and s_i is state information the adversary will use to make the subsequent query. When we assume that such an A is a *polynomial-time adversary*, we assume that not only is A a polynomial-time algorithm, but that A will ultimately make only polynomially many queries before halting.

Adversary views. If A is an adversary, we define $\text{View}_A\{x_1 \leftarrow M_1, \ldots, x_n \leftarrow M_n\}$ to be a random variable representing the randomness, inputs, and outputs of the adversary A through the computation of the values $x_1, \ldots, x_n$ according to the given probabilistic experiment. Presumably, some of the probabilistic assignment sources M_i involve the adversary A, or the view would be trivial.

Binary trees. We use string notation to specify nodes in a binary tree. ϵ will be the root of the tree. If v is a node in the tree, $v0$ will be the left child of v while $v1$ will be the right child. Values that are stored in a tree at each node will have this string as a subscript; for example, a_ϵ would be the value of a stored at the root node ϵ. If the depth of the tree is bounded by k, the longest strings that refer to nodes in the tree will be of length k. We mean by a *prefix* of a string s any string ω (including s) such that there is a string s' such that $\omega s' = s$. Note that if ω is a prefix of s, then ω will be a node that lies on the path from ϵ to s in a binary tree.

3 Definitions

Our goal in this section is to rigorously define transparently updatable zero-knowledge databases.

3.1 Mechanics

As with zero-knowledge databases, updatable zero-knowledge databases rely on a public random string σ, the *reference string.* This string must have length polynomial in k, the security parameter.

There are three types of tasks the prover will have to be able to perform. First of all, she will have to be able to commit to the database initially. Second, she will have to be able to issue proofs of membership or non-membership in the database for any key. Finally, she will have to be able to issue updates to the database.

A verifier should be able to verify proofs and to update proofs.

Transparently updatable database systems We say that a quintuple of Turing machines, $(\mathsf{Commit}, \mathsf{Prove}, \mathsf{DBUpdate}, \mathsf{Verify}, \mathsf{PUpdate})$, constitute a transparently updatable database system or TUDB system if none of the machines retain state information after an execution and their computation on common inputs 1^k, a unary string called the *security parameter*, and σ, a binary string called the *reference string*, proceeds as follows:

- The database commitment algorithm is Commit. On input $(D, 1^k, \sigma)$, Commit produces two outputs: (1) a string PK, called D's *public key* (or *commitment*), and (2) a string SK, called D's *secret key.*
- The database proof algorithm is Prove. On input $(D, 1^k, \sigma, PK, SK)$, and an additional input $x \in \{0,1\}^*$, Prove outputs a string π_x, called D's *proof* about x.
- The database update algorithm is $\mathsf{DBUpdate}$. On input $(D, 1^k, \sigma, PK, SK)$, an additional input $x \in \{0,1\}^*$, and a value $y \in \{0,1\}^* \cup \{\perp\}$, $\mathsf{DBUpdate}$ computes a new public key PK' and a new secret key SK' for the updated database in which $D(x) = y$, and a string U called the *update information* about x and y, which will be used to update proofs.
- The proof verifying algorithm is Verify. On input $(1^k, \sigma, PK)$ and an additional $x \in \{0,1\}^*$ together with its proof π_x, Verify outputs either a string $y \in \{0,1\}^*$ (meaning that it believes $y = D(x)$, $\perp$ (meaning that it believes that x is outside D's support), or *reject* (meaning that it detected cheating).
- The proof update algorithm is $\mathsf{PUpdate}$. On input $(1^k, \sigma, PK, PK', U)$, and an additional $x \in \{0,1\}^*$ together with its proof π_x, $\mathsf{PUpdate}$ outputs either a new proof π'_x, which will be called the *updated proof about* x, $\perp$ (meaning that the update given by PK', U was about x and so the proof cannot be updated), or *reject* (meaning that it detected cheating).

3.2 Security properties

Updatable zero-knowledge databases must satisfy certain security properties: completeness, soundness, and zero-knowledge. We first describe the desired properties informally, and then formalize our definitions.

Completeness dictates that if the prover and verifier are honest, then for any database, if the prover updates the database any number of times, then gives the verifier a proof about x, and then updates the database any number of times, the verifier may update their proof and obtain a valid one, except with negligible probability, so long as $D(x)$ was not updated after the proof was issued.

Soundness guarantees that the prover is in fact committed to a particular database. That is, given the reference string σ it should be hard for any prover to come up with a PK and any element for which it can prove two different values.

The *zero-knowledge* property of updatable zero-knowledge databases is trickier to describe. Ideally, the adversary should learn nothing more than the values of elements for which a proof has been obtained (and possibly updated), and that updates have occurred. However, we have not been able to realize this full level of security, and instead offer a weaker but acceptable notion of security. Each key x that might be included in the database will have a *pseudonym* $N(x)$. Instead of revealing only that an update has occurred, we reveal that an update has occurred about the key relating to a particular pseudonym. Thus, the pattern of updates is revealed (since the pseudonym is constant for a constant x, so repeated updates on keys can be discovered). In addition, the link between a value x and its pseudonym $N(x)$ will be revealed by Prove. However, we require that no information beyond this be revealed.

This alone does not constitute a high enough level of security: $N(x)$ could reveal information about x. One particular N that is desirable is one that answers 1 to its first input, 2 to its second distinct input, and so on. We call this pseudonym the *pattern pseudonym* N_P, as revealing $N_P(x)$ for many x is equivalent to revealing the pattern of values.

To say this more clearly, a system is zero-knowledge with respect to pseudonym N if, even given any adversary A and any database D the views of A in each of the following two experiments are indistinguishable.

1. First, a random reference string σ is chosen. Then, D is chosen by A and given to the prover, who creates an updatable zero-knowledge database based on D and σ, committing to it with PK while keeping SK private. Then the adversary adaptively chooses a sequence of strings $x_1, x_2, \ldots$ where either $x_i = \mathsf{Query}(x)$ or $x_i = \mathsf{Update}(x, y)$. When x_i is a query, the prover returns a proof π_i that either x is in the database or that x is not in the database. When x_i is $\mathsf{Update}(x, y)$, the prover updates so that $D(x) = y$ and sends PK_i, U_i to the adversary.
2. The simulator Sim, on input only the security parameter k, produces a string σ of the proper length, and a public key PK. The adversary adaptively chooses a sequence of strings $x_1, x_2, \ldots$, where either x_i is either $\mathsf{Query}(x)$ or $\mathsf{Update}(x, y)$. If $x_i = \mathsf{Query}(x)$, the simulator is told x, $N(x)$, and $D(x)$, (where D is up to date, starting with the initial D), and must compute π_i. If x_i is an update $\mathsf{Update}(x, y)$, the simulator is given $N(x)$ and must compute SK_1, PK_1, U_1, while D is updated so that $D(x) = y$. Note that the pseudonym function N is not part of the adversary or the simulator here, but rather is thought of as an oracle that is only called when the game specifies.

In the first scenario, there is no pseudonym function. In the second, the pseudonym function exists, however, the adversary is not directly aware of its presence; the adversary specifies updates $\mathsf{Update}(x, y)$ which get translated into $N(x)$ for the simulator.

