Searchable Encryption

Xaitheng Yang
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
yang3792@morris.umn.edu

ABSTRACT

This paper provides an overview of searchable encryption, including its uses, and relevant technical knowledge surrounding efficiency and security. These topics are explored within multiple different searchable encryption scheme implementations in order to identify differences.

Keywords

searchable encryption, forward privacy, backward privacy

1. INTRODUCTION

As cloud services become more prevalent in the modern world, the storage of data on off-site servers is becoming an inevitability. This cloud storage provides an ease of access and acts as a convenient form of disaster recovery, by providing a way to back up data. However, these benefits aren't without cost. The storage of information on third party servers opens up the potential for exposing this information to others. Those with malicious intent could abuse this exposed information. To avoid this, data is often encrypted. However, this causes difficulty for many services, as the need to retrieve or update this data becomes much more complicated with the requirement to decrypt data before it can be handled.

The specific demands of situations like these has created the necessity for new methods of security to protect data. Solutions that fill this need can be constructed with techniques such as oblivious RAM, which relies on preventing information leakage by hiding memory access patterns. However, these solutions are, in their current forms, impractical and slow [4].

A more practical method that meets these needs is searchable encryption. Searchable encryption is a class of structured encryption. It enables clients to perform keyword searches on encrypted documents while preserving the privacy of the database [4]. Regardless of the implementation, some amount of information is leaked by searchable encryption schemes, but efficiency is gained in exchange for this. This leakage has been due to the use of deterministic encryption which enables the server to easily find matches between encrypted tokens without expensive computation [1].

The leakage of these searchable encryption schemes can have dangerous consequences, as even a small leakage can be used to break the privacy of some schemes. Because of this, an emphasis on security definitions and new searchable encryption schemes is paramount.

UMM CSci Senior Seminar Conference, November 2018 Morris, MN.

2. BACKGROUND

2.1 Encryption Basics

For the purposes of this paper, encryption is the process of encoding information in such a way that only authorized users can access it. To all others who attempt to access the content, it is unintelligible.

Encryption is performed through the use of an encryption key. An encryption key is a string of bits created for encoding and decoding information. Depending on the type of encryption, this key can be used for only encoding, only decoding, or both functions. Data which is unencrypted shall be referred to as plaintext, and data which has been encrypted shall be referred to as ciphertext [5].

An important part of encryption is often randomness. For this paper, any reference to randomness can be assumed to be the result of a pseudo-random function. A pseudo random function is a polynomial-time computable function that is indistinguishable from a true random function by any probabilistic polynomial-time adversary. In other words, it is not random, but seems random to anything trying to comprehend its pattern at a certain level [4].

2.2 Databases

A database, for the purposes of this paper, can be seen as a structured set of data which holds documents, and indexes. The document is the data being stored in the database, and the index is an identifier for it. Queries are instructions that can be sent to a server containing a database. They can be used to add, retrieve, update, or delete data within the database, depending on the database's functionality. The source of the queries is called the *client* and the receiver of the queries is called the *server*.

2.3 Searchable Encryption

Searchable encryption is a class of structured encryption for search structures such as search indexes or search trees. It allows for the performance of queries on its encrypted data without having to decrypt the data. This is done through the use of keywords, which are assigned to indexes in a database. This is done prior to any searches on the database. Search queries are run on these keywords, which identify the documents to be operated on or retrieved. However, searchable encryption necessarily leaks some amount of information, as it trades security for functionality and efficiency [1].

Information leakage by searchable encryption schemes has been shown to allow for leakage abuse attacks, and even full plaintext recovery of encrypted databases. This has led to the need for searchable encryption schemes which do not leak pattern revealing query information. Such schemes, while more robust with regards to security, are less performance efficient than their information leaking counterparts.

2.3.1 Dynamic Schemes

A performance capability sought by some searchable encryption schemes is being dynamic. A searchable encryption scheme is said to be *dynamic* if an individual is able to add encrypted documents efficiently (without having to decrypt the database, then re-encrypt it) [3].

