Privacy-Enhanced Searches Using Encrypted Bloom Filters

Steven M. Bellovin
smb@research.att.com
AT&T Labs Research

William R. Cheswick
ches@lumeta.com
Lumeta

Abstract

It is often necessary for two or more or more parties that do not fully trust each other to selectively share data. We propose a search scheme based on Bloom filters and Pohlig-Hellman encryption. A semi-trusted third party can transform one party's search queries to a form suitable for querying the other party's database, in such a way that neither the third party nor the database owner can see the original query. Furthermore, the encryption keys used to construct the Bloom filters are not shared with this third party. Provision can be made for third-party "warrant servers", as well as "censorship sets" that limit the data to be shared.

1 Introduction

It is often necessary for two or more or more parties that do not fully trust each other to selectively share data. For example, two intelligence agencies may wish to let each other query their databases, while only disclosing clearly relevant documents to the other party. Even then, there may be be restrictions that must be observed.

Assume there are two principals, a querier and an information provider. Ideally, we would like the following properties to hold:

a. The querier gains no knowledge of the contents of the provider's database, except for documents that are matched by valid queries.

b. The provider gains no knowledge of the contents of the queries; if possible, that should include inferences based on the documents retrieved.

c. An independent party may restrict the set of legitimate queries.

d. No third parties may gain any knowledge of the queries or the documents.

Conventional search techniques do not have these properties. We propose a new scheme that does satisfy our requirements, based on encrypted Bloom filters [1].

We will often speak of some party's "filter", in the singular. While that will sometimes be the case, in general each party will have very many filters, one per document. In addition, there may be "union filters" for distinct collections of documents, or for optimization in searching. Fundamentally, little of that matters for our purposes, though we do discuss some aspects of this. When we say "search Bob's filter", we really mean "search all of the Bloom filters owned by Bob and protected by his key". In some situations, such a search can be optimized; in other cases, it cannot.

It should be emphasized that our scheme is not a total solution to the problem of information sharing. Just knowing that someone has some information on a topic is useful. Indeed, an absence of public discussion has been useful; in 1940, the lack of American publications on nuclear physics persuaded the Soviets that atomic bomb research was under way [2].

We are concerned with solving one problem: providing a controlled way for one party to learn something about documents owned by others, without disclosing the query. Exactly what happens next will vary, depending on the circumstances; we outline a few possibilities, but we emphasize that there are many more ways matters can proceed from that point.

Section 2 reviews the basics of Bloom filters. Section 3 explains encrypted Bloom filters, and shows how to use Pohlig-Hellman encryption to satisfy one of our requirements. Further privacy-protecting enhancements are discussed in Section 4.

This is a draft paper and should not be mirrored or archived.

From there, we move to more practical concerns. Section 5 shows how a semi-trusted third party can be provided with certain special keys, while never having any keys to permit decryption of messages. Section 6 discusses system design considerations for this work. We conclude with a discussion of related work and some final thoughts.

2 Bloom Filters

A Bloom filter is a very efficient way to store information about the existence of a record in a database. It is susceptible to false positives, but the probability of a false positive can be made as small as desired.

A Bloom filter is an array B of m bits, initialized to zero. It requires a set of n independent hash functions H_i that produce uniformly distributed output in the range $[0, m-1]$ over all possible inputs.

To add an entry W to the filter, calculate

$$\begin{aligned} b_1 &= H_1(W) \\ b_2 &= H_2(W) \\ \ldots & \\ b_n &= H_n(W) \end{aligned}$$

$$\forall i, 1 \leq i \leq n, \text{set } B[b_i] = 1$$

To check if W is in the database, the same b_i are calculated and bits $B[b_i]$ are examined. If any of the bits are 0, the entry does not exist; if all are 1, the record probably does exist.

The values for m and n are dependent on the number of records to be indexed and on the desired upper bound on the false positive rate; see [1] for details. If, as suggested there, we pick m and n so that approximately half the array is populated, the false positive rate is $.5^n$. But these parameters are not particularly important to this work, and will not be discussed further here. (Some considerations relating to the density of 1 bits are discussed in Appendix B.)

3 Queries with Encrypted Bloom Filters

3.1 Group Ciphers as Hash Functions

To produce an encrypted Bloom filter, we use an encryption algorithm for our hash functions. For example, we could define

$$H_i(W) = \{W\}_{k_i}$$

or

$$H_i(W) = \{W \| i\}_k$$

where $\{X\}_k$ denotes the encryption of plaintext X using key k. This would be expected to work well, as modern cryptosystems are designed to produce uniformly distributed pseudo-random output across the range of input space. In fact, a usual criterion (besides unpredictability – a sin qua non for encryption!), changing one input bit should change approximately half of the output bits.) A Bloom filter scheme using encryption for hash functions has been described by Goh [3].

But that requires distribution of the k_i to all parties, which does not meet one of our original goals. Instead, we use a specialized form of encryption function where operations can be done on encrypted data. In particular, we will employ a cipher that forms an Abelian group over its keys when encrypting any given element. More formally, we employ a cipher such that for all encryption input values W, the set of all keys k forms a group under the operation composition of encryption of W.

Groups are closed, so

$$\{\{X\}_k\}_j = \{X\}_{j \circ k}$$

for all j and k and some operator $\circ$. Such encryption algorithms are not common; indeed, for at least one purpose—enhancing security via iterated encryption—they are undesirable. But at least one such algorithm exists.