The concept of pseudonyms seems inevitable in any zero-knowledge database construction. A zero-knowledge database is in some sense a committed tree, and a particular element must have a unique place to reside (so that we can prove non-membership), which can be thought of as its pseudonym. Furthermore, we cannot use zero-knowledge proofs that reveal nothing about the data structure – the user has to learn enough to allow them to update, but this seems to be the only way to avoid revealing pseudonyms. We have not been able to conceive of a system that does not use pseudonyms, or that uses them but does not reveal them.

We say a transparently updatable database is secure if it is complete, sound, and zero-knowledge with respect to the pattern pseudonym N_P. We say it is secure with respect to N if it is complete, sound, and zero-knowledge with respect to N. Thus, while we may talk about security with respect to other pseudonyms, we regard N_P as the only truly acceptable one.

Efficiency properties In order for us to consider an updatable zero-knowledge database efficient, we ask that:

– The running time of the procedure that generates the initial commitment may depend on the size of the database, but all other running times must be independent of the size.
– None of the sizes of the outputs other than SK may depend on the number of updates.
– None of the running times of any of the verifier algorithms may depend on the number of updates that have been performed (in a sense limiting total performance to linear in the number of updates, since some procedures are performed once per update).

3.3 Formal definitions

We formalize our definitions in appendix A.

4 Preliminaries

Before we present our construction, we first review some crucial building blocks used in our construction. Some of our text follows closely from the preliminaries section from [9].

4.1 Updatable commitments

Here, we define updatable commitments. In an ordinary commitment scheme, there are two algorithms: $\mathcal{C}$, which takes a message m as input and produces c and d, where c is the commitment, and d is the information used to open the commitment later, and $\mathcal{V}$, which takes a commitment c, a message m, and a decommitment d, and checks whether c was a commitment to m, using d. Note that there may also be public parameters which are inputs to all algorithms, but for clarity we simplify.

In an updatable commitment, there will be one more algorithm: $\mathcal{U}$, which takes a message m and decommitment information d, and produces a commitment c, where d will be the decommitment information used to open c. The binding property is defined in the natural way. The hiding property is essentially that commitments be indistinguishable under a chosen message attack, where the adversary may ask for commitments, updated commitments, and decommitments of his choice, so long as he doesn't ask for a decommitment of the challenge or any message derived from the challenge through updates.

Our construction. Our construction is quite simple. Given a secure perfectly binding commitment scheme and a secure pseudorandom permutaiton P, we can construct a simple computationally hiding, perfectly binding commitment scheme as follows:

$\mathcal{C}(m)$: generate a key K for the pseudorandom permutation, a random string IV, and compute c_1, a commitment to K under the commitment scheme and d_1, the related decommitment information, and c_2, the evaluation of the pseudorandom permutation on $m \oplus IV$ with key K. Output $c = (c_1, c_2, IV)$, and $d = (K, c_1, d_1)$.
$\mathcal{V}((c_1, c_2, IV), m, (K, c_1, d_1))$: check that c_1 is a commitment to K using d_1. If not, reject. Then, check that $c_2 = P_K(m \oplus IV)$. If so, accept, if not, reject.
$\mathcal{U}(m, (K, c_1, d_1))$: compute $c_2 = P_K(m)$ and output $c = (c_1, c_2)$.

It is clear that any commitment is a commitment to one specific value, since c_1 specifies a unique K, and given that K, c_2 specifies a unique m. Furthermore, c_2 is the encryption of the one-block message m under CBC mode, so if this scheme is not hiding, then either the PRP is not pseudorandom or the underlying commitment scheme is not hiding. This is true even if K is used for many different commitments, so long as K is never revealed.

4.2 Mercurial commitments

Mercurial commitments were introduced recently by Chase et al [2] with direct application to zero-knowledge sets and databases. A mercurial commitment is a commitment scheme in which there are two kinds of commitments and two kinds of ways to decommit.

- A "hard commitment" is a commitment to a particular value. It can only be decommitted to that value, whether the decommitment is a hard or a soft one.
- A "soft commitment" is a commitment to no value. It can never be hard-decommitted, but it can be soft-decommitted to any value.

A mercurial commitment scheme is secure when it is hiding (in the sense that the type of a commitment is kept secret as well as the value if the commitment is a hard commitment) and binding (in the sense that the committer cannot break the above rules.) Mercurial commitments have a non-interactive commitment and decommitment, but require the public random string model. In fact, they also have a trap-door property: if the public random string is chosen by a simulator, the simulator can avoid the binding properties.

4.3 Pedersen's commitment scheme

Pedersen's commitment scheme [12] assumes the availability of a public quadruple (p, q, g, h), where p and q are prime, $q|p-1$ and g and h are generators for G, the cyclic subgroup of Z_p^* of order q, for which computing discrete logarithms is assumed to be hard.

The commitment and verification algorithms are defined as follows, where all operations are performed modulo p:

$\mathcal{C}((p, q, g, h), m)$: randomly select $r \in Z_q$ and output (c, r), where $c = g^m h^r$ is the commitment string, and r is the (for the time being secret) proof.
$\mathcal{V}((p, q, g, h), c, m, r)$: If $c = g^m h^r$, then accept; else, reject.

This commitment scheme is perfectly hiding and computationally binding.

The mercurial commitment scheme used in [9] is based directly on this commitment scheme. Instead of using g as the base to compute g^m directly, we use a different base for each commitment: g^e for a hard commitment or h^e for a soft commitment, and publish the base that we use as part of the commitment (where e is random). A soft decommitment consists of publishing r; then, it can be checked that $c = b^m h^r$ where b is the base being used. A hard decommitment involves publishing r as well as e, so that it can also be checked that $g^e = b$.

4.4 CHLMR zero-knowledge databases

The following is a summary of the general zero-knowledge database construction of Chase, Healy, Lysyanskaya, Malkin, and Reyzin [2].

ZK databases. The construction works in the public random string model, that is, there is a common random reference string σ.

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

- A value v_ω. If $\omega = H(x)$ for some $x \in [D]$ then $v_\omega = H(D(x))$. If $|\omega| = k$ but $\omega \neq H(x)$ for any $x \in [D]$ then $v_\omega = H(\perp)$. If ω is an internal node, the value v_ω is defined recursively as $H(c_{\omega 0} c_{\omega 1})$ where c_ω is defined below. Essentially, the values v_ω make the tree a Merkle tree.
- A commitment c_ω which is either a soft commitment or a hard commitment to v_ω.
- Decommitment information d_ω for the commitment c_ω.

The commitment to the database is the commitment c_ϵ from the root node ϵ.

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

1. $D(x)$, so that $H(D(x))$ is the value $v_{H(x)}$.
2. For every ω that is a prefix of $H(x)$, c_ω and a hard decommitment of c_ω, and
3. For every ω that is a sibling along the path from ϵ to $H(x)$, the value c_ω.

The verifier uses this to construct the values v_ω for every ω that is a prefix of $H(x)$, and then checks the hard decommitments.

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

1. For every ω that is a prefix of $H(x)$, c_ω and a soft decommitment of c_ω to v_ω, and
2. For every ω that is a sibling along the path from ϵ to $H(x)$, the value c_ω.

