Private Database Queries Using Somewhat Homomorphic Encryption

Dan Boneh[1], Craig Gentry[2], Shai Halevi[2], Frank Wang[3], and David J. Wu[1]

[1] Stanford University
{dabo,dwu4}@cs.stanford.edu
[2] IBM Research
craigbgentry@gmail.com, shaih@alum.mit.edu
[3] MIT
frankw@mit.edu

Abstract. In a private database query system, a client issues queries to a database and obtains the results without learning anything else about the database and without the server learning the query. While previous work has yielded systems that can efficiently support disjunction queries, performing conjunction queries privately remains an open problem. In this work, we show that using a polynomial encoding of the database enables efficient implementations of conjunction queries using somewhat homomorphic encryption. We describe a three-party protocol that supports efficient evaluation of conjunction queries. Then, we present two implementations of our protocol using Paillier's additively homomorphic system as well as Brakerski's somewhat homomorphic cryptosystem. Finally, we show that the additional homomorphic properties of the Brakerski cryptosystem allow us to handle queries involving several thousand elements over a million-record database in just a few minutes, far outperforming the implementation using the additively homomorphic system.

1 Introduction

Enabling private database queries is an important research problem that arises in many real-world settings. The problem can be thought of as a generalization of symmetric private information retrieval (SPIR) [3,8] where clients can retrieve records by specifying complex queries. For example, the client may ask for the records of all people with age 25 to 29 who also live in Alaska, and the server should return these records without learning anything about the query. The client should learn nothing else about the database contents.

In this work we explore the use of somewhat homomorphic encryption (SWHE) [5] for the design of private database query protocols. In particular, we show that certain polynomial encodings of the database let us implement interesting query types using only homomorphic computations involving low-degree polynomials. There are now several encryption schemes [1,2] that efficiently support the necessary low-degree homomorphic computations on encrypted data needed for our constructions.

M. Jacobson et al. (Eds.): ACNS 2013, LNCS 7954, pp. 102–118, 2013.

Unfortunately, being a generalization of SPIR, private database queries is subject to all the same inherent inefficiency constraints as SPIR. To understand these limitations let us consider the two parties involved in the basic setup: the client and the server. The server has a database and the client has a query. We seek a protocol that gives the client only those records that match its query without the server learning any information about the query. In this setting the server must process the entire database for every query; otherwise, it would learn that the unprocessed records do not match the query. Moreover, the server has to return to the client as much data as the number of records in the database, or else the database would learn some information about the number of records that match the query. Thus, for large databases, the server is forced to do a considerable amount of work, rendering such systems impractical in most scenarios.

To overcome these severe limitations we modify the basic model a bit and consider a setting in which the database server is split into two entities called the "server" and the "proxy." Privacy holds as long as these two entities do not collude. This approach was taken by De Cristofaro et al. [4], who designed a system that supported private evaluation of a few simple query types and demonstrated performance similar to a non-private off-the-shelf MySQL system. However, the architecture of De Cristofaro et al. could not handle conjunctive queries: for instance, the client could ask for all the records with `age=25 OR name='Bob'`, but could not ask for the records with `age=25 AND name='Bob'`. Another multi-party architecture for performing private database queries is proposed in [13]. In this case, the server constructs an encrypted document index which is stored on an index server (e.g., "proxy" in our setting). To submit queries, the client interacts with a query router. One of the limitations of this scheme is that for each query, the server has to perform a computation on each record in the database, which does not scale well to very large databases.

In this work, we develop protocols that can efficiently support conjunction queries over large databases using an architecture similar to [4]. We rely on somewhat homomorphic encryption schemes [1,2] that efficiently support low-degree homomorphic computations on encrypted data.

1.1 Security Model

The functionality that our protocol implements gives the client the indices of the records that match its query. The client should learn nothing about the data beyond this set and the server and proxy should learn nothing about the query beyond what is explicitly leaked.

More precisely, security for the client means that if the client issues one of two adversarially-chosen queries with the same number of attributes, the adversarial server cannot distinguish between them. Security for the server means that for any fixed query and two adversarially-chosen databases for which the query matches the same set of records, the client cannot distinguish the two databases.

In this paper, we adopt the honest-but-curious security model. Our protocols can be enhanced to handle malicious adversaries using generic tools such as [10]. It is an interesting open problem to design more efficient protocols in the ma-

licious settings specific to the private database queries problem. Security holds as long as the server and the proxy do not collude. This is very similar to the assumptions made in [13].

1.2 Our Protocol

The protocol and tools we present in this work are aimed at revealing to the client *the indices* of the records that match its query, leaving it to a standard follow-up protocol to fetch the records themselves. The approach that underlies our protocol is to encode the database as one or more polynomials and then manipulate these polynomials using the client's query so as to obtain a new polynomial whose roots are the indices of the matching records. This representation is well suited for conjunction queries, since it allows us to use techniques similar to the Kissner-Song protocol for (multi-)set intersection [11].

In our protocol, the three parties consist of a client with a query, a proxy that has an inverted index for the database, and a server that prepared the inverted index during a pre-processing step and now keeps only the keys that were used to create this inverted index. Specifically, the server keeps some "hashing keys" and the secret key for a SWHE scheme. For every attribute-value pair (a, v) in the database, the inverted index contains a record $(\mathsf{tg}, \mathsf{Enc}(A(x)))$ where tg is a tag, computed as $\mathsf{tg} = \text{Hash}(\text{“}a = v\text{”})$, and $A(x)$ is a polynomial whose roots are exactly the records indices r that contain this attribute-value pair.