Suppose that Alice wishes to query Bob's Bloom filter (or Bloom filter collection) for some word W. Using her Bloom filter key k_A, she calculates $V_{W_A} = \{W\}_{k_A}$. She then sends V_{W_A} to a semi-trusted third party Ted. Ted does not know any keys k_i; however, for each pair i and j, he knows

$$r_{i,j} = k_j \circ k_i^{-1}$$

Note that k_i^{-1} must exist if the cipher is indeed a group.

Ted then uses $r_{A,B}$ to transform V_{W_A} into V_{W_B}:

$$\begin{aligned} \{W\}_{K_B} &= \{V_{W_A}\}_{r_{A,B}} \\ &= \{\{W\}_{k_A}\}_{r_{A,B}} \end{aligned}$$

and returns that to Alice. Alice then sends this value to Bob, and receives in return a bit vector with the answer. Thus, Alice can query Bob's database without disclosing the query and without knowing Bob's key.

3.2 Using Pohlig-Hellman Encryption for Bloom Filters

In Pohlig-Hellman encryption [4], we encrypt under key k by raising the message to the kth power modulo some large prime p:

$$\{X\}_k = X^k \bmod p.$$

The key k must be relatively prime to p; we achieve that by choosing p to be a prime of the form $2p' + 1$ where p' is also prime, and mandating that all keys be odd and not equal to p'. In addition, given that $x^{p-1} = 1 \bmod p$ (see, for example, [5]), we restrict keys to be less than $p - 1$, and do all exponent arithmetic modulo $(p - 1)$.

The decryption key d is chosen such that $kd \equiv 1 \bmod (p-1)$; d can be calculated efficiently by Euclid's Algorithm.

Pohlig-Hellman encryption is an Abelian group; a proof is sketched in Appendix A.

Suppose that we have $\{X\}_k$ and wish to produce $\{X\}_j$. Let $r = j \cdot k^{-1} \bmod (p-1)$, where k^{-1} is the is the decryption key corresponding to k, i.e., the multiplicative inverse of $k \bmod (p - 1)$. Then

$$\begin{aligned} \{\{X\}_k\}_r &= (X^k)^r \bmod p \\ &= X^{kr} \bmod p \\ &= X^{k \cdot j \cdot k^{-1}} \bmod p \\ &= X^{k \cdot k^{-1} \cdot j} \bmod p \\ &= (\{X\}_k)^{k^{-1} \cdot j} \bmod p \\ &= (\{\{X\}_k\}_{k^{-1}})^j \bmod p \\ &= X^j \bmod p \\ &= \{X\}_j \end{aligned}$$

Pohlig-Hellman encryption is expensive, since it requires exponentiation modulo a large prime p. But such encryption naturally produces a large output value. We can use that to generate our entire family of hash values. If B is m bits long, we use successive chunks of length $\lceil \log_2 m \rceil$ bits for our different hash values. For security, p will be at least 1024 bits long. If we want a false positive rate of less than $10^{-6} \approx 2^{-20}$—much higher than is commonly necessary—we need 20 hash functions, which means that our bit array can be 2^{50} bits long—far more than enough.

3.3 Encrypted Values and Hash Sets

A set of hash values b_i can be represented in two different ways: as the result of the Pohlig-Hellman encryption (which will will designate as PH form), in which case it is a single, large number, or as a set of Bloom filter indices. Both representations can be used; however, they are not equivalent.

As shown above, it is easy to transform an encryption value into a set of indices; the inverse transformation is not possible, because information, and in particular the ordering, is lost.

Note also that the values and the ordering of the different b_i for a given W will differ for different values of k. We illustrate this by a toy Pohlig-Hellman cipher using modulus 65267, and extracting 4-bit chunks as our indices.

If we encrypt 42 with $k = 537$, we get 19648, or $4CC0_{16}$, yielding an index set of $\{0, 4, 12\}$. (Since these values are indices, we've sorted the set and suppressed duplicates.) Encrypting 42 with a key of 17 yields 6362, or $18DA_{16}$, for a key set of $\{1, 8, 10, 13\}$.

Consider again the ciphertext $4CC0_{16}$. If we re-encrypt that with $k = 31$ — which is equivalent to encrypting 42 with $k = 537 \cdot 31 \bmod 65266 = 9129$ — we get $385B_{16}$. But if we change even a single bit of the input value — say, to $4CC1_{16}$ — and encrypt with $k = 31$, we get the very ciphertext different $F959_{16}$.

This is not surprising for an encryption algorithm, of course, but it underscores an important point: the individual segments of a PH number cannot be manipulated individually. PH values can only be manipulated arithmetically and as a whole, such as by transformation into the corresponding PH value for another key. When in set form, the usual set operations of union and intersection can be performed, but one cannot switch back to the arithmetic domain. As is discussed in the following section, we will use the different forms in different places.

4 Enhancing Security

4.1 Salts

The scheme as presented has a number of deficiencies. If nothing else, Bob knows k_B and can decrypt V_{W_B} to learn W. We can prevent that easily enough by first calculating $W' = G(W)$, where G is a cryptographic hash function. That has the added benefit of expanding the query word, thus turning it into a better base for the exponentiation.