The verifier checks as before, except that the verifier uses $D(x) = \perp$, and that the decommitments are soft.

The key to the efficiency of the construction is the use of mercurial commitments. If ordinary commitments were to be used, the entire tree of depth k would have to be computed, which is clearly exponential. However, the tree is constructed so that soft commitments are used for any node that has no descendents in the data set, which allows the prover to not compute those parts of the tree ahead of time, but allows the prover to compute those parts of the tree when necessary, *and* be able to decommit.

4.5 Verifiable random functions

Verifiable random functions or VRFs were first presented by Micali, Rabin, and Vadhan [10], and subsequent constructions appear in [6, 3]. A verifiable random function consists of four algorithms: a key generating algorithm GenVRF that produces a pair (PK, SK) on input 1^k, an algorithm $\mathsf{ComputeVRF}$ that computes $f_{SK}(x)$, an algorithm $\mathsf{ProveVRF}$ that gives proofs π that a value $y = f_{SK}(x)$ is correctly generated from x, and an algorithm VerVRF that verifies proofs, with the following informal properties:

1. If (PK, SK) are generated from GenVRF, and y is generated from $\mathsf{ComputeVRF}(SK, x)$ and π is generated from $\mathsf{ProveVRF}(SK, x)$, then $\mathsf{VerVRF}(PK, x, y, \pi)$ will accept.
2. f_{SK} is a pseudorandom function, even to an adversary that may request both outputs and outputs with proofs, so long as the two sets of queries do not overlap.
3. No adversary can produce a (PK, SK) pair for which it can give proofs that will be verified for incorrect values.

In particular, note that no adversary should be able to compute $f_{SK}(x)$ given x and PK.

5 Our construction

We describe our construction incrementally. First, we describe how to go about updating a CHLMR database efficiently. Then, we go on to describe how to provide update information that will allow proof holders to update their proofs. Then we give a construction with an unspecified pseudonym N and prove security relative to N. We then prove security in the random oracle model and discuss issues that arise relative to implementing the random oracle.

5.1 Updating a CHLMR database

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

Our first goal is to efficiently compute a new commitment to a CHLMR database with the updated value. This is fairly easy to do, and natural. Essentially, we just change the values at the leaf we are interested in, and update the internal nodes of the tree to maintain the required structure. To update the value $D(x)$, we regenerate the commitment $c_{H(x)}$ and from this recompute the values and commitments in the tree going up along the path from $H(x)$ to ϵ, leaving everything else the same. Now, for every prefix ω of $H(x)$, the value v_ω may change, so the value c_ω may also change. The set owner then publishes c_ϵ anew.

In order to make this fit all the properties of a ZK database, we must be careful when adding an element to the set that all its ancestors are hard commitments. Thus, when we add an element to the set that was previously not in the set, we must make commitments along the path hard commitments, even if they were previously soft commitments. In fact, we can simply make all commitments in any update hard commitments, to simplify.

5.2 A simple mechanism for updating proofs

Now, the updated database is a CHLMR database, just as was constructed before.[3] The next step is to determine what information is necessary to allow proof holders to update their proofs. Since a proof is essentially a hash path in the tree along with decommitments to the values along that path, and the only internal nodes or commitments that have changed are the ones along the path from ϵ to $H(x)$, we could just publish all the commitments at the updated internal nodes. However, this is not quite sufficient, because decommitments are necessary for the proofs to be complete. To solve this, we need to modify our mercurial commitment scheme so that it is updatable, but the requirements are a little more complex than the requirements for an updatable commitment. Specifically, we need to be able to update such that (1) the updated commitment is always a hard commitment, and (2) the holder of a decommitment (soft or hard) can update their decommitment to a new one of the same type.

Under general assumptions, the best known mercurial commitment is only computationally hiding. In order to make an updatable one, we need to combine a mercurial commitment scheme and an updatable commitment scheme as follows. Instead of publishing only the mercurial commitment c, we also publish c_H and c_S where c_H is an updatable commitment to the hard decommitment of c (or a random string if it is a soft commitment), and c_S is an updatable commitment, initially to a random string, but after any updates, to a soft decommitment of c. A hard decommitment involves opening c_H, while a soft decommitment involves opening c_S, and also giving a soft decommitment to c. This means a verifier will notice a difference between opening an original commitment and opening an updated one, but this will be acceptable for our means. Updating the commitment (c, c_H, c_S) is done by replacing c with a fresh commitment and updating c_H and c_S to be commitments to their new appropriate values.

We can also make the MRK mercurial commitment updatable in this way, simply by reusing r. When we update a commitment, we always make it hard, so we also publish e. It is worth noting that this is not as hiding as we might like such a commitment to be in isolation, since (for instance) the ratio between j^m and $(g^e)^{m'}$ is revealed, and an unbounded adversary could learn information from this. This costs us perfect zero-knowledge in our construction, but under the DDH assumption, this is still hiding. We should also note that updating commitments in this way does not give a mechanism for the verifier to determine m', but, in our application, m' can be derived from other information.

5.3 Attaining zero-knowledge with respect to N

Now we have a system where after an update we have a zero-knowledge database, and proofs can be updated. However, the updates do not preserve secrecy. The issue has to do with the pseudonym we use. Here, we use $H(x)$ as a pseudonym. In order to more carefully discuss the issue of our choice of pseudonym, we specify this construction by describing it in terms of an unspecified pseudonym $N(x)$.

[3] Except, some commitments might be hard that don't need to be hard commitments, but by the properties of mercurial commitments, this is an indistinguishable change

Commit$(D, 1^k, \sigma)$: Run the database commitment algorithm but instead of using $H(x)$ to define an element's position in the tree, use $N(x)$.

Prove$(D, 1^k, \sigma, PK, SK, x)$: run the database proof algorithm, looking for x at position $N(x)$ to obtain π_x.

DBUpdate$(D, 1^k, \sigma, PK, SK, x, y)$: create a new commitment $c_{N(x)}$ to $v_{N(x)} = H(y)$. Recursively, for each ω that is a prefix of $N(x)$, update c_ω to be a hard commitment of v_ω. Compute $PK' = c_\epsilon$, update SK' by remembering all the new decommitment information, and compute $U = \{\omega, c_\omega\}$ for all prefixes ω of $N(x)$.

Verify$(1^k, \sigma, PK, x, \pi_x)$: run the proof verifying algorithm to verify π_x, using $N(x)$ instead of $H(x)$, and check the value given as $N(x)$ to be sure it is correct.

PUpdate$(1^k, \sigma, PK, PK', U, x, \pi_x)$: if U is an update about $N(x)$, output $\perp$. (Note that $N(x)$ would be known from π_x.) Otherwise, for every ω that is a prefix of $N(x)$ and is included in U, we have a decommitment to the old c_ω, so we update our decommitment. For every ω that is a sibling along the path, we change our value of c_ω to the value of c_ω given in the update U. Finally, we check our updated proof, and reject if it does not yield the same value, otherwise we outpud π'_x, our updated proof.

Theorem 1. This scheme is a secure zero-knowledge transparently updatable database with respect to N.