2.3.2 Forward Privacy

One notion of security sought by searchable encryption schemes, forward privacy, considers privacy of the database and update queries. More specifically, a searchable encryption scheme is said to be forward private if update queries to a server don't reveal information about the modifications they carry out [2].

2.3.3 Backward Privacy

In contrast, backward privacy considers the privacy of the database and updates to it during search queries. In other words, a searchable encryption scheme is backward private if it limits the information a server can learn about deleted data from further search queries on the database. [2].

Backward privacy under this definition can be broken into three types. In order of decreasing strength, depending on how much information is leaked due to inserted and deleted entries [2]:

I. Backward privacy with insertion pattern:
leaks the documents currently matching a keyword, when they were inserted, and the total number of updates on the keyword.

II. Backward privacy with update pattern:
leaks the documents currently matching a keyword, when they were inserted, and when all the updates on the keyword happened (but not their content).

III. Weak backward privacy:
leaks the documents currently matching a keyword, when they were inserted, when all the updates on the keyword happened, and which deletion update canceled which insertion update.

To demonstrate the differences between these notions of backward privacy, consider the following series of queries in order of arrival:

- add to index 1, “pepperoni and “pineapple”
- add to index 2, “pepperoni”
- delete from index 1, “pepperoni”
- add to index 3, “pineapple”

If we then consider a search query on “pepperoni after this series of queries (see table 1), we can observe what information is leaked depending on the type of backward privacy. In

Index	Keyword	Document
1	~~Pepperoni~~ Pineapple	Data 1
2	Pepperoni	Data 2
3	Pineapple	Data 3

Table 1: Backward Privacy Example Database

a scheme which fits the first notion of backward privacy, information about index 1 is leaked, revealing that it matches the given keyword and the time at which this entry was added. In addition, it is also revealed that three updates occurred for “pepperoni. A scheme which fits the second notion reveals everything the first did, as well as the time at which the three updates on “pepperoni occurred. The third notion reveals everything the first and second did, while also revealing that index 1 had “pepproni” removed from it at the time of the third query [2].

2.3.4 Adversaries

The notions of forward and backward privacy above shall primarily be considered against what is known as a persistent adversary. This type of adversary monitors the actions on a server from the beginning. The main example of this the server itself, which the database is stored on. This is why leakage prevention at all stages is crucial. However, there are other adversaries that may be considered, specifically, a late-persistent adversary, and a snapshot adversary.

A late-persistent adversary is one which is not able to monitor the actions of a server initially, but begins to at some point later on. At this point, it acts in the same way a persistent adversary would, continuously monitoring any actions performed.

A snapshot adversary is one which is only able to monitor the actions on a server at a specific moment or moments in time. Unlike the previous adversaries, it doesn't persist in its monitoring.

Due to their inherent limitations, these two adversaries are considered to be weaker than a persistent adversary, as they monitor the actions on a server less [2].

3. SEARCHABLE ENCRYPTION SCHEMES

In this section, we will consider multiple encryption schemes with different levels of privacy. While each of these schemes have mathematical proofs of correctness and security, these are out of the scope of this paper.

3.1 Dual Dictionary

The dual dictionary scheme was developed by Kim, et al. [4]. As its name suggests, it proposes a new data structure to handle indexes, called dual dictionary. This data structure allows for efficient updates. In addition to this, the scheme achieves forward privacy.

3.1.1 Dual Dictionary Data Structure

The dual dictionary data structure consists of linked dictionaries to represent both inverted and forward indexes. An inverted index maintains lists of document identifiers per keyword. A forward index, maintains keyword lists per document. By incorporating both in the dual dictionary data structure, the benefits of both are gained - the ability to search efficiently, and the ability to update efficiently [4].

