

Bristol Cryptography Blog

A blog for the cryptography group of the University of Bristol. To enable discussion on cryptography and other matters related to our research.

Friday, November 15, 2013

How to Search on Encrypted Data, in Practice

With the rise of cloud computing and storage in the last decade or so, many people have raised concerns about the security issues of outsourced data. And to match this, cryptographers have come up with many proposals aiming to solve all of these problems. The most well-known of these methods, Fully Homomorphic Encryption, can in theory offer a perfect solution to this, but is extremely inefficient and not currently practical for most real-world situations. Other solutions include Oblivious RAM and special forms of identity-based encryption, but these are both still pretty expensive.

Symmetric Searchable Encryption (SSE), which we looked at in this week's study group, stands at the other end of the scale. It is *extremely practical*, capable of handling extremely large (hundreds of GB) databases, and even has performance comparable to MySQL. However it does compromise somewhat on leakage. It is similar in some ways to deterministic encryption, but ensures that leakage only occurs when a query is made (rather than as soon as the database is uploaded), and has some clever techniques to minimise the quantity of this leakage.

Our study group focussed on the [Crypto 2013 paper](#) by Cash *et al.*, which also describes their state-of-the-art implementation. In this post I'll cover the basic concepts and mechanisms, and explain exactly what leaks and when.

For a basic overview of the many ways in which encrypted search can be done, I recommend checking out [Seny Kamara's blog](#), which currently has a [series](#) of articles running on this topic.

Scenario

A client holds a database (or collection of documents) and wishes to store this, encrypted, on a third party server. The client later wants to search the database for certain keywords and be sent an encrypted list of documents that contain these keywords. Note that we're not searching the entire contents of the database, only a set of keywords that have been indexed in advance (just as Google doesn't look in every HTML file on the web for your search terms, instead using precomputed data structures based on keywords). Also note that the server only returns a set of encrypted identifiers, rather than the documents themselves. In the real world the client would then request the documents from the server.

The remarkable aspect about the Crypto paper is that it aims to do the above in time proportional to the number of documents matching the *least frequent keyword* in the search query -- independent of the database size! Efficiency-wise, this seems optimal -- you can't do much better this, even on plaintext data. The compromise, of course, comes with leakage, which we'll look at shortly.

Warm-up: Single Keyword Search

Let's start with the simple scenario of searching for a single keyword.

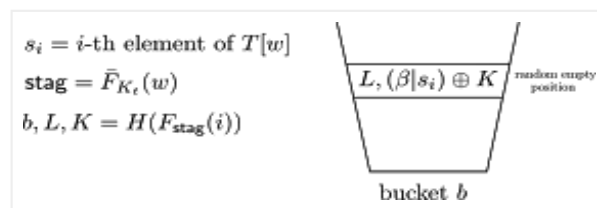
To enable search time proportional to the number of documents matching the keyword, the database first needs to be structured in a special way. It is arranged into a table T , so that for every keyword w , $T[w]$ contains a list of indices of all the documents containing w , encrypted under the key K_w , obtained by applying a PRF to w . That is:

$$T[w] = \{ \text{Enc}_{K_w}(\text{ind}) : \text{ind} \in \text{DB}(w) \}$$

where $\text{DB}(w)$ is the set of indices of documents containing w .

Now this is not yet secure. Firstly, the keywords will be leaked to the server as they are not encrypted. Secondly, even if the keywords were somehow hidden, the structure of T will reveal the *frequencies* of all keywords in the database, which the server could use to perform frequency analysis. The way around these problems is to store T in a special data structure that hides the association between keywords and documents.

The data structure, called a TSet, consists of a number of fixed size *buckets*. Each document index in $T[w]$ is randomly assigned to a bucket, by using a hash function H and PRFs F and \bar{F} applied to w and the index of $T[w]$, as illustrated below. A bit β is concatenated with the entry of $T[w]$ to signal when the last element of $T[w]$ is reached. This is all then XORed with a key (also derived from H), and added to a random empty position in the bucket, alongside a label L . L is also obtained by hashing, and is needed to identify the correct entry within a bucket. Modelling H as a random oracle ensures that the bucket and key are truly random, and using the key as a one-time pad hides all information about $T[w]$ from the server.



The bucket sizes need to be chosen to ensure that the probability of overflow isn't too high - as long as this probability is reasonably small, the setup can just be repeated with different PRF keys until everything works. This needs a storage overhead of around 2-3x.

Searching

After this setup is done by the client, they upload the TSet data structure to the server. Searching for a keyword w is done by:

1. Client sends the search tag $\text{stag} = F_{Kw}(w)$ to Server
2. Server sets $\beta = 1$, $i = 0$. While $\beta = 1$, do
 - Derive a bucket, label and key by hashing $F_{\text{stag}}(i)$
 - Search the bucket for an entry containing the label
 - XOR with the key to obtain β , s_i
 - Send s_i to the client, increment i
3. Client decrypts the indices s_i

Leakage

Let's stop a minute and think about how much leakage is going on here. The server can learn:

- The indices of all documents containing w (the *access pattern*)
- When repeated queries have been made (the *search pattern*)

Compare this with a simple solution using deterministic encryption, where the table T is stored with each w encrypted deterministically. In this case the server can straight away learn the keyword frequencies, before any queries have been made, whereas using the TSet as described above, this information is only gradually revealed with each query.

