

Experimental Evaluation of Forward Secure Dynamic Symmetric Searchable Encryption using the Searchitect Framework

Ines Kramer, Silvia Schmidt, Manuel Koschuch^a and Mathias Tausig

Competence Centre for IT Security, University of Applied Sciences FH Campus Wien, Vienna, Austria

Keywords: Searchable Encryption, Dynamic Symmetric Searchable Encryption, Implementation, Framework, Forward Privacy.

Abstract: In this work we present a prototype implementation of a framework for searchable encryption (SE), “Searchitect”. Our framework can be used to extend applications with search functionality over encrypted data in a protocol agnostic approach, hopefully paving the way for a broader and easier adoption of this promising privacy enhancing technology. Furthermore, it allows for easy comparison and evaluation of different implementations of SE schemes. We discuss dynamic searchable encryption schemes, supporting efficient updates of an encrypted index, as well as forward secure schemes that guarantee additional security properties, which resist file injection attacks. We evaluate the performance characteristics of two implementations of existing forward secure schemes, DynRH and Sophos. Our results show that the DynRH implementation is outperforming Sophos in terms of efficiency in the execution time of the search and update protocol, but needs more bandwidth for a search request. In addition, we augment an existing cloud-storage application with SE functionality using our framework, showing the negligible additional effort required by the implementers to accomplish this.

1 INTRODUCTION AND RELATED WORK

Symmetric searchable encryption (SE) is a privacy enhancing technology (PET) which provides a secure search functionality on outsourced encrypted data. The searched content stays oblivious to the server, even if it gets compromised. The emerging need for this kind of technology is based on the rising trend of the usage of cloud infrastructure in industries and home automation. Furthermore, the introduction of the general data protection regulation (GDPR) enforces privacy protection.

Since the seminal work of (Song et al., 2000) a vast number of schemes have been proposed compared to the rare number of reported usable implementations. Up to our knowledge only a few research projects on SE openly published source code such as (Kamara and Moataz, 2017; Popa, 2014; Popa et al., 2015; Bost, 2016a; Bost et al., 2017; Hoang et al., 2017; Hoang et al., 2018). The comprehensive survey on SE techniques by (Bösch et al., 2014) states that schemes differ in efficiency, performance and se-

curity levels. It might be insufficient to compare different approaches just on a theoretical basis. Choosing an appropriate and secure SE scheme for a use case without a practical comparison and realistic evaluation may be burdensome for a software developer without specific knowledge in this domain.

Our work tries to close this gap by providing a server/client framework called Searchitect, which

- Allows an easy integration of a searchable encryption scheme for testing purposes,
- Offers metrics for a comparison between schemes in performance and storage space, and
- Provides an interface for integration of SE technology in existing applications.

Further, we evaluate two proposed forward secure dynamic symmetric searchable encryption schemes (FS-DSSE). We also show how to integrate SE functionality in an existing data storage application. Our approach (Haböck et al., 2018) is hybrid in the sense that it does not matter how the original data is encrypted, but it has to be possible to map to the encrypted data. Therefore, the application can handle the data encryption and storage in its own way. The client needs to generate a searchable data structure - known as en-

^a  <https://orcid.org/0000-0001-8090-3784>

encrypted database (EDB) - with its own keys, which are concealed from the server.

Since our primary objective are efficient search queries we focus on symmetric searchable encryption schemes based on an inverted index. This approach was first proposed by (Curtmola et al., 2006), the difference between a forward and inverted plaintext index is shown in Table 1.

Table 1: Forward and inverted index.

Forward index	Inverted index
{doc1:{new,test,...}}	{new:{doc1,doc3}}
{doc2:{error,test,...}}	{test:{doc1,doc2}}
{doc3:{new,...}}	{error:{doc2}}

Inverted index based schemes provide a search performance that is linear to the number of documents containing the keyword, whereas forward index based scheme such as (Goh, 2004) just achieve a search time linear to the number of documents. Unfortunately, inverted encrypted indexes are more complex in handling updates.

Furthermore, we limit the scope of this work to dynamic schemes which support efficient updates. Due to the fact that data processed by applications is steadily growing and outsourced to the cloud over time, SE technology has to support addition and preferably also modification and deletion of existing data. File injection attacks such as presented in (Zhang et al., 2016) compromise the privacy of client queries by injecting a small portion of new primed documents into the encrypted database. These attacks break the security assumptions of most known dynamic schemes, only forward private schemes resist them. This property was first introduced by (Stefanov et al., 2013) and is also known as forward secure. Forward security ensures that newly added data remains hidden to the server until it gets revealed by a later query, even if the server might have learned some secrets during previous queries. Therefore, the Searchitect framework is shaped to integrate forward secure dynamic symmetric searchable encryption schemes.

The remainder of this paper is now structured as follows: Preliminary definitions of dynamic symmetric and forward secure searchable encryption are given in Section 2. The Searchitect framework is described in Section 3 and the implementation and integration of schemes is given in Section 4. Section 5 provides the experimental evaluation and results. Finally, we discuss our findings in Section 6.

2 DYNAMIC SYMMETRIC SEARCHABLE ENCRYPTION

Dynamic searchable encryption schemes provide an efficient and secure solution for adding and deleting data to and from an encrypted index structure.