Even with that, Bob can do a dictionary attack on W' to learn what Alice is looking for. This is especially easy for successful queries, since by definition Bob has already created entries for W' in his Bloom filter. We solve this by "salting" the query. We can do this in two ways, by modifying the Bloom filter-specific query or by including dummy words.

The first is superficially the most appealing. As before, Alice sends Ted an encrypted query, in PH form. Ted performs the transformation to Bob's key, and converts the transformed value to set form. This set is modified by deleting some entries, leaving n' valid ones, and add in some other set of random values. Ted knows which the random values are, and can ignore the response values for them. While deleting some valid indices will increase the false positive rate, if n is large enough that issue can be minimized.

The problem is that there is still information leakage. A successful query will match n' of the n bits belonging to the target word; if Bob has an inverted index of the Bloom filter, he can see what the query word is, because it will be the only one with a high hit count. Alice's defense is to ensure that some other word or words have similarly high hit counts.

If we use random values for our extra indices, we need to calculate how many extra indices must be used to ensure a reasonable probability of other words matching. Intuitively, it is reasonably clear that a signficant number of extras are necessary, since each word is associated with only n bits, and $n \ll m$. In fact, the scheme probably will not work in practice. A more detailed quantitative analysis, presented in Appendix B, shows why it fails.

The second technique is simpler: salt the query by selecting other, uninteresting words that are likely to be in Bob's database. The danger would be in correlations unknown to Alice; the dummy words may select the same documents as the target words. Furthermore, over a series of several searches, the dummy words should fit some recognizable pattern: given two query sets of *miniskirt, poodle, uranium, houndfish* and *plutonium, privacy, cotton, snake* it would be pretty clear what the topic of interest was.

polonium	0, 1, 2, 10, 13, 47
oralloy	10, 15, 16, 26, 35, 43
beryllium	4, 6, 10, 18, 18, 20
neutron	0, 2, 11, 25, 41, 43
Goldschmidt	1, 16, 19, 28, 42, 44
Kistiakowsky	4, 4, 10, 14, 36, 44
Meitner	12, 13, 22, 25, 27, 36
Szilard	11, 16, 33, 38, 43, 43

Figure 1: A sample Bloom filter of 48 elements; each search word has six bits set. The collisions within some search terms are an artifact of the small size of this filter.

Both problems are illustrated by the sample Bloom filters shown in Figure 1. A query for, say, "polonium" would generate a vector of 0, 1, 2, 10, 13, 47; if we used a query size of four elements, all four entries would, of course, select 1 bits in the filter.

Suppose, though, that we sent the random entries 8, 12, 17, 27, 30, 37, 42, and 47 as well. Looking at the inverse map (Figure 2), we see that "Meitner" has a hit count of two and "Goldschmidt" a count of one; "polonium", at four, would stand out as the real query.

On the other hand, a real use of this technique would be looking for certain documents in a large set; someone unaware of the history of fission weapon design [2] might not realize that most documents containing the word "polonium" would also contain the word "beryllium". Bob might not know which term was really of interest; it almost doesn't matter, since either might reveal the questioner's real intent. Still, this is the best option, especially because the connection between the two is not symmetric: in this particular example, there are likely to be many documents with the word "beryllium" but not "polonium".

The remaining question here is who should generate the dummy query terms. Ted cannot; he does not have a Pohlig-Hellman key that can be transformed to Bob's key. Bob cannot, since he could easily detect use of his own dummy words. That leaves Alice, which imposes a knowledge requirement on her: she needs to know enough about Bob's database to generate plausible dummy words.

0	polonium, neutron
1	polonium, Goldschmidt
2	polonium, neutron
4	beryllium, Kistiakowsky
6	beryllium
10	polonium, oralloy, beryllium, Kistiakowsky
11	neutron, Szilard
12	Meitner
13	polonium, Meitner
14	Kistiakowsky
15	oralloy
16	oralloy, Goldschmidt, Szilard
18	beryllium
19	Goldschmidt
20	beryllium
22	Meitner
25	neutron, Meitner
26	oralloy
27	Meitner
28	Goldschmidt
33	Szilard
35	oralloy
36	Kistiakowsky, Meitner
38	Szilard
41	neutron
42	Goldschmidt
43	oralloy, neutron, Szilard
44	Goldschmidt, Kistiakowsky

Figure 2: An inverse filter, showing the words that map to each bit.

Note that if Alice issues many queries, the dummy elements must be consistent each time. Put another way, telling a consistent set of lies is hard.

4.2 Warrant Servers and Censorship Sets

Under certain circumstances, it may be desirable to restrict the scope of some queries. For example, a police officer pursuing an investigation may be restricted to query terms listed in a warrant. Similarly, there may be some queries that Bob will answer for, say, Carol but not Alice. We can solve these problems with warrant servers and censorship sets.

A warrant server is another party to the dialog. T transforms all queries to the warrant server's key. The warrant server also needs to have some authoritative knowledge of the permissible terms, encrypted under its own Pohlig-Hellman key; there are many obvious ways to accomplish this, including digitally signed warrant messages and local copies of authoritative databases. Regardless, the warrant server deletes from the query all impermissible terms and sends the result back to T for transformation to B's key.

