algorithms

MDPI

Article

Searchable Data Vault: Encrypted Queries in Secure Distributed Cloud Storage

Geong Sen Poh [1,*], Vishnu Monn Baskaran [2], Ji-Jian Chin [2], Moesfa Soeheila Mohamad [1], Kay Win Lee [1], Dharmadharshni Maniam [1] and Muhammad Reza Z'aba [3]

1 Information Security Lab, MIMOS Berhad, 57000 Kuala Lumpur, Malaysia; soeheila.mohamad@mimos.my (M.S.M.); kw.lee@mimos.my (K.W.L.); dharmadh.maniam@mimos.my (D.M.)
2 Faculty of Engineering, Multimedia University (MMU), 63000 Cyberjaya, Malaysia; vishnu.monn@mmu.edu.my (V.M.B.); jjchin@mmu.edu.my (J.-J.C.)
3 Network Security Lab, MIMOS Berhad, 57000 Kuala Lumpur, Malaysia; reza.zaba@mimos.my
* Correspondence: gspoh@mimos.my; Tel.: +60-3-8995-5000

Academic Editors: Sye Loong Keoh and Khin Mi Mi Aung
Received: 28 February 2017; Accepted: 3 May 2017; Published: 9 May 2017

Abstract: Cloud storage services allow users to efficiently outsource their documents anytime and anywhere. Such convenience, however, leads to privacy concerns. While storage providers may not read users' documents, attackers may possibly gain access by exploiting vulnerabilities in the storage system. Documents may also be leaked by curious administrators. A simple solution is for the user to encrypt all documents before submitting them. This method, however, makes it impossible to efficiently search for documents as they are all encrypted. To resolve this problem, we propose a multi-server searchable symmetric encryption (SSE) scheme and construct a system called the searchable data vault (SDV). A unique feature of the scheme is that it allows an encrypted document to be divided into blocks and distributed to different storage servers so that no single storage provider has a complete document. By incorporating the scheme, the SDV protects the privacy of documents while allowing for efficient private queries. It utilizes a web interface and a controller that manages user credentials, query indexes and submission of encrypted documents to cloud storage services. It is also the first system that enables a user to simultaneously outsource and privately query documents from a few cloud storage services. Our preliminary performance evaluation shows that this feature introduces acceptable computation overheads when compared to submitting documents directly to a cloud storage service.

Keywords: searchable symmetric encryption; searching encrypted data; cloud security

1. Introduction

It is now common practice for an individual to share or backup documents using cloud storage services. Examples of services include Google Drive, Microsoft OneDrive, Apple iCloud, Dropbox and Amazon S3. This provides an individual with easy access to his/her data anywhere and anytime. In addition, more and more organizations from both the private and government sectors are moving their data and computations to the cloud. In the US, for instance, both Amazon and Microsoft provide specialized cloud services called AWS GovCloud and Azure Government to government agencies. These services are physically isolated from the regular cloud services offered by both providers and need to adhere to strict government security regulations. The existence of such services is probably attributable, in part, to the surge in usage of cloud-based services in the government sector.

According to a report by security audit firm Netwrix, the number of cloud adoptions in the surveyed organizations increased from 43% (2015) to 68% (2016) [1]. The report also states that cloud

security is a primary concern for 70% of organizations worldwide. In a report published in 2016, Cisco anticipates that by the year 2020, 59% of the world's Internet users will use personal cloud storage, which is an increase from 47% in 2015 [2]. These trends are not uncommon since a cloud storage service brings numerous benefits to the user. The service, however, is not without its downsides.

One common property inherited by all the current cloud services is that the user does not have total control of the privacy of the stored documents. This is inevitable since the provider is expected to have read access privileges in order to be able to search through documents related to the user's query. A seemingly trivial solution would be for the user to encrypt all documents prior to storing them on the cloud. However, the user now loses the ability to search these documents. The workaround is either to download and decrypt all documents locally or give the encryption key to the cloud provider. The former approach is extremely inefficient while the latter raises privacy concerns.