For the definition of the protocols of dynamic SE schemes we follow a slightly modified approach of (Cash et al., 2014) and (Bost, 2016b) as shown in Figure 1 and Table 2. The database *DB* consists of keyword/document-identifier pairs also called records, which have been extracted from the documents using an indexer.

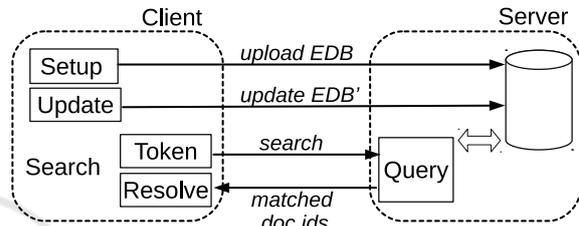


Figure 1: Protocols supported by a DSSE scheme.

The Resolve step in the Search protocol is optional and only needed for schemes which are resource hiding. They do not reveal the document identifiers to the server, thus they need to provide a decryption step.

2.1 Forward Secure Schemes

The first forward secure scheme was presented by Chang and Mitzenmacher in (Chang and Mitzenmacher, 2005); it is based on a forward index. A scheme with sub-linear search time was proposed in (Stefanov et al., 2013) using a path ORAM (Oblivious RAM) like construction, which lacks in performance. An efficient but not forward secure construction Dyn2Lev has been proposed by Cash et al in (Cash et al., 2014) using keyword dependent keys for encryption. We found a forward secure implementation DynRH2Lev within the Clusion library (Kamara and Moataz, 2017) published by Kamara and Moataz, where a query contains one search token for each keyword/document identifier pair in the result set. Sophos - published by Bost (Bost, 2016b) - uses asymmetric trapdoor permutations to make the update tokens unlinkable to the server. Another approach to achieve forward and backward privacy is based on range constrained cryptographic primitives (Bost et al., 2017). The forward secure scheme is achieved similar to the ARQ-EQ scheme of (Poddar et al., 2016). Puncturable encryption has been designed by Green and Miers to enable forward se-

Table 2: DSSE protocol definition.

Setup	$(1^\lambda, DB) \rightarrow (K, EDB)$ Setup runs at the client and expects a security parameter λ along with the DB as an input; consequently, it outputs randomly generated key material K along with the encrypted index EDB .
Search protocol between client and server	
<i>Token</i>	$(w, K, \sigma) \rightarrow ST$ Token runs at the client and takes a keyword w , the key material K and optional a client state σ as an input; consequently, it outputs a search token ST .
<i>Query</i>	$(ST, EDB) \rightarrow ids/encids$ Query runs at the server and takes the search token ST and the encrypted index EDB as an input; consequently it delivers all matching document-identifiers ids or encrypted ids : $encids$
<i>Resolve</i>	$(encids, K) \rightarrow ids$ The client decrypts the encrypted identifiers using the key material K .
Update	$(op, K, DB', \sigma) \rightarrow EDB'$ The client performs the Update procedure on the input of the desired operation (<i>add</i> or <i>delete</i>), the key material K , the new inserted database DB' , and the client state σ . Then it sends the output - an updated encrypted data-structure - to the server, where it is processed and results in an updated encrypted database EDB'

cure asynchronous messaging (M. D. Green and I. Miers, 2015). Using puncturable encryption a weak form of backward security can be achieved, but the computation effort for the pairings lacks in efficiency. A scheme proposed by Kim et al uses a dual dictionary, which has the advantages of both forward and inverted indexes at the same time (Kim et al., 2017). This enables efficient deletions with a continuous space reclamation after each search query, but has no backward security guarantees. The empirical evaluation of the search performance of this scheme compared to Sophos reveals that it lacks in performance and storage efficiency.

Another line of work called TWORAM and ODSE examines ORAM based schemes (Garg et al., 2016; Hoang et al., 2017; Hoang et al., 2018). Hoang, Yavuz, Durak, and Guajardo give several ORAM

and distributed PIR (Private Information Retrieval) based approaches which achieve forward and backward privacy. Instead of using a generic ORAM they leverage specific bandwidth-efficient oblivious access techniques such as read-only multi-server PIR for search and write-only ORAM for updates. Their experimental evaluation shows that this overcomes the limitations of prior ORAM schemes regarding communication costs.

To our knowledge there is still a lack of schemes providing efficient backward privacy, where the server can not learn any information from a deletion update with space reclamation on the EDB (Bost et al., 2017).

Unfortunately, all efficient forward secure schemes need to keep track of the state at the client. Thus, an additional synchronization effort for the client state is needed if an application allows the usage of multiple devices.

2.2 Selection of SE Schemes

We used following metrics to compare above schemes:

- Storage size at client and server
- Search & update time and communication complexity
- Parallelization of computation
- Efficient deletion handling

Taking into account efficiency of performance and storage size, we chose to compare DynRH and Sophos within our framework.