Dic_1 - Forward Index

DL	SL
$document\_1\_DL_1$	$pepperoni\_SL_1$
$document\_1\_DL_2$	$pineapple\_SL_1$

Dic_2 - Inverted Index

SL	(DL, Index)
$pepperoni\_SL_1$	($document\_1\_DL_1$, 1)
$pineapple\_SL_1$	($document\_1\_DL_2$, 1)

Table 2: Example dictionaries

3.1.2 Dual Dictionary Construction

In order to see how the dual dictionary data structure works, consider a document containing keywords $w_1 ... w_t$. A t number of labels are generated for each keyword, which shall be referred to as delete labels. These generated labels shall be represented as DL_i, where i is the current keyword number. Each of these labels are generated with a secret key corresponding to the document index. The document index is known because the client keeps a count of the number of documents added to the database.

Each document added to the database has its own key that is kept by the client. For example, if a document was added to an empty database with the keywords *pepperoni* and *pineapple*, two delete labels would be generated for this document, DL_1 and DL_2. This would be done using a secret key specific to index 1 because it is the first document to be added. This separation by key allows for a document added to index 2 to have its own DL_1 and DL_2, which differ from the document in index 1.

A second set of labels are generated using a secret key corresponding to a specific keyword, as well as the number of documents matching that same keyword in the database. Each keyword has its own key that is kept by the client, and a count of the number of documents matching a keyword is kept. These labels shall be referred to as search labels, and represented as SL_j where j is the current document. So, with the same document in the previous example, we would generate SL_1 for *pepperoni*, and SL_1 for *pineapple*, which differ because they are generated with their own keys, but are both the first search label for each keyword.

Both of these labels are stored as a pair, (DL_i,SL_j) in a dictionary - Dic_1. This dictionary reflects a forward index. In the second dictionary, Dic_2, the labels are stored with the document index (SL_j,(DL_i, index)). This reflects an inverted index.

In order to retrieve documents containing a keyword, the search label is calculated for every document matching the keyword (from 1 to the number of documents), and Dic_2 is searched for these labels. So, in our example table, if we searched for *pineapple*, we would generate the search label for pineapple ($pineapple\_SL_1$). We would only generate one because there is only one document in the database with the keyword *pineapple*. We would then search Dic_2 for this label. The result of this would be the pair of delete label and index. The delete label isn't used, and the index is used to identify which document to retrieve.

To delete a document with a given index, DL_i is calculated for the index, and then Dic_1 is searched, then using the result, SL_j is used to search Dic_2, and the results are deleted from the database and dictionaries. So, in our example table, if we wished to delete the document at index 1, we would generate the delete labels for index 1. In this case there would be two because the document matches two keywords ($document\_1\_DL_1$, $document\_1\_DL_2$). We would then search Dic_1 for this delete label. The result of this would be two search labels corresponding to the keywords that match the document ($pepperoni\_SL_1$, $pineapple\_SL_1$). These would then be used to search Dic_2 for the corresponding delete labels and indexes. The results of each of these searches would then be deleted from Dic_1, Dic_2, and the document matching the index would also be deleted.

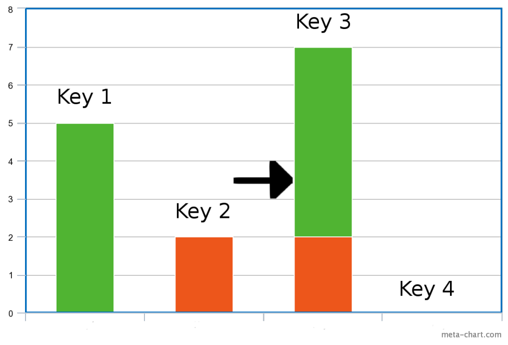

Figure 1: Dual Dictionary key usage before and after search

This set up is necessary for forward privacy because it hides which keywords correspond to which keyword document pairs. More specifically, the labels for a specific document can't be identified as being related because they are generated on the client side. However, after a search, the server does know which labels correlate with another. In order to achieve forward privacy, a fresh key replaces the current key after each query. The key which is replaced is based on what is used for a search. This new key is used generate the labels for newly added documents after the query, which makes old searches unusable.