Proof. Due to space constraints, we only provide a proof sketch here. A more detailed proof may be found in appendix C.

Completeness of this construction should be clear. Since the form of any database commitment and proof are just as in [2] except with a different scheme to assign database locations to database keys, soundness here follows from the soundness of their construction and the uniqueness of the mapping $x \mapsto N(x)$.

For zero-knowledgeness we must show a simulator that has the required properties. First of all, the simulator generates σ so that the mercurial commitment simulator can be used (that is, the simulator can break the binding property of the scheme). The simulator then generates a soft commitment c_ϵ and publishes it.

When the simulator is asked for a proof that $D(x) = y$ and is given x and $N(x)$, it simply does exactly as the CHLMR database simulator does, except that the path is a path from ϵ to $N(x)$. When the simulator is asked to update a value with a given pseudonym n, it performs an update just as DBUpdate would, using $y = \epsilon$, creating c_ω values for each ω that is a prefix of n for which c_ω was not already determined in a proof. (Note that DBUpdate does not need to know x if it knows $N(x)$.)

The values given in the proofs issued by the system are just sequences of commitments, decommitted to the correct values, so the distribution of the proofs given by the simulator and those given by the real prover are indistinguishable. The distribution of updates is also identical except that the simulator always sets $y = \epsilon$. However, the only value that depends directly on y is $c_{N(x)}$ which is a (fresh) commitment, so in fact the distribution of update strings is also indistinguishable. Thus, we achieve zero-knowledge.

5.4 Attaining security in the random oracle model

We now have a system that gives a transparently updatable zero-knowledge database with respect to N for an unspecified N. Unfortunately, we cannot simply specify $N = N_P$ and be done, because N_P cannot be computed in a way verifiable to the user. This problem can be solved by assuming the random oracle model. The idea is that we use a random oracle that may be controlled by the simulator to compute $N(x)$. It should be clear that a random oracle computed on x and a random oracle computed on $N_P(x)$ are identical. Thus, the simulator simulates the random oracle on input $N_P(x)$ by evaluating a random function on it. By doing this, the simulator may naturally compute $N(x)$ knowing only $N_P(x)$. Thus, such a simulator shows that if we use a random oracle as $N(x)$, our construction is secure.

5.5 Implementing the random oracle

Using the random oracle model has significant problems. First of all, random oracles are generally implemented by collision-resistant hash functions, but this cannot always be done securely. There is also an issue of pseudonym collisions, which we discuss this issue in appendix D.

Most importantly, though, we cannot simply use a public hash function here, because doing so would allow the adversary to query the pseudonym function, but it was one of our security requirements that the adversary not be able to do this. Ideally, the adversary should only be able to learn if a particular update was about x by querying the database at x.

The pseudonym function we propose to use is $H^*(x) = H(f(H(x)))$ where H is a hash function and f is a verifiable random function. We will still assume that H is a random oracle, but now, even if H is a random oracle, the adversary cannot query H^*. Before we jump into the security proof for this pseudonym, we must modify our construction slightly, because $H^*(x)$ cannot be computed by the verifier.

- In Commit we also run GenVRF and make the public key PK_f part of the public key, and keep SK_f as part of the secret key.
- In $\mathsf{Prove}(D, 1^k\sigma, PK, SK, x)$, we also give $\pi'_x = \mathsf{ProveVRF}(SK_f, H(x))$ and $z = \mathsf{ComputeVRF}(SK_f, H(x))$.
- In Verify, we additionally run $\mathsf{VerVRF}(PK_f, H(x), z, \pi'_x)$ and check that $H^*(x) = H(z)$ before accepting.

This fits nicely into our original specification; we are simply expanding the idea of what it means to check that $H^*(x)$ is correctly computed.

Theorem 2. This construction is secure in the random oracle model.

Proof. Again, we give only a sketch of the proof, due to space constraints. See appendix C for a full proof.

Completeness is already established by our proof of Theorem 1. To prove soundness, we need only note that the pseudonym $H^*(x)$ that will be verified is unique, from the soundness property of the VRF.

Zero-knowledge is more of a challenge. We give a simulator with respect to N_P that gives us computational zero-knowledge. First, the simulator makes σ and the database commitment c_ϵ just as the previous simulator does. The simulator then runs GenVRF to generate (PK_f, SK_f), and publishes (PK_f, c_ϵ) as the database commitment.

The simulator must answer three kinds of messages: random oracle queries, database queries, and update queries. The simulator maintains two random functions, H and H', with the idea that $H'(N_P(x)) = H(f(H(x)))$. When the simulator receives an update query, it computes $H^*(x) = H'(N_P(x))$. When the simulator receives a database query, the simulator computes $H(x)$, and then computes $z = f_{SK_f}(H(x))$, and then sets $H(z) = H'(N_P(x))$ and fakes a proof that the value stored at $H^*(x) = H'(N_P(x))$ is y, just as the simulator does in theorem 1.

The illusion that $H'(N_P(x)) = H(f_{SK_f}(H(x)))$ is maintained as long as $H(z)$ is not already defined to be something else when the simulator tries to set $H(z) = H'(N_P(x))$. However, if this happens with non-negligible probability, it must be because either we have found an f-collision with non-negligible probability, or because the adversary has queried $H(z)$ separately. In either case, we can use such an adversary to break the pseudorandomness of f. Because ultimately, the zero-knowledge property of our scheme may be defeated by defeating the pseudorandomness of f, we only get computational zero-knowledge.

We note that if we restrict the adversary a bit further, we can actually remove the random oracle assumption. Specifically, if we require that whenever the adversary requests an update about x, that either the adversary has already queried the database at x, or the adversary will *never* query the database about x, then we can prove zero-knowledge without the random oracle. We can also remove the random oracle if we use general NIZK proofs. We discuss this further in appendix E.

6 Efficiency

Our proposal for the mecahnics of a transparently updatable database embeds the idea that for each update (even of a single element) to the database, a public update string is published, and that for each update string that is published, each user updates each of their proofs. Given this syntax, our performance is optimal in terms of the number of updates: each update induces additional work for both the database owner and the user, but the amount of work per update is independent from the number of updates. However, the total amount of work a user must do to maintain a proof is linear in the number of updates. In appendix F we describe some minor efficiency improvements along these lines.

7 Conclusion and open problems

We have given a secure construction of a transparently updatable zero-knowledge database that is both efficient and practical in the random oracle model. For our construction to be secure, we must assume the existence of a VRF, and that mercurial commitments exist. The most practical construction that arises from this work is the extension of the original Micali-Rabin-Kilian construction, which requies the discrete logarithm assumption. These two assumptions can be combined by using the VRF of Dodis and Yampolskiy [3], which relies on a more restrictive assumption than the discrete logarithm assumption.

Some open problems that may be of interest would be to construct:

- Zero-knowledge transparently updatable databases with stronger security or more general assumptions
- More efficient and/or perfect zero-knowledge opaque updates.
- Zero-knowledge databases the can be efficiently updated both transparently and opaquely.

Acknowledgements

The author sincerely and deeply thanks Susan Hohenberger, Anna Lysyanskaya, Silvio Micali, and Adam Smith for their helpful comments. The author also especially wishes to thank the anonymous program committee members who have provided useful feedback.