DynRH - The dynamic construction called Π_{Bas} in (Cash et al., 2014) is based on a simple dictionary, which contains label-value pairs. The label consists of a keyed hash value derived from a keyword specific key and a keyword counter, that stores the occurrence of the keyword per document. Whereas the value of a pair is calculated by the encryption of the document identifier with another keyword dependent key. In the forward secure version of the Clusion library (Kamara and Moataz, 2017) the document identifiers are encrypted twice, deterministic and probabilistic - each time with a different, keyword dependent key. Therefore, this scheme is resource hiding, because plaintext document identifiers remain hidden to the server. The forward private search is achieved by sending one search token for each expected match. The number of expected matches is indicated by a keyword specific counter stored within the client state. This forward secure scheme will be called DynRH in this paper, the pseudo-code can be found in Appendix 6.2

Sophos - Bost’s approach is based on a dictionary - similarly to DynRH - and our version keeps track of the keyword counters in a client state - similarly to DynRH (Bost, 2016b). The main difference is how the labels - also called update tokens UT_i - are derived and a Search token is issued. *Sophos* achieves forward privacy by keeping the update tokens unlinkable until a search query is issued. The scheme makes use of an extra layer of asymmetric encryption using trapdoor permutations π . The server is able to process a search query using the search token, the public key, and a keyword constrained key to re-calculate all previous update tokens of a specific keyword. However, the same parameters prevent the server from predicting the next update token without a secret key. Whereas the client is able to issue a searchtoken ST_c using the number of the keyword specific counter inverse trapdoor permutations. This scheme is resource revealing, because the document identifiers are encrypted just once; the encryption is done by applying an XOR operation with the search token. The drawbacks of this method are the computational effort for the asymmetric cryptographic primitives, and the sequential processing, which cannot be parallelized. We slightly modified the original scheme to increase the efficiency for batched updates at the client as shown in Appendix 6.2.

Comparing DynRH to *Sophos*, we noticed that both schemes are add-only schemes. They do not provide an efficient deletion handling and both need to keep track of a client state. The most remarkable differences are data privacy, used cryptographic primitives, and resulting performance. DynRH is resource hiding whereas *Sophos* is not. Furthermore, the former uses symmetric encryption contrary to the latter which uses asymmetric trapdoor permutations. Hence, DynRH requires more storage space and bandwidth while *Sophos* needs more computational resources.

3 SEARCHITECT FRAMEWORK

The client-server framework provides support for multiple users to upload and update their encrypted index and search for a keyword in this index. Furthermore, it offers some basic user account management and query authentication. It achieves the following non-functional requirements:

- Secure communication - authentication and encryption,
- Openness - easy integration of new schemes, no dependency on programming language,

- Easy to deploy and test,
- Dynamically updated documentation of the web-service API.

3.1 Client-side Searchitect Framework

The client Searchitect API is exposed to the primary application that enables it to perform all Searchitect related queries through this interface. The client library contains all generic classes such as the indexer, which is used to extract keyword/document-identifier pairs from documents resulting in a plain-text inverted index. Each SE scheme is integrated by its own client plugin. Hence, each plugin has to implement the same interface to the client library.

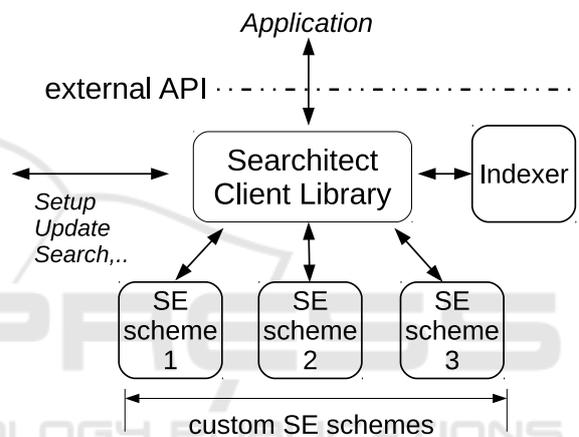


Figure 2: Client-side Searchitect framework, source: (Haböck et al., 2018).

3.2 Server-side Searchitect Framework

The server offers a web-service based on a micro-services architecture, which consists of a gateway and several SE backend modules. The external API exposed to the client is provided by a gateway, which accomplishes basic user-management tasks and query authentication. For these tasks the gateway holds a database with user accounts containing randomly generated identifiers of the EDBs and the related backend module. This enables the gateway to authorize access and forward the queries of SE protocols to the specific SE backend module. Each backend module will handle the storage of the EDB in its own flavor.

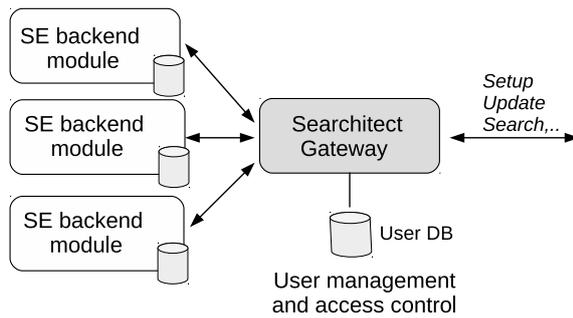


Figure 3: Server-side Searchitect framework, modified source: (Haböck et al., 2018).

4 IMPLEMENTATION AND INTEGRATION

Currently, all code is written in Java 10. However, the communication interfaces between client and server accept JSON formatted data through RESTful APIs; this offers interoperability and loosely coupled systems. Client queries are secured by TLS 1.2 in order to guarantee privacy. At the moment we use a username-password authentication and JSON web tokens (JWT) for the authorization of all subsequent client requests.