An example query supported by our protocol is:

$$\texttt{SELECT} \star \texttt{FROM db WHERE } a_1 = v_1 \texttt{ AND } \cdots \texttt{ AND } a_t = v_t.$$

Given this query, the client (with oblivious help from the server) computes the tags $\mathsf{tg}_i = \text{Hash}(\text{“}a_i = v_i\text{”})$ for $i = 1, \ldots, t$ and sends them to the proxy. The proxy fetches the corresponding encrypted polynomials $A_i(x)$ from the inverted index, chooses random polynomials $R_i(x)$ of "appropriate degrees" and computes the encrypted polynomial $B(x) = \sum_{i=1}^{t} R_i(x)A_i(x)$. The proxy returns the encrypted B to the client, who again uses oblivious help from the server to decrypt B, and then factors it to find its roots, which are the indices of the matching records (with high probability).

One drawback of this protocol is that the proxy can tell when two different queries share the same attribute-value pair (since the client will send the same tag in both). In Section 3.3, we show that using quadratic-homomorphic encryption, we can mitigate this drawback somewhat, providing a privacy/bandwidth tradeoff that the client can tune to its needs.

Bandwidth reduction and other optimizations. Another drawback of the protocol above is that the degree of the encrypted polynomial B returned by the proxy (which determines the size of the response) depends on the *largest* number of records that match any of the attribute-value pairs in the query. For example, if the client query was "`SELECT ⋆ FROM db WHERE gender='male' AND zipcode=12345`," the response size will be at least as large as the number of males in the database, even if there are only a few people with zipcode 12345.

In Section 3.2, we describe how to reduce this degree (and bandwidth) by observing that the minimum-degree polynomial that encodes the intersection is the gcd of the A_i's. We show that the somewhat homomorphic properties of the cryptosystem can be used to approximate the gcd. Our discussion here will lead to a storage/homomorphism tradeoff. We present additional optimizations in Section 3.3. In Section 3.4 we show that we can take advantage of homomorphic batching [6,14]) to further speed up the computation.

Implementation and performance results. We implemented our three-party protocol using both the additive homomorphic Paillier cryptosystem [12] and a variant of Brakerski's system [1] that supports a single multiplicative homomorphism. Our implementation, described in Section 4, shows that the use of multiplicative homomorphisms greatly improves performance and bandwidth over the strictly additive implementation using Paillier.

2 Preliminaries

2.1 Homomorphic Encryption

Fix a particular plaintext space $\mathcal{P}$ which is a ring (e.g., $\mathcal{P} = \mathbb{F}_2$). Let $\mathcal{C}$ be a class of arithmetic circuits over the plaintext space $\mathcal{P}$. A somewhat homomorphic (public-key) encryption relative to $\mathcal{C}$ is specified by the procedures $\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}$ (for key generation, encryption, and decryption, respectively) and the additional procedure Eval that takes a circuit from $\mathcal{C}$ and one ciphertext per input to that circuit, and returns one ciphertext per output of that circuit.

The security requirement is the usual notion of semantic security [9]: it should be hard to distinguish between the encryption of any two adversarially-chosen messages, even if the public key is known to the adversary. The functionality requirement for homomorphic schemes [5] is that for every circuit $\pi \in \mathcal{C}$ and every set of inputs to π, if we choose at random the keys, then encrypt all the inputs, then run the Eval procedure on these ciphertexts and decrypt the result, we will get the same thing as evaluating π on this set of inputs (except perhaps with negligible probability). An important property of SWHE schemes is *circuit privacy*, which means that even the holder of the secret key cannot learn from the evaluated ciphertext anything about the circuit, beyond the output.

In this work we use "low degree" somewhat homomorphic encryption, namely homomorphic encryption schemes relative to the class of low degree polynomials. While our basic protocol requires only additive homomorphism, some of our optimizations require that the scheme support polynomials of higher degree.

2.2 Polynomial Arithmetic and Set-Intersection

We provide a brief overview of the techniques underlying the Kissner-Song set-intersection protocol [11]. Our setting is different than that considered in [11], hence also our use of these techniques is somewhat different. Roughly, Kissner and Song considered the case where each party has a set and they want to

compute the intersection of all their sets. In our case we have one party holding all the sets (the server), and another party that determines which of these sets should participate in the intersection (the client).

The idea behind the Kissner-Song protocol is to fix a large field $\mathbb{F}$ and represent a set $S \subset \mathbb{F}$ by a polynomial A_S that has zeros in all the elements of S, that is $A_S(x) = \prod_{s \in S}(x - s)$. To compute the intersection of many sets S_i, we construct a polynomial B whose zeros are the intersection of these sets. Clearly, if some point $s \in \mathbb{F}$ is contained in all the sets S_i, then $A_{S_i}(s) = 0$ for all i, and therefore, if we compute B as a linear combination of the A_{S_i}'s, then also $B(s) = 0$. On the other hand, if $A_{S_i}(s) \neq 0$ for some i and B is a *random* linear combination of the A_{S_i}'s, then with high probability $B(s) \neq 0$.

The Kissner-Song approach is therefore to choose the field $\mathbb{F}$ sufficiently larger than the "universe" U of valid points (e.g., we have $S_i \subseteq U \subsetneq \mathbb{F}$), then take B to be a random linear combination of the A_{S_i}'s, and show that with high probability, the only roots of B that come from U are the ones corresponding to the intersection of the S_i's. The following lemma is easy to prove using the above arguments:

Lemma 1. *Fix a finite field $\mathbb{F}$ and a "universe" $U \subset \mathbb{F}$, let $S_1, \ldots, S_t \subseteq U$ be subsets of the universe and for each S_i, let $A_{S_i}(x) = \prod_{s \in S_i}(x - s)$.*

(i) Let $\rho_1, \ldots, \rho_{t-1}$ be random scalars in $\mathbb{F}$, let $A'(x) = A_{S_t} + \sum_{i<t} \rho_i A_{S_i}(x)$, and denote the set of roots of A' by $S_{A'}$. Then $\Pr[S_{A'} \cap U = \bigcap_i S_i] \geq 1 - |U|/|\mathbb{F}|$.

