Zero-Knowledge Sets*

Silvio Micali† Michael Rabin‡ Joe Kilian§

Abstract

We show how a polynomial-time prover can commit to an arbitrary finite set S of strings so that, later on, he can, for any string x, reveal with a proof whether $x \in S$ or $x \notin S$, without revealing any knowledge beyond the verity of these membership assertions.

Our method is non interactive. *Given a public random string, the prover commits to a set by simply posting a short and easily computable message. After that, each time it wants to prove whether a given element is in the set, it simply posts another short and easily computable proof, whose correctness can be verified by any one against the public random string.*

Our scheme is very *efficient; no reasonable prior way to achieve our desiderata existed. Our new primitive immediately extends to providing zero-knowledge "databases."*

1. Introduction

Sets are perhaps the most fundamental notion in mathematics, and ever since [7], zero-knowledge proofs have been extensively studied. Yet, to date, no general and satisfactory zero-knowledge representation of sets exists. We thus wish to provide one by putting forward *zero-knowledge sets*.

Intuitive Goals We wish to enable one to (1) arbitrarily choose a finite set, S, of finite strings, (2) compute a commitment, C_S, to S, and (3) given an arbitrary sequence of strings, $x_1, x_2, \ldots$, prove non-interactively (relatively to C_S) whether x_i belongs to S or not without revealing more knowledge about S than just truthfully asserting "$x_i \in S$" or "$x_i \notin S$"", whichever the case may be).

Following the non-interactive zero-knowledge paradigm of [2] and [4], such non-interactive proofs of "membership" and "non-membership" in S are relative to a publicly available random string.

Notice that, in particular, our zero-knowledge desiderata imply that S's cardinality should remain hidden to the maximum extent possible. For instance, after proving that 10 different strings belong to S, then the only information deducible about S's cardinality should be that it is at least 10; not even a ridiculously large upper bound on S's cardinality should be deducible. Indeed, hiding the size of S poses the most difficult technical obstacles to our construction.

Technical Difficulties Constructing zero-knowledge sets is not trivial. The main difficulty lies in proving that an arbitrary string x does not belong to our arbitrarily constructed set S as identified by its commitment value C_S. For instance, quite straightforwardly, one could prove in zero knowledge the following two sub-statements: (1) "there exists a (secret) value n and (secret) strings $x_1, \ldots, x_n$ such that committing to them yields the value C_S, and (2) "$x \neq x_1, \ldots, x \neq x_n$ ". Unfortunately, such an approach is bound to betray an upper bound on S's cardinality. In fact, the number of bits transmitted in a zero-knowledge proof, whether interactive [7] or non-interactive [4], grows with the size of the corresponding statement: in our case, with "$x_1, \ldots, x_n$", and thus with S's cardinality. One may attempt to get around this size problem using PCP-based techniques (c.f., [12, 16, 13]); however, such techniques do not appear to lead to practical solutions.

Main Result Via a different approach, we prove the following

Theorem: Zero-knowledge sets exist if the discrete logarithm problem is hard.

Perhaps surprisingly, in light of the above very stringent security requirements, our construction of zero-knowledge sets is not only efficient, but actually *very practical:* A set S is committed to by performing at most $2k$ collision-free hashings and $2k$ modular exponentiations for each of its elements, where k is the security parameter. Each proof of membership in S is computed by a mere table look up, and each proof of non-membership is S is computed by at most $2k$ collision-free hashings and $2k$ modular exponentiations. The same amount of work is required by a verifier for the verification of a proof π_x of membership or non-

† Laboratory for Computer Science, MIT, Cambridge, MA 02139.

‡ Department of Applied Science, Harvard University, Cambridge, MA 02138. Research supported at Harvard Univ. by NSF Grant ITR 0205423

§NEC Laboratories, America. joe@nec-labs.com.

membership always requires $2k$ collision-free hashings and $2k$ modular exponentiations.

From Zero-Knowledge Sets to Zero-Knowledge (Elementary) Databases By an elementary database (EDB for short) we mean a partial function D mapping a (sub)set of *keys* into *values*. While databases have many complex functionalities, our EDBs have just an elementary one. Namely, if D is an EDB, then the only envisaged operation is querying D with a key x and obtain in response either the special symbol $\perp$ (if no value is associated to x) or the only value $D(x)$ associated with x.

Our construction of zero-knowledge sets immediately yields the construction of a *zero-knowledge EDB*. The latter consist of a way to (1) commit to an elementary database D and then (2) $\forall x \in \{0,1\}^*$, prove whether x is indeed an existing key in D (and if so what is the value $D(x)$ associated with it), without revealing any undue knowledge. That is, without providing more knowledge than that obtainable from a trusted party who, knowing D, on a given sequence of input strings $x, y, \ldots$, truthfully states the status of D relative to these strings: e.g., "x is not a key in D," "y is a key of D and its corresponding value is $D(y) = v$," etc.

Using a small amount of interaction, Ostrovsky, Rackoff and Smith [19] have very recently constructed zero-knowledge databases with more powerful functionalities. For instance, they show how to handle *range queries* (roughly, after committing to D they prove that, for an input interval $[a, b]$, either D does not contain any keys in it, or which keys are contained in it, and which are their corresponding values).

Road Map In Section 2, we give our basic notation. In Section 3, we define zero-knowledge elementary databases. In Section 4, we give some basic preliminaries. In Section 5, we give an informal exposition of our protocols. In Section 6, we make some final comments. In Section A of the appendix, we give a more "pseudo-code" presentation of our protocols.

A more detailed version of this paper can be found in the Cryptology ePrint Archive (`http://eprint.iacr.org/`).

2. Notation

We shall follow, verbatim, [4] and [9]. Namely,

Strings. We denote the empty word by **e**, and the concatenation of two strings x and y by $x|y$ (or more simply by xy). If α is a binary string, then $|\alpha|$ denotes α's length; $\alpha_1 \cdots \alpha_i$ denotes α's i-bit prefix.

Integer representation. We denote by N the set of natural numbers: 0, 1, 2, ..., Unless otherwise specified, a natural number is presented in its binary expansion (with no *leading* 0s) whenever given as an input to an algorithm. If $n \in N$, we denote by 1^n the unary expansion of n (i.e., the concatenation of n 1's).

Negligible functions. By $\epsilon(\cdot)$ we represent any function than vanishes faster than the inverse of any fixed positive polynomial. That is, for any positive polynomial P,

$$\lim_{k\to\infty} P(k)\epsilon(k) = 0.$$

Probabilistic algorithms. If A is a probabilistic algorithm, then for any input x, the notation "$A(x)$" refers to the probability space that assigns to the string σ the probability that A, on input x, outputs σ. An *efficient* algorithm is a probabilistic algorithm running in expected polynomial time.

Probabilistic assignments. If S is a probability space, then "$x \stackrel{R}{\leftarrow} S$" denotes the act of choosing an element x at random according to S. If F is a finite set, then the notation "$x \stackrel{R}{\leftarrow} F$" denotes the act of choosing x uniformly from F.

Probabilistic experiments. If p is a predicate, and $S_1, S_2, \ldots$ are probability spaces, then the notation $\Pr[x_1 \stackrel{R}{\leftarrow} S_1;\ x_2 \stackrel{R}{\leftarrow} S_2;\ \ldots\ :\ p(x_1, x_2, \ldots)]$ denotes the probability that $p(x_1, x_2, \ldots)$ will be true after the ordered execution of the probabilistic assignments $x_1 \stackrel{R}{\leftarrow} S_1;\ x_2 \stackrel{R}{\leftarrow} S_1;\ \ldots$

Probability spaces. If $S, T, \ldots$ are probability spaces,

the notation $\{x \stackrel{R}{\leftarrow} S; y \stackrel{R}{\leftarrow} T; \cdots : (x, y, \cdots)\}$ denotes the new probability space over $\{(x, y, \cdots)\}$ generated by the ordered execution of the probabilistic assignments $x \stackrel{R}{\leftarrow} S$, $y \stackrel{R}{\leftarrow} T, \cdots$.