4.1 Client-side Implementation

The client has to process the biggest part of the workload. The indexer of the Clusion library supports keyword extraction from different file formats using specific libraries such as Apache PDFBox or Apache POI. This multi threaded process is using the Apache Lucene library and does some basic filtering of stop words. We had to adapt the indexing code to fit the specific application data to our test sets. A difference to the Clusion version is that our map to a document identifier should not be bigger than 20 bytes. We used a shortened path to the document. Each SE scheme has a separate client plugin that needs to implement the same interface method signatures in order to get integrated into the client library.

4.2 Server-side Implementation

Server-side code is built on the Spring Boot (version 2.06) template, which facilitates dependency management, configuration, and deployment using the embedded application server Apache Tomcat. For continuous compilation and building of all server instances we use Docker (Docker-compose version 1.18.0).

4.2.1 Searchitect Gateway

The gateway consists of three main controllers for:

- Authentication which issues a JWT authorization token
- User account management
- Repository controller which forwards the SE client queries.

Currently both, user management and access control database, are stored in memory using an H2 database. We added some security related classes to the Spring Security library that manage authentication without any external identity provider.

4.2.2 SE Backend Module

A Java dictionary object in memory was used for the first implementation of the SE backend module. Consequently, the search time performance slowed down. Therefore, we considered a fast key-value store as data structure which is able to persist the EDBs in the backend-module similar to (Bost, 2016a). The open source library RocksDB is designed for efficient random access in memory and SSD I/O operations (Team, 2018). RocksJava offers a Java API in an extra Java package. We use the HashSkipListMemTable, because the Java API does not provide an implementation of a Cuckoo hash table as for C++, which is optimized for point lookup. The HashSkipListMemTable contains a fixed-sized array of buckets and each bucket points to a skiplist, the default value is 1,000,000 buckets.

4.3 Integration of SE Schemes

We integrated two schemes, i.e. DynRH and Sophos. The original design is extensible with a new scheme by adding code for the client-plugin, a backendmodule and their common classes. Further the new backendmodule name and port need to be added to the application properties of the gateway.

4.3.1 DynRH

The Clusion library uses cryptographic primitives of the Bouncy Castle library. AES 128 in CTR mode was chosen for encryption and a keyed SHA256 hash for the pseudo-random function (PRF). All keys are derived from the keyword using the PRF. The document identifiers are encrypted twice, one time with deterministic encryption using a synthetic initialization vector (SIV) and another time with a random IV in the probabilistic encryption. We slightly modified

Table 3: Difference between DynRH and Sophos implementations.

Dynamic Variants	DynRH	Sophos
Storage	RocksDB + Client state	
Resource	Hiding	Revealing
Id Encryption	2 x AES_CTR	XOR $H_K(ST_i)$
Label Derivation	Keyed hash	Trapdoor permutation
Label/Value Sizes	24/52 byte	44/32 byte
FS-Search	Bandwidth	Calculation effort

the implementation of the Clusion library and limited the size of a document identifier to 20 bytes in order to reduce the storage space, which resulted in 36 bytes encrypted values.

4.3.2 Sophos

The Secure Computation API (SCAPI) is an open source library for secure two-party and multiparty computation (Ejgenberg et al., 2012). It also provides RSA trapdoor permutations, which we extended to support fast multiple inversions using the Chinese remainder theorem. For the PRF we make use of SHA256 and Blake2 of the Bouncy Castle library. Further, we fixed the maximal length of a document identifier and subsequently implemented a 24 bytes fixed length XOR operation. Our implementation of the Sophos scheme is entirely deterministic, because our design does not use any randomness (except once in the key derivation); i.e. the same key material and plaintext index (DB) result into exactly the same EDB.

Both backendmodules use RocksDB as persistent database. At the client we have the same size of client state, if both schemes process the same numbers of keywords. The main difference between the two implemented schemes lies in the encryption of document identifiers, the derivation of the labels, and how the forward secure search (FS-Search) is achieved as shown in Table 3.

Our implementation of the framework is open source and publicly available at <https://gitlab.com/Searchitect>.

4.4 Application Integration

Further we integrated the prototype of our client library into a secure cloud synchronization application as a proof of concept. This integration didn't require any fundamental changes of the underlying application and basically boiled down to merely extending the user interface with the search functionality as well as finding appropriate injection points for triggering

the update process. The prototype still lacks in handling multiple EDB repositories and efficient and secure key management while it enables an encrypted search over one vault.

5 EXPERIMENTAL EVALUATION AND RESULTS

Our tests focus on the feasibility of the searchable encryption technology in a working application. Therefore, all measurements except the EDB storage size are taken at the client machine. The communication between client and gateway is transport encrypted with TLS 1.2. Search and Update requests are sent to the gateway and after an authorization check forwarded to the appropriate backend module instance. For our experiments we used for the server an Intel Xeon E5-2620 with 8 cores, 64 GB RAM, and an SSD SATA3 running a Docker container in a virtual machine on Ubuntu 18.04 with 16 GB memory. The client hardware environment was a DELL Latitude E470 with an Intel(R) Core(TM) I5-6300U CPU running with 16 GB RAM and an ATA LITEON CV3-8D512 SSD hard drive running Ubuntu 18.04 (64 bit).

