Performance Analysis of Searchable Symmetric Encryption Schemes on Mobile Devices

Dennis Y. W. Liu[1], Chi Tsiu Tong[1], and Winnie W. M. Lam[2]

[1] Department of Computing, The Hong Kong Polytechnic University
[2] Department of Mathematics and Information Technology, The Education University of Hong Kong
csdennis@comp.polyu.edu.hk, tony.ct.tong@connect.polyu.hk, winnielam@eduhk.hk

Abstract. In the age of cloud computing, it is common for individuals to store their digital documents to the cloud so that they can access them anytime and anywhere. While the demand of cloud storage continues to grow, the associated security threat has caught attention of the public. Searchable encryption (SE) attempts to ensure confidentiality of data in public storage while offering searching capability to the end users. Previous studies on SE focused on the security and performance on desktop applications and there were no discussions of those on mobile devices, where their memory, computational and network capabilities are limited. In this paper, we implemented three recent SE schemes and evaluated their performance in Android mobile devices in terms of indexing time and searching time, under 1) exact-match, 2) partial search, and 3) multi-keyword queries. We realize that each of the schemes has its performance advantages and disadvantages. Since user experience is one of the major concerns in mobile App, our findings would help App developers to choose the appropriate SE schemes in their applications.

Keywords: Searchable encryption schemes, Android devices, Encryption scheme performance

1 Introduction

These days, it is common for mobile device users to outsource their data to cloud storage providers (CSP). By utilizing the managed service provided by CSPs, data owners can minimize the cost and efforts of hosting storage servers and accessing their data from virtually everywhere. Despite the fact that the storage outsourcing approach sounds attractive, there are increasing number of people worrying about data security and user privacy issues. Common cloud settings often operate in a black-box manner and require the data owners to trust the service providers. If inadequate security measures are adopted, sensitive data could be leaked to unauthorized parties. Furthermore, the rise of big data analytics has made personal data unprecedentedly valuable. Malicious CSPs can monitor user activities and look into users' data anytime without notifying the data owner. As a result, cloud users' privacy is compromised. We believe aforementioned security threats are the major barriers that prevent individuals, businesses, and government from migrating sensitive data to the cloud.

To ensure data secrecy in a cloud environment, the most straightforward method is to encrypt the data before sending it to the cloud. By transforming data into a scrambled format, the secret key holder remains the only one who can access the original information. To instantiate a search, the client must download the documents, decrypt them and perform searching. If the document is lengthy or the document collection is large, it could be time-consuming and wasting network bandwidth, which are particular the major concerns for mobile device users. The idea of searchable encryption emerged and the primary goal is to make searching feasible on encrypted data while retaining data confidentiality and eliminates the need for downloading entire document to local storage first. In the past decades, numerous researchers [15, 3–5, 8] have contributed to the topic and proposed different constructs to achieve the goal. Prior studies focus on improving the searching efficiency and enhancing the data security, and less effort has been put on exploring the impact of these schemes on mobile devices, where their memory, computational and network capabilities are limited, which compared with their desktop/laptop counterparts. Given that searchable encryption schemes usually involve complicated computations, it would place a burden on the mobile devices and eventually lead to an undesired user experience.

In this paper, we propose a software implementation of three recent searchable encryption schemes [10, 2, 12] and benchmark their performances in Android mobile devices, in terms of 1) exact-match, 2) partial search, and 3) multi-keyword queries. We believe that our findings could give insights to mobile App developers the impact of adopting SSE schemes in applications of mobile devices.