References

1. Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Noninteractive zero-knowledge. *SIAM Journal on Computing*, 20(6):1084–1118, December 1991.
2. Melissa Chase, Alex Healy, Anna Lysyanskaya, Tal Malkin, and Leonid Reyzin. Mercurial commitments with applications to zero-knowledge sets. In *Advances in Cryptology—EUROCRYPT 2005*, Lecture Notes in Computer Science. Springer-Verlag, 22 – 26 May 2005.

3. Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *8th International Workshop on Theory and Practice in Public Key Cryptography*, volume 3386 of *lncs*, pages 416–432. Springer-Verlag, 2005.
4. Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(1):691–729, 1991.
5. J. Kilian. Efficiently committing to databases. TR 97-040, NEC Research Institute, 1997.
6. Anna Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In Moti Yung, editor, *Advances in Cryptology—CRYPTO 2002*, Lecture Notes in Computer Science. Springer-Verlag, 2002.
7. Ralph C. Merkle. A certified digital signature. In G. Brassard, editor, *Advances in Cryptology—CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1990, 20–24 August 1989.
8. 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, 11–13 June 1997. IEEE Computer Society.
9. Silvio Micali, Michael Rabin, and Joseph Kilian. Zero-knowledge sets. In *44th Annual Symposium on Foundations of Computer Science*, Cambridge, MA, October 2003. IEEE.
10. 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.
11. R. Ostrovsky, C. Rackoff, and A. Smith. Efficient proofs of consistency for generalized queries on a committed database. In *Proceedings of ICALP 2004*, 2004.
12. Torben Pryds Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In D. W. Davies, editor, *Advances in Cryptology—EUROCRYPT 91*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526. Springer-Verlag, 8–11 April 1991.

Appendix A: Formal definitions for opaque updates

These definitions are closely derived from [9]. Here, we formalize the definitions described in section 3.2.

Updatable database simulators

Let Sim be a probabilistic polyonomial-time oracle Turing machine. We say that Sim is an updatable database *simulator* (or UDB simulator) if it computes as follows, relative to an *external* database D and pseudonym function N:

1. In its first execution, Sim^N outputs three strings, σ, PK, and SK.
2. In a subsequent execution on input SK and a triple $(x, D(x), N(x))$, $\mathsf{Sim}^N(SK, x, D(x), N(x))$ outputs a string π_x.
3. In a subsequent execution on input SK and n, $\mathsf{Sim}^N(SK, n)$ computes PK', SK', U where SK' becomes the new secret key, and PK' and U are outputs. When this happens, D may change at up to one input, namely an x such that $N(x) = n$.

Transparently updatable zero-knowledge databases

Let ($\mathsf{Commit}, \mathsf{Prove}, \mathsf{DBUpdate}, \mathsf{Verify}, \mathsf{PUpdate}$) be a TUDB system where all the Turing machines in the quintuple run in probabilistic polynomial time. We say that (Commit, Prove, $\mathsf{DBUpdate}$, Verify, $\mathsf{PUpdate}$) is a *zero-knowledge transparently updatable database system* (or ZKTUDB system) if there exists a UDB simulator Sim and a constant c such that

1. *Completeness.* $\forall$ database $D, \exists \nu$ negligible such that $\forall k, \forall r, s, t$ such that $0 \leq s \leq r \leq k^c$,

```
Pr[ $\sigma \leftarrow \{0,1\}^{k^c}; (PK, SK) \leftarrow \mathsf{Commit}(D, 1^k, \sigma);$
    $x_1 \leftarrow \{0,1\}^{\leq t}; y_1 \leftarrow \{0,1\}^{\leq t}; \ldots ; x_r \leftarrow \{0,1\}^{\leq t}; y_r \leftarrow \{0,1\}^{\leq t}; x \leftarrow \{0,1\}^{\leq t}$
    $(PK', SK', U) \leftarrow \mathsf{DBUpdate}(D, 1^k, \sigma, PK, SK, (x_1, y_1)); PK \leftarrow PK'; SK \leftarrow SK';$
    $D(x_1) \leftarrow y_1; \ldots ;$
    $(PK', SK', U) \leftarrow \mathsf{DBUpdate}(D, 1^k, \sigma, PK, SK, (x_s, y_s)); PK \leftarrow PK'; SK \leftarrow SK';$
    $D(x_s) \leftarrow y_s; \pi_x \leftarrow \mathsf{Prove}(D, 1^k, \sigma, PK, SK, x);$
    $(PK', SK', U) \leftarrow \mathsf{DBUpdate}(D, 1^k, \sigma, PK, SK, (x_{s+1}, y_{s+1})); SK \leftarrow SK'; D(x_{s+1}) \leftarrow y_{s+1};$
    $\pi_x \leftarrow \mathsf{PUpdate}(1^k, \sigma, PK, PK', U, x, \pi_x); PK \leftarrow PK';$
    $\ldots ;$
    $(PK', SK', U) \leftarrow \mathsf{DBUpdate}(D, 1^k, \sigma, PK, SK, (x_r, y_r)); SK \leftarrow SK'; D(x_r) \leftarrow y_r;$
    $\pi_x \leftarrow \mathsf{PUpdate}(1^k, \sigma, PK, PK', U, x, \pi_x); PK \leftarrow PK';$
    $y \leftarrow \mathsf{Verify}(1^k, \sigma, PK, x, \pi_x) :$
    if $\exists l$ such that $s < l \leq r$ and $x_l = x$ then $\pi_x = \perp$, otherwise $y = D(x)] > 1 - \nu(k)$.
```

Here, s is the number of updates before the proof is given, and r is the number of updates total.

2. *Soundness.* $\forall x \in \{0,1\}^*$ and $\forall P'$ probabilistic polynomial time, $\exists \nu$ negligible such that $\forall k$,

```
Pr[ $\sigma \leftarrow \{0,1\}^{k^c}; (PK, x, \pi_1, \pi_2) \leftarrow P'(1^k, \sigma);$
    $y_1 \leftarrow \mathsf{Verify}(1^k, \sigma, PK, x, \pi_1); y_2 \leftarrow \mathsf{Verify}(1^k, \sigma, PK, x, \pi_2) :$
    $\mathit{reject} \notin \{y_1, y_2\} \wedge y_1 \neq y_2] \leq \nu(k),$
```