In order to protect the privacy of documents while at the same time allowing a user to efficiently search them, we introduce a system based on our multi-server SSE scheme that we call the searchable data vault (SDV). The SDV enables a user to store encrypted documents in the cloud and to retrieve them in an efficient and privacy-preserving manner. Our main contributions are as follows:

1. SDV allows documents, or pieces of an encrypted document, called blocks, to be stored in different cloud storage services, in contrast to existing schemes and systems that focus on outsourcing to a single storage. It ensures with high probability that no single storage provider has a complete set of blocks, which it may use to learn additional information about the document. As far as we know, it is the first searchable encryption system that provides such a feature.
2. A core component of SDV is a controller that manages query indexes, document submission, and retrieval. The controller is designed in such a way that the underlying searchable symmetric encryption (SSE) scheme, utilized for efficient search, is "pluggable". It means existing schemes that cater for single storage provider can be adapted for use with our system. The controller may be implemented and placed in a query gateway. This is the case for our deployment.
3. We further propose a multi-server SSE scheme, which is adapted from [3], with an implementation for SDV.
4. The query index is designed to cater to a two-level dictionary structure, where the basic level is for a single-word query. An upper level can be included for future extension to more expressive queries (e.g., ranked, range, conjunctive).

Figure 1 illustrates the architecture of SDV, which consists of three main modules that provide the mechanisms contributing to our main results:

- *SDV app:* This module provides an interface for the SDV system to interact with the users for document upload and retrieval. In our deployment, it is also integrated with a unified authentication platform [4] for the purpose of user registration and authentication.
- *SDV controller:* This module maintains a database that contains credentials of users and cloud servers. It also creates a query index table, and manages the submission and retrieval of encrypted documents.
- *SDV API:* This module contains an implementation of our multi-server SSE scheme. The scheme divides and encrypts blocks of documents, and generates index entries for query purposes.

Figure 2 further illustrates our modular approach in designing the SDV controller, so that the underlying searchable encryption scheme can be replaced without too much modification in the system, as well as providing flexible extension to various search functionalities. The idea is to obtain only the index entries from the searchable encryption scheme. Creation of the query index table is done at the SDV controller, instead of the common approach of creating the index table by the SSE scheme. As for search functionalities, our intuition is to maintain a basic query index table with filename as the keyword token. This token links to the encrypted blocks and the servers that store these blocks. By doing so, query module based on information retrieval techniques, or expressive searchable encryption mechanisms, for example in [5,6] can be customized to return filenames matching the

query keywords. These filenames can then be passed to the basic index table for document retrieval. We discuss our multi-server SSE scheme utilizing the filename as a keyword in Section 4.

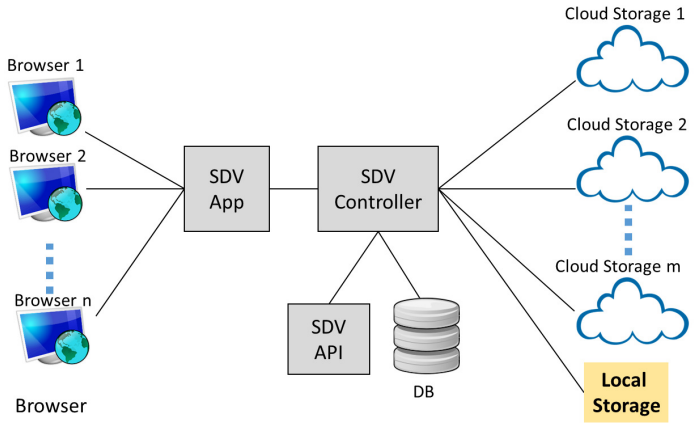


Figure 1. The searchable data vault (SDV) consists of three main components: (**a**) SDV application; (**b**) SDV controller; and (**c**) SDV API. Users access the system through a web browser. The SDV app provides the interface to interact with the users. The SDV controller manages the credentials database, and the core functionalities through interaction with the SDV API, which consists of the searchable symmetric encryption (SSE) scheme, and distributes encrypted documents to storage servers. The servers may include a mix of third-party cloud storage and local in-house storage servers.