Elementary Databases. An EDB D is a subset of $\{0,1\}^* \times \{0,1\}^*$ such that $(x, v_1) \in D$ and $(x, v_2) \in D$ implies $v_1 = v_2$.

If D is an EDB, by the expression "$[D]$" we denote the *support* of D, that is, the set of finite binary strings x for which, for some $v \in \{0,1\}^*$, $(x, v) \in D$.

To indicate that x is not in the support of and EDB D, we write "$D(x) = \perp$". By writing "$v = D(x)$" or "D(x)=v" (where y is any symbol other than $\perp$) we informally mean that $x \in [D]$ and v is a string, indeed the unique string such that $(x, y) \in D$.

3. The Notion of Zero-Knowledge EDB

Since the notion of a zero-knowledge set is essentially a special case of that of a zero-knowledge EDB, we only define the latter. We start with the intuitive version.

3.1 The Informal Notion

Mechanics Though not employing non-interactive zero-knowledge proofs directly, zero-knowledge EDBs have a similar mechanics. Namely, they rely on a public random string σ, the *reference string*. This string is polynomially long in k, the security parameter controlling the probability of error or successful cheating.

In the initial committing phase, on input an EDB D (i.e., a binary string encoding the relevant subset of $\{0,1\}^* \times \{0,1\}^*$) and σ, a prover P easily computes a pair of matching keys, PK and SK. Key PK constitutes P's commitment to D, and is thus made public; key SK enables P to prove the value of $D(x)$ for any $x \in$ KEYS, and is thus kept secret. (In addition to some short cryptographic information, SK may include a description of D.)

In the subsequent proving phase, given any string x, P, using SK, quickly produces on his own (i.e., without any interaction) a proof π_x of either $D(x) = \perp$ or $D(x) = v$, whichever is the case. Any verifier can check the correctness of π_x by running an efficient algorithm on inputs x, the alleged proof π_x, "D's description" PK, and the reference string.

Security Zero-knowledge EDBs enjoy completeness, soundness and zero-knowledge.

Completeness simply guarantees that, with above mechanics, one can commit to any EDB D, and then, for any key x, prove the correct value of $D(x)$.

Soundness guarantees that the prover cannot lie about the value of $D(x)$. Namely, PK commits the prover to a partial function $D : \{0,1\}^* \to \{0,1\}^*$ in the sense that, in polynomial time, no one can find (1) a string x together with (2) a proof, relative to PK, of $D(x) = \perp$ and $D(x) = v \in \{0,1\}^*$, or (2′) a proof, relative to PK, of $D(x) = v_1$ and $D(x) = v_2$ for $v_1 \neq v_2$.
[1]

Zero-knowledge guarantees that the knowledge obtainable by seeing a commitment to a EDB D and then a sequence of proofs for the value of D at strings x, y, ..., coincides with that obtainable without seeing any thing about D at all, except for asking a trusted party about strings x, y, etc., and receiving in response his truthful but unproved assertions: e.g., "$D(x) = \perp$, $D(y) = v \in \{0,1\}^*$, ..."

Technically, the latter condition is expressed by saying that a zero-knowledge EDB D has a polynomial-time simulator that, without knowing anything about the set, provides to any verifier exactly the same view that he might receive from the true prover (who knows D). Namely, the verifiers' views produced by the following two games are *indistinguishably distributed*:

Game A. First, a random reference string σ, whose length is a fixed polynomial in the security parameter k, is made public. Then, an EDB D is chosen by the adversary and handed to the honest prover P. Later, based on D and the reference string σ, P computes a commitment PK to D along with a proving key SK. Finally, the adversary chooses a sequence of strings x_1, x_2, ..., for which P produces proofs for the correct value of D, $\pi_{x_1}, \pi_{x_2}, \ldots$

(The latter sub-process is *adaptive*. Namely, after seeing PK, the adversary chooses x_1 and the prover computes π_{x_1}; after seeing π_{x_1} the adversary chooses x_2 and the prover computes π_{x_2}; and so on.)

The verifier's view then consists of the strings σ, PK, $x_1, \pi_{x_1}, x_2, \pi_{x_2}, \ldots$

Game B. The efficient simulator SIM, on input the security parameter k, computes a string σ' of the proper length, a public key PK' and secret key SK'. After seeing PK', the adversary chooses x_1, the simulator is told (without proof!) the correct value of $D(x)$ for the same EDB D of Game 1, and computes π'_{x_1}; after seeing π'_{x_1} the adversary chooses x_2, the simulator is told (without proof!) the correct value of $D(x)$, and computes π'_{x_2}; and so on.

The verifier's view then consists of the strings σ', PK', $x_1, \pi'_{x_1}, x_2, \pi'_{x_2}, \ldots$

By $|D|$ we denote the sum of the lengths of the keys in $[D]$ and their corresponding values.

3.2 The More Formal Notion

Elementary Database Systems We say that a triple of Turing machines, (P_1, P_2, V), constitute a *EDB system* if neither machine retains 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 first algorithm to compute is P_1. On input $(D, 1^k, \sigma)$, P_1 produces two outputs: (1) a string PK, called D 's *public key* (or *commitment*), and (2) a string SK, called D 's *secret key*, and halts forever.

  (Note: SK may always include 1^k, σ, and D.)

- The second algorithm to compute is P_2. On input $(D, 1^k, \sigma, PK, SK)$, and an additional input $x \in \{0,1\}^*$, P_2 outputs a string π_x, called D's *proof* about x.

[1] Note that the computational nature of soundness is necessary in our setting. Namely, our very stringent zero-knowledge requirements imply that, for each choice of the security parameter k, *a commitment to D should always have the same size (depending only on k), no matter how many strings may be in the $[D]$, and no matter how long these strings may be.* In turn, this implies that a commitment to D must be only *computationally binding.* Indeed, the number of bits necessary to pin-down a set S exactly cannot be shorter than S 's Kolmogorov complexity, and thus any perfectly binding commitment to a set must grow with the set.

- The third algorithm to compute is algorithm V. On input $(1^k, \sigma, PK)$ and an additional $x \in \{0,1\}^*$ together with its proof π_x, V outputs either a string $y \in \{0,1\}^*$ (meaning that it believes $y = D(x)$), *out* (meaning that it believes that x is outside D's support), or $\perp$ (meaning that it detected cheating).

If (P_1, P_2, V) is an EDB system, we refer to P_1 as the database *committer*, to P_2 as the database *prover*, and to V as the database *verifier*.

Elementary Database Simulators Let SIM be a probabilistic polynomial-time Turing machine capable of making oracle calls. We say that SIM is an EDB *simulator* if, given oracle access to a database D, it computes as follows:

- In its first execution, $SIM^D(1^k)$ makes no oracle calls and outputs three strings, σ', PK', and SK'

  (Respectively, a "fake" reference string, D's "fake" public key, and D's "fake" secret key)

- In each subsequent execution, on input SK' and a string $x \in \{0,1\}^*$, $SIM^D(SK', x)$ calls its oracle only about string x, receives $D(x)$ in response, and outputs a string π_x.