To demonstrate this, consider a database which contains a set, A, of identifiers which match the keyword *pepperoni*, and are encrypted with key_1. After a search that includes *pepperoni*, a set, B, of new documents are added to the database that match the keyword *pepperoni*. The new identifiers for these documents are encrypted with a fresh key, key_2. After a search for *pepperoni*, both key_1 and key_2 are used to retrieve the documents, and then both sets A and B are encrypted with a new fresh key, key_3. At this point, the combined set of A and B can be treated as A was in the beginning of the scenario, as they are the set of previously searched entities. Any new additions to the database can then be treated as B was in the beginning of the scenario, and so on. As a result of this, key_1 and key_2 can't be used to search for results inserted after they are revealed, and may be discarded by the client (see figure 1) [4].

3.1.3 Dual Dictionary Considerations

While this construction allows for efficient search and updates, and explicit deletion, there is a cost. Specifically, the data structure requires about twice the amount of storage as a scheme that didn't implement both. In addition to this, the Dual Dictionary scheme by itself isn't backward private because when a document is deleted from the database, it is actually deleted, as opposed to marked as deleted or inaccessible. It also performs deletions in the same query that requests deletions. This causes the scheme to leak information on deleted documents after they are deleted.

3.2 Fides

Fides is a forward and backward private dynamic searchable encryption scheme developed by Bost et al. [2]. It is a combination of a forward private scheme, $\Sigma o\psi o\zeta$ (pronounced sohpos), and a technique for backward privacy, two-roundtrip.

3.2.1 $\Sigma o\psi o\zeta$

$\Sigma o\psi o\zeta$ is a dynamic searchable encryption scheme which ensures forward privacy developed by Bost et al. [1]. Its security is achieved through the use of tokens, called update and search tokens. For the purposes of this paper, a token can be seen as a unique identifier.

$\Sigma o\psi o\zeta$ operates by requiring client side storage of the keywords used and related search tokens. These search tokens are generated by using the number of documents already in the database that match the keyword, and a secret key. The number of documents for each keyword is kept track of by the client. Importantly, the tokens are generated using a one-way trapdoor permutation, which is a mathematical function which is easy to compute in one direction, but difficult to compute in the opposite direction without special information.

This one-way trapdoor permutation allows the server to compute all previous search tokens, given a search token for a specific keyword, but not compute any future search tokens. So, if there are three documents matching *pineapple*, then there are three relevant search tokens. If the server is given the most recent token, it can calculate the previous two, but it cannot calculate a fourth. In contrast, if the client was adding another document to the database matching *pineapple*, then the client would calculate the fourth token to keep as the most recent, and discard the third. If there are no documents in the database matching a keyword, a search token is randomly generated and paired with a keyword.

When a document is added to the database, its index is computed based on the keyword it matches, and the number of documents already in the database that match given the keyword. Then this index is paired with an update token generated using the current search token of the keyword. The document, and the pair are what is sent to the server.

Whenever a search query is performed, a search token corresponding with the keyword being searched for is sent to the server. This then allows the server to recompute all previous search tokens for the keyword. These search tokens are then used to find their paired update tokens. These update tokens are used to identify the document indexes, which then identify the documents to be retrieved [1]. So, returning to our previous example, if there were three documents matching the keyword *pineapple*, and a search query was sent for *pineapple*, the query would contain the most recent search token for *pineapple*. The server would then compute the previous two search tokens for *pineapple*. Now, with all three, the server would find the three update tokens that correspond to the search tokens. The document indexes that are paired with these update tokens are then used to identify which documents to retrieve.

This scheme achieves forward privacy through its use of tokens. This prevents the server from learning which keywords correspond to which keyword document pairs.