(ii) Let R_1, R_2 be random polynomials in $\mathbb{F}[x]$ of some given degrees $d_1, d_2 \geq 0$. Let $B(x) = A_1(x)R_1(x) + A_2(x)R_2(x)$, and S_B be the set of roots of B. Then $\Pr[S_B \cap U = S_1 \cap S_2] \geq 1 - |U|/|\mathbb{F}|$.

The harder part is to show that the random linear combination B does not leak information on the A_{S_i}'s beyond their intersection. For this to hold, the coefficients of the linear combination cannot be scalars in $\mathbb{F}$, they must be themselves polynomials of high-enough degree. Specifically, we use the following lemma which is a slight generalization of [11, Lemma 1]:

Lemma 2. *Fix a finite field $\mathbb{F}$ and two co-prime polynomials $A_1(x), A_2(x) \in \mathbb{F}[x]$, of degrees $d_1 = \deg(A_1)$ and $d_2 = \deg(A_2)$. Also, fix some integer $D_1 \geq d_1 - 1$, and let $D_2 = d_2 + D_1 - d_1$. Next, choose uniformly at random a degree-D_2 polynomial $R_1(x) \in \mathbb{F}[x]$ and a degree-D_1 polynomial $R_2(x) \in \mathbb{F}[x]$ and set $B(x) = A_1(x) \cdot R_1(x) + A_2(x) \cdot R_2(x)$. Then, $B(x)$ is distributed uniformly among all the polynomials of degree $d_1 + D_2 = D_1 + d_2$ over $\mathbb{F}$.*

Proof. Omitted due to space constraints. See appendix of the full version. □

Corollary 1. *Fix a finite field $\mathbb{F}$ and two polynomials $A_1(x), A_2(x) \in \mathbb{F}[x]$, with degrees d_1 and d_2, respectively. Let $G(x) = \gcd(A_1(x), A_2(x))$. Also fix some integer $D_1 \geq d_1 - 1$, and let $D_2 = d_2 + D_1 - d_1$. Then choosing uniformly at random a degree-D_2 polynomial $R_1(x) \in \mathbb{F}[x]$ and a degree-D_1 polynomial $R_2(x) \in \mathbb{F}[x]$ and setting $B(x) = A_1(x) \cdot R_1(x) + A_2(x) \cdot R_2(x)$, the polynomial*

$B(x)$ is distributed uniformly among all the polynomials of degree d_1+D_2 over $\mathbb{F}$ which are divisible by $G(x)$.

Proof. Follows by applying Lemma 2 to the co-prime polynomials $A'_1(x) = A_1(x)/G(x)$ and $A'_2(x) = A_2(x)/G(x)$. □

Intersection of two sets. If $A_{S_1}(x)$, $A_{S_2}(x)$ are polynomials that represent sets S_1, S_2, respectively, then $\gcd(A_{S_1}, A_{S_2})$ is the polynomial that represents their intersection. In this case, Corollary 1 says that setting $B = A_{S_1}R_1 + A_{S_2}R_2$ for R_1, R_2 of "appropriate degrees" yields a random multiple of $G(x)$ that leaks "no information" about A_1, A_2 beyond their intersection and the sum of their sizes.[1]

Intersection of many sets. In this setting, we are given the polynomials A_{S_i}, $i = 1, 2, \ldots, t$, with $d_i = \deg(A_{S_i})$. Without loss of generality, let d_t be the largest degree. We first choose random scalars, $\rho_i \in \mathbb{F}$ for $i = 2, \ldots, t$, and compute the degree-d_t polynomial $A'(x) = A_{S_t}(x) + \sum_{2 \le i < t} \rho_i A_{S_i}(x)$. Then we choose two random polynomials $R_1(x)$ of degree $d_t - 1$ and $R'(x)$ of degree $d_1 - 1$ and set $B(x) = A_{S_1}(x)R_1(x) + A'(x)R'(x)$.

Clearly $\gcd(A_{S_1}, A_{S_2}, \ldots, A_{S_t})$ divides $\gcd(A_{S_1}, A')$. Also Lemma 1 (applied to $U = S_1$ and $S'_i = S_i \cap S_1$) implies that with probability at least $1 - d_1/|\mathbb{F}|$ we have $\gcd(A_{S_1}, A') = \gcd(A_{S_1}, A_{S_2}, \ldots, A_{S_t})$. It follows from Corollary 1 that when the size of $\mathbb{F}$ is super-polynomially larger than d_1, the distribution of $B(x)$ is statistically close to uniform over the degree-$(d_1 + d_t - 1)$ polynomials divisible by $\gcd(A_{S_1}, A_{S_2}, \ldots, A_{S_t})$.

Reducing the degree. To reduce the degree of the resulting polynomials, instead of using $A'(x) = \sum_i \rho_i A_{S_i}(x)$, we compute the polynomial $A''(x) = A'(x) \bmod A_{S_1}(x)$ of degree $d_1 - 1$. Choosing at random $R_1(x)$ of degree $d_1 - 1$ and $R''(x)$ of degree d_1, we set $B(x) = A_1(x)R_1(x) + A''(x)R''(x)$. Correctness and secrecy follow from the observation that since $A''(x) = A'(x) \bmod A_{S_1}(x)$, $\gcd(A_{S_1}, A'') = \gcd(A_{S_1}, A')$.

3 The Three-Party Protocol

In this section, we describe the three-party setting that we adopt in this paper (which is similar to the "Isolated-Box" architecture in [4]). In this architecture, in addition to the client and server there is a third party, a proxy, that holds an "encrypted" inverted index of the database records. For each attribute-value pair in the database, the proxy holds a tag that identifies the pair, along with a set of record indices that contain the pair. Specifically, for each attribute-value pair in the database (e.g., "`name=Joe`"), the inverted index contains the following:

$$\langle \mathsf{PRF}_s(\text{“}\texttt{name=Joe}\text{”}), \quad \text{encrypted-set-of-record-indices} \rangle \tag{1}$$

[1] We can pad to a pre-determined degree to hide the information about the sizes.

where the PRF key s is held by the server and the set of record indices contains all the records where the attribute "name" has value "Joe."