Note that the warrant server never sees the plaintext of any query terms. These are agreed upon offline, and are encrypted by the warrant authorizer (e.g., a judge). The warrant server performs its operations on the encrypted form of the query.

B's censorship set is now applied. Any terms that Bob will not permit Alice to query are now deleted. The resulting query is then salted and sent along to B.

Note that both warrant servers and censorship sets are specific to both the source and the destination of the query. Alice may be allowed to ask different questions of Bob than of Carol; similarly, Bob may be willing to disclose more to one than to the other.

4.3 Index Servers

Another approach to protecting queries is to use "index servers". Bob sends his Bloom filters to an index server; each document is tagged with an opaque name. As before, Alice sends her queries to Ted; Ted transforms them to Bob's key. Instead of being routed to Bob, though, they're sent to the index server. The index server performs the Bloom filter matches and sends back the document names. Alice (or Ted) can then ask Bob for these documents.

The advantage of this scheme is that Bob never sees the queries, and hence cannot perform any sort of guessing attack. The index server doesn't know Bob's key, and hence can't build a dictionary. Dummy queries may still be necessary if Alice wants to prevent Bob from even knowing the topic of the investigation.

5 Provisioning Pohlig-Hellman Encryption Transformation Keys

We must now consider how to store the necessary r values in T. The exact mechanisms will vary for different encryption algorithms; here, we take advantage of the mathematical tractability of Pohlig-Hellman encryption.

We have a set of queriers/publishers, Q. While not everyone will publish data and not everyone will query for data, both types need keys k_q; accordingly, we treat them the same way.

As noted, the relationship between the keys needs to be known. While this could be done by having all $q \in Q$ send their keys to the trusted party T, this would create a security risk if T were not fully trustworthy. A simple, but not altogether satisfactory, alternative is to have a second trusted party, T', which calculates the $r_{i,j}$ and sends the to T. If T' is not part of subsequent conversations between the q and T—this can be ensured by conventional cryptography—there is no ongoing risk. This scheme will work for all group ciphers. But the presence of many keys in one place is worrisome. Instead, we use a blinding mechanism.

To calculate the ratio $r_{b,a}$ between two keys k_a, k_b, $a, b \in Q$, both A and B set up a secure channel to T. They each generate random blinding factors F_a, F_b; additionally, T generates F_{ta} and F_{tb}, $1 \leq F_x < p-1$. The following messages are sent over pairwise encrypted channels, with all arithmetic being done modulo $(p-1)$. (For simplicity, we write a/b or $\frac{a}{b}$ to mean $a \cdot b^{-1}$ where b^{-1} is the inverse of b in the Abelian group of integers modulo $(p-1)$.)

$$\begin{aligned}
A &\rightarrow T : k_A \cdot F_a & (1)\\
B &\rightarrow T : k_B \cdot F_b & (2)\\
T &\rightarrow A : F_{ta} & (3)\\
T &\rightarrow B : F_{tb} & (4)\\
A &\rightarrow B : F_a \cdot F_{ta} & (5)\\
B &\rightarrow A : F_b \cdot F_{tb} & (6)\\
A &\rightarrow T : (F_a \cdot F_{ta})/(F_b \cdot F_{tb}) & (7)\\
B &\rightarrow T : (F_b \cdot F_{tb})/(F_a \cdot F_{ta}) & (8)
\end{aligned}$$

From messages 1 and 2, T can calculate $\frac{k_A \cdot F_a}{k_B \cdot F_b}$. From that and message 8, T can calculate

$$\begin{aligned}
&\frac{k_A \cdot F_a}{k_B \cdot F_b} \cdot \frac{F_b \cdot F_{tb}}{F_a \cdot F_{ta}}\\
= &\frac{k_A}{k_B} \cdot \frac{F_{tb}}{F_{ta}}
\end{aligned}$$

But T knows F_{ta} and F_{tb}, and can therefore calculate $r_{A,B} = K_A/K_B$. A similar calculation can be done using message 7; the results will match if A and B are honest.

We thus see that T never knows any party's encryption keys. But can they be recovered from the ratio values? Fortunately, that appears to be impossible, too.

Assume that we have three parties, Alice, Bob, and Carol, possessing keys K_A, K_B, and K_C. T therefore knows

$$\begin{aligned}
r_{A,B} &= K_A/K_B\\
r_{B,C} &= K_B/K_C\\
r_{C,A} &= K_C/K_A
\end{aligned}$$

We wish to solve for K_A in terms of the ratios.

Simplifying these equations, we get

$$\begin{aligned}
K_A &= K_B \cdot r_{A,B}\\
K_B &= K_C \cdot r_{B,C}\\
K_C &= K_A \cdot r_{C,A}
\end{aligned}$$

Substituting the second and third equations into the first, we get

$$K_A = ((K_A \cdot r_{C,A}) \cdot r_{B,C}) \cdot r_{A,B}$$

which yields the rather unsatisfactory insight that

$$1 = r_{C,A} \cdot r_{B,C} \cdot r_{A,B}$$

We are thus left with a situation where T can transform encrypted queries from one key to another, but cannot generate queries or decrypt them.