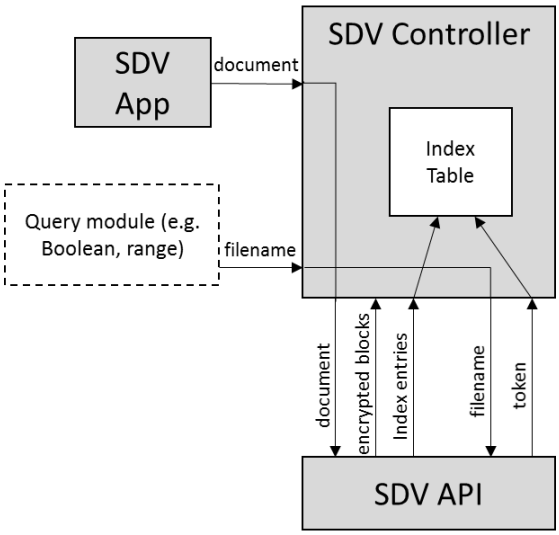


Figure 2. SDV controller—indexing and search: The SDV app submits a document to the SDV controller, which forwards the document to the SDV API. The SDV API then executes the underlying SSE scheme. The document is divided into blocks. The encrypted blocks, together with an index entry of the document are returned to the SDV controller, which then inserts the index entry to a query index table, and forwards the encrypted blocks to their respective servers based on the index entry. The index entry contains a token generated based on filename, and also information on where to store/retrieve the encrypted blocks of the document with the filename. The search then is simple: submit a filename to the SDV controller and a token is generated by the SDV API to search the index table. Given such a setting, a query module for more expressive search (e.g., Boolean) can be added as an additional module, as long as the search result for this module is a list of filenames.

2. Related Work

2.1. Systems

One of the earliest comprehensive systems created for privacy-preserving queries on encrypted data is CryptDB, proposed by Popa et al. [7]. CryptDB is a system for SQL-based encrypted queries to the database. It utilizes order-preserving encryption, homomorphic encryption, and Song et al. [8]'s SSE scheme, and can be considered as one of the early systems for providing such services. CryptDB provides layer-based mechanisms (called onions of encryption), in which different encryption schemes (e.g., SSE) are selected for different types of queries e.g., SEARCH) to ensure the most secure approach is used to perform encrypted queries in encrypted records. Lau et al. [9] then proposed a system for mobile platforms known as M-Aegis. The system puts a transparent layer on top of existing applications, such as GMail and WhatsApp, so that it can be integrated without changing the look-and-feel of these applications. The transparent layer encrypts a message before it is passed to the underlying application, and allows for querying and decrypting the encrypted message, through a searchable encryption scheme. Other systems include BlindSeer [10,11] and Mylar [12]. Blindseer is a practical system that caters for private SQL-based queries to the database, as in CryptDB, but uses different techniques (i.e., garbled circuit and tree) and allows arbitrary Boolean queries. It deploys an index server to manage indexes and queries. Mylar, on the other hand, is a multi-user system that provides searchable encryption using a pairing-based construct. Commercial systems that also provide searchable encryption include Skyhigh Networks [13], CipherCloud [14] and Bitglass [15]. These systems use a simple SSE approach, such as attaching keyword tokens to an encrypted document, as stated in [16]. We remark that all these systems focus on outsourcing to a single storage provider.

2.2. Searchable Symmetric Encryption (SSE)

Song et al. [8] introduced practical schemes for searching encrypted data using only symmetric primitives. Goh [17] then proposed an index-based structure, which can be deployed for searchable encryption and proved the structure secure through the notion of chosen keyword attack. Chang and Mitzenmacher [18] also proposed a searchable encryption scheme using index structure and defined a simulation-based security model. Later, Curtmola et al. [19] provided a formal security model that improved on the previous definitions by Goh [17], and Chang and Mitzenmacher [18]. A scheme achieving sublinear search time using inverted index was also proposed. Most of the subsequent SSE schemes, such as schemes in [5,20–27] adopted or extended the model proposed by Curtmola et al. [19]. All these schemes are based on a single server.

More recently, schemes that involve more than one server have been proposed, such as in [3,28,29]. In Bösch et al. [28], a query proxy is introduced in addition to the storage server. The query proxy operates in such a way that every query is different from the previous queries, even when a keyword is queried many times. The main idea is for the query proxy to re-encrypt the query and re-process the index every time a query is submitted. Due to this, the scheme may not be scalable. Ishai et al. [29] in their proposal also introduced an additional server known as a helper server. The main objective of their proposal is to minimize leakage and cater for a wide variety of search functionalities such as range queries, that also scale to large databases. Relatively less computationally efficient oblivious RAM (ORAM), private information retrieval (PIR) and multi-party computation (MPC) mechanisms are deployed to a small part of the database. In contrast to the above proposals that deploy a helper or proxy server, Poh et al. [3] introduced SSE schemes that allow for a message to be divided into blocks and randomly distributed to many servers. By doing so, leakage can be further reduced as compared to schemes with one storage server. Kuzu et al. [30] proposed a distributed SSE scheme, which also distributes data blocks into many arrays of commodity hardware commonly deployed in a cloud storage server by utilizing the underlying cloud storage architecture. We note that it is still a single server scheme since it involves only one cloud server, where block distributions are performed internally on the arrays of disks of the cloud server. We note that our system is based on the techniques of SSE.