When the client wants to fetch the records with `name=Joe`, it engages in a protocol for oblivious-PRF-evaluation with the server and learns the tag $\mathsf{PRF}_s(\text{“}\texttt{name=Joe}\text{”})$. It then engages in a protocol with the proxy to learn the set of indices corresponding to this tag. To make a conjunction query, the client sends multiple tags to the proxy and at the end of the protocol, learns the records in the intersection of all the sets.

3.1 Our Basic 3-Party Protocol

The task of computing conjunctions is closely related to set intersection. Indeed, an attribute-value pair (e.g., "`name=Joe`") implicitly defines a set of records that contains this pair. The proxy needs to send the intersection of all these sets to the client, without learning anything about the sets themselves.

Using the technique of Kissner and Song described in Section 2.2, we represent each set as a polynomial whose roots are the elements of that set. Thus, in the row of the inverted index with tag $\mathsf{PRF}_s(\text{“}\texttt{name=Joe}\text{”})$, we do not store the set of indices S containing this attribute-value pair, but rather the polynomial $A_S(x) = \prod_{s \in S}(x - s)$, encrypted using our SWHE scheme. Note that the SWHE scheme is used to encrypt each *coefficient* of the polynomial A_S. To issue a conjunctive query (say, "`name=Joe`" and "`age=28`"), the client does the following:

1. Use oblivious-PRF-evaluation to obtain from the server the tags $\mathsf{tg}_1, \ldots, \mathsf{tg}_t$ corresponding to each of the attribute-value pairs. The client sends all the tags to the proxy.
2. The proxy collects the encrypted polynomials A_i corresponding to the tags tg_i and then computes a polynomial $B(x)$ as a "random linear combination" of the $A_i(x)$'s:
   (i) Letting $d_i = \deg(A_i)$ and assuming that the A_i's are ordered by degree ($d_1 \leq d_2 \leq \cdots \leq d_t$), the proxy first chooses random scalars $\rho_2, \ldots, \rho_{t-1}$ and computes the degree-d_t polynomial $A'(x) = A_t + \sum_{2 \leq i < t} \rho_i A_i(x)$.
   (ii) Then the proxy chooses two random polynomials $R_1(x)$ of degree $d_t - 1$ and $R'(x)$ of degree $d_1 - 1$ and sets $B(x) = A_1(x)R_1(x) + A'(x)R'(x)$. The proxy uses the additive homomorphism of the scheme to compute the encrypted coefficients of the polynomial B from the encrypted coefficients of the A_i's and the plaintext ρ_i, R_1 and R'. The proxy sends the encrypted $B(x)$ to the client.
3. The client and server engage in another protocol to decrypt $B(x)$ (encrypted under the server's key). At the conclusion of this protocol, the client knows $B(x)$ and the server knows nothing.
4. The client factors $B(x)$ and finds its roots, which are the indices of the records that the client is interested in. While $B(x)$ may have superfluous roots, we use a large-enough space so that with high probability these roots are identified as invalid and discarded.

Once the client knows the indices of the records that match its query, it can use PIR/ORAM protocols to fetch the encrypted records, then engage in another oblivious decryption protocol with the server to decrypt them.

Security. Secrecy against an honest-but-curious proxy is ensured by the fact that the tags do not leak to the proxy anything about the attribute-value pairs that were used to generate them (because the tag-generation function is pseudo-random), and the encrypted polynomials do not leak anything due to the semantic security of the SWHE cryptosystem. Note that our security model only ensures privacy for a single query. If the client issues multiple queries then the proxy may learn relations between these queries. We briefly discuss multiple queries in Section 3.3.

Secrecy against an honest-but-curious client follows from Corollary 1 and the circuit-privacy property of the SWHE scheme. Specifically, Corollary 1 implies that the polynomial B by itself does not leak anything about the A_i's beyond their intersection (and the size $d_1 + d_t$), and circuit-privacy of the cryptosystem means that the evaluated ciphertext encrypting B does not leak anything else.

3.2 Reducing Communication via Modular Reduction

The communication complexity of the basic solution above is determined by the degree of the polynomial B, which is tied to the size of the largest set in the intersection (e.g., the highest degree d_t). Using some more homomorphic operations, we can make the degree of B as low as $2d_1 - 1$, namely it can be tied to the size of the smallest set S_1 rather than the largest set S_t.

To this end, we use the optimization from Section 2.2, where instead of using $A'(x) = A_t(x) + \sum_{2 \le i < t} \rho_i A_i(x)$, the proxy uses $A''(x) = A' \bmod A_1(x)$. We note that given the encrypted coefficients of both the polynomial $A'(x)$ of degree d_t and the *monic* polynomial $A_1(x)$ of degree d_1, we can homomorphically reduce A' modulo A_1 as long as our SWHE scheme supports formulas of degree $d_t - d_1$. To see this, notice that given the encryption $\mathsf{Enc}(\alpha'_{d_t})$ of the top coefficient of A', we can reduce the degree of A' by one by setting $A'' = A' - \alpha'_{d_t} \cdot A_1(x) \cdot x^{d_t - d_1}$. Clearly the degree of A' is one less than that of A' and it satisfies $A'' \equiv A' \pmod{A_1}$.