If some party D were to collude with T, T could read queries by transforming them to D's key. To defend against this, a querier A can blind messages to T by superencrypting with some nonce key N_A, and then decrypting the transformed query. Because Pohlig-Hellman encryption and decryption are commutative—the cipher is, as noted earlier, an Abelian group over the keys—the message can be successfully unblinded. Let $V' = (V)^{N_A} \bmod p$, where V is the query encrypted with A's key that is sent to T to be transformed to a query encrypted for B.

$$\begin{aligned}
(V')^{R_{A,B}} &= ((V)^{N_A})^{R_{A,B}} \bmod p\\
&= (V)^{N_A \cdot R_{A,B}} \bmod p\\
&= ((V)^{R_{A,B}})^{N_A} \bmod p\\
&= (\{V\}_{K_B})^{N_A} \bmod p
\end{aligned}$$

This value can be decrypted using the decryption key corresponding to N_A; $\{V\}_{K_B}$ can be used to generate a query to B as described earlier.

The remaining roles are the generation of the prime p and the certificate authority used for the initial setup with T. Neither of these is particularly critical. Any party can verify that p is prime, that it's of the form $2p' + 1$, and that it's long enough to protect against solutions to the discrete log problem modulo p.

The certificate authority is almost as simple. While A, B, and T want some assurance that they're talking to the right parties, the result of a failure does not leak any information about queries. The most likely result of an impersonation is a failed query, which is undesirable. Thus, some reliable CA should be used. But there is one further danger. In a real-world implementation of this scheme, a succesful query for a desired document would likely result in a request for retrieval of that document. But this is more or less inherent in the problem statement. While the documents could, presumably, be stored elsewhere in encrypted form, the problem of finding the key would remain. We thus reject this solution.

At first glance, the number of transformation keys that T must have appears to be a problem; it is, after all, quadratic in the number of parties. That may not matter too much — for realistic uses, the number of parties is likely to no more than a few thousand, and calculating and storing a few million keys is not a challenge with modern hardware. Still, if it comes an issue, there is a solution. T's role can be partition, with each T_i serving some set of parties. For each party A served by a T_A, there would be a transformation key r_{A,T_A}; a routing key r_{T_A,T_B} would link the trusted parties. A query sent from A to T_A would be transformed three times: to T_A's key, to T_B's key, and then to B's key.

To be sure, this does require that each trusted party have a Pohlig-Hellman key, which violates one of our design principles. But this key only needs to be retained long enough for it to engage in the provisioning dialog with the other T_i; after that, it can be discarded, thus preserving security.

The set of trusted parties does not need to be fully connected. Instead, they can be linked in any sort of network; standard routing techniques can be used to direct the query to the proper party.

6 System Design Considerations

6.1 Roles

We can now look at some systems-level issues. We begin by considering who the different parties are, and what their trust properties are.

First, of course, there are sets of queriers and information owners. Some parties may do both, but these roles are independent. While in some sense it does not matter if a single key is used for both sorts of operation, in general it's better to use separate keys. Thus, to the system they would appear to be two separate parties.

In fact, in a real it would be better to let each person who generates queries have an individual key. This permits finer-grained authorization and auditing. The catch is the n^2 nature of the provisioning process.

Information providers have a more complex problem. Except for very small agencies, building the index is problematic: it is undesirable for large numbers of sensitive documents to exist in one place at one time. There are two easy solutions. First, each group can be given the same Pohlig-Hellman key to use in generating its own bit array; the collection of bit arrays can then be treated as a document collection by the outside world's contact. Second, each group could have its own Pohlig-Hellman key, and send a set of PH-form values to the central contact point; it would use a specialized set of transformation keys to convert these values to a common Pohlig-Hellman key, and build the bit map from them. (This is, in effect, a special case of the distributed T role discussed earlier.)

The trust relationship between queriers and providers is complex. Clearly, they do not trust each other unreservedly. This lack of trust may be due to legal strictures, organizational "turf battles", or simply the need for compartmentalization of sensitive data. But within certain bounds, they are willing to share certain classes of information if suitable need is demonstrated. In other words, the details here are political, not technical; nevertheless, these details are likely to be the major driver of any actual implementation of this scheme.

A second role is that of the warrant server. This role can be split, as long as the intermediary has authoritative knowledge of which server handles requests from which queriers and for which providers. Note that T does not need a full set of transformation keys for the warrant servers; rather, it only needs keys to map requests from each querier and to each provider associated with that warrant server.

The most complex role is that of the trusted third party. While Ted never sees any queries or any data, he is the ultimate arbiter of who does get to see what. The set of transformation keys stored by Ted is the functional discriminator of what providers any given querier can reach; if no transformation key exists, no queries can be made. In the simplest design, Ted is also responsible for routing queries to the proper warrant servers, though there are clearly alternative topologies that would remove that responsibility: queriers could sent their requests to the warrant servers directly; they in turn would transmit the filtered requests to Ted.

In some designs, Ted must also send responses and even encrypted documents back to queriers. This imposes some bandwidth constraints; in some cases, one must trust Ted not to invent keys that it can use to decrypt documents; see Section 6.2 for details.

Index servers, if used, have some of the same properties as Ted. They never see any confidential information; however, they're responsible for routing requests accurately. They also need to be more trustworthy; a subverted index server could select documents that don't match Alice's queries, thus betraying Bob. On the other hand, they presumably don't know the actual names or contents of the documents they might betray.