2 Related Work

2.1 Symmetric Searchable Encryption

The development of symmetric searchable encryption (SSE) can be traced back to 1996 when Oblivious RAMs (ORAM) [8] were first proposed. ORAM is a computing technique that was originated for software protection, but the concept can also be applied to searchable encryption. By using ORAM, one can modify a function's memory access pattern without changing its input and output behavior. Recall that the key objectives of SSE are retaining data secrecy and protecting user privacy, ORAM can hide the underlying data structure and access pattern so that an optimal level of privacy can be achieved. The drawback is that it requires $O(log\ n)$ rounds per read/write, which is not an efficient approach. Since ORAM-based searching algorithms are computational expensive in nature, researchers started to explore other feasible methods in addressing the efficiency problem. One of the research direction is relaxing certain security attributes (e.g., leaking the search pattern) in return for better performance. Song et. al. [15] proposed an SSE scheme that can secure the data being outsourced, together with substantial queries and corresponding result. Their scheme can be classified into two phases: the setup phase and the search phase. During the setup phase, each document is tokenized into words. Every word is encrypted into "search token" using a combination of deterministic encryption with random factor (pseudorandom number generated by the word position within that document). The client concludes the setup phase by pushing encrypted documents and the "search tokens" to the server. When performing searches, the client first encrypts the search term into token and send the token to the server. The server performs matching with saved search tokens and returns the document(s) accordingly. As the server does not have access to any plaintext, nor the encryption key, the data remains confidential. This scheme requires $O(n)$ operations for encryption and search for a document with length n. It is comparatively faster and more efficient.

The method from [15] enjoys a significant efficiency improvement over ORAM based schemes. But the performance remains a problem when the documents are large. To resolve this problem, they suggested to use keyword-as-key hash tables to store pointers to documents (a.k.a. inverted indexes to documents). Goh [7] argues that this method could leak partial information of the encrypted documents when the hash table is being updated. Adversaries can perform chosen-ciphertext attacks (CCA) and deduce the encrypted content by analyzing the update pattern of the indexes. As an alternative, Goh suggested using Bloom Filter can fix this update loophole. In Goh's Z-IDX construction, it does not store the keyword directly, but it runs each keyword through multiple hash functions and maps the output into an arbitrary number of bits (e.g., m-bit array). When performing a search, we just need to run the search phase with the same procedure again. By comparing the result with the stored m-bit array, then we can easily determine whether the document contains the keyword. Because of hash collisions, the use of Z-IDX could bring false positive to the search result. But the actual index size is much smaller and it allows quick comparison when perform searching.

Another interesting topic under SSE is about the capability to perform partial searches on encrypted data. When searching, it is common for us to enter only certain part of the keyword and ask the search engine to return all matched results. The previous SSE schemes mentioned does not address the substring searching problem explicitly. To facilitate substring searching in these SSE schemes, a simple method is to include all substrings when constructing the index, but this generally brings drawbacks that could outweigh the benefits in return. For example, in Song et. al.'s [15] construction, this would dramatically increase the index size and computation rounds per keyword (for a string with length n, there will be an extra $n*(n+1)/2-1$ substrings to be stored/processed). This also means that there will be more computations during the search phase. As for Goh's [7] Z-IDX scheme, it can amplify the false positive rate in the search result and the extra overhead could slow down the index-building process.

2.2 Profiling Methods under Android Platform

There exist many profiling tools in Java that allow us to gain insights into applications under different aspects. While the Java profilers for the desktop platform are well-developed, the story is totally different for the Android platform. Despite both platforms can be coded with Java, the internal architecture of two platforms are different. Android is designed to run on power-constrained devices, it is equipped with a specialized virtual machine called Dalvik Virtual Machine (DVM). Different from the Java Virtual Machine (JVM), DVM is characterized by its register based design and its proprietary format of bytecode (.dex). In addition, the Dalvik VM supports a subset of the standard Java library only, in which some useful benchmarking packages (e.g. the *java.lang.instrumentation* package) are missing or have limited functionalities. For these reasons, most of the existing Java profiling tools and frameworks are not compatible with Android.

At present, the most effective way to profile an Android App is using the built-in Android Monitor [1] under Android Studio. It provides a graphical representation of key aspects (e.g., CPU, memory, and GPU usage) of the DVM runtime and includes several tools which can capture and save information while the App is running. The Android Monitor is useful in analyzing an App, but it does not provide a structured way to store the profiled data for later inspection.