However, reducing modulo A_1 can be done using more limited homomorphism if the proxy is given not just the encryption of A_1 but also some other ciphertexts. For example, suppose the proxy is given the encryption $\mathsf{Enc}(x^i \bmod A_1)$ for $i = d_1 + 1, d_1 + 2, d_1 + 3, \ldots, d_t$. Then given the encryptions of all the coefficients of A', $\mathsf{Enc}(\alpha'_0), \ldots, \mathsf{Enc}(\alpha'_{d_t})$, the proxy computes the encryption of the reduced polynomial as $\mathsf{Enc}(A' \bmod A_1) = \mathsf{Enc}(\sum_{i=0}^{d_t} \alpha'_i (x^i \bmod A_1))$. Since the proxy has the encryptions of all the α'_i's and the $(x^i \bmod A_1)$'s, then it is enough if our SWHE scheme supports only quadratic formulas, such as [7,1].

The above two procedures for computing polynomial modular reduction represent two extremes on the storage/homomorphism tradeoff. Perhaps a better tradeoff can be obtained by storing only logarithmically many encrypted polynomials corresponding to A_1, and using a SWHE scheme supporting formulas

of degree $O(\log d_t)$. Denoting $\Delta = d_t - d_1$, the proxy is given the encryptions $\mathsf{Enc}(x^{d_1+2^i} \bmod A_1)$ for $i = 0, 1, \ldots, \lceil \log \Delta \rceil$. Given these encryptions and the encryptions of the coefficients of A', reducing A' modulo A_1 homomorphically can be done in $\lceil \log \Delta \rceil$ steps. See appendix of full version for more details.

3.3 Other Optimizations and Variations

Returning two polynomials. The most expensive operation that the client performs in our protocol is factoring the polynomial B. Even with the bandwidth reduction trick from above, its degree is still twice as large as the degree of the smallest A_i, which can be much higher than the degree of the gcd of the A_i's.

A simple trick that can be used here is to have the proxy send to the client two encrypted polynomials. Namely, after the proxy computes the polynomial A' in Step 2(i), it repeats Step 2(ii) twice, that is, choose polynomials R_1, R' and S_1, S' and set $B(x) = A_1(x)R_1(x) + A'(x)R'(x)$ and $C(x) = A_1(x)S_1(x) + A'(x)S'(x)$. The proxy sends the encrypted B and C to the client, who engages in an oblivious decryption protocol with the server to decrypt both. Then the client computes the gcd of the two polynomials B and C, and with high probability this polynomial is the gcd of all the A_i's, which hopefully has much lower degree than B, C themselves.

Obscuring relations between different queries. One problem with the basic solution above is that the client sends to the proxy all the tags $\mathsf{tg}_i = \mathsf{PRF}_s(\text{attr}_i = \text{value}_i)$, so the proxy can tell when a given tg_i is used in multiple queries. This problem can be mitigated by adding spurious tags to the request, but without changing the result of the final intersection. The idea is to have the client send to the proxy pairs (tg_i, s_i) where tg_i is a tag for an attribute-value pair and s_i is an encryption of a bit $\sigma_i \in \{0, 1\}$. By using a quadratic-homomorphic encryption scheme (such as [7]), the proxy can choose its randomizers $R_i(x)$ and compute an encryption of the polynomial $B(x) = \sum_i R_i(x) \cdot (\sigma_i \cdot A_i(x))$. The client will send some spurious tags tg_i with $\sigma_i = 0$, thus obscuring the tags that it is really interested in, but without changing the result of the intersection.

3.4 Speedups via Batching

One appealing optimization that applies to the protocol in this paper is to use "batch homomorphic encryption" where a single ciphertext represents a vector of encrypted values and a single homomorphic operation on two such ciphertexts applies the homomorphic operation component-wise to the entire vector. This way, for the cost of a single homomorphic operation we get to compute on an entire vector of encrypted plaintexts. This is a cryptographic analogue of the Single Instruction Multiple Data (SIMD) architecture and is supported by recent fully homomorphic encryption systems [1,14,2,6].

We take advantage of batching in our context by splitting the database into a few small partial databases and running the same query against all parts

in parallel. When using the techniques from [14,2,6] (for the ring-LWE-based homomorphic encryption) we can pack in each ciphertext ℓ different plaintext elements (where ℓ is typically in the range of 500-10,000). We can then break an r-record database into ℓ smaller databases, each with $\approx r/\ell$ records.

In the three-party setting, with each tag $\mathsf{tg}_i = \mathsf{PRF}_s(\text{“}\mathsf{attr}_i = \mathsf{val}_i\text{”})$, we keep encryptions of ℓ different polynomials, one for each part of the database. These are placed in the ℓ “plaintext slots” of the ciphertexts, so the number of ciphertexts that needs to be kept is only as large as the degree of the largest of these ℓ polynomials. (If the records are split between the parts uniformly, then we expect this degree to be roughly a factor of ℓ smaller than it would be if we keep everything as a single database.) A client query will still be processed in the exact same way as in the previous sections, but now the client will get back from the proxy not a single encrypted polynomial $B(x)$ but ℓ different polynomials $B_j(x)$, one for each of plaintext slot. The client gets the decryption of all these B_i's from the server, factors them all, and takes the union of their roots to be the set of records that match the query.

4 Implementing the Three-Party Protocol

We implemented the basic three-party protocol from Section 3 using both the Paillier cryptosystem [12] and a variant of Brakerski's leveled homomorphic system [1]. Because the Paillier cryptosystem only supports additive homomorphism, we can only support the basic protocol, without the batching (Section 3.4) and modular reduction optimizations (Section 3.2). In contrast, Brakerski's leveled homomorphic scheme supports a bounded number of homomorphic additions and multiplications. To demonstrate the effectiveness of our optimizations we conducted a set of experiments with batching and modular reduction using Brakerski's cryptosystem. Since most of our described optimizations pertain specifically to the problem of oblivious set intersection, we focus our experimental analysis on this portion of the three-party protocol.

In this section, we show that support for batching (Section 3.4) in Brakerski's system is critical for evaluating large queries. Specifically, for large queries, the Paillier system becomes intractable, leaving the Brakerski system as the only suitable option. We also demonstrate that the modular reduction optimization (Section 3.2) yields substantial reductions in *both* computation time and network bandwidth on queries where there is a large disparity in the sizes of the record sets corresponding to the tags. In one case, we show a 4X improvement in *both* processing time and bandwidth using modular reduction.