6.2 Protecting Document Retrieval

The actual document retrieval can be a crucial feature of total system design: by seeing which documents are actually retrieved, Bob can learn soemthing of the query terms. Here we briefly sketch a retrieval protocol layered on top of our encrypted Bloom filter mechanism.

As we noted earlier, there are many possible ways to proceed at this point. The constraints will be both technical and policy-oriented. For example, the scheme we are about to outline specifies that all actual document retrieval be done via T. This would require that T have high-bandwidth links to all servers. If that were not the case, a different solution would be needed.

From a policy perspective, Bob may not wish to transmit documents to Ted, even in encrypted form. Instead, Alice may be allowed to present a request for a set of documents to a competent human arbiter; he or she would decide if they were relevant, and would then provide them to Alice only under carefully controlled conditions.

Again, there are endless possibilities. Here we describe one possible scheme, with a few variants thrown in for good measure.

We first describe the notion of *sealing*. A sealed message is one created by some party, and encrypted and authenticated in such a way that only that party can read or verify the message. Initialization vectors or random padding are used to prevent dictionary attacks on sealed messages. There are many obvious ways to do sealing; similar schemes are often used with Web cookies [6]. Sealing is used here as an optimization to permit stateless operation by servers; an obvious alternative is local caching of such messages.

Initially, Alice prepares a query list. The query list is a set of hashed, PH-encrypted search terms; each query is flagged as real or dummy. The list is sent to Ted (all messages in this protocol are pairwise encrypted), along with a newly-generated public key embedded in a certificate. Alice can either remember the corresponding private key or send along a sealed copy of it.

Ted sends the query list, including the flags, to the warrant server. The warrant server compares the query list with the warrant; for any unauthorized terms, the flag is set to "dummy". The warrant server also signs the certificate; Alice's name never appears in it, thus preserving her anonymity. The altered list is returned to Ted.

Next, Ted applies Bob's censorship constraints. Again, invalid queries are not deleted from the list; rather, their flags are reset. Ted prepares a sealed copy of this list, including the flags.

If random index padding is to be used, each entry in the query list is converted to set form, and a random subset of each set of indices is generated. Each index is paired with a pointer to the corresponding query list entry; this part is sealed. The final list of indices and pointers is randomly permuted; this list, along with the sealed version of the query list and Alice's public key, are sent to Bob. If random index padding is not used, Bob can get the list in PH form.

Bob now matches the set of indices against his document collection. Any document matched by more than some threshhold number of indices is flagged as eligible for retrieval. Bob sends back to Ted a list of these documents; each document is associated with the set of indices that selected it. Note that each such index is paired with Ted's sealed pointer. Bob's reply message is accompanied by a sealed copy of the query; this will later permit Bob to verify that a valid query was made for some documents.

Ted now filters Bob's results, according to its own list of what Alice is entitled to see. Note that this filtering can include an enforceable minimum number of hits on any word, to rule out false positives from the Bloom filter. We thus can enforce a quantitative notion of "probable cause".

Ted then asks Bob for those documents, as well as a few others to disguise the actual topic of interest. Bob encrypts these documents with Alice's public key, thus denying Ted any knowledge of what they actually contain. As final step, only the authorized documents are sent back to Alice.

Clearly, many other variants are possible for this phase. Ted could blend together several separate queriers' requests, each querier could send along many different public keys, etc.

The remaining issue is security against traffic analysis. Standard techniques, such as Mixnets [7], can be used as a defense.

We get a different set of tradeoffs if we use index servers. Exporting the filters should not pose a security risk, because of the cryptographic mechanisms used to generate them. On the other hand, there is the issue of greater trust in some outside party. The decision on using index servers must be based on the relative trust parties have in such a third party versus their confidence that information providers will not go to great lengths to ascertain the subject of queries.

6.3 Performance Issues

The performance of a system based on this design is limited by two factors: the speed of Pohlig-Hellman encryption, and the ability of a site to rapidly search many Bloom filters.

In general, encryption speed is not likely to be a major issue. There are off-the-shelf chips that can perform 25,000 modular exponentiations per second; T could be equipped with such hardware. Beyond that, T's role is easily replicated. No querier is likely to generate nearly that many queries per second; information providers do not need to do any Pohlig-Hellman operations.

Beyond that, there will be some overhead to set up secure pairwise connections. While this may not be a trivial issue, these connections can be amortized over many queries and responses. In addition, off-the-shelf Web SSL accelerators and load balancers can be used as needed.

A linear search of a large collection of Bloom filters—say, one per document—is likely to be more expensive. A better solution is to use hierarchical filters, where each one at a higher level is composed of the logical union of its child filters. There are obvious optimizations at this point, including letting each group of documents be hosted on a separate departmental search server.

There is a more subtle optimization that we can do if the queries arrive in PH form. A single PH-form query can be split into different size pieces, to accomodate different Bloom filter sizes. Thus, we can separate documents by size (more precisely, by the number of indexed search terms), and use different sizes of Bloom filters for each size range. This may achieve a considerable performance improvement; recall that Bloom filters give optimal performance for a 1's density of .5 [1], and small documents will not achieve that density. [1] also provides a performance analysis of Bloom filters, though the access patterns here are more complex than are considered in that paper.

Alice is likely to incur considerable expense generating many private/public key pairs for query retrieval. Each document retrieved may require a separate pair; at the least, each query would require one in some scenarios, as was discussed above.