Another approach in profiling Android App is to introduce extra code to the App. For instance, one could create two variables to store the time before and after the invocation of methods. By calculating the elapsed time, developers can get the actual processing time of that method. This approach allows customizable code measurement, but it requires modification of existing code. Practically, it is not well suited for Apps that have complex structure because it can cause more confusion with the code base. Some developers have spotted this problem and introduced AspectJ [6]. AspectJ is an open source library that supports Compile Time Weaving (CTW). During compilation, it modifies existing Java code behind the scene and introduces extra behavior to existing functions. Recent AspectJ development on Android platform allows us to benchmark and monitor function calls easily. In the Hugo plugin [17], Jake Wharton consolidated the AspectJ library into a Gradle plugin which can be easily integrated with existing Android Apps. Through this plugin, developers can "annotate" their Java methods and get the parameters and execution time printed to the console. In our analysis, we employ similar techniques in capturing function calls as well as measuring the execution time.

Our Results. In this paper, we give our results of performance analysis of three SSE schemes on Android devices in terms of indexing time and searching time, under 1) exact-match, 2) partial search, and 3) multi-keyword queries. The three schemes are implemented in Java so that it is completely compatible and testable in Android devices.

3 Analysis Approach

3.1 Introduction

Our analysis is divided into several phases. In the initial phase, a library skeleton, an Android client App and a web service are created. At the beginning, these three objects contain only a few classes and simple user interface. Then, three SSE schemes are chosen and implemented in Java. We integrate the newly implemented schemes into the library skeleton. Performance tests are conducted to test the actual impact when running on our Android phone. We made use the AspectJ library in measuring CPU time. In the next phase, we proceed to develop the corresponding Android App (client) and the supportive web service (server). The client allows users to upload documents stored in their phone to cloud storage in an encrypted format. It also provides the interface for searching uploaded documents. As for the web service, it mainly supports three primary functions: 1) document indexes storage; 2) search request handling; 3) statistics (for both client/server) recording for evaluation. To ensure the maximum degree of compatibility and promote code reuse, both the client and server applications are written in Java.

3.2 Software Architecture

As searchable encryption schemes assume a client-server setting, we followed this design when implementing our system. The system can be spitted into two logical parts - the **Client** and the **Server**. The **Client** consists of four components. Our **Android App** uses the Android API for the user interface and other built-in functions. The **SSE Client Library** contains a subset of the logics and functions that would be used in SSE schemes. Our App employs the **Google Play Service API** to authenticate Google accounts and storing encrypted documents to the cloud. These encrypted documents were stored at user's own Google Drive storage. We target on Android API level 19 as the minimum API because it supports both Dalvik and Android Runtime (ART). We could switch between ARTs and compare the performance difference efficiently. The **Server** part contains four components. The **Web Service** runs a Java Web application and utilizes Tomcat API for HTTP service. The **SSE Server Library** contains utility functions which allows the Web Service to perform comparison between search tokens and stored search indexes. Taking into account that the search indexes could be large in size, the Web Service stores the search indexes at the database server to facilitate near-constant time of retrieval. The **JDBC Library** were used for database related operations.

3.3 Hardware Architecture

Similar to our software design, we adopt client-server approach in hardware architecture. There are three servers: 1) **Query Server**, 2) **Database Server** and 3) **Cloud-based Storage Server**. When the end user uploads a document, the **Android Client** sends both the encrypted document and search indexes to the network. The search indexes will be forwarded to the **Query Server**. The **Query Server** stores the received search indexes into the **Database Server** for later search operations. As for the encrypted document, it will be routed to and stored into the **Cloud-based Storage Server**. In this paper, we are referring Google Drive as the **Cloud-based Storage Server**. For search queries, they are issued by the **Android Client**. The **Android Client** passes the query to the **Query Server**. **The Query Server** processes the query and performs retrieval with the **Database Server**. **Database Server** returns matched information to the **Query Server**. The **Query Server** returns formatted result to the **Android Client**. Based on search result, documents are retrieved from the **Cloud-based Storage Server** to **Android Client**.