4.1 Homomorphic Encryption Schemes

Paillier cryptosystem. Recall that the Paillier cryptosystem works over $\mathbb{Z}^*_{n^2}$ for an RSA-modulus n of unknown factorization. The scheme has plaintext space $\mathcal{P} = \mathbb{Z}_n$ and ciphertext space $\mathbb{Z}^*_{n^2}$. The scheme is additively homomorphic, with

Table 1. Parameters used to achieve 128-bit security in the Brakerski system. The false positive rate is fixed at 10^{-3}.

Experiment	Ring Modulus Φ_m	Plaintext Slots $\varphi(m)$	Plaintext Modulus p	Ciphertext Modulus q
NoMR	$m = 5939$	$\varphi(m) = 5938$	$p = 1000032577$	$\log_2 q = 181$
MR, MRNoKS	$m = 7867$	$\varphi(m) = 7866$	$p = 1000021573$	$\log_2 q = 238$

homomorphic addition implemented by multiplying the corresponding ciphertexts in $\mathbb{Z}_{n^2}^*$. Similarly, we can homomorphically multiply a ciphertext $c \in \mathbb{Z}_{n^2}^*$ by a constant $a \in \mathbb{Z}_n$ by computing $c^a \bmod n^2$.

Brakerski's leveled homomorphic cryptosystem. We also use the ring-LWE-based variant of Brakerski's scale-invariant homomorphic cryptosystem [1]. Specifically, our implementation operates over polynomial rings modulo a cyclotomic polynomial. Let $\Phi_m(x)$ denote the m^{th} cyclotomic polynomial. Then, we work over the ring $R = \mathbb{Z}[x]/\Phi_m(x)$. Specifically, we take our plaintext space to be $\mathcal{P} = R_p = \mathbb{Z}_p[x]/\Phi_m(x)$ and our ciphertext space to be $R_q = \mathbb{Z}_q[x]/\Phi_m(x)$ for some $q > p$. In this scheme, our secret keys and ciphertexts are *vectors* of elements in R_q. Homomorphic addition is implemented by adding the corresponding ciphertexts. We can multiply a ciphertext $\mathbf{c}$ by a constant $a \in R_p$ by computing $a\mathbf{c}$. Finally, homomorphic multiplication is performed using a tensor product. Note that when we homomorphically multiply two ciphertexts, the resulting ciphertext is encrypted under a tensored secret key. Using a technique called *key-switching*, we can transform the product ciphertext into a regular ciphertext encrypted under the original secret key. We refer readers to [1] for further details.

As noted in Section 3.4, one of the main advantages of using a ring-LWE-based homomorphic scheme is the fact that we can pack multiple plaintext messages into one ciphertext using a technique called batching. To use batching we partition a database with r records into ℓ separate databases, each containing approximately r/ℓ records. Correspondingly, the the degrees of the polynomials in each database are reduced roughly by a factor of ℓ. In our implementation, $\ell \geq 5000$, so this translates to a substantial improvement in performance.

We now consider a choice for the plaintext modulus p for use in the Brakerski scheme. From Lemma 1, we have that the probability of a false positive (mistaking an element not in the intersection to be in the intersection) is given by $|U| \,/\, |\mathbb{F}_p|$. If we tolerate a false positive rate of at most $0 < \lambda < 1$, then we require that $|\mathbb{F}_p| \geq \frac{1}{\lambda} |U| = \frac{r}{\lambda}$, where r is the number of records in the database. Additionally, to maximize the number of plaintext slots, we choose p such that $p = 1 \pmod m$. To summarize, we choose our plaintext modulus p such that $p = 1 \pmod m$ and $p \geq \frac{r}{\lambda}$.

4.2 Experimental Setup

We implemented the three-party protocol using both the Paillier and Brakerski cryptosystems as the underlying homomorphic encryption scheme. Our

implementation was done in C++ using the NTL library over GMP. Our code was compiled using `g++ 4.6.3` on Ubuntu 12.04. We ran all timing experiments on cluster machines with multicore AMD Opteron processors running at 2.1 GHz. The machines had 512 KB of cache and 96 GB of available memory. All of our experiments were conducted in a single-threaded, single-processor environment. Memory usage during the computation generally stayed below 10 GB.

In the Paillier-based scheme, we used a 1024-bit RSA modulus for all of our experiments. For the Brakerski system, we chose parameters m, p, q to obtain 128-bit security and a false positive rate of $\lambda = 10^{-3}$. See appendix of full version for derivation of parameters. Since the Brakerski system supports both the batching and modular reduction optimizations described in Section 3.4 and Section 3.2, respectively, we considered three different experimental setups to assess the viability of these optimizations. Below, we describe each of our experiments. The parameters used in our SWHE scheme for each setup are given in Table 1.

NoMR: *Brakerski scheme without modular reduction.* In the NoMR setup, we just used the batching capabilities of the Brakerski system. Note that this setup only required homomorphic addition, and *not* homomorphic multiplication, and thus, allowed us to use smaller parameters in the Brakerski system.

MR: *Brakerski scheme with modular reduction.* In the MR setup, we considered the modular reduction optimization from Section 3.2. In the final step of the three-party protocol, the proxy computes the polynomial $B(x) = A_1(x)R_1(x) + A'(x)R'(x)$ where $\deg(A_1) \leq \deg(A')$. When we perform modular reduction, we compute $A'(x) \pmod{A_1(x)}$ followed by $B(x) \pmod{A_1(x)}$. This optimization reduces the degree of the polynomial $B(x)$ that the proxy sends to the client as well as the cost of the computation of $B(x)$. To perform this optimization, the SWHE scheme must support at least one multiplication, thus requiring larger parameters for security. Consequently, each homomorphic operation takes longer, but since we are performing fewer operations overall, the modular reduction can yield substantial gains for certain queries. Due to the cost of homomorphic multiplications, we just consider the case of doing a single multiply.