3. *Zero-knowledge with respect to N.* $\forall A$ acceptable adversaries, $\forall k$, $\text{View}(k) \approx \text{View}'(k)$[4] where

```
View$(k)$ =
    View$_A\{\sigma \leftarrow \{0,1\}^{k^c}; (D, s_0) \leftarrow A(1^k, \sigma);$
    $(PK, SK) \leftarrow \mathsf{Commit}(D, 1^k, \sigma); z_0 \leftarrow PK;$
    $(w_1, s_1) \leftarrow A(s_0, z_0);$
    If    $w_1 = \mathsf{Update}(x_1, y_1),$
          $(PK'_1, SK'_1, U_1) \leftarrow \mathsf{DBUpdate}(D, 1^k, \sigma, PK, SK, x_1, y_1); SK \leftarrow SK'_1; PK \leftarrow PK'_1;$
          $D(x_1) \leftarrow y_1; z_1 \leftarrow (PK'_1, U_1);$
    Else  if $w_1 = \mathsf{Query}(x_1)$, $\pi_1 \leftarrow \mathsf{Prove}(D, 1^k, \sigma, PK, SK, x_1); z_1 \leftarrow \pi_1;$
    $(w_2, s_2) \leftarrow A(s_1, z_1);$
    $\ldots\}$
```

and

```
View$'(k)$ =
    View$_A\{(\sigma, PK, SK) \leftarrow \mathsf{Sim}^N(1^k); (D, s_0) \leftarrow A(1^k, \sigma);$
    $z_0 \leftarrow PK;$
    $(w_1, s_1) \leftarrow A(s_0, z_0);$
    If    $w_1 = \mathsf{Update}(x_1, y_1),$
          $(PK'_1, SK'_1, U_1) \leftarrow \mathsf{Sim}^N(SK, N(x_1)); SK \leftarrow SK'_1; PK \leftarrow PK'_1;$
          $D(x_1) \leftarrow y_1; z_1 \leftarrow (PK'_1, U_1);$
    Else  if $w_1 = \mathsf{Query}(x_1)$, $\pi_1 \leftarrow \mathsf{Sim}^N(SK, x_1, D(x_1), N(x_1)); z_1 \leftarrow \pi_1;$
    $(w_2, s_2) \leftarrow A(s_1, z_1);$
    $\ldots\}$
```

[4] As usual, $\approx$ may refer to computational indistinguishability (in which case the system is said to be "computationally zero-knowledge"), statistical closeness ("statistical zero-knowledge"), or equality ("perfect zero-knowledge"). For computational indistinguishability, A must be a polynomial-time adversary. For statistical or perfect indistinguishability, we do not limit A's power.

Appendix B: Opaquely updatable zero-knowledge databases

We define opaquely updatable zero-knowledge databases, and present a solution following ideas from Rackoff, Ostrovsky, and Smith [11] that is inefficient and relies on general non-interactive zero-knowledge proofs. We do not present any practical, efficient method better than simply committing the updated database from scratch; indeed, we view this as an important open problem.

An opaquely updatable database system (or OUDB system) is a quadruple of algorithms (Commit, Prove, DBUpdate, Verify) which satisfy the properties properties of a TUDB system, except that DBUpdate outputs only PK', SK'.

Zero-knowledge opaquely updatable databases are defined similarly to transparently updatable ones. Let (Commit, Prove, DBUpdate, Verify) be a UDB system where all the Turing machines in the quadruple run in probabilistic polynomial time. We say that (Commit, Prove, DBUpdate, Verify) is a *zero-knowledge opaquely updatable database system* (or ZKOUDB system) if there is a UDB simulator Sim and a constant c such that the following four properties are satisfied:

1. *Perfect completeness.* $\forall$ database $D, \forall r, \forall$ sequences of updates $(x_1, y_1), \ldots, (x_r, y_r)$, and $\forall x \in [D] \cup \{x_1, \ldots, x_r\}$,

```
$Pr[$  $\sigma \leftarrow \{0,1\}^{k^c}; (PK, SK) \leftarrow \mathsf{Commit}(D, 1^k, \sigma);$
       $(PK', SK', U) \leftarrow \mathsf{DBUpdate}(D, 1^k, \sigma, PK, SK, (x_1, y_1)); PK \leftarrow PK'; SK \leftarrow SK';$
       $D(x_1) \leftarrow y_1; \ldots;$
       $(PK', SK', U) \leftarrow \mathsf{DBUpdate}(D, 1^k, \sigma, PK, SK, (x_r, y_r)); PK \leftarrow PK'; SK \leftarrow SK';$
       $D(x_r) \leftarrow y_r; \pi_x \leftarrow \mathsf{Prove}(D, 1^k, \sigma, PK, SK);$
       $y \leftarrow \mathsf{Verify}(1^k, \sigma, PK, x, \pi_x):$
       $y = D(x)] = 1.$
```

2. *Soundness.* (Commit, Prove, DBUpdate, Verify) satisfies the soundness property of a ZKTUDB.
3. *Zero-knowledge.* (Commit, Prove, Verify) satisfies the zero-knowledge properties of a ZK database. We actually want zero-knowledge to hold for an adversary that can adaptively ask for queries and updates, but we capture the difference in our definition of update secrecy.
4. *Update secrecy.* For all appropriate A, $\mathrm{View}(k) \approx \mathrm{View}'(k)$ where:

```
$\mathrm{View}(k) =$
    $\mathrm{View}_A\{\sigma \leftarrow \{0,1\}^{k^c}; (D, s_0) \leftarrow A(\sigma); (PK, SK) \leftarrow \mathsf{Commit}(D, 1^k, \sigma);$
    $z_0 \leftarrow PK; (w_1, s_1) \leftarrow A(s_0, z_0);$
    If    $w_1 = \mathsf{Update}(x_1, y_1),$
          $(PK', SK') \leftarrow \mathsf{DBUpdate}(D, 1^k, \sigma, PK, SK, x_1, y_1); SK \leftarrow SK'; PK \leftarrow PK';$
          $D(x_1) \leftarrow y_1; z_1 \leftarrow PK';$
    Else  if $w_1 = x_1, \pi_1 \leftarrow \mathsf{Prove}(D, 1^k, \sigma, PK, SK, x_1); z_1 \leftarrow \pi_1;$
    $(w_2, s_2) \leftarrow A(s_1, z_1);$
    $\ldots\}$
```

and

```
$\mathrm{View}'(k) =$
    $\mathrm{View}_A\{\sigma \leftarrow \{0,1\}^{k^c}; (D, s_0) \leftarrow A(\sigma); (PK, SK) \leftarrow \mathsf{Commit}(D, 1^k, \sigma);$
    $z_0 \leftarrow PK; (w_1, s_1) \leftarrow A(s_0, z_0);$
    If    $w_1 = \mathsf{Update}(x_1, y_1),$
          $D(x_1) \leftarrow y_1; (PK', SK') \leftarrow \mathsf{Commit}(D, 1^k, \sigma); SK \leftarrow SK'; PK \leftarrow PK';$
          $z_1 \leftarrow PK';$
    Else  if $w_1 = x_1, \pi_1 \leftarrow \mathsf{Prove}(D, 1^k, \sigma, PK, SK, x_1); z_1 \leftarrow \pi_1;$
    $(w_2, s_2) \leftarrow A(s_1, z_1);$
    $\ldots\}$
```

Again, appropriate adversaries are polynomial-time adversaries for computational indistinguishability, and unbounded adversaries otherwise.