Conjunctive queries - naively

For the case of conjunctive queries, i.e. searches of the form w_1 AND w_2 AND w_3 , things are more complex. Naively, the client could perform each search separately, then compute the intersection themselves. However, this has two main drawbacks. For queries where one of the keywords is very common, e.g. "gender=male", this would require a large proportion of the database indices to be sent to the client. Moreover, the server still learns the encrypted indices of every document matching one of the keywords, which is more leakage than just the access pattern of the result.

Alternatively, you might try and have the server perform the intersection. The client could send the server the secret key K_w (derived from a PRF) for each keyword, but this would still leak all indices matching at least one keyword.

Conjunctive queries - less leakage

To get around these issues, they propose a modified protocol called Oblivious Cross-Tags (OXT), which allows the server to compute the intersection without learning (much) more information than the indices of documents matching the *entire* query. Impressively, they manage to do this in time proportional to the *least frequent* keyword in the query.

For this to work, another data structure, called an XSet, is needed. The XSet is essentially a set of all 'encrypted' pairs (w, ind) , where w is a keyword appearing in document ind . The TSet from the single-query search is also augmented with an 'encrypted' index for each entry. During search, the TSet is used for the first keyword, which gives the server the encrypted indices matching this keyword. The client also sends some tokens allowing the server to compute encryptions of (w, ind) for the rest of the keywords and the encrypted indices already recovered. These pairs are then searched for in the XSet (implemented as a bloom filter) and the resulting matches sent to the client.

The cleverness lies in the special way in which the XSet pairs and the extra indices in the TSet are encrypted. Their initial idea was to use an interactive protocol based on DDH to give the search tags for the correct terms and indices to the server, but this turns out to be insecure, and also needs an extra round of computation. By precomputing the first protocol message and making some other neat changes, they get around both of these issues. I'll leave you to figure out the details from the [paper](#) (pages 12-15).

To make the running time only dependent on the least frequent keyword, you have to ensure that this keyword is always searched with the TSet, and the XSet is used for all other keywords. By storing a small state with some basic keyword statistics, it's straightforward for the client to do this every time.

Leakage

Leakage-wise, this improves a lot on the naive methods. The only situation where the server might be able to learn more than desired is when the client sends two queries with *different* least-frequent keywords, but the *same* most frequent keywords. In this case, say when $(w1, w2)$ and then $(w1', w2)$ are queried, the server can compute the indices matching $(w1, w1')$. Also, note that in a conjunctive query of several terms, e.g. $(w1, w2, w3)$, the server can learn the number of documents matching $(w1, w2)$ and $(w1, w3)$ in addition to the number of matches for the entire query (this leakage doesn't seem to be mentioned in the paper, but came up in our discussion).

Conclusion

I think this technology is a really cool way of increasing the security of deterministic encryption, whilst maintaining plaintext-like search speeds, and it seems like a good alternative to things like [CryptDB](#), which leaves much to be desired regarding security. Even so, I would advise caution to anyone considering this kind of technology. We've recently been reminded of the perils of deterministic encryption (and several other bad security practices) with the leakage of [Adobe's](#)

[password database](#), so a careful assessment of the scenario is essential.

It's worth noting that in many cases the adversary is not the storage provider, but a 3rd party who may have obtained access to their servers. Using SSE ensures that nothing can be learned when seeing a snapshot of the database, and it would typically be much more difficult for an attacker to also gain access to the individual queries. For these scenarios it could really make a lot of sense to deploy SSE.

Clearly this is an area that needs more research to assess the suitability of specific applications. Also it would be really nice to see if other technologies like FHE and ORAM can be used to reduce the leakage, whilst still maintaining a reasonable level of efficiency.

Peter Scholl at 2:18 PM

Share

G+1

10

No comments:

[Post a Comment](#)

Links to this post

[Create a Link](#)









































[Home](#)



[View web version](#)

Contributors

-  [Tesleem Fagade](#)
-  [Joey Green](#)
-  [Elisabeth Oswald](#)
-  [Emmanuela](#)
-  [Guy Barwell](#)
-  [Jonathan O'Connell](#)
-  [Eduardo Soria-Vázquez](#)
-  [geo](#)
-  [ΑΠΑΝ](#)
-  [Gareth Davies](#)

-  Carolyn Whitnall
-  Jia Liu
-  Jake Longo Galea
-  Joop van de Pol
-  Christopher Sherfield
-  Bin Liu
-  Yan Yan
-  Emmanuela
-  Tim Wood
-  Theodoros Spyridopoulos
-  Dan Page
-  Anamaria Costache
-  Ryan Stanley
-  Srinivas Vivek
-  Dan Martin
-  Nigel Smart
-  Susan Thomson
-  David McCann
-  Luke Mather
-  Guillaume Scerri
-  Peter Scholl
-  Theo Tryfonas
-  David
-  Bogdan
-  Martijn Stam
-  Marcel Keller
-  Sergiu Bursuc
-  Marco Martinoli

Powered by [Blogger](#).