3.4 Experiment Design

Our experiments on SSE schemes are coded with Java. Experiments on client device are conducted on an LG G2 D802 (Snapdragon 800 with 2GB RAM, Android 4.4.2). This model was publicly available in 2013 and comes with a quad-core CPU. Given most of the Android phones nowadays usually come with 4+ core CPU and few GBs of RAM, we believe the capability of D802 can reflect the performance of mainstream Android phone in the current market. For the experiments on the server, they are conducted on a laptop computer (i7 2620M Dual Core 2.7GHz, 8GB RAM, Windows 7 x64), which hosts the web service and the database server on the same machine. In our settings, both the Android phone (WiFi) and the laptop (wired) are connected to the same LAN, so network latency is minimized. To offer a common ground for evaluation, we have selected the alt.atheism from the 20 Newsgroup Data Set [16] as the testing data. It consists of 1000 ASCII encoded documents which sized from 438 bytes to 51 kilobytes.

These 1000 text-based documents are stored in the internal storage of the client device. For each document, the client parses the content and tokenizes every line into keywords. The extracted keywords were then converted into lowercase and duplicates are removed (or "normalized"). The client processes each unique keywords (the actual method varies for different SSE schemes) and transmits the search indexes to the server. Since mobile devices usually have memory constraints, the search indexes will be uploaded to the search server upon creation and purged from the memory immediately. This approach keeps the memory usage to the minimum. As for the document files, the client performs encryption on each document and tranfers them to the cloud storage. Same as the method we used in creating search indexes, the encrypted documents will be discarded immediately once they have been uploaded to the cloud storage. Note that three implemented SSE schemes share a common method for encrypting documents, so the performance differences in file-encryption can be neglected. We use 128-bit AES in CBC mode for file-based encryption and PKCS7 padding is used to fill up empty bytes within ending block so the resulting encrypted documents would be slightly larger. All of the aforementioned tasks are carried in single-threaded, sequential approach.

We have implemented three schemes which vary in terms of search index construction and query processing. In order to provide a comprehensive comparison of each scheme, our analysis also covered the performance of server side. We developed automated tests for simulating client/server communication. The test metrics include the search time of implemented SSE schemes and the required storage for storing the search indexes. Since all three implemented SSE schemes support multi-keyword searching (either implicitly or explicitly), we would also evaluate the extra overhead when more than one keyword is submitted. For our server environment, we host a single instance PostgreSQL 9.5 as our database server. We developed a PostgreSQL extension using the PL/Java [16] library.

3.5 Experiment Implementation

For the client-side experiments, we make use of AspectJ library in measuring the actual CPU time of the testing subject. To minimize confusion on existing classes, we have created a separate Android module for storing AspectJ related code. This module contains three Java classes StopWatch, PrefMon, and TraceAspect. The StopWatch class is a Java bean which contains two variables, one for storing the start time and the other one stores the end time of method execution. The PrefMon class is a marker interface which is used to annotate methods to be tested. In our experiments, information will be captured from the annotated methods only. During compilation, methods annotated by PrefMon will have additional code "injected". The code injection process is done behind the scene and does not require intervention by the developer. The injected code exposes runtime information to the TraceAspect class. Through the TraceAspect class, we can learn runtime information about the method being invoked (e.g., method name, signature, parameters and their data type) or manipulate the parameters on the fly.

Since our goal is to measure the execution time, we create a StopWatch variable for storing the timestamp before/after the function calls. By computing the delta between two timestamps, we get the elapsed time of each function call. The TraceAspect class consolidates captured information (e.g., execution time, method name, input size, and number of keywords) and uploads to the remote server for further analysis and evaluation. Similar to the way we handled the search indexes, these key information will be pushed to the server and discarded immediately at the client device.