Zero-Knowledge Elementary Databases Let (P_1, P_2, V) be an EDB system whose Turing machines run in probabilistic polynomial time. We say that (P_1, P_2, V) is a *zero-knowledge EDB* (ZK EDB, for short) if there exists a positive constant c such that:

1. *Perfect Completeness.* $\forall$ database D and $\forall x \in [D]$,

$$Pr\left\{\begin{array}{l}\sigma \stackrel{R}{\leftarrow} \{0,1\}^{k^c}; (PK, SK) \stackrel{R}{\leftarrow} P_1(1^k, \sigma, S);\\ \pi_x \stackrel{R}{\leftarrow} P_2(x, SK):\\ V(1^k, \sigma, PK, x, \pi_x) = D(x)\end{array}\right\} = 1.$$

2. *Soundness.* $\forall x \in \{0,1\}^*$ and $\forall$ efficient algorithms P', $\varepsilon(k)$ is negligible, where $\varepsilon(k) =$

$$\Pr\left\{\begin{array}{l}\sigma \stackrel{R}{\leftarrow} \{0,1\}^{k^c}; (PK', \pi_1', \pi_2') \stackrel{R}{\leftarrow} P'(1^k, \sigma):\\ V(1^k, \sigma, PK', x, \pi_1'), V(k, \sigma, PK', x, \pi_2') \neq \perp \wedge\\ V(1^k, \sigma, PK', x, \pi_1') \neq V(k, \sigma, PK', x, \pi_2').\end{array}\right\}$$

3. *Zero-Knowledge.* There exists a database simulator SIM such that $\forall$ Turing machine Adv, $\forall k \in N$, and $\forall$ databases D: View$(k) \approx$ View$'(k)$, where

$$\begin{array}{l}\text{View}(k) =\\ \quad \{\sigma \stackrel{R}{\leftarrow} \{0,1\}^{k^c};\ (PK, SK) \stackrel{R}{\leftarrow} P_1(1^k, \sigma, S);\\ \quad (x_1, s_1) \stackrel{R}{\leftarrow} Adv(1^k, \sigma, PK);\\ \quad \pi_{x_1} \stackrel{R}{\leftarrow} P_2(x_1, SK);\\ \quad (x_2, s_2) \stackrel{R}{\leftarrow} Adv(1^k, \sigma, PK, s_1, \pi_{x_1});\\ \quad \pi_{x_2} \stackrel{R}{\leftarrow} P_2(x_2, SK);\\ \qquad \vdots\\ \quad : PK,\ x_1,\ \pi_{x_1},\ x_2,\ \pi_{x_2},\ \ldots\}\end{array}$$

and

$$\begin{array}{l}View'(k) =\\ \quad \{(\sigma', PK', SK') \stackrel{R}{\leftarrow} SIM(1^k);\\ \quad (x_1, s_1) \stackrel{R}{\leftarrow} Adv(1^k, \sigma, PK);\\ \quad \pi'_{x_1} \stackrel{R}{\leftarrow} SIM^D(SK', x_1);\\ \quad (x_2, s_2) \stackrel{R}{\leftarrow} Adv(1^k, \sigma, PK, s_1, \pi_{x_1});\\ \quad \pi'_{x_2} \stackrel{R}{\leftarrow} SIM^D(SK', x_2);\\ \qquad \vdots\\ \quad : PK',\ x_1,\ \pi'_{x_1},\ x_2,\ \pi'_{x_2},\ \ldots\}\end{array}$$

As usual, various flavors of zero-knowledge arise, depending on whether $\approx$ denotes computational indistinguishability, statistical closeness, or equality.

4. Preliminaries for Our EDB Construction

4.1 Primes, Generators, and the Discrete-Logarithm Assumption

Let p be a prime and q be a prime divisor of $p-1$. Then Z_p^*, the multiplicative group modulo p, is cyclic. and furthermore has a cyclic subgroup G of order q. Furthermore, given p and q it is a straightforward and standard exercise to generate a random element $g \in G$ that generates G (all $g \in G$ save the identity generate G). Given g and $x \in Z_q$, it is easy to compute g^x. The *discrete-logarithm assumption* (DLA for short) formalizes the widely believed assumption that inverting this permutation is indeed computationally intractable.

DLA Let (p_k, q_k) be a family of prime pairs such that $|q| = k, |p| = O(k)$, and $q|p-1$; let G_k and generator g_k be as above. Then, $\forall$ efficient algorithm A the function $\varepsilon(k)$ is negligible, where

$$\varepsilon(k) = \Pr\left\{\begin{array}{l}x \stackrel{R}{\leftarrow} Z_{q_k}; y = g_k^x:\\ A(p_k, q_k, g_k, y) = x\end{array}\right\}$$

4.2 Commitment Schemes

We restrict ourselves to discussing non-interactive commitment schemes, as they are the ones used herein. Informally, such schemes comprise two phases (the *commit* and *verification* phases) and involve two parties (the committer and the verifier).

Both parties share as a common input a random string σ. The commit phase is executed first. In it, the committer, given input m (the *message*, outputs a *commitment string* c, which is made public, and a *proof* r, which she keeps secret. During the verification phase the committer simply publicizes the message m and the proof r, which the verifier checks against σ.

Semantically, at the end of the commit phase, (1) the verifier does not know anything yet about the message, and (2) the Sender "cannot change the message." In the verification phase, the verifier either (a) learns the correct message, if the committer is honest, or (b) learns that the sender has cheated —and suspends any judgment about the message's value.

In our application, we only define commitment schemes in which the committer is polynomially bounded and the verifier unrestricted. More formally,

Definition Let COMMIT and VERIFY be probabilistic algorithms, where COMMIT is polynomial-time. We say that (COMMIT,VERIFY) is a *perfectly hiding, non-interactive commitment scheme* if the following three properties hold:

1. *Completeness:* $\forall \sigma, m \in \{0,1\}^*$,

$$Pr\left\{\begin{array}{l}(c,r) \stackrel{R}{\leftarrow} \text{COMMIT}(\sigma,m): \\ \text{VERIFY}(\sigma,c,m,r) = m\end{array}\right\} = 1$$

2. *Soundness:* $\forall$ efficient algorithm COMMIT$'$, $\varepsilon(k)$ is negligible, where

$$\varepsilon(k) = \Pr\left\{\begin{array}{l}\sigma \stackrel{R}{\leftarrow} \{0,1\}^k;\ (c,r,r') \stackrel{R}{\leftarrow} \text{COMMIT}'(\sigma): \\ \perp \neq \text{VERIFY}(\sigma,c,r) \neq \text{VERIFY}(\sigma,c,r')\end{array}\right\}$$

3. *Zero knowledge:*

$$\forall m_1, m_2 \in \{0,1\}^*,\ \mathcal{C}(\sigma,m_1) = \mathcal{C}(\sigma,m_2),$$

where, for $m \in \{0,1\}^*$,

$$\mathcal{C}(\sigma,m) = \{(c,r) \stackrel{R}{\leftarrow} \text{COMMIT}(\sigma,m) :\ c\}.$$

We remark that, though we have defined commitment schemes so as to be able to handle messages of arbitrary length, we shall use them only for messages whose length is fixed and actually shorter than $|\sigma|$. Indeed, we shall use the following commitment scheme (which will prove to be implementable in the common-random-string model).

Pedersen's Commitment Scheme Pedersen's commitment scheme [20] does not assume the existence of public random string σ, but, rather, 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. (Thus, some additional work is necessary in our model, in order to uniquely from a public random string the correct public quadruple.)

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

PED_COMMIT$((p,q,g,h),m)$: 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.

PED_VERIFY$((p,q,g,h),c,m,r)$: If $c = g^m h^r$, then accept; else, reject.

Clearly, the verifier will always accept values that are committed to and revealed as above. For any m, c is distributed uniformly over G. Thus, c carries no information about m. Notice that being able to find efficiently proofs that the same c is the commitment to two different messages implies the ability to compute efficiently compute the discrete logarithm of h in base g.