5.1 Testset Generation and Tests

We examined two main test cases, one with synthetic generated data and another with realistic data.

Synthetic - The synthetic generated testset includes 100 plaintext update indexes, where each update index contains the same 100 keywords with 100 newly inserted random order document maps. The test iterates over all 100 updates, the time used to process one update is measured at the client. After each update all 100 keywords are searched and an average search time is calculated and stored in a report.

Realistic - For the realistic testset we extracted the biggest archive from the publicly available ENRON email dataset (William W. Cohen, 2012) and grouped it into 282 updates each containing 100 documents.

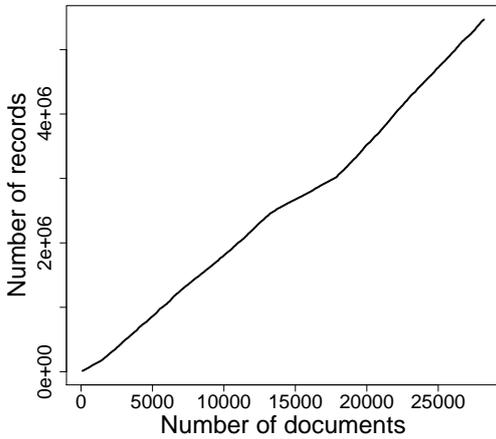


Figure 4: Growth of the database records with incremental updates containing 100 email documents.

In order to create the plain text inverted index, we needed to adapt the indexer of the client library to map document identifiers of the size of 20 bytes and filter out email tags of the keywords. The growth of the records in the DB with incremental updates is shown in Figure 4.

The mean time used for the index extraction process for an update of 100 documents was 106.74 ms. It took on average 0.0055 ms per record, but this value is strongly dependent on the data and occurrence of the keywords in the texts. After each index extraction of an update, an additional wordlist containing keywords with a small, medium, and big result set in the updated database has been created. The test loops over all updates and the update related wordlist is searched after each update and the running times are saved in a report.

5.2 Performance Evaluation

A record of the EDB consists of a label and encrypted document identifier. Table 4 gives an overview of our testing results for the EDB storage space and the running time for the update and search protocol on a per record base. The measured running time at the client includes also an averaged time for authentication and network traffic communication overhead.

5.2.1 Storage Size of the EDB

We logged the storage space usage of the EDB with incremental updates at the server. Although there is no difference between the 76 byte sized records for both schemes, RocksDB occupies in case of the DynRH implementation 5.99 % more space than in case of Sophos as shown in Figure 5. The size in bytes for one record differs between the synthetic and

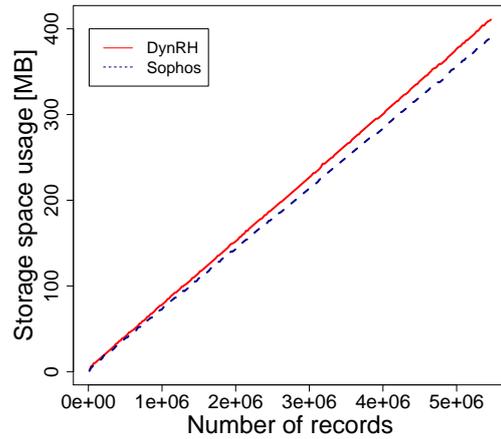


Figure 5: Storage space usage of RocksDB with continuous updates tested with the realistic test set.

realistic test set. This may be caused by the database management of RocksDB.

5.2.2 Update Time

The measured time for an update in the synthetic test set stayed nearly constant over time, except for two spikes in the 37th and 73rd update in DynRH and 36th and 72nd in Sophos which needed between 700 and 1500 ms longer. We can trace back this behavior to the use of the HashSkipListMemTable implementation of RocksDB. The performance difference of the accumulated update running time between DynRH and Sophos for the realistic testset is shown in Figure 6.

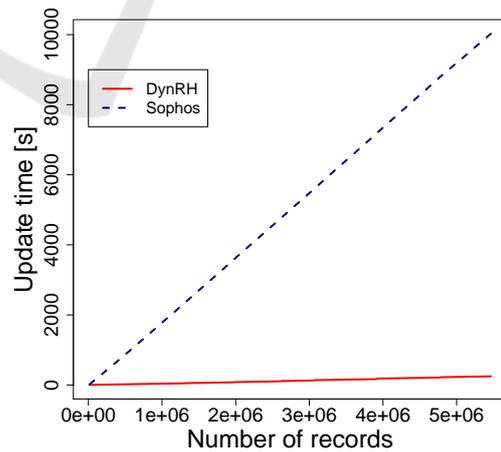


Figure 6: Accumulated difference in running time for incremental updates of DynRH implementation compared to Sophos one tested with the realistic test set.

An average update processed by the Sophos implementation takes about 41 times longer than with DynRH. This lack of performance in the Sophos im-

Table 4: Test results presented in storage size, update time and search time calculated as mean value over all measurements on a per record base.

	Synthetic		Realistic	
	DynRH	Sophos	DynRH	Sophos
Storage size/record [bytes]	85.45	78.61	79.58	75.08
Update time/record [ms]	0.03838	1.34523	0.04458	1.83433
Search time/record [ns]	1.10473	1.60002	21.13343	29.17961

plementation is caused by the expensive secret key computations of the asymmetric trapdoor permutation.