For the server side experiments, we used the JUnit [13] library and Apache HttpComponents [11] for our tests. We first extracted all unique keywords (around 350K keywords, extracted with the same logic as search indexes construction) from our testing dataset and put them into one set, S. During our tests, we randomly sampled one thousand keywords from S, created and submitted the search token to the server. We used the *org.apache.http.impl.client.CloseableHttpClient* to submit POST requests to the server. Then, we counted the duration starting from the time that the client issues a request to the point when the server returns the result to the client.

4 Our SSE Code Implementations

4.1 SUISE

We adopted the SSE scheme suggested by Hahn and Kerschbaum [10] and implemented SUISE as our first SSE scheme. SUISE is IND-CCA2 secure. It consists of six operations and most computationally intensive operations such as index construction and file encryption are handled by the client. In contrast, the server side does not involve many complex operations apart from the search operations. SUISE features dynamic secure index and efficient query processing. The client is allowed to change the document contents after initial outsourcing and alter the search index accordingly. Hash table is used in inverted indexing to facilitate quick searches. Another key design is the use of caching on improving substantial query performance. In their proposal, the client is the only one who possesses the keys. Any other parties including the storage and search server do not have access to the keys. The scheme makes use of the keyed hash function and random salt in creating search index that is "blind". Therefore, even the communication channel or servers are compromised, attackers still cannot access to the original information.

Moreover, there is possibility of enabling substring searching for this scheme by enumerating all substrings when building index and then we can match it during the search phase. On the basis of this idea, we further extended SUISE to support substring search and named them as SUISE2. SUISE2 supports prefix-based partial search.

4.2 VASST16

For the second SSE scheme, we implemented the scheme by Vasudha Arora and S.S. Tyagi [2] (VASST16). Unlike other traditional Boolean-based SSE schemes, the document sorting feature is incorporated into this scheme. The relevancy sorting feature is essential to any retrieval system, especially when searching through a large document collection. The scheme uses TF-IDF (Term Frequency and Inverse Document Frequency) as weighting method. Every document is sorted by their TF-IDF scores in descending order, and Top-K (where K is the desired number of result) results are returned to the client. Since the TF-IDF method does not only cater the individual term occurrences but also looking into the overall document frequency within the whole document collection, the system is designed to scan through whole document collection to compute the IDF component. Therefore, it can take a considerable amount of RAM and it does not fit well in memory constrained settings. In addition, if any new documents are being added to the document collection, the TF-IDF score would become inaccurate because the total number of document (IDF) has changed. In other words, we need to rebuild the search index and update the server again. To address this problem, we have modified the original scheme to fit our settings. In our implementation, we delegated the IDF calculation to the server. When creating search index, we computed only the term frequency of the keywords only. We used the tuple T = (SearchToken, Frequency) to store the encrypted index accompany with its term frequency.

Because we skipped the IDF component during the index phase, we computed it at search time instead. The search server is responsible for computing the IDF component upon request. We first retrieved 1) the number of matched documents, 2) the total number of documents including those unmatch items and 3) matched index entries and their frequency from the database. Based on this information, we calculated the TF-IDF score of each document and sorted them in descending order. All these calculation were done at the server side. In the end, the top-10 most relevant documents are returned.

Similar to SUISE2 which supports substring searching. We have implemented another version, called VASST16-2, that supports prefix-based partial search, so that the performance of the schemes chosen can be properly compared.

4.3 CHLH15

When searching is performed, it is common for us to enter only certain parts of keyword for searching. From the user perspective, these kinds of partial search method are intuitive and helpful in locating relevant documents. While these partial search method can bring users great convenience, they also come with a cost. A classic example is that database queries with partial search components generally run slower than exact-match. In the field of searchable encryption, the same concept applies. Encryption attempts to transform user data into scrambled and randomized message to secure data secrecy, but the process itself also breaks the relationship between character and word. In our first implementation, we tried a way to retain the character-word relationship by enumerating all possible substrings within a word. The drawback of this approach is that it can take a considerable amount of index storage, and is inefficient for both index construction and search.