MRNoKS: *Brakerski scheme with modular reduction but without key switching.* When we homomorphically multiply two ciphertexts in the Brakerski system, we obtain a tensored ciphertext (e.g., a higher-dimensional ciphertext) encrypted under a tensored secret key. Normally, we perform a key-switching operation that transforms the tensored ciphertext into a new ciphertext encrypted under the normal secret key. If left unchecked, the length of the ciphertexts grows exponentially with the number of successive multiplications. Thus, the key-switching procedure is important for constraining the length of the ciphertexts. In our application, we perform a single multiplication, and so the key-switching procedure may be unnecessary. Since the key-switching operation has non-negligible cost, we can achieve improved performance at the expense of slightly longer ciphertexts (and thus, increased bandwidth) by not performing the key switch.

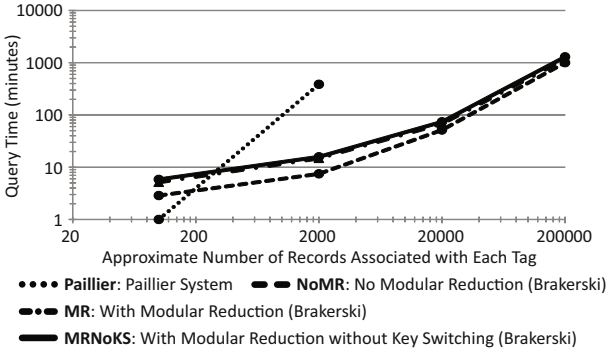

Fig. 1. Timing tests on *balanced* queries using the Paillier cryptosystem and the three setups of the Brakerski cryptosystem described in Section 4.2. All queries were conducted over a database consisting of 10^6 records. Each query consisted of five tags; the approximate number of records associated with each tag is indicated on the plot above. Note that the running time with Paillier became too large when the database had more than 2,000 records per tag and as a result the Paillier line stops at 2,000.

Query type. In each of our experiments, we operated over a database with 10^6 records and performed queries consisting of five tags. Let $d_1 \leq d_2 \leq \cdots \leq d_5$ denote the number of elements associated with each tag $\mathtt{tg}_1, \ldots, \mathtt{tg}_5$. We profiled our system on two different sets of queries: *balanced* queries and *unbalanced* queries. In a balanced query, the number of elements associated with each tag was approximately the same: $d_1 \approx d_2 \approx \cdots \approx d_5$.

In an unbalanced query, the number of elements associated with each tag varies significantly. Specifically, d_1 is at most 5% of d_5. As discussed in Section 1, queries like these where we compute an intersection of a large set with a much smaller set are very common and so, it is important that we can perform such queries efficiently. For each query, we measured the computation time as well as the total network bandwidth required by each of our setups. Note that due to the poor scalability of the Paillier system, we were not able to perform the full set of experiments using the Paillier cryptosystem.

4.3 Experimental Results

Balanced queries. In the first set of experiments, we considered the run-time and bandwidth requirements for performing balanced queries. In particular, we constructed a database with 10^6 records and where each tag in the database was associated with approximately d records (for d ranging from 100 to 200,000). We executed these queries on the four different setups described above (Paillier, NoMR, MR, and MRNoKS). Our timing and bandwidth measurements are summarized in Fig. 1 and Fig. 2. Because the query execution time dominated the cost of the computation, we just present the cost of performing the query.

We compare the computational cost and network bandwidth required by each of our setups described in Section 4.2 for evaluating balanced queries. From

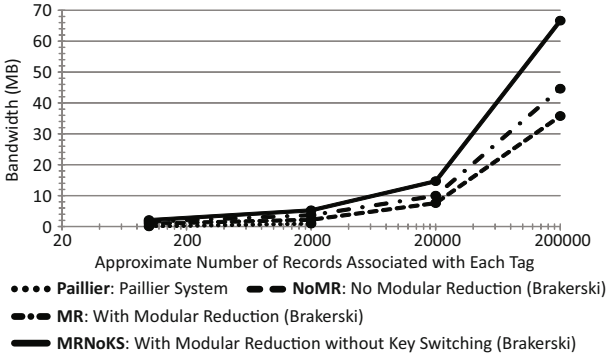

Fig. 2. Bandwidth measurements on *balanced* queries using the Paillier cryptosystem and the three different setups of the Brakerski cryptosystem. Same setup as in Fig. 1.

Fig. 1, we see that the Paillier system is faster for small queries involving sets of several hundred records. This is due to the simplicity and low computational overhead of the Paillier cryptosystem compared to Brakerski's leveled homomorphic cryptosystem. However, the run time scales quadratically with the size of the underlying sets, so for queries with over 2,000 elements, the Paillier system becomes completely impractical. While the performance using Brakerski's system also scales quadratically with the number of records, batching allows us to split the main database $\mathcal{D}$ into ℓ slices, each with approximately $\frac{|\mathcal{D}|}{\ell}$ records. Thus, we were able to reduce the degree of the polynomials we needed to multiply by a factor of approximately $\ell > 5000$. In turn, batching allows for approximately a factor of ℓ increase in the number of records the system could handle. Using Brakerski's system, we are able to handle queries for tags consisting of 200,000 records. These results also indicate that in terms of both bandwidth and computation time, the modular reduction optimization from Section 3.2 is ineffective when we have *balanced* queries. This is because the modular reduction optimization is designed for cases where there is a large disparity between the sizes of the smallest and largest sets. When the size of each set is approximately equal, the larger parameters needed to support the modular reduction optimization coupled with the computational cost of performing the optimization resulted in worse performance overall. Thus, for balanced queries, it is advantageous to just use the Brakerski system without additional optimizations.