3.2.2 Two-Roundtrip

In a generic two-roundtrip scheme, what is stored on the server isn't a document index. Instead a ciphertext which includes an index and an operation (specifically addition, or deletion) is stored. This ciphertext is plaintext encrypted with a key specific to a given keyword. The server only sees the resulting ciphertexts as the keys are never revealed to it.

In this scheme, whenever a search query on a keyword is performed, a set of matching encrypted document indexes are returned. The client will then need to decrypt this set, and remove deleted indexes, in order to be left with the final set of indexes matching the keyword. If the client wished to delete a document, it would mark it as deleted at this point, but would not remove it from the set. The ciphertext and indexes can then be used for retrieval of documents and insertion of updated documents in a second roundtrip to the server - hence the name *two-roundtrip*. In addition to this, the client sends a ciphertext of the same indexes encrypted with a new key. The server replaces its previous ciphertext with this new one.

As described above, backward privacy with update pattern is achieved by the generic two-roundtrip scheme. This is because documents matching a keyword are leaked, as they correspond with the ciphertexts, and the time of insertions and updates are leaked. However, because operations are contained within the ciphertexts, and document indexes are re-encrypted after each search, the content of updates are not leaked [2].

3.2.3 Fides Construction

When $\Sigma o\psi o\zeta$ and two-roundtrip are put together, they function similarly to $\Sigma o\psi o\zeta$ in its first trip. The client sends the most recent search token. The server calculates all previous search tokens and uses them to find their corresponding update tokens. The difference occurs at this point. Instead of the server finding indexes, it finds ciphertext. Because it lacks the key, it must return the ciphertext to the client. The client will then need to decrypt the ciphertext for the relevant indexes and operations. The client would then encrypt the ciphertext with a new key. It would then send the new ciphertext and the indexes to the server for it to replace its old ciphertext and retrieve the relevant documents, as the two-roundtrip method specifies (see table 3) [2].

By combining the methods above, Fides achieves forward privacy, and backward privacy with update pattern. Its forward privacy is achieved through $\Sigma o\psi o\zeta$'s token system. Its backward privacy is achieved through the two-roundtrip methodology, which keeps the content of updates hidden within ciphertext.

3.2.4 Fides Considerations

This scheme as described has two main drawbacks. The first is that it requires the second round trip. In contrast it is common for searchable encryption schemes to return actual documents. The second drawback is that deleted elements are never actually deleted on the server side, only marked as deleted. This causes the communication cost, and the client side work to remain large even when documents are deleted. In order to mitigate the second drawback, a cleanup procedure can be sent along with the second trip. Specifically, when the second trip is made, the indexes marked as deleted

First Trip

Step	Client		Server
1	Search Token	$\rightarrow$	Calculate Search Tokens
2			Find Ciphertext with Corresponding Update Tokens
3	Decrypt Ciphertext	$\leftarrow$	Ciphertext

Second Trip

Step	Client		Server
4	Indexes and New Ciphertext	$\rightarrow$	Update Ciphertext
5		$\leftarrow$	Documents

Table 3: Fides Illustration

can be removed from the database. In addition to this, the new ciphertext that is sent to the server is made only for the non-deleted indexes [2].

3.3 Janus

Janus is a searchable encryption scheme developed by Bost et al. [2]. It uses puncturable encryption with incremental punctures to achieve weak backward privacy. In exchange for only having weak backward privacy, it has the benefit of not requiring client storage or multiple roundtrips.

3.3.1 Puncturable Encryption

A puncturable encryption scheme allows one to *puncture* the secret key to prevent the decryption of some messages. More specifically, the plaintexts are encrypted and attached to a *tag*. When the secret key is punctured, it is punctured on a set of tags, so decryption of those specific tags are impossible. This is done through the use of a puncture function, $Puncture(SK, tag)$ [2]. So, if a key SK_0 could decrypt ciphertext with the tags 1234 and 5678, one could puncture it so that it could no longer decrypt entities tagged 1234. To do this, the key would be modified to be SK_1 through the use of the puncture function, $Puncture(SK_0, 1234)$.