4.3 Collision-free Hash Functions

Informally, a collision-free hash function is a polynomial-time computable function H mapping binary strings of arbitrary length into reasonably short ones, so that it is computationally infeasible to find any *collision (for H)*, that is, any two different strings x and y for which $H(x) = H(y)$. Popular candidate collision-free hash function is the standardized *secure hash function* [18] and Rivest's MD5 [21]. Formally, collision-free hash functions are easy to sample function families. Namely,

Definition Let KG (for key generator) be a probabilistic polynomial-time algorithm, $KG : 1^* \to \Sigma^*$, and let E (for evaluator) be a polynomial-time algorithm, $E : \Sigma^* \times 1^* \to \Sigma^*$ (more precisely, $E : \Sigma^* \times 1^k \to \Sigma^k$ for all positive integers k). We say that the pair (KG, E) is a *collision-free hash function* if $\forall$ efficient algorithm A, $\varepsilon(k)$ is negligible, where

$$\varepsilon(k) = \Pr\left\{\begin{array}{l}h \stackrel{R}{\leftarrow} KG(1^k); (x,y) \stackrel{R}{\leftarrow} A(h) : \\ x \neq y \ \wedge\ E(h,x) = E(h,y)\end{array}\right\}.$$

Pedersen's Hash Functions Collision-free hash function exist under the discrete-log assumption. In particular, Pedersen's commitment function may be viewed as the collision intractable hash function: $H = H_{pqgh} : (Z_q)^2 \to Z_p$ so defined: $H(ab) = g^a h^b \bmod p$. If $|q| > \ell$ and $|p| < k$, we can by standard coding tricks treat H as a hash function from $\{0,1\}^{2\ell}$ to $\{0,1\}^k$. By Fouvry's theorem [6], and Bach's

algorithm [1], we can efficiently (and provably) find such p, q where $\ell \approx (2/3)k$; hence, H compresses can be used to compress strings of size $\approx (4/3)k$ to strings of length k. (In practice, we can make $|p| = \alpha|q|$, for any desired alpha.) One may iterate this construction in the standard way (e.g., using Merkle trees) to create collision intractable hash functions from $\{0,1\}^*$ to $\{0,1\}^k$.

An "all or nothing" property of Pedersen's hash function

We note that the hash function computed above has the following "all or nothing" property: Given a single collision ((x_1, x_2) such that $H(x_1) = H(x_2)$), one can compute for *any* x a sibling y such that $H(x) = H(y)$. This property follows from the fact that any collision allows one to compute $\log_g h$. We use this property to show that the (perfect) completeness and zero-knowledge properties of the protocol are unconditional.

4.4 Pseudorandom Functions

Goldreich, Goldwasser, and Micali [8] have shown how to simulate a random oracle from a-bit strings to b-bit strings by means of a construction using a *seed*, that is, a secret and short random string. They show that, if pseudorandom generators exist [3, 22], then there exists a polynomial-time algorithm $GGM(\cdot,\cdot,\cdot,\cdot)$ such that, letting s denote the seed, the function $GGM(s, 1^a, 1^b, \cdot) : \{0,1\}^a \to \{0,1\}^b$ *passes all efficient statistical tests for oracles*. That is, to an observer with sufficiently limited computational resources, accessing a random oracle from $\{0,1\}^a$ to $\{0,1\}^b$ is provably indistinguishable from accessing (as an oracle) $GGM(s, 1^a, 1^b, \cdot)$, even if algorithm GGM is publicly known (provided that s is still kept secret).

As with hash functions, we can easily modify the domain and range of the pseudorandom functions to whatever is convenient. For ease of exposition, we suppress all such standard coding issues.

4.5 Trees and Merkle Trees

Binary Trees We denote by $\mathcal{T}_k$ to be the complete binary tree with 2^k leaves (we shall more simply write $\mathcal{T}$ when k is clear from context). We define the *level* of a vertex as its distance from the root. We label each of the 2^i nodes of $\mathcal{T}$ of level i with an i-bit string such that the vertex labeled v has children labeled $v0$ and $v1$ (thus, the root has label **e**, and its children are labeled 0 and 1). Equivalently, we define the parent of a vertex v, $\text{PARENT}(v)$, as follows: $\text{PARENT}(vb) = v$ for any bit b.

We identify the vertices of $\mathcal{T}$'s by their labels (e.g., given a leaf $x = x_1 \cdots x_n$, the path from the root to x is $\mathbf{e}, x_1, x_1x_2, \ldots, x_1 \cdots x_k = x$.) Similarly, if S is a subset of $\{0,1\}^k$, we identify S with the subset of the leaves of $\mathcal{T}_k$ having the same labels, and define

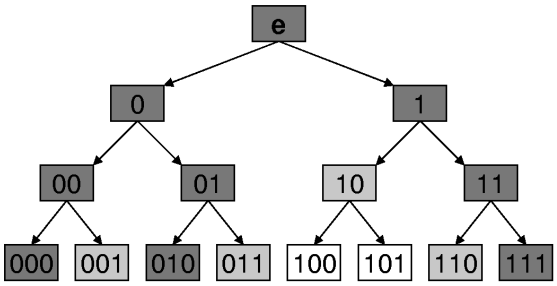

Figure 1. The complete labeled binary tree $\mathcal{T}_3$, and the subtrees $\text{TREE}(S)$ and $\text{FRONTIER}(S)$, for $S = \{000, 010, 111\}$. The darkly shaded vertices comprise $\text{TREE}(S)$, and the lightly shaded vertices comprise $\text{FRONTIER}(S)$.

- $\text{TREE}(S)$ to be the subtree of $\mathcal{T}_k$ consisting of the union of all the paths from the root to the leaves in S; if S is empty, $\text{TREE}(S) = \mathbf{e}$; and
- $\text{FRONTIER}(S) = \{v | v \notin \text{TREE}(S) \text{ and } \text{PARENT}(v) \in \text{TREE}(S)\}$;
  if S is empty, $\text{FRONTIER}(S) = \{\mathbf{e}\}$.
  (See Figure 1.)

We denote the complement of a bit b by $\bar{b}$. We say that distinct vertices $v_1, v_2 \in \mathcal{T}$ are *siblings* if they have the same parent, and denote by $\bar{v}$ the unique sibling of v ($\bar{\mathbf{e}}$ is undefined). Hence, $v = \omega b$ has sibling $\bar{v} = \omega\bar{b}$.

Merkle Trees Given a collision-free hash function H, a subtree T of $\mathcal{T}_k$ is turned into a Merkle tree M by "storing" in every node v of $\mathcal{T}_k$ a value (i.e., binary string) V_v in the following manner: any childless node can store any nonempty binary string, but any other node must store the value $H(ab)$ whenever its left child stores a and its right child stores b: that is, $V_v = H(V_{v0}V_{v1})$. The value stored in the root of M can be viewed as a commitment to all values stored in M's nodes. A proof that node x stores the value V_x (i.e., a proof of $V_x = y$) consists of x's *authentication path:* the sequence of values stored in the siblings of the nodes (except the root) along the path from the root to x. (That is, the sequence of $V_{\bar{\alpha}}$ for all nonempty prefixes α of x.)

For instance, if x is a leaf, then x's authentication path consists of k values, $v_1, \ldots, v_k$, and verifying whether x

stores y is done as follows. Letting x_j be the jth bit of x and $u_k = y$, compute the values $u_{k-1}, \ldots, u_0$ as follows: if $x_j = 1$, set $u_{j-1} = H(v_j u_j)$; else, set $u_{j-1} = H(u_j v_j)$. Finally, check whether the so computed u_0 equals the value $V_{\mathbf{e}}$ stored in the root.

It is immediately seen that, once M's root value is made public, one cannot efficiently compute authentication paths proving that two different values are stored in the same node of M without efficiently finding an H-collision.