5.2.3 Search Time

In the synthetic test we compared the average search speed of the same 100 keywords after each update, the result is shown in Figure 7. We discovered the

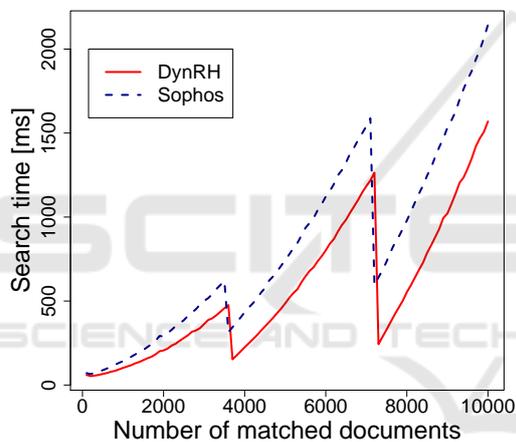


Figure 7: Running time of search queries with continuous increasing result matches tested with the synthetic generated test set.

two update time spikes. The restructuring of the HashSkipListMemTable in these updates improved the search performance. Tested with the synthetic testset the search protocol in the DynRH implementation is 30.96% faster than the Sophos one.

Realistic Testset: The running time for 13,305 search queries in relation to the number of matching document identifiers is shown in Figure 8. Just a few keywords are contained in each update and the restructuring of the database management of RocksDB is obvious. We took an average over all search queries of the realistic testset and compared the DynRH implementation to Sophos. DynRH is 27.57% more efficient than Sophos tested with the realistic dataset.

Further, we extracted test results for search queries with one matched document and had a look if the search time stays constant over time with an in-

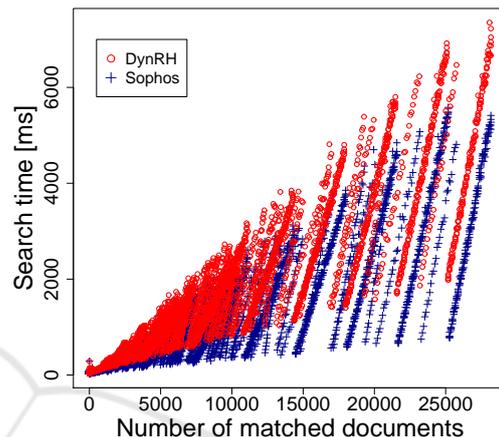


Figure 8: Running time of DynRH and Sophos search queries with continuous increasing result matches tested with the realistic test set.

creasing encrypted database. This extraction is shown in Figure 9, the search time in both schemes keeps relatively constant. Remarkably, there seems to be an optimal size for search queries in the middle.

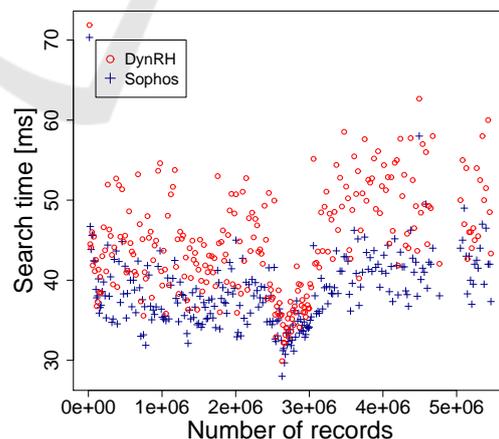


Figure 9: Running time of DynRH and Sophos search queries with continuous increasing result matches tested with the realistic test set.

5.3 Comparison of the Results

There are several points which make it difficult to compare our outcomes to other reported test results such as in (Bost, 2016b; Kim et al., 2017; Cash et al., 2014).

- The used coding language differs - we used Java, which lacks in performance compared to C++.
- Our tests focus on feasibility in the usage of SE technologies in an application, whereas others focus on performance of the raw algorithms without communication overhead of authentication and/or transport encryption.
- The testing environment of this work differs compared to (Cash et al., 2014).

6 CONCLUSION

The Searchitect framework presents our approach to integrate searchable encryption technology in real world applications. Further, it provides a stable environment to compare implementations of different SE schemes. We examined the performance difference of two integrated forward secure implementations of the DynRH scheme and Sophos. Due to the lack of efficient backward secure schemes, both are add-only schemes. Forward secure schemes can be extended by a basic deletion handling by using the first byte indicating if the document identifier is added or deleted. If the performance of the client is a key factor the DynRH implementation is preferred over the Sophos one. The search queries of DynRH will cost significantly more bandwidth. A work around could be using the keyword counters in the client state to predict the significance of the keyword in advance to avoid costly requests.

For our future evaluations we plan to replace the HashSkipListMemTable with a Cuckoo hash table to determine its influence on the system performance. In addition we want to examine the impact of switching to a native C(++) implementation as well as add new backward secure schemes (i.e. schemes supporting document deletion with space reclamation).