3.3.2 Incremental Puncture

A straightforward implementation of puncturable encryption cause punctured keys to grow with the number of punctures. To avoid this unlimited key growth, a method called incremental puncture can be used. To show how it works, one can imagine a secret key, SK, after n punctures as

$$SK_n = (sk_0, sk_1, ..., sk_n)$$

In other words, a key is a set of key parts. The incremental puncture method requires a puncture algorithm such that

$$Puncture(SK_n, tag) = (sk_0, sk_1, ..., sk_n, sk_{n+1})$$
$$\text{where } IncPuncture(sk_0, tag) = (sk_0, sk_{n+1}).$$

If a puncturable encryption scheme fits this definition, the client will only need to store the initial keypart, sk_0, of the key. This is because only sk_0 is needed to generate any future keypart using the $IncPuncture()$ function. This allows the server to store all but sk_0 safely, because without sk_0 the server cannot construct the full key. This is important to Janus' security because it ensures the server can't decrypt anything in the database until Janus allows it. Furthermore,

Addition Instance

Keyword	Encrypted Indexes
pepperoni	$e_{pepperoni}(1)$
pepperoni	$e_{pepperoni}(2)$
pineapple	$e_{pineapple}(2)$
sausage	$e_{sausage}(3)$

Deletion Instance

Keyword	Key Part
pepperoni	$sk_1^{pepperoni}$

Table 4:
Janus Database Example

puncturable encryption is how Janus deletes things from the database, as a document with a punctured tag will be rendered inaccessible [2].

3.3.3 Janus Construction

Janus can be constructed using any forward secure searchable encryption (e.g. $\Sigma o\varphi o\varsigma$, Dual Dictionary, etc.). It is constructed using two instances of the forward secure searchable encryption scheme. The first instance is used for additions, to store newly inserted indexes, encrypted with the puncturable encryption scheme. The second instance is used for deletion, storing the punctured key elements. A representation of the two instances can be seen in table 4 ($e_{keyword}(n)$ represents an encrypted index n with the key corresponding to $keyword$).

When inserting new entries, the client encrypts them with the puncturable encryption scheme. Each keyword has its own encryption key, and the client stores the initial key part of each key, sk_0, matched with the keyword. So, the client uses the key corresponding to the keyword used for the entry. This entry also receives a tag generated by a pseudo-random function based on the keyword and the index. This ciphertext is then inserted as a new entry matching the keyword into the addition instance. So, if an individual wanted to insert a document matching the keyword *chicken*, the plaintext index in the database would be encrypted with a key corresponding to *chicken*, and be tagged using the document index. The keyword, *chicken*, and the ciphertext would then be added to the insertion instance.

To delete an entry, the client computes the tag for the entry using the same pseudo-random function, and incrementally punctures the key corresponding to the relevant keyword. This is done using the $IncPuncture()$ function. The client then pushes the new keypart to the deletion instance. So, to delete the *chicken* entry in the previous example, the tag for the entry would be calculated. This would then be put through the function, $IncPuncture(sk_0^{chicken}, tag)$, to generate the key part $sk_1^{chicken}$. This key part is then added to the deletion instance along with the keyword.

When the client sends a search query, the associated sk_0 for the relevant keyword is included. Both instances are searched for the keyword, and as a result, the server obtains the encrypted indexes from the insertion instance, and all the corresponding keyparts from the deletion instance. The server will then be able to decrypt all the indexes that aren't punctured (i.e. not deleted). In other words, deleted documents aren't actually deleted from the server, just rendered inaccessable by puncturing the key.

As a consequence of this scheme, the server will learn the indexes of a given keyword, and its secret key. Therefore, any future insertions of a keyword after a search must be encrypted with a new key. However, as the server has already learned the indexes of a given keyword, there is no reason to re-encrypt them. So, to boost performance with no loss to security, Janus caches these indexes in order to increase storage locality.