Opaquely updatable construction

To create an opaquely updatable zero-knowledge database, following Rackoff, Ostrovsky, and Smith [11], we modify the CHLMR construction as follows. Instead of sending a proof π_x to the verifier, we give $D(x)$ and a non-interactive zero-knowledge proof of knowledge relative to σ of knowledge of π_x such that π_x is a valid proof. To update, we just update the values where required, but do not publish any of the updated values. We clearly have zero-knowledge: in order to simulate, we just randomly create c_ϵ initially and each time we are asked to update we create a new random commitment, and any time we are asked to give a proof, we provide a faked non-interactive zero-knowledge proof. Furthermore, c_ϵ form a random commitment whether or not they were generated from $\mathsf{DBUpdate}$, so we have update secrecy as well, and soundness and completeness follow from these same properties of CHLMR databases.

However, such non-interactive zero-knowledge proof systems are also only computational zero-knowledge. In addition, much effort was taken by Micali, Rabin, and Kilian to avoid both computational zero-knowledge and the need for general non-interactive zero-knowledge proofs. The large amount of inefficiency added to the system may even overbalance the objection to the solution of recommitting the database from scratch. We consider it a significant open problem to construct an efficient and practical opaquely updatable zero-knowledge databases.

Appendix C: Detailed proof of security

Proof of Theorem 1. To prove theorem 1, we must make a minor additional asusmption, and prove several things.

First of all, note that when an update occurs, the only difference between the secret information in our construction and the secret information in a CHLMR database is that in our construction, it may be that for some internal nodes ω which have no descendents in the tree, c_ω is a hard commitment rather than a soft one. However, that is unimportant as proofs involving such an ω as a node on the path will always be of nonmembership, and so only soft decommitments will be revealed.

To prove completeness, note that when the database is updated, part of an old proof about a different element will include path elements that have changed. However, such path elements are always published as part of the update information, so they can simply be replaced. Thus, the updated proof is valid. The only possible snag we can run into is that if $N(x) = N(x')$ then an update about x' would prevent a proof about x from being properly updated. Barring this, as long as no updates have occurred about the element x since π_x was issued, π_x may be updated successfully. To deal with this issue we must assume that $N(x)$ is such that collisions are unlikely to occur. This is certainly the case for all N we use.

To prove soundness, note that if a cheating prover were to be able to produce relative to a random σ a public key PK and two valid proofs π_1 and π_2 proving different results about $D(x)$ for some particular x, then this same prover would violate the soundness of CHLMR databases.

To prove zero-knowledge, we describe the simulator. The simulator must do five things: it must create the string σ, it must provide the initial commitment, and it must provide proofs and updates when requested.

- To produce σ, PK, or to produce a proof that $D(x) = y$, the simulator runs just as the CHLMR simulator does, except using $N(x)$ instead of $H(x)$ to determine the location of key pairs.

– To produce an update on a pseudonym n, computes $v_n = H(\epsilon)$ and computes a new commitment c_n.
  The simulator then updates all the commitments along the path from ϵ to n from soft to hard commitments, with the proper values to maintain the Merkle tree structure. The simulator incorporates any new decommitment information into SK'.

Now, to prove that the view provided to the adversary in the real model is identical to that in the ideal model, we describe the view of the adversary. In the real world, the adversary sees the random string σ, and then after specifying D, the commitment c_ϵ. Then, for each proof query, the adversary sees a proof about x which consists of an appropriate value $v_{N(x)}$ and random commitments c_ω to appropriate values, forming a hash authentication path to the root. For each update query, the adversary sees a pseudonym $N(x)$, a new commitment at $N(x)$, and for each proper prefix ω of $N(x)$, a random updated commitment c_ω. Furthermore, in the case of the discrete logarithm-based scheme, the adversary also sees e for each such ω, which shows that all these commitments are hard commitments.

In the ideal world, the adversary sees the simulated σ, followed by a distribution exactly the same as in the real world, except that $c_{N(x)}$ is a commitment to $H(\epsilon)$ rather than $H(y)$. However, these commitments are hiding so this is indistinguishable from the view of the adversary in the real world. In fact, in the case of the discrete logarithm-based scheme, the views are identical, since the only difference is in what c_ω commits to where ω is a leaf, but c_ω is a perfectly hiding commitment. Furthermore, the distribution of real σ values is identical to the distribution of simulated σ values by the perfect zero-knowledge property of the Micali-Rabin-Kilian simulator.

Proof of Theorem 2. To prove that the construction using $N(x) = H(f_{SK_f}(H(x)))$ is strongly secure, we must prove that it satisfies completeness, soundness, and computational zero-knowledge with respect to N_P in the random oracle model.

Completeness is already established by the completeness proof of Theorem 1; the only difference here is that a VRF proof must be verified (note that indeed, $N(x)$ here is unlikely to have collisions). However, $N(x)$ does not change when x is updated, so this part of the proof may remain the same. To prove soundness, we need only note that the pseudonym $N(x)$ that will be verified is unique from the soundness property of the VRF.

Zero-knowledge is more of a challenge. We give a simulator with respect to N_P that gives us computational zero-knowledge. First, the simulator makes σ and the database commitment c_ϵ just as the CHLMR simulator does. The simulator then runs GenVRF to generate (PK_f, SK_f), and publishes (PK_f, c_ϵ) as the database commitment. We must be careful to note here that N_P is not available as an oracle to the simulator, but $N_P(x)$ is given without x for any update query, and $N_P(x)$ is given with x for any database query. $H^*(x)$ here refers to the value used in the construction; the actual pseudonym we are considering is $N_P(x)$.

The simulator maintains two random functions: H and H', with the idea that $H'(N_P(x)) = H(f(H(x)))$. Whenever we say the simulator must "compute" (say) $H(x)$, the simulator looks to see if it has ever set $H(x)$ to any particular value. If so, it outputs that value. If not, it generates a random value of the correct length, and notes the correspondence with x. There can never be a problem with the simulator computing a value $H(x)$ or $H'(x)$.

When the simulator receives an update query, it computes $H'(N_P(x))$, and uses this value as $H^*(x)$.

When the simulator receives a database query on $x, y, N_P(x)$, the simulator computes $H(x)$, and then computes $z = f_{SK_f}(H(x))$, and then attempts to set $H(z) = H'(N_P(x))$. That is, if H is not defined at z, $H(z)$ is set to be the value computed from $H'(N_P(x))$. Otherwise, if H' is not yet defined at $N_P(x)$, $H'(N_P(x))$ is set to be the value computed from $H(z)$. If $H(z)$ and $H'(N_P(x))$ are already defined and equal to each other, the simulator sets nothing. However, if $H(z)$ and $H'(N_P(x))$ are already defined and unequal, the simulator aborts. If the simulator does not abort, it fakes a proof that the value stored at $H^*(x) = H(z) = H'(N_P(x))$ is y, just as the MRK simulator does, and provides the value z along with $\mathsf{ProveVRF}(SK_f, H(x))$ that $z = f_{SK_f}(H(x))$.

We must prove two things. First, in cases in which the simulator doesn't abort, the adversary cannot distinguish between the simulator and the real prover. We can assume without loss of generality that the adversary will always make a database query about every value x that he asks us to update before he halts (doing so will only increase the probability that the simulator aborts). If the simulator hasn't aborted by the time the adversary halts, we can reconcile H' into H, since all values $H'(N_P(x))$ will have been set equal to $H(z)$ for some z (because the adversary has queried all points for which we have a pseudonym). Thus, this simulator is doing exactly what the simulator in our previous proof does: it accurately computes $H^*(x)$ in every case and simulates proofs and updates according to this. Thus, the view produced by such a simulator is identical to the view produced by the real prover.