7 Related Work

Song, Wagner, and Perrig described a scheme for searching for sequences of words in encrypted files [8]. But search time is linear in the size of the documents.

Boneh et al.'s Searchable Public Key Encryption [9] is a mechanism for tagging messages with a few keywords that can be searched for. However, it doesn't scale to searches over the entire document.

Goh's scheme [3] is the closest to ours, in that it employs Bloom filters with encryption used for the hash functions. However, it requires that all parties share all keys. The paper gives several elegant mechanisms for executing more powerful searches; most of those schemes apply to our method as well. In particular, he described using binary searches on collections of documents to speed up retrieval. He also described how to use Boolean combinations of terms in queries; while those will work for our scheme, they pose compliance checking problems for warrant servers. The suggestion of deleting some of the indices (Section 4.1), as well as the problems with that scheme, were also noted by Goh.

The encryption property we need was dubbed *universal re-encryption* by Golle et al. in [10]. A similar scheme was called *atomic proxy encryption* by Blaze, Bleumer, and Strauss [11]. Both of these use public key cryptosystems, rather than symmetric ones. While public key schemes would work for encrypted Bloom filters, we do not need the other properties of public key cryptography. (On the other hand, Pohlig-Hellman encryption is comparable in cost to many public key schemes.)

8 Conclusions

We have described a scheme for protected searches among mutually suspicious parties, without the need for a trusted intermediary. The current design uses Pohlig-Hellman encryption, which is rather expensive, but this is not a requirement. Most of the design and analysis would apply to any other cipher where the keys form an Abelian group. If we omit the blinding of queries, we can drop the requirement for commutativity.

In fact, we don't even need encryption for much of it; a keyed hash function that formed a group would suffice. But that would require that T know $r_{A,B}$ and $r_{B,A}$ for all pairs; with Pohlig-Hellman encryption, the two are multiplicative inverses. It is tempting to use RSA encryption with a common modulus as the hash function; that would permit use of efficient public exponents such as $2^{32}+1$. Unfortunately, that runs afoul of Simmons' attack [12, 13].

There is one aspect that is tightly tied to the arithmetic properties of Pohlig-Hellman encryption: the scheme for (and analysis of) provisioning T. This aspect would have to be rethought if a different cipher were to be used. We note again, though, that the provisioning role and the query transform role are separable.

There are a number of applications for this scheme beyond what we have presented here. One intriguing one is for use in discovery proceedings in civil lawsuits. During discovery proceedings, each party is entitled to some of the other side's documents, but only if they're demonstrably relevant. Our technology provides an efficient scheme for performing such searches.

Other applications are feasible if the retrieval enhancements from Section 6.2 are used. One is a secure peer-to-peer file-sharing network. By broadcasting a salted query to numerous servers, Ted can find who has a certain file — song? — without knowing what is being requested. Similarly, Bob, Carol, et al. do not know who is requesting things, nor even what is actually being requested.

PH-encrypted Bloom filters can also be used to implement part of Anderson's Eternity Service [14]. Anderson suggests that an index is necessary, but doesn't suggest how to provide one.

There is a large body of literature on secure or anonymous document sharing, such as Publius [15]; it is likely that most of those schemes could be integratd with our secure search mechanism. How to do that is not the focus of this work. We do note, though, that security is a total systems property; some such layer is likely a necessary component.

Acknowledgments

Jeff Lagarias performed the analysis of query index padding, and made other observations that form the core of Appendix B. Rebecca Bellovin corrected a number of errors in the equations.

References

[1] B.H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[2] Richard Rhodes, *The Making of the Atomic Bomb*, Simon & Schuster, Inc., 1987.

[3] Eu-Jin Goh, "Secure indexes for efficient searching on encrypted compressed data," Cryptology ePrint Archive, Report 2003/216, 2003, `http://eprint.iacr.org/2003/216/`.

[4] Stephen C. Pohlig and Martin Hellman, "An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance," *IEEE*

Transactions on Information Theory, vol. IT-24, pp. 106–110, 1978.

[5] Ivan Niven, Herbert S. Zuckerman, and Hugh L. Montgomery, *An Introduction to the Theory of Numbers*, John Wiley & Sons, 1991.

[6] D. Kristol and L. Montulli, "HTTP state management mechanism," RFC 2965, Internet Engineering Task Force, Oct. 2000.

[7] David L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[8] Dawn Song, David Wagner, and Adrian Perrig, "Practical techniques for searches on encrypted data," in *Proceedings of IEEE Symposium on Security and Privacy*, May 2000, pp. 44–45.

[9] "Searchable public key encryption," Cryptology ePrint Archive, Report 2003/195, 2003, `http://eprint.iacr.org/2003/195/`.

[10] P. Golle, M. Jakobsson, A. Juels, and P. Syverson, "Universal re-encryption for mixnets," 2002.

[11] Matt Blaze, G. Bleumer, and Martin Strauss, "Divertible protocols and atomic proxy cryptography," in *Proceedings of Eurocrypt '98*, 1998, Lecture Notes in Computer Science.

[12] Gustavus J. Simmons, "A "weak" privacy protocol using the RSA crypto algorithm," *Cryptologia*, vol. 7, no. 2, pp. 180–182, 1983.