REFERENCES

- Bösch, C., Hartel, P., Jonker, W., and Peter, A. (2014). A Survey of Provably Secure Searchable Encryption. *ACM Computing Surveys*, 47(2):1–51.
- Bost, R. (2016a). OpenSSE schemes. <https://gitlab.com/sse/sophos>. [accessed on 2018-04-15].
- Bost, R. (2016b). Sophos - Forward Secure Searchable Encryption. Published: Cryptology ePrint Archive, Report 2016/728.
- Bost, R., Minaud, B., and Ohrimenko, O. (2017). Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. Published: Cryptology ePrint Archive, Report 2017/805.
- Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., and Steiner, M. (2014). Dynamic searchable encryption in very-large databases: Data structures and implementation. In *In Network and Distributed System Security Symposium (NDSS14)*.
- Chang, Y.-c. and Mitzenmacher, M. (2005). Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS 2005*, volume 3531 of *LNCS*, pages 442–455. Springer Berlin Heidelberg.
- Curtmola, R., Garay, J., Kamara, S., and Ostrovsky, R. (2006). Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *Computer and Communication Security CCS'06*, pages 79–88. ACM.
- Ejgenberg, Y., Farbstein, M., Levy, M., and Lindell, Y. (2012). Scapi: The secure computation application programming interface. <https://cyber.biu.ac.il/scapi/>.
- Garg, S., Mohassel, P., and Papamanthou, C. (2016). TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption. In Robshaw, M. and Katz, J., editors, *Advances in Cryptology – CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14–18, 2016, Proceedings, Part III*, pages 563–592. Springer Berlin Heidelberg, Berlin, Heidelberg. DOI: 10.1007/978-3-662-53015-3_20.
- Goh, E.-J. (2004). Secure Indexes. *IACR Cryptology ePrint Archive*, 2003:216:1–18.
- Haböck, U., Koschuch, M., Kramer, I., Schmidt, S., and Tausig, M. (2018). Searchitect - a developer framework for hybrid searchable encryption (position paper).
- Hoang, T., Yavuz, A. A., Durak, B. F., and Guajardo, J. (2017). Oblivious Dynamic Searchable Encryption via Distributed PIR and ORAM. Published: Cryptology ePrint Archive, Report 2017/1158.
- Hoang, T., Yavuz, A. A., Durak, F. B., and Guajardo, J. (2018). Oblivious Dynamic Searchable Encryption on Distributed Cloud Systems. In Kerschbaum, F. and Paraboschi, S., editors, *Data and Applications Security and Privacy XXXII*, Lecture Notes in Computer Science, pages 113–130. Springer International Publishing.
- Kamara, S. and Moataz, T. (2017). Encrypted systems lab: The Clusion library. <https://github.com/encryptedsystems/Clusion>. [accessed on 2017-09-05].
- Kim, K. S., Kim, M., Lee, D., Park, J. H., and Kim, W.-H. (2017). Forward Secure Dynamic Searchable Symmetric Encryption with Efficient Updates. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1449–1463, Dallas, Texas, USA. ACM.

- M. D. Green and I. Miers (2015). Forward Secure Asynchronous Messaging from Puncturable Encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320.
- Poddar, R., Boelter, T., and Popa, R. A. (2016). Arx: A Strongly Encrypted Database System. Published: Cryptology ePrint Archive, Report 2016/591.
- Popa, R. A. (2014). Cryptdb. "https://css.csail.mit.edu/cryptdb/". [accessed on 2017-09-09].
- Popa, R. A., Stark, E., Helfer, J., Valdez, S., Zeldovich, N., Kaashoek, M. F., and Balakrishnan, H. (2015). Mylar. https://github.com/strikeout/mylar. [accessed on 2017-09-09].
- Song, D. X., Wagner, D., and Perrig, A. (2000). Practical techniques for searches on encrypted data. In *S&P 2000*, pages 44–55. IEEE.
- Stefanov, E., Papamanthou, C., Shi, E., and Encryption, S. (2013). Practical Dynamic Searchable Encryption with Small Leakage.
- Team, F. D. E. (2013-2018). Rocksdb. https://rocksdb.org/. [accessed on 2017-07-07].
- William W. Cohen, MLD, C. (2012). Enron email dataset, may 7, 2015 version of dataset. https://www.cs.cmu.edu/~.enron/.
- Zhang, Y., Katz, J., and Papamanthou, C. (2016). All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. *IACR Cryptology ePrint Archive*, 2016:172.

APPENDIX

6.1 Implemented DynRH Scheme

Algorithm 1 shows a version of the Π_{Bas} scheme of (Cash et al., 2014) found in the Clusion library (Kamara and Moataz, 2017). For brevity we omitted the Setup algorithm, where we used a modified version of DynRH2Lev of the Clusion library. F represents a pseudo random function (PRF) and Enc stands for a symmetric encryption scheme.

6.2 Implemented Sophos Scheme

Our implementation presents a slightly modified version of Sophos scheme (Bost, 2016b) supporting batched updates, this is shown in Algorithm 2. F, H_1, H_2 are PRF and the trapdoor permutation is symbolized by π , where π^{-c} is the c th iteration of the inversion of the trapdoor function.