Second, if the simulator aborts with non-negligible probability, we can break the security of the VRF as follows. On input a VRF public key PK_f, we act as the simulator with the given adversary in this experiment, except we give PK_f as the VRF public key instead of generating it ourselves, and we implement the simulator. Note that we only ever need to query f_{SK_f} right before we ask for a proof about it. After some number of queries, the probability that the next value we ask for will cause an abort is non-negligible, so instead of asking for $f_{SK_f}(H(x))$ that time, we pick a random z such that $H(z)$ is defined, and guess that $f_{SK_f}(H(x)) = z$. We try this with the given oracle (which is either the VRF or a random oracle), and if we are correct, we say that the oracle is a VRF, otherwise, we guess at random. If the oracle is the VRF, and an abort would have been caused, then we have a $1/p(k)$ probability of guessing the right z, where $p(k)$ is the polynomial determining how many inputs have been queried from H. Thus, if the probability of an abort at the given step is $1/q(k)$, then the probability that we break the VRF is $(1/2)(1/(p(k)q(k))) + (1/4)(1 - (1/p(k)q(k)) + 1/2(1 - \nu(k))$ which is at least $1/2 + 1/(4p(k)q(k)) - \nu(k)$ for some negligible ν.

If the probability of an abort is non-negligible, it is non-negligible at some particular query. Thus, there is some reduction that breaks the security of the VRF.

Appendix D: Pseudonym collisions

In the work of Micali, Rabin, and Kilian, the Pedersen hash function is used to assign pseudonyms to database elements. One attractive property of using the Pedersen hash function is that if a pseudonym collision occurs, the database owner learns the discrete logarithm of h to the base g, and then may continue proving what would otherwise be impossible: for instance, that $D(x) = y$ and $D(x') = y' \neq y$ when $H(x) = H(x')$. This allows the database to have size that is unrelated to any security parameters.

If, as we propose, we replace $H(x)$ by $N(x) = H(f(H(x)))$ for some verifiable random function f, we lose this property: N could encounter collisions either from H-collisions or from f-collisions. The former would be fine while the latter would be a problem. In practice, it is acceptable to limit honest users to polynomial-size databases, in which case collisions are negligibly likely. However, we can preserve this property through some extra effort, which has a minimal impact on efficiency.

Due to space constraints, we do not give the full details of this construction. The basic idea is that we use a public-key cryptosystem, and include two public keys: one from the cryptosystem and one from a verifiable random function. Then, instead of computing $a = f(x)$, we compute $E_{PK_e}(x; a)$, that is, we encrypt a under the encryption public key, using a as the randomness. A proof consists of a and the proof that $a = f(x)$ was properly generated by the VRF. This may not be pseudorandom, but in our construction it is sufficient to have unpredictability of the full answer, and this construction does achieve that.

When we use this injective verifiable unpredictable function, we get a pseudonym function that only has collisions when they are collisions of the hash function. Thus, any pseudonym collisions can be worked around.

It is worth noting, however, that the properties of the Pedersen hash function are nice, yet we are assuming in our (main) security proof that the hash function we use is a random oracle. In our opinion, the nice properties of the Pedersen hash are worth having, and this will probably not cause a significant security problem. However, we are unwilling to assume that the Pedersen hash function is a random oracle.

Appendix E: Removing the random oracle assumption

If we are willing to assume certain conditions on the adversary, we can give a construction that is secure without the random oracle assumption. The conditions are as follows:

- If the adversary first inquires about x in a database query, it may in the future ask for more database queries about x as well as updates about x.
- If the adversary first inquires about x in an update query, it may only ask for more updates about x in the future.

It may seem at first glance that we can assume this without loss of generality: any successful adversary could always make more queries, and thus, make a database query immediately before any update query so as to always comply with the conditions. The problem with this is that since the simulation is actually a game of *three* parties: the adversary, the simulator, and the functionality that provides pseudonyms, the simulator actually must interact with the functionality more than normal to handle adversaries that don't hold to these conditions, which means that the simulator must learn more, which is not acceptable. It is important that in our simulation, the *simulator* not be able to get any more information out of the pseudonym-providing functionality than the adversary would.

Given that all adversaries meet these restrictions, we remove the random oracle assumption as follows: Again, we use the pseudonym function $H^*(x) = H(f(H(x)))$. To simulate, this time without being able to control H as a random oracle, we do as follows: if x is a value that is first mentioned in a database query, we actually compute $H(f(H(x)))$. If x is a value that is first mentioned in an update query, we know that the adversary will never make a database query about this particular x, so we compute $H^*(x) = H(R(N(x)))$ where R is a random function that we maintain, and where $N(x)$ is the pseudonym of x. If the adversary can distinguish between this simulator and a real adversary then either the adversary managed to find an H-collision, (for example, if $H(x) = H(x')$, so the adversary could detect this simulator by making a database query on x and then an update query on x', which should give the same pseudonym), or all inputs that should be given to f are distinct between the two types, in which case, the probability of distinguishing is exactly the probability of distinguishing the VRF from a random function.

We should note that although the restriction on the adversary is nontrivial, such adversaries still represent a significant class of adversaries. What's more, since we use the same construction here as in Theorem 2, we have actually proved security of that construction in two different ways: one, with the random oracle model, the other, with these restrictions on the adversary.

However, we can remove the random oracle model without weakening our assumptions if we give up efficiency. Instead of using a VRF, we can simply commit to a key K for a PRP using a commitment that becomes part of the database commitment, and then use $f_K(H(x))$ as $N(x)$, and prove correctness of this using a general NIZK proof. The advantage of this is that the simulator can fake NIZK proofs of false theorems, so the simulator can simply pretend that $F(N_P(x)) = f_K(H(x))$ where F is a random function, and fake proofs when necessary.

Appendix F: Efficiency improvements

Multi-pair updates

Suppose the database owner wants to update the database at n pairs simultaneously. A fairly obvious method presents itself: make an update for each pair individually, and publish all the update information together. This saves space, since some updated nodes will overlap. Asymptotically, the number of nodes updated becomes $O(n(k - \log n)k)$, which represents some savings over the one-at-a-time approach, which is asymptotically $O(nk^2)$.

Multi-proof updates

Suppose a proof owner has n proofs and an update is issued. If two proofs overlap (that is, $N(x)$ and $N(x')$ share a common prefix), the change in the updated proofs for x and x' can be computed more quickly by computing the change in the common portion of those two proofs together, then computing the change in the remaining portion of each. More generally, if a user holds n proofs, updating each separately would take time $O(nk^2)$, but by combining the work, this is reduced to time $O(n(k - \log n)k)$.

The analysis for both of these methods is based on the observation that an average case instance of n random strings will have the first $\log n$ bits in common with a newly chosen random string. Thus, if each string translates to a path of length k, the expected sum of the length of all paths is $k + k - \log 0 + \ldots + k - \log(n-1) < nk - (n/2)\log(n/2) = O(n(k - \log n))$. The additional factor of k accounts for the length of the data per node.