Unbalanced queries. We also considered the case where the underlying sets are unbalanced, that is, cases where the smallest set contains at most 5% of the number of records in the largest set. Due to the poor scalability of the Paillier system, we only performed the queries using our three Brakerski setups. Our results are summarized in Fig. 3 and Fig. 4.

When working with unbalanced queries, the modular reduction optimization (with or without key switching) reduces the necessary bandwidth. Despite the fact that each individual ciphertext is larger when we perform modular reduction

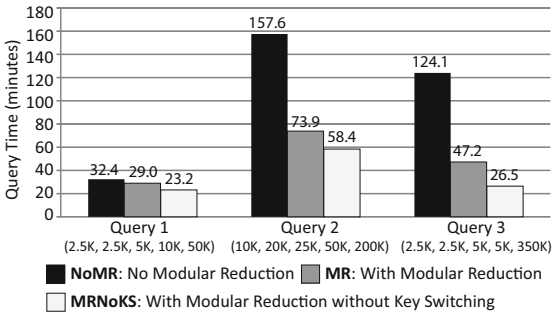

Fig. 3. Timing tests on *unbalanced* queries using the three different setups of the Brakerski system (described in Section 4.2). All queries were conducted over a database consisting of 10^6 records. Each query consisted of five tags; the number of records associated with each tag is shown in parenthesis in the corresponding graphs.

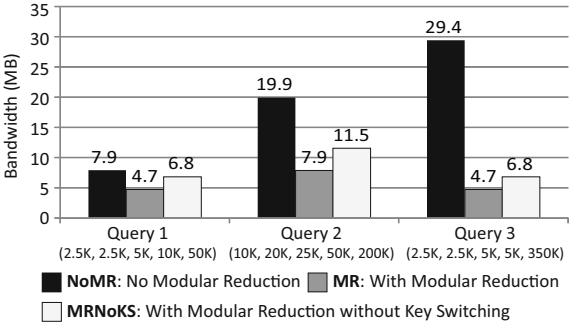

Fig. 4. Bandwidth measurements on *unbalanced* queries using the three different setups of the Brakerski system. Same setup as in Fig. 3.

(due to the larger parameters in the Brakerski system), the polynomials also have much lower degree (degree given by $2d_1 - 1$ rather than $d_1 + d_5 - 1$). The larger the difference between d_1 and d_5, the more substantial the bandwidth reduction. Furthermore, performing modular reduction also translated to faster query processing. Recall that in the last step of the proxy computation, the proxy multiplies a polynomial of degree $d_5 - 1$ with one of degree $d_1 - 1$. If we use modular reduction, the multiplication is instead performed on two polynomials of degree d_1 and $d_1 - 1$. From our experiments, we see that when $d_1 = 10{,}000$ and $d_5 = 200{,}000$ (Query 2), the MRNoKS setup is about 2.7 times faster. When this gap is even larger with $d_1 = 2{,}500$ and $d_5 = 350{,}000$ (Query 3), we observe that the MRNoKS setup is almost 4.7 times faster than the NoMR system. Even with key switching in this case (Query 3), modular reduction still reduces the run time by a factor of 2.6. In both MR and MRNoKS, the bandwidth on this very unbalanced query is reduced by more than a factor of 4 compared to the baseline without the modular reduction optimization.

To summarize, performing the modular reduction optimization is greatly beneficial, both in terms of computation time as well as in terms of network bandwidth, when there is a large difference between the sizes of the underlying sets. As we have demonstrated, it is possible to achieve over a 4X improvement in *both* computation time and network bandwidth on certain queries, making modular reduction a very viable optimization in practice.

5 Conclusion

This paper presents new protocols and tools that can be used to construct a private database query system supporting a rich set of queries. We showed how a polynomial representation of the database allows for efficient evaluation of private conjunction queries. The basic schemes only require an additively homomorphic system like Paillier, but we showed that significant performance improvements can be obtained using a stronger homomorphic system that supports both homomorphic additions and a few homomorphic multiplications. Our experiments quantify this improvement showing a real-world example where lattice-based homomorphic systems can outperform their factoring-based counterparts.

Acknowledgements. This work is supported by IARPA via DoI/NBC contract number D11PC20202. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

References

1. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R. (ed.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Heidelberg (2012)
2. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. In: Innovations in ITCS 2012 (2012)
3. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. J. ACM 45(6), 965–981 (1998)
4. De Cristofaro, E., Lu, Y., Tsudik, G.: Efficient techniques for privacy-preserving sharing of sensitive information. In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) Trust 2011. LNCS, vol. 6740, pp. 239–253. Springer, Heidelberg (2011)
5. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009), `http://crypto.stanford.edu/craig`
6. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 465–482. Springer, Heidelberg (2012)

7. Gentry, C., Halevi, S., Vaikuntanathan, V.: A simple BGN-type cryptosystem from LWE. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 506–522. Springer, Heidelberg (2010)
8. Gertner, Y., Ishai, Y., Kushilevitz, E., Malkin, T.: Protecting data privacy in private information retrieval schemes. In: STOC 1998, pp. 151–160 (1998)
9. Goldwasser, S., Micali, S.: Probabilistic encryption. Journal of Computer and System Sciences 28(2), 270–299 (1984)
10. Ishai, Y., Prabhakaran, M., Sahai, A.: Founding cryptography on oblivious transfer – efficiently. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 572–591. Springer, Heidelberg (2008)
11. Kissner, L., Song, D.: Privacy-preserving set operations. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 241–257. Springer, Heidelberg (2005)
12. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)
13. Raykova, M., Cui, A., Vo, B., Liu, B., Malkin, T., Bellovin, S.M., Stolfo, S.J.: Usable, Secure, Private Search. IEEE Security and Privacy, 53–60 (October 2012)
14. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations (2011), Manuscript at `http://eprint.iacr.org/2011/133`