5. Our Informal Construction of ZK EDBs

5.1 Committing to database values

Recall that the committer receives three inputs: (1) the security parameter k, (2) the public and random reference string σ —whose length is polynomial in k— and (3) the EDB D as a list of pairs (x, v) —where $x \in [D]$ and $v = D(x)$. We find it useful to describe our committer in terms of the following steps.

Number-Theory Step We omit an intuitive description of this step (making non-trivial use of computational number theory) and are satisfied of just discussing the goals it achieves at a very high level.

In this step the committer "extracts" from σ two quantities: (a) a quadruple (p, q, g, h) as demanded by Pedersen's commitment scheme, so that no one —including the committer— will know $\log_g h$, the discrete log of h in base g modulo p, and (b) a collision-free hash function $H : \{0,1\}^* \to \{0,1\}^k$. This extraction is deterministic and polynomial-time. Thus, it can be easily replicated by any possible verifier, yielding the very same (p, g, h) and H. We also require that one can simulate the creation (p, q, g, h) such that $\log_g h$ is known; this is a standard construction and we omit details in this extended abstract.

Tree-Pruning Step In this conceptual step the committer computes from D the following subtree T of $\mathcal{T}_k$: $T = \text{TREE}(H([D])) \cup \text{FRONTIER}(H([D]))$. That is, first he constructs a subtree T' by putting in it, whenever $D(x) = y$, node $H(x)$ together with all the nodes from $\mathcal{T}_k$'s root to x. Then, he obtains T by adding to T' all the nodes of $\mathcal{T}_k$ whose parent is in T'.

The commitment to our EDB is then obtained by associating to and storing various quantities in T's nodes.

leaf-m-value step In this step the committer associates to each leaf x of T a value m_x as follows: if $x \in H([D])$ then $m_x = H(D(x))$ (i.e., if $D(x) = y$ then he associates $H(y)$ to leaf $H(x)$); else, $m_{H(x)} = 0$. Note that $H(y)$ always is a k-bit value, and thus different from 0.

Considering the entire tree $\mathcal{T}_k$, we call a node *full* if it is the ancestor of at least one leaf x' corresponding to $H([D])$ (i.e,, if there exist strings x and y such that $D(x) = y$ and $H(x) = x'$); else, we call it *empty*.

node-h-value step To each node $v \in T$, the committer associates a random exponent e_v in G (the group generated by g and h), and then stores in v the following value h_v: if $v \in T'$, then $h_v = h^{e_v}$ (in which case no one knows the discrete log of h_v in base g). Else, $h_v = g^{e_v}$.

Thus the committer does not know $\log_g h_v$ for any full v, but does know $\log_g h_v$ for all empty v. Furthermore, revealing e_v such that $h_v = h^{e_v}$ is a proof that one does not know $\log_g h_v$ (assuming one doesn't know $\log_g h$).

leaf-commitment step The committer stores in every leaf v of T a commitment c_v computed as follows. First he associates to v a random element r_v in Z_p^*, and then computes and stores in v a value c_v computed as a Pedersen commitment to m_v (the value already associated to v) using the Pedersen quadruple (p, q, g, h_v): that is, $c_v = g^{m_v} h_v^{r_v}$ modulo p.

Note that, if leaf v is full then c_v is a *genuine* commitment, that is, the committer cannot decommit c_v to any string other than m_v, because it does not know the discrete log of h_v in base g. Else, if leaf v is empty, c_v is a *fake* commitment, that is, the committer can decommit c_v to any string he wishes, because it knows $\log_g h_v$.

Merkle-commitment step By now all nodes n of T store a value h_n, and all leaves v of T have an associated value m_v and store a commitment value c_v. In this step the committer associates a value m_u to and stores a commitment c_u to every internal node u of T. He proceeds in a recursive, bottom-up fashion. Namely, if u is an internal node whose left child $u0$ and right child $u1$ already have stored commitments, then the committer stores in u the value $m_u = H(c_{u0}, h_{u0}, c_{u1}, h_{u1})$ and stores in u the commitment c_u computed (as for a leaf) by first associating to u a random element r_u in Z_p^*, and then computing the Pedersen commitment $c_u = g^{m_u} h_u^{r_u}$ modulo p (using the generator h_u previously stored in u).

Finally, the commitment to the EDB D, c_D, consists of the commitment c_ϵ and the generator h_ϵ stored in the root.

Net Result It should be noted that committing to D is quite fast: it essentially requires three modular exponentiations and one hashing for each element in D's support. It should also be noted (which requires some proving) that this complex operation satisfies a Merkle-tree like property. Namely, as long as he operates in polynomial time, then with high probability the committer cannot change any quantity (i.e., m_v, e_v, h_v, r_v, and c_v) associated to or stored in any node v of T' without also changing c_ϵ. The opposite (except for c_v) is however true for every node v in T's frontier.

5.2 Proving database values

Recall that the EDB prover coincides with the EDB committer, and thus he has in memory the committer's tree T and all values associate to and stored in its nodes.

For proving statements of the form $D(x) = y$, it will suffices from the prover to retrieve T's values, but, for proving statements of the form $D(x) = \perp$, it will be necessary for the prover to augment T with new nodes and and their relative associated and stored values. We start with the simpler case.

Proving $D(x) = y$. To prove $D(x) = y$, the prover produces x and y and reveals a proof π_x consisting of: for every node v along the path $P_{H(x)}$ from leaf $H(x)$ to the root, (1) the values m_v, e_v, h_v, r_v, and c_v and —except for $v = \epsilon$, the root— (2) the values c_u and h_u for v's sibling, u.

Such a proof is verified by (a) checking for the leaf of $P_{H(x)}$ that $m_{H(x)} = H(y)$; (b) checking recursively, for every other node of $P_{H(x)}$, that $m_v = H(c_{v0}, h_{v0}, c_{v1}, h_{v1})$; (c) checking for every v in $P_{H(x)}$ that $h_v = h^{e_v}$ and $c_v = g^{m_v} h_v^{r_v}$ modulo p; and (d) verifying that $c_\epsilon, h_\epsilon = c_D$.

Proof π_x is convincing because it is hard for an adversary to "prove" both $D(x) = y$ and $D(x) = y'$ for $y \neq y'$. This is so because it is hard to a malicious prover to two distinct strings α and β such that $H(\alpha) = H(\beta)$, because H is collision-resistant, and because the prover cannot decommit any of the above c_v in any other way, because Pedersen's commitment is computationally binding as long as one does not know the discrete log of h_v in base g, and because the prover shows that he does not know this discrete log (in fact he proves that he knows the discrete log, e_v, of h_v in base h, and he does not know that of h in base g).

Proving $D(x) = \perp$. To prove that $D(x) = \perp$, the prover computes $H(x)$, and then "moves" in $\mathcal{T}_k$ from the root towards leaf $H(x)$ until he finds the last node u that also belongs to the current subtree T. (Note that u may or may not coincide with leaf $H(x)$, but is always the case that $m_u = 0$ and that c_u is a fake Pedersen commitment.)

Because of the way D is committed to, proving that c_u could be decommitted to 0 would also prove that leaf $D(x) = \perp$; however, unless u is also a leaf of $\mathcal{T}_k$, this would also reveal additional knowledge: namely that "below" u are do not correspond to D's support, something that in turn provides information about the size of D's support.

Thus, the prover will enlarge the current T by incorporating in it the subtree T_u of $\mathcal{T}_k$ rooted at u and consisting of (1) the subpath from u to $H(x)$ together with (2) all of the siblings of the nodes in this subpath (except the root u). Such incorporation will require computing values m_v, e_v, h_v, r_v, and c_v for each node v in T_u (computed similarly to the other nodes of T, except that, like for u, all such nodes v will have store a value h_v —for which the prover knows the discrete log in base g— and a fake commitment c_v) and then obtaining a new tree T by "decommitting" c_u so as to seamlessly weld T_u into the old T. Let us now see how the prover associates values to the *new* nodes v of T_u (i.e., those other than u).