[13] Steven M. Bellovin and Michael Merritt, "Augmented encrypted key exchange," in *Proceedings of the First ACM Conference on Computer and Communications Security*, Fairfax, VA, November 1993, pp. 244–250.

[14] R. Anderson, "The eternity service," in *Proceedings of Pragocrypt '96*, 1996.

[15] Aviel D. Rubin Marc Waldman and Lorrie Faith Cranor, "Publius: A robust, tamper-evident, censorship-resistant, web publishing system," in *Proc. 9th USENIX Security Symposium*, August 2000, pp. 59–72.

A Proof that Pohlig-Hellman Encryption is a Group

We sketch a proof that Pohlig-Hellman encryption is indeed a group, and in particular an Abelian group, for the operation of composition. We assume that the modulus p is a large prime of the form $2p' + 1$, where p' is also prime.

The requirements for a group are the existence of an identity element, the existence of an inverse for all set members, closure, and associativity.

Fairly obviously, the identity element is encryption with the key 1. The existence of inverses for all keys is shown in [4].

To show that the set is closed, we must show that encryption with any two keys yields another valid key. A Pohlig-Hellman key is an integer k relatively prime to $p-1$ and $1 \leq k \leq p-2$.

$$\begin{aligned} \{\{x\}_k\}_j &= (x^k)^j \bmod p \\ &= (x^{jk}) \bmod p \end{aligned}$$

We thus have closure if jk yields a suitable integer.

For j and k to be relatively prime to $p-1$, they must be odd. Since j and k are odd, their product is odd. Per [4], we reduce the product modulo $p-1$, an even number. The result of that operation is always odd, and by definition of modulus will yield a value less than $p-1$.

We must also show that jk is relatively prime to $p-1$. Since $p = 2p' + 1$, this reduces to showing that jk is relatively prime to $2p'$; since jk is odd, we merely need to show that jk is relatively prime to p'. Assume it isn't. By definition of a prime number, this implies that j or k is a multiple of p'. However, members of the set are all relatively prime to $2p'$, and hence to p'.

The remaining criterion is associativity.

$$\begin{aligned} \{\{x\}_{kj}\}_i &= \{(x^{kj} \bmod p)\}_i \\ &= (x^{kj})^i \bmod p \\ &= x^{kji} \bmod p \\ &= (x^k)^{ji} \bmod p \\ &= (\{x\}_k)^{ji} \bmod p \\ &= \{(\{x\}_k)\}_{ji} \end{aligned}$$

Finally, for encryption to be an Abelian group, we must show that it is commutative.

$$\begin{aligned} \{\{x\}_k\}_j &= \{x^k \bmod p\}_j \\ &= (x^k)^j \bmod p \\ &= x^{kj} \bmod p \end{aligned}$$

$$
\begin{aligned}
&= (x^j)^k \bmod p \\
&= \{x^j \bmod p\}_k \\
&= \{\{x\}_j\}_k
\end{aligned}
$$

B Disguising Search Terms via Partial Queries

As noted, it is tempting to try to disguise queries by converting them to set form, deleting some indices, and inserting some random values. Unfortunately, the approach does not work very well.

This is reasonably clear intuititively: the size of the bit array is quite large relative to the number of hash functions that would be used. This implies that for any search word, the density of "productive" bits is low. For a false positive, the random indices would have to hit a significant number of of these widely-scattered bits; this is improbable.

Jeff Lagarias of AT&T Labs Research has analyzed it in more detail, and more quantitatively. Most of the following discussion, and in particular the formula, is due to his insights and derivations.

First, remember that the ultimate goal of most queries is to find some particular *document* that matches the specified criteria. If the target document is short, its Bloom filter will be very sparse; accordingly, there will be very few words that can be matched, and virtually none that will be hit at all by random indices.

Even for the nominal 1's density of .5, the odds are low. Suppose we want to pad a query in set form with random indices. To achieve a 50% probability of hitting a single word, a lower bound on the number of pad entries we would need is

$$\frac{c}{n} \cdot m^{1-\frac{1}{h}} \cdot h \log h$$

where c is a constant between $2/3$ and 1 and h is the number of bits we think we need to hit.

It is clear from the equation that the fundamental problem is the relationship between the size of the filter and the number of hash functions used. If we fix h at 2, thus minimizing the $m^{1-\frac{1}{h}}$ term, the equation reduces to $c'/n \cdot \sqrt{m}$. While this is sublinear in m, even $\sqrt{m}$ is likely to be sufficiently larger than n that this scheme is impractical.

Thus, we cannot even ameliorate the problem by choosing an unrealistically small h. Apart from the fact that h has to be at least as great as the number of hits we need for legitimate queries, our total filter size would have to be extremely small. In other words, this scheme can only work if we are willing to tolerate a large number of false positives over a very small population of terms.

We summarize this in Table 1. Bear in mind that for false positives to be effective, there would need to be several for each query, thus increasing these values even further.

Table 1: The number of dummy indices for a 50% probability of a single false positive, as a function of the Bloom filter size (m) and the number of hits we need to be persuasive. All calculations were done assuming $n = 20$ and $c = 1$.

	m			
h	$10,000$	100,000	1,000,000	10,000,000
6	1,158	7,889	53,752	366,213
8	2,630	19,724	147,913	1,109,191
10	4,583	36,407	289,191	2,297,130
12	6,920	57,120	471,477	3,891,598