Janus, by virtue of requiring forward private searchable encryption schemes, is forward private. Weak backward privacy is achieved because the server only has access to the decryption keys of a keyword during the search query for the keyword. In addition to this, the key only allows the server to decrypt the entries that have been added since the last search for the keyword, because the key is changed after every search. For instance, after a search for *chicken*, anything new added to the database with the keyword *chicken* would have its index encrypted with a new key. The old key can be discarded, as its results have already be cached. Because the keys are switched the deleted indexes remain hidden. However, the server is able to determine which inserted entries were later deleted, which is why weak backward privacy is the strongest level that can be achieved [2].

3.3.4 Janus Considerations

The previous considerations of forward and backward privacy focused on one type of adversary, a persistent adversary. However, the Janus scheme is vulnerable in ways the other schemes are not. This is due to the scheme's caching of search results. Because of this, information can be leaked to a snapshot or late-persistent adversary, as it would be able to see the cached results. In other words, without needing to monitor the databases activities from the beginning, these adversaries can acquire the same information that a persistent adversary would have. The result of this is that an adversary considered weaker than a persistent adversary could still lead to leakage-abuse attacks. In contrast a scheme like Fides would have stronger security against a snapshot or late-persistent adversary than it would against a persistent adversary. This is because these adversaries acquire less information than a persistent adversary would have [2].

This issue with Janus can be solved by encrypting the cache and storing it using history-independent data structures. Specifically, the cache would be encrypted with a key not maintained at the server, which is sent by the client. This would occur whenever the client sends a search query. The server would then decrypt the cache, and discard the result and key after the query was processed. In addition to this, history-independent data structures would be used to hide information regarding the size of current or discarded values, and the time of insertion/modification of data. This would prevent the cache from exposing information to late-persistent and snapshot adversaries, rendering them weaker than a persistent adversary [2].

4. RESULTS

While all the schemes are forward private and dynamic, Dual Dictionary isn't backward private. In contrast, it is the only scheme which easily supports explicit deletion. Fides requires a seperate clean up procedure to achieve this, and Janus doesn't support this at all.

When comparing levels of backward privacy, Fides reaches a high level, backward privacy with update pattern. However, in order to achieve this it requires two roundtrips, which significantly increases communication cost. In addition to this, the two-roundtrip method breaks with conventional expectations of a database, which normally return documents after a single search query. In contrast, Janus doesn't require two roundtrips, and performs conventionally, but is only weakly backward private. In addition to this, it requires additional fixes to be protected against weaker adversaries (see table 5).

SE Scheme	FP	BP	Other Considerations
Dual Dict.	✓	X	Two dictionaries takes twice the space
Fides	✓	With update pattern	Two roundtrips increases communication cost
Janus	✓	Weak	Vulnerable to weak adversaries

Table 5:
Summary of Schemes

5. CONCLUSIONS

There is still a lot of room for improvement in the security of searchable encryption schemes. Although there are schemes which achieve ideal levels of security, the performance costs are higher than ideal, as demonstrated by Fides. In contrast, schemes which attain ideal performance are often less than ideal with regards to security, as demonstrated by Janus. No perfect solution exists yet.

Acknowledgments

Thank you to Elena Machkasova, my advisor and senior seminar professor, for her expertise, advice, and feedback. I would also like to thank Shawn Seymour for providing feedback.

6. REFERENCES

[1] R. Bost. $\Sigma o\varphi o\varsigma$: Forward secure searchable encryption. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, pages 1143–1154, New York, NY, USA, 2016. ACM.

[2] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, pages 1465–1482, New York, NY, USA, 2017. ACM.

[3] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, pages 668–679. ACM, 2015.

[4] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, pages 1449–1463, New York, NY, USA, 2017. ACM.

[5] H. D. . H. Knebl. Introduction to cryptography: Principles and applications. Springer Berlin Heidelberg, Heidelberg, Germany, 2007.