In each new node v of T_u the prover stores the value h_v so computed: first, he randomly chooses an exponent $e_v \in Z_P^*$, and then computes g^{e_v} modulo p. (Thus, by construction, he will know the discrete log of h_v in base g!) For each new leaf v of T_u, he sets $m_v = 0$, chooses $r_v \in Z_p^*$ at random, and stores in v the "fake Pedersen commitment" $c_v = g^0 h_v^{r_v}$ modulo p. Then, he processes all other new nodes v of T_u, so as to compute the values m_v, e_v, h_v, r_v, and c_v, in a recursive, bottom-up fashion: namely, if v is an internal node whose left child $v0$ and right child $v1$ already have been processed, then the prover associates to v the value $m_v = H(c_{v0}, h_{v0}, c_{v1}, h_{v1})$ and stores in v the commitment c_v computed (as for a leaf) by first associating to v a random element r_v in Z_p^*, and then computing $c_v = g^{m_v} h_v^{r_v}$ modulo p, using the generator h_v already associated to v.

The prover now "welds T_u into T" as follows. He computes $m_u = H(c_{u0}, h_{u0}, c_{u1}, h_{u1})$, and then using the fact that he knows the discrete log of h_u in base g, he decommits c_u (originally a fake commitment to 0) to m_u: that is, he computes a new r_u such that $c_u = g^{m_u} h_u^{r_u}$ modulo p.

To prove $D(x) = \perp$, the prover produces x and reveals a proof π'_x consisting of: for every node v along the path $P_{H(x)}$ from leaf $H(x)$ to the root, (1) the values m_v, h_v, r_v, and c_v and —except for $v = \epsilon$, the root— (2) the values c_u and h_u for v's sibling, u.

Such a proof is verified by (a) checking for the leaf of $P_{H(x)}$ that $m_{H(x)} = 0$; (b) checking recursively, for every other node of $P_{H(x)}$, that $m_v = H(c_{v0}, h_{v0}, c_{v1}, h_{v1})$; (c) checking for every v in $P_{H(x)}$ that $c_v = g^{m_v} h_v^{r_v}$ modulo p; and (d) verifying that $c_\epsilon, h_\epsilon = c_D$.

5.3 Soundness is preserved

Note that proof π_x is syntactically identical to a proof of the type $D(x) = y$, except that $m_{H(x)}$ will be 0 rather than a k-bit value $H(y)$ and all values e_v are omitted: that is, the prover does not reveal how he constructed the values h_v (and thus checking their construction is skipped during verification). Of course, if the verifier ever witnessed a proof of the type $D(x) = y$, he will have seen how *some* of those h_v were actually constructed (e.g., h_ϵ, the value associated to the root). But for some other generators (e.g., $h_{H(x)}$) he would have never seen how it was constructed. Further, by knowing the strategy of the prover, the verifier will know that a non-empty subset of the latter values (including $h_{H(x)}$) have actually been used to generate fake Pedersen commitments. The presence of such fake commitments

may actually raise some concerns about the *soundness* of the overall construction. They do *not*, however, enable the prover to find both a proof π_x of $D(x) = y$ and a proof π'_x of $D(x) = \perp$. This can be informally argued as follows.

Let π_x consist of values m_v, h_v, r_v, c_v **and** e_v and sibling values c_u and h_u; and let π'_x consist of values m'_v, h'_v, r'_v, c'_v and sibling values c'_u and h'_u. We distinguish two cases:

Case 1: $(c_{H(x)}, h_{H(x)}) = (c'_{H(x)}, h'_{H(x)})$. In this case, the prover has found the $\log_g h$. In fact, $m_{H(x)} = H(y)$ and $m'_{H(x)} = 0$ are two different Pedersen decommitments of $c_{H(x)} (= c'_{H(x)})$, relative to the same g and $h_{H(x)} (= h'_{H(x)})$. Thus by the way Pedersen commitment works, this implies that the prover has found $\log_g h_{H(x)}$. This, per se, may not be hard to find, because the prover has chosen $h_{H(x)}$. However, the prover has also included in proof π_x the value of $\log_h h_{H(x)}$. Given this value and $\log_g h_{H(x)}$, it is easy to compute $\log_g h$, which violates the discrete logarithm assumption, because h is not chosen by the prover, but randomly selected in G (via the random reference string).

Case 2: $(c_{H(x)}, h_{H(x)}) \neq (c'_{H(x)}, h'_{H(x)})$. In this case, the prover has found a collision for H. Notice that leaf $H(x)$ belongs to path $P_{H(x)}$, and so does root ϵ. But for root ϵ, $(c_\epsilon, h_\epsilon) = (c'_\epsilon, h'_\epsilon)$, because the commitment to D, c_D, consists of c_ϵ and h_ϵ and π_x and π'_x are convincing proofs relative to the same c_D. Thus, let u be the first node v of $P_{H(x)}$ (starting from the root) such that (a) $c_v = c'_v$ and $h_v = h'_v$ and (b) for v's child in $P_{H(x)}$, $v0$ without loss of generality, $(c_{v0}, h_{v0}) \neq (c'_{v0}, h'_{v0})$. We distinguish two subcases. *Subcase 2.1:* $m_u = m'_u$. In this subcase, the prover has found a collision for H, because $m_u = H(c_{u0}, h_{u0}, c_{u1}, h_{u1})$ and $m'_u = H(c'_{u0}, h'_{u0}, c'_{u1}, h'_{u1})$. *Subcase 2.2:* $m_u \neq m'_u$. In this subcase, using the same reasoning of Case 1, the prover has found $\log_g h$.

5.4. Odds and Ends

After making precise the informal description above, one can prove for our construction all the claimed efficiency and zero-knowledge properties of ZK EDBs, but not their perfect completeness. If $(x', y'), (x'', y'') \in D$ and $H(x') = x = H(x'')$, then a honest prover is forced to provide an incorrect answer for at least one key, since either $m_x = H(y')$ or $m_x = H(y'')$.[2]

To guarantee perfect completeness, one needs an additional idea. Rather than being "independent" of one another, in the current attempt H and COMMIT will be *all-or-nothing matched*. As usual, $H : \{0,1\}^* \to \{0,1\}^k$ is deterministic (because we want that each possible key x' determines a unique leaf $x = H(x')$ that may store information about $D(x')$) and COMMIT is probabilistic (because we want the scheme to be perfectly hiding). However, H and COMMIT are so correlated: though finding any H-collision is hard, if *any* H-collision is found, then *any* commitment string c generated by COMMIT can be de-committed arbitrarily. That is, COMMIT is trap-door, and any H-collision yields a trapdoor, t, for COMMIT.

Thus, assume that $(x', y'), (x'', y'') \in D$, that $D(x''') = \perp$, that $x = H(x') = H(x'') = H(x''')$, and that —without loss of generality— the honest prover processes (x', y') first — thus computing $m_x = H(y')$ — and (x'', y'') later. Then he will not reset m_x when processing (x'', y''). Rather, having found a H-collision, he will compute and store the trapdoor t for COMMIT. During the proving process, the honest prover produces a proof π_x (i.e., c_v, m_v, r_v for each prefix v of x, and $c_{\bar{v}}$ for every nonempty prefix v of x) for $D(x') = y'$ as usual. But, should he need to prove that $D(x'') = y''$ (respectively, $D(x''') = \perp$), he will produce the same π_x above, except for the values m_x and r_x that he so computes: using trapdoor t, he computes a proof r''_x (respectively, r'''_x) for "c_x decommits to $H(y'')$ (respectively, 0)" and sets $m_x = H(y'')$ and $r_x = r''_x$. Fortunately, under the DLA, we can construct *very efficient* such $H -$ COMMIT pairs! Indeed, we can use Pedersen's scheme as the underlying commitment scheme.[3]

A Possible Trade-off As discussed so far, the memory (but not the computation) required during the proving process grows with the number of new proofs of non-membership. In fact, to keep consistency among such proofs, the prover must store and re-use some node-dependent random values (e.g., r_v and e_v). To avoid, such growth (or to enable different servers given the same secret key of a committed database to prove facts about it without coordinate with one another). We can use pseudorandom functions [8] to solve this problem (namely, any random value associated to a node v is computed pseudo-randomly on input v, and all computed values about v are re-computed) but degrade the zero-knowledge property from perfect to computational. The latter approach is what we actually adopted in the posted pseudo-code.