To solve this problem, Changhui Hu and Lidong Han [12] proposed a new scheme (CHLH15). In their proposal, they feed each individual characters of keyword plus its position into a Bloom Filter (Index BF) and then store this filter to the cloud. During the search phase, they use the same method to create another Bloom Filter (Trapdoor BF) and submit to the server. The server side performs bitwise AND operation with the Index BF and Trapdoor BF. If the bitwise result does not equal to the keyword filter, the document does not contains the keyword; if the bitwise result matches with the keyword filter, this document *may* contain the keyword. Originally in our CHLH15 implementation, we attempted to use the Bloom Filter (BF) from the Guava library [9]. However, we encountered an "Android Dex Over 64K Methods" error when compiling it with Android Studio. This is because Guava contains too many classes and methods that cannot be compiled into Android's Dex format. Eventually, we chose Magnus Skjegstad's Bloom Filter [14] as our concrete BF implementation. Magnus's implementation contains only one class and it depends on the Java/Android API only, which is relatively easy to integrate into existing code. In addition, Magnus's implementation provides sophisticated control over several performance-critical parameters, such as expected false positive rate and number of expected elements.

Back to our CHLH15 implementation details. Due to the memory constraints, we have modified CHLH15 to fit into mobile device settings. In original construct, only single Index BF is created and the whole document collection will share the same BF. In our implementation, we will create a separate set of Bloom Filter for each individual document. In other words, for a collection of m documents, we will get m Bloom Filter entries instead of one. After our modification, the required index space is now having a linear relation with the total number of document, so there will be a performance penalty if the document collection is large.

5 Performance Comparison

In this section,we evaluate the three schemes in terms of indexing and search speed under single keyword and multi-keyword queries.

For single keyword exact-match queries, SUISE has the fastest indexing speed among all implemented schemes. On average, it takes 100ms to compute, which is 20% of CHLH15 and 10% of VASST16. CHLH15 has the medium performance. It takes 0.5ms by average to process a single document. VASST16 has the worst indexing performance, which takes 1s because of the computational expensiveness in the 2nd round encryption. From the searching point of view, VASST16 has the fastest response time. On average, it takes half a second to process a single-keyword query. The second rank belongs to CHLH15, where it takes nearly 1s to return the search result. For the slowest scheme, SUISE takes 1.8s to process.

For single keyword partial-search queries, after including all prefixes during the indexing phase, SUISE2 remains the fastest schemes in indexing documents. Its performance is slightly better than CHLH15, with approximately 100ms faster. VASST16-2's indexing speed remains the last, which takes around 4s to index a document. For the search time, CHLH15 has the best performance in terms of prefix-based queries. It requires 900ms to process, which is around 50% of VASST16-2 and 14% of SUISE2. As the number of index entries increase, more comparisons have to be done at the server side. From these tests, we can identify the competitive advantages of CHLH15 over other schemes. Because the number of index of CHLH15 is linear to the document count, it prevents an excessive amount of index entries from slowing down the processing time of searching.

For multi-keyword exact-match queries, the searching performance of SUISE scales with the number of the keyword used. For any new keyword added, the search time of SUISE is increased by 1000ms. In contrast, VASST16 and CHLH15 are less sensitive to the keyword count. The performance of these schemes does not deteriorate much when additional keywords submitted.