Algorithm 1: DynRH Source: derived from (Cash et al., 2014; Kamara and Moataz, 2017) and modified.

```

UPDATE( $id, DB, K, \sigma$ ): ▷ Runs at client
 $K^+ \leftarrow F(K, 3)$ 
for  $w \in DB$  do
   $K_1^+ \leftarrow F(K^+, 1 \parallel w); K_2^+ \leftarrow F(K^+, 2 \parallel w);$ 
   $K_3 \leftarrow F(K, 4 \parallel w); SIV \leftarrow F(K, 5);$  ▷ Derive
 $K_3$  and ( $SIV$ )
   $encid \leftarrow \text{Enc}(K_3, SIV, id, size);$  ▷ Deterministic
e.
   $c \leftarrow \text{Get}(\sigma, w);$  ▷ Get client keyword counter
  if  $c = \perp$  then  $c \leftarrow 0$ 
  for  $id \in DB(w)$  do
     $l \leftarrow F(K_1^+, c); d \leftarrow \text{Enc}(K_2^+, encid);$  ▷
    Probabilistic encryption
     $c = c + 1;$  ▷ Increment counter
    Add( $l, d$ ) to List  $L$ 
    Replace( $\sigma(w, c)$ ) ▷ Update client state
  send  $L$  to the server
SEARCH( $K, w$ ):
procedure TOKEN( $K, w, \sigma$ ) ▷ Runs at client
   $K^+ \leftarrow F(K, 3); K_1^+ \leftarrow F(K^+, 1 \parallel w);$ 
   $K_2^+ \leftarrow F(K^+, 2 \parallel w);$ 
   $c \leftarrow \text{Get}(\sigma, w)$ 
  for  $i = 0$  to  $c$  do
     $ST_i \leftarrow \text{Get}(\gamma^+, F(K_1^+, i));$ 
    Send ( $K_2^+, ST_1, \dots, ST_c$ ) to server
procedure QUERY( $(K_2^+, ST_1, \dots, ST_c, \gamma^+)$ ) ▷ Runs at
server
  for  $ST_i \in ST_1, \dots, ST_c$  do
     $d \leftarrow \text{Get}(\gamma^+, ST_i); encid \leftarrow \text{Dec}(K_2^+, d)$ 
    add  $encid$ 's to List  $encidList$ 
procedure RESOLVE( $encidList, w, K$ ) ▷ Runs at
client
   $K_3 \leftarrow F(K, 4 \parallel w); SIV \leftarrow F(K, 5);$ 
  for  $encid \in encidList$  do
     $id \leftarrow \text{Dec}(K_3, SIV, id);$  Add  $id$  to  $idList$  ▷
  Decrypt the  $ids$ 
  return  $idList$ 

```

Algorithm 2: Sophos forward secure scheme - modified implemented version. Source: derived from (Bost, 2016b).

SETUP(): ▷ Runs at client
 $K_s \xleftarrow{\%} \{0, 1\}^\lambda$;
 $(sk, pk) \leftarrow \text{KeyGen}(1)^\lambda$;
 $\sigma, \gamma \leftarrow \text{empty map}$
return $((\gamma, pk), (K_s, sk), \sigma)$

UPDATE($\{\text{add}, DB, \sigma\}$): ▷ Runs at client
for $w \in DB$ **do**
 $K_1 \leftarrow F(K_s, 1 \parallel w)$; $K_2 \leftarrow F(K_s, 2 \parallel w)$; ▷
 Derive keys
 $ST_0 \leftarrow \text{Map}(K_2)$; ▷ Map K_2 to a group element
 $c \leftarrow \text{Get}(\sigma, w)$ ▷ Derive counter c from state
if $c = \perp$ **then** $c \leftarrow 0$; $ST_c \leftarrow ST_0$
 $\text{Add}(\sigma, (w, c))$
else
 $ST_c \leftarrow \pi^{-c}(sk, ST_0)$; ▷ Get search token ST_c
 by multiple trapdoor inversion
for $id \in DB(w)$ **do**
 $ST_c \leftarrow \pi^{-1}(sk, ST_c)$; $c \leftarrow c + 1$; ▷ Derive
 next ST_c and increment counter c
 $UT_c \leftarrow H_1(K_1, ST_c)$
 $d \leftarrow id \oplus H_2(K_1, ST_c)$ ▷ XOR encryption
 $\text{Add}(UT_c, d)$ to List L ▷ Insert in random
 order
 $\text{Replace}(\sigma, (w, c))$
 Send L to the server ▷ Server adds L to γ

SEARCH(K, σ, w):
procedure **TOKEN**(K_s, σ, w) ▷ Runs at client
 $K_1 \leftarrow F(K_s, w)$; $K_2 \leftarrow F(K_s, 2 \parallel w)$
 $c \leftarrow \text{Get}(\sigma, w)$; ▷ $c = |DB(w)|$
if $c = \perp$ **then**
return 0
 $ST_0 \leftarrow \text{Map}(K_2)$; $ST_c \leftarrow \pi^{-c}(sk, ST_0)$; ▷
 Calculate search token
 send (K_1, ST_c, c) to the server.

procedure **QUERY**(K_1, ST_c, c, γ) ▷ Runs at server
for $i = 0$ to c **do**
 $UT_i \leftarrow H_1(K_1, ST_i)$; $d \leftarrow \text{Get}(\gamma, UT_i)$;
 $id \leftarrow d \oplus H_2(K_1, ST_i)$
 Add id to $idList$
 $ST_{i-1} \leftarrow \pi(pk, ST_i)$; $i = i + 1$; ▷ Calculate
 trapdoor function
 Send $idList$ to Client