6. Final Thoughts

Additional Privacy Given our anthropocentric perspective, zero-knowledge sets and databases are particularly attractive when dealing with sets and databases about *people*. In such applications, the privacy of "membership" information is just one of the desiderata, though often be the most difficult one to enforce in a totally provable way. For instance, it is not hard to enhance our construction so as to ensure that

[2](Alternatively, setting $m_x = H((x, y'), (x'', y''))$ generates a ZK problem, since proving that $(x'y') \in D$ proves also that $(x'', y'') \in D$.)

[3]Note that a solution to the small leakage problem could have been obtained as follows: for each pair $(x'y') \in D$, store $y = H(y')$ in leaf $P(x)$, where $P(\cdot)$ denotes a prefix-free encoding. Such a solution is however, much less efficient whenever long keys may be used to address the database, than just hash and insert!

- It is impossible to prove, without x's cooperation, whether person $x \in D$ —and, if so, which his record $D[x]$ is. (This may be useful if D consists of the medical records of the patients of an hospital.)
- It is possible to prove portions of record $D[x]$ in "isolation," that is, without revealing anything about the other portions. (E.g., within $D[x]$, one can separate financial information from medical information.)
- Only certain given entities can read certain given portions (and only such portions) of the record, $D[x]$, of person x.
- A database D can be distributed across a multiplicity of servers, which can interface with the broad "outside" so as to both produce and prove any value $D[x]$, but without understanding it.

Details of these simple but powerful privacy enhancements will be given in the final paper.

Open Questions We hope that non-interactive zero-knowledge sets (and databases) will attract further research. In particular, we hope that the following questions will be answered:

- Is it possible to *update* a zero-knowledge set at a low additional cost (e.g., logarithmic in the size of the set)?
- Is it possible to handle multiple *provers* (e.g., ensuring that their sets are "independent")?
- Is it possible to construct zero-knowledge sets under *weaker* complexity assumptions?

In Sum Zero-knowledge sets and databases provide a new *mathematical* primitive, with enormous *application potential*.

References

[1] E. Bach. How to generate random integers with known factorization. In *Proceedings of the fifteenth annual ACM Symposium on Theory of Computing*, pages 184–188, ACM Press, 1983.

[2] M. Blum, P. Feldman, and S. Micali. Proving security against chosen ciphertext attacks. In S. Goldwasser, editor, *Proc. CRYPTO 88*, pages 256–268. Springer-Verlag, 1988. Lecture Notes in Computer Science No. 403.

[3] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.

[4] M. Blum, A. De Santis, S. Micali, and G. Persiano. Noninteractive zero-knowledge. *SIAM J. Comput.*, 20(6):1084–1118, 1991.

[5] I. B. Damgard, T. P. Pedersen, and B. Pfitzmann. On the existence of statistically hiding bit commitment schemes and fail-stop signatures. *Lecture Notes in Computer Science*, 773:250–??, 1994.

[6] E. Fouvry. Theoreme de Brun-Titchmarsh; application au theoreme de Fermat. *Invent. Math.*, 79:383–407, 1985.

[7] S. Goldwasser, S. Micali, and C. Rackoff. "The Knowledge Complexity of Interactive Proof Systems", *SIAM J. Comput.*, 18 (1):186–208, 1989.

[8] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the Association for Computing Machinery*, 33(4):792–807, October 1986.

[9] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attack. *SIAM Journal on Computing*, 17:281–308, 1988.

[10] S. Halevi and S. Micali. Practical and provably-secure commitment schemes from collision-free hashing. In *Proc. 16th International Advances in Cryptology Conference – Crypto '96*, pages 201–215, 1996.

[11] A. Kalai. Generating random factored numbers, easily. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)*, pages 412–412, New York, January 6–8, 2002. ACM Press.

[12] J. Kilian. A note on efficient zero-knowledge proofs and arguments. In Proc. 24th Ann. ACM Symp. on Theory of Computing, pages 723–732, Victoria, B.C., Canada, May 1992.

[13] J. Kilian. Improved efficient arguments. In Proc. 15th International Advances in Cryptology Conference – CRYPTO '95, pages 311–324, 1995.

[14] J. Kilian Efficiently Committing to Databases TR #97-040, NEC Research Institute, 1997.

[15] R. 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.

[16] S. Micali. Computationally Sound Proofs, In proceedings, 35*th IEEE Symposium on Foundations of Computer Science*, 1994.

[17] S. Micali, M. Rabin Hashing on Strings, Cryptography, and Protection of Privacy. In *Proceedings Compression and Complexity of Sequences* IEEE Computer Society, Los Alamitos, CA, June 11-13, 1997, p. 1. (First presented at Berkeley Symp. on Randomness, 1996.)

[18] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, April 1995. Supersedes FIPS PUB 180 1993 May 11.

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

[20] T. Pedersen. Noninteractive and information-theoretic secure verifiable secret sharing. *Lecture Notes in Computer Science*, 576:129–140, 1991.

[21] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, MIT, RSA Data Security, April 1992.

[22] A. C. Yao. Theory and applications of trapdoor functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.

A. Detailed Construction of Zero-Knowledge Databases

A.1. Construction-Dependent Terminology about Binary trees

Given a database D with support S, we label each vertex $v \in \mathcal{T}_k$ with a collection of values, described as follows. We first generate (pseudo)random elements $a_v, b_v \in Z_q$. We similarly generate $e_v \in Z_q$ pseudorandomly, with the requirements that $e_v \neq 0$ and $e_{\mathbf{e}} = 1$. These pseudorandom values have little conceptual meaning, but are used to generate the "meaningful" values, described below. We generate them pseudorandomly to simplify the memory requirements of our protocol, as they can be implicitly generated once and for all.

We "hash" the support S of D as follows. If $D(x) = y$ we say that $D(v) = y$, where $v = H(x) \in \{0,1\}^k$ is a vertex of $\mathcal{T}_k$. We ignore the unlikely possibility of a collision (see Section 5.4). We hereafter treat S as being defined over $\{0,1\}^k$ when convenient.

The "meaningful" values we associate with v are m_v, c_v, h_v and r_v. Here, m_v is a hash of some of the values for v's children (as with a Merkle tree), c_v is a Pedersen commitment to m_v, with generator h_v, and r_v is the value needed to prove that m_v was indeed the committed value. We define these as follows:

- $m_v = \begin{cases} H(y) & \text{if } |v| = k \text{ and } D(v) = y \\ 0 & \text{if } |v| = k \text{ and } D(v) \text{ is undefined} \\ H(c_{v0}, h_{v0}, c_{v1}, h_{v1}) & \text{if } |v| < k. \end{cases}$
- $h_v = h^{e_v}, r_v = a_v$ and $c_v = g^{m_v} h_v^{r_v}$ if $v \in \text{TREE}(S)$, and
- $h_v = g^{e_v}, c_v = g^{b_v}$ and $r_v = \frac{b_v - m_v}{e_v}$ if $v \notin \text{TREE}(S)$.