6 Performance Analysis

For SUISE, preliminary test results show that file-based encryption is reasonably efficient at the client side, but the index construction could be a problem when the document is large. The construction time is linear to the number of words, which is not desirable for lengthy documents. Moreover, we have explored the possibility of enabling partial search in SUISE without extensive modification on the scheme. Results show that it is feasible, but at the cost of heavy processing at both the client and server sides. Prolonged indexing time and query response time can lead to an undesirable user experience. For VASST16, the document sorting feature from VASST16 has caught our attention, but we have experienced difficulties in realizing it due to memory constraints. In addition, it is observed that Android's "big number" constructs are fairly computationally expensive. Accordingly, the 2nd level encryption of the VASST16 has suffered performance penalty at the client side. Despite its usefulness in finding a needle in a haystack, it seems that our Android implementation needs further optimization before it reaches an acceptable performance. As for our 3rd SSE scheme, test results show that CHLH15 have satisfactory index creation speed and quick response time. Nevertheless, its sensitiveness to the false positive rate and immutable nature have restricted the use cases. For instance, developers must be aware of the number of items to be indexed or the underlying data structure will get saturated. This limited the scalability and extensibility when deploying on a real application. In the light of our comparison result, it is observed that SUISE has the best indexing performance under exact match queries, and has minimal impact on the client devices. Nevertheless, due to the complexity of the search token-index comparison, it requires the longest process time for searching. Application users may not feel comfortable with the prolonged search time. For functional reasons, VASST16 requires the most computational time at the client side but the scheme does not show advantages that could overweight the cost in return. Yet, the existence of VASST16 proves that document relevancy sorting under SSE is feasible, though further development and optimization is necessary. CHLH15 balanced the indexing time and searching time under partial search queries. Users can obtain a reasonable response time in searching while having minimal computation overhead on their devices.

7 Conclusion

The rise of cloud data outsourcing brings us convenience and cost saving, but it also leads to security and privacy problems. Existing proposal of searchable encryption schemes attempts to strike a balance between usability and data privacy, but at the cost of high computational requirement. While this may be acceptable in the desktop/laptop environment, the actual impact on mobile devices remains unanswered. In an effort to answer the question, we have chosen three recent SSE schemes from literature, implemented them and compare their performance in Android mobile devices. We explored several existing Java profiling tools and found that most of them are not compatible with the Android platform due to structural differences in the runtime environment. We developed a new module for capturing and storing the benchmarking results. We performed an analysis on the indexing time and searching time, under 1) exact-match, 2) partial search, and 3) multi-keyword queries. We believe that our research provides insights to mobile App developers about the computational requirements of various SSE schemes for applications in mobile devices.

References

1. Android: Android monitor basics. https://developer.android.com/studio/profile/am-basics (2018), [Online; accessed 4-May-2018]
2. Arora, V., Tyagi, S.: An efficient multi-keyword symmetric searchable encryption scheme for secure data outsourcing. International Journal of Computer Network and Information Security **8(11)**, 65–71 (2016)
3. Bellare, M., Boldyreva, A., O'Neill, A.: Deterministic and efficiently searchable encryption. In: Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings. pp. 535–552 (2007)
4. Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A.: Order-preserving symmetric encryption. In: Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings. pp. 224–241 (2009)
5. Boneh, D., Sahai, A., Waters, B.: Functional encryption: Definitions and challenges. In: Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings. pp. 253–273 (2011)
6. Foundation, E.: The aspectj project. http://www.eclipse.org/aspectj/ (2018), [Online; accessed 4-May-2018]
7. Goh, E.: Secure indexes. IACR Cryptology ePrint Archive **2003**, 216 (2003)
8. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)
9. Google: google/guava. https://github.com/google/guava (2018), [Online; accessed 4-May-2018]
10. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014. pp. 310–320 (2014)
11. Hc.apache.org: Apache httpcomponents. https://hc.apache.org/ (2018), [Online; accessed 4-May-2018]
12. Hu, C., Han, L.: Efficient wildcard search over encrypted data. Int. J. Inf. Sec. **15**(5), 539–547 (2015)
13. Junit.org: Junit – about. https://junit.org/junit4/ (2018), [Online; accessed 4-May-2018]
14. Skjegstad, M.: Magnuss/java-bloomfilter. https://github.com/magnuss/java-bloomfilter (2018), [Online; accessed 4-May-2018]
15. Song, D.X., Wagner, D.A., Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000. pp. 44–55 (2000)
16. Tada.github.io: Postgresql pl/java pl/java: stored procedures, triggers, and functions for postgresql. https://tada.github.io/pljava/ (2018), [Online; accessed 4-May-2018]
17. Wharton, J.: Hugo. https://github.com/jakeWharton/hugo (2018), [Online; accessed 4-May-2018]