A.2. The Database Committer

COMMIT_DATABASE(D, σ, k)

1. The committer uniquely identifies from the reference string σ a prime p, two generators g and h for Z_p^*, and a hash function n H as follows:

   First, the committer parses σ as $\sigma = \sigma_1 \cdots \sigma_{k^2} \tau_1 \cdots \tau_{k^2}$; where (i) each σ_ℓ has length equal to the number of coin tosses sufficient (with high probability) for Bach's algorithm to output a random k-bit integer in factored form, and (ii) each τ_ℓ has length $O(k)$.

   Second, the committer finds the first integer i such that $BACH(1^k)$ with coin tosses σ_i outputs a k-bit integer n_i, in factored form, such that $n_i + 1$ is prime and n_i has a prime factor q of size at least $2(k+1)/3$; and sets $p = n_i + 1$.[4]

   Third, using the factorization of $p - 1$, the committer finds the first two integers a and b such that $g = \tau_a^{(p-1)/q}$ and $h = \tau_b^{(p-1)/q}$ are generators of G, the subgroup of Z_p^* or order q (treating τ_a and τ_b as numbers mod p). It lets $H = H_{pqgh}$ be the corresponding Pedersen hash function.

   Finally, if the above initialization procedure fails to produce the desired values for σ (i.e., it runs out of random bits), it simply stops. We choose σ large enough (details omitted) so that such a failure occurs with probability at most 2^{-k}; in such an insignificant case we have the verifier always accept.

2. The committer chooses a seed $s \stackrel{R}{\leftarrow} \{0,1\}^k$, thus implicitly defining (but not explicitly computing) a pseudorandom $GGM(v, \cdot)$, mapping each vertex label v to a triple (a_v, b_v, e_v), as described in Section A.1.

3. Let S be the "hashed" support of D. For each vertex $v \in \text{FRONTIER}(S)$, the committer computes c_v and h_v according to the definition given in Section A.1 (note that $v \in \text{FRONTIER}(S)$ implies $v \notin \text{TREE}(S)$. These values depend solely on (a_v, b_v, e_v).

   Then, for $v \in \text{TREE}(S)$, the committer computes m_v, h_v, r_v and c_v (according to Section A.1). Note that for any vertex $v \in \text{TREE}(S)$, these values depend on the database entries, (a_v, b_v, e_v) and on the c and h values computed for the children of v ($v0$ and $v1$). Also note that any child of $v \in \text{TREE}(S)$ is either in $\text{TREE}(S)$ or $\text{FRONTIER}(S)$. Thus, the committer may compute m_v, h_v, r_v and c_v for the leaves of

[4] It follows from Fouvry's theorem [6] that only $O(k)$ expected n_i need be generated.

TREE(S), and then successively visit v after it has visited v's children (those in FRONTIER(S) have already been visited).

4. Finally, the committer outputs the public key PK consisting of $c_{\mathbf{e}}$, the commitment at the root node of the tree, and the secret key SK consisting of the values $a_v, b_v, e_v, m_v, h_v, c_v, r_v$ for each node $v \in$ TREE$(S) \cup$ FRONTIER(S).

   [If in any of the operations described above, the committer observes a collision: (x_1, y_1) and (x_2, y_2) such that

   $$g^{x_1} h_v^{y_1} = g^{x_2} h_v^{y_2},$$

   where $h_v = h^{e_v}$, it computes

   $$\log_g h = \frac{y_2 - y_1}{e_v(x_1 - x_2)},$$

   and stores this value.]

   (Comment: TREE$(S) \cup$ FRONTIER(S) contains less than $3kn$ nodes if if S has cardinality n.)

A.3. The Database Prover

PROVE(x, SK)

If $D(x) = y$, the prover executes PROVE_DATA(D, x, y, SK, σ).

If $D(x) =$ *out* , the prover executes PROVE_EMPTY(x, SK).

[If in the execution of COMMIT_DATABASE*, the committer computed $d = \log_g h$, the prover may perform these operations trivially (details omitted).]*

PROVE_DATA(D, x, y, SK, σ) */* prove that $D(x) = y$ */*

The prover outputs the statement "$D(x) = y$" and the proof π_x consisting of the stored values

- $e_v, r_v, c_v, c_{v0}, h_{v0}, c_{v1}, h_{v1}$ for each $v = v_0, \ldots, v_{k-1}$ (where $\mathbf{e} = v_0, \ldots, v_k = H(x)$ is the sequence of vertices from $\mathbf{e}$ to $H(x)$); and
- $e_{H(x)}, r_{H(x)}, c_{H(x)}, y$.

Note that these values were computed during the execution of COMMIT_DATABASE.

PROVE_EMPTY(x, SK) */* prove that $D(x)$ is undefined */*

The prover outputs the statement "$D(x) =$ *out*" and the proof π_x obtained by computing from scratch (given the definition in COMMIT_DATABASE) or retrieving from storage the following values

- $h_v, r_v, c_v, c_{v0}, h_{v0}, h_{v1}, h_{v1}$ for each $v = v_0, \ldots, v_{k-1}$ (where $\mathbf{e} = v_0, \ldots, v_k = H(x)$ is the sequence of vertices from $\mathbf{e}$ to $H(x)$); and
- $h_{H(x)}, r_{H(x)}, c_{H(x)}$.

(Comments: (1) The value $h_{\mathbf{e}} = h$ always, and is thus not computed nor sent. (2) Values must be recomputed from scratch only for those nodes in the sub-path, SP, from a node in FRONTIER$([D])$ to leaf $H(x)$. All nodes u in SP do not belong to TREE$([D])$, and thus their c_v values do not depend on their children's values, but are "locally" computed as a function of u. Therefore, since the length of SP is at most k, the values of at most $2k$ nodes must be computed from scratch, either locally or based on previously computed "from scratch" values. This entails that $O(k)$ GGM evaluations, hashings and exponentiations mod p, suffice to compute π_x. (3) In this exposition, the same value appears in multiple locations; the prover sends them only once.)

A.4. The Database Verifier

VERIFY$(x, \mathsf{statement}, \pi_x, PK, \sigma)$

- The verifier extracts the quantities p, q, g, h, H from the reference string σ just as the honest prover (if it cannot do so, it simply accepts); then
- If statement= "$D(x) = y$", the verifier executes VERIFY_DATA(x, y, π_x, PK).

  If statement= "$D(x) =$ *out*", then verifier executes VERIFY_EMPTY(x, π_x, PK).

VERIFY_DATA$(x, y, \pi_x, PK, \sigma)$

1. For each node v such that the verifier receives e_v, the verifier computes $h_v = h^{e_v}$, and checks that this is consistent with the stated values of h_v if given to it (verifier also checks that $h_{\mathbf{e}} = h$). For $v = v_0, \ldots, v_{k-1}$, the verifier computes $m_v = H(c_{v0}, h_{v0}, c_{v1}, h_{v1})$ and $m_{v_k} = m_{H(x)} = H(y)$.
2. For $v = v_0, \ldots, v_k$, the verifier computes PED_VERIFY$_{p,g,h_v}(c_v, m_v, r_v)$, and rejects if any of these tests rejects. Otherwise, the verifier accepts.

VERIFY_EMPTY$(x,$ *out*$, \pi_x, PK, \sigma)$:

1. For $v = v_0, \ldots, v_{k-1}$, the verifier computes

   $$m_v = H(c_{v0}, h_{v0}, c_{v1}, h_{v1})$$

   and $m_{v_k} = m_{H(x)} = 0$.
2. For $v = v_0, \ldots, v_k$, the verifier computes PED_VERIFY$_{p,g,h_v}(c_v, m_v, r_v)$, and rejects if any of these tests rejects. Otherwise, the verifier accepts.