2.3. PEKS and General Mechanisms

There are also public key-based schemes, which were introduced by Boneh et al. [31] and termed public key encryption with keyword search (PEKS). In addition, fully hidden searchable encryption scheme can potentially be achieved using building blocks such as fully homomorphic encryption [32] and ORAM [33,34], but schemes based on these will be less efficient compared to SSE schemes. For a comprehensive survey, we refer interested readers to [35].

3. Definition

The main engine of SDV is a multi-server SSE scheme in the SDV API. In this section, we provide definition of such schemes as defined in [3], and present our scheme in the subsequent section.

3.1. Notations

Table 1 provides notations used to describe the definition, and the proposed SSE scheme.

Table 1. Notations used in the description of the SSE scheme.

Notation	Description
D	$D = \{D_1, \ldots, D_n\}$, is a set of n documents.
D_i	A document i.
S	$S = \{S_1, \ldots, S_s\}$, is a set of s servers.
S_x	A server x.
C	$C = \{\mathbf{c}_1, \ldots \mathbf{c}_s\}$, is a set of lists of s encrypted blocks.
$id(x)$	An identifier of an item x.
K	A set of cryptographic keys.
I	A query index with a $I[key] = value$ structure, in which given a key, the $value$ is returned.
tok	A keyword token.
$\mathbf{t}$	A temporary list to hold the identifiers of the encrypted blocks.
$F_{k_f}(.)$	A keyed pseudorandom function with key k_f. This is used to generate keyword token.
$\mathcal{E}$	$\mathcal{E} = (\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$ is a symmetric encryption scheme where `Gen` is the key generation function, `Enc` the encryption function and `Dec` the decryption function.
`op`	The document addition or removal operation of the update algorithm in a SSE scheme, `op` $\in \{$`Doc+`, `Doc-`$\}$.
`in`	The information submitted to the update algorithm. `in` can be a document D_i for `Doc+` or an identifier $id(D_i)$ for `Doc-`.
$\mathcal{L}^l$	A leakage profile that states information leakage of the l phase in a SSE scheme (i.e., document size, index size), where $l \in \{Setup, Search, Update\}$.
$\texttt{negl}(\lambda)$	A negligible function with security parameter λ.
$\mathcal{S}$	A simulator.
$\mathcal{A}$	An adversary.
B, $\lvert B \rvert$	List of blocks, and the number of blocks in the list.
b	Length (or size) of a block.

3.2. Multi-Server SSE Scheme

A multi-server SSE scheme consists of three algorithms, namely `Setup`, `Search` and `Update`.

We note that this is similar to a single server SSE scheme, for example, as proposed in [19,23,26]. The only difference is the process of division/combination of documents and distribution over multiple storage servers within each algorithm.

`Setup`. The algorithm partitions a document D_i into a list of blocks B and then encrypts and assigns an identifier $id(.)$ and storage location to every block. Input to this algorithm is the set of documents D and a keyword index table which map each document D_i to a list of keywords. Output of this algorithm is a set of cryptographic keys K to be stored securely by the user, index I and sets of encrypted blocks C.

`Search`. This is an interactive protocol initiated by the user which involves the storage servers. The user provides a keyword as input. The protocol takes the cryptographic keys K generated from `Setup` to produce a search token *tok*. The token is used to extract the list of blocks and their locations from the index I. Next, the listed blocks are retrieved from the respective storage servers and returned to the user. Finally, the user decrypts the encrypted blocks and reconstructs the documents.

`Update`. This is the algorithm for adding (`Doc+`) or removing (`Doc-`) documents in the storage servers. The input to the algorithm is an object (e.g., a document D_i), and the operation to be executed on the object. Output of this algorithm is an updated index, and a set of encrypted blocks (for document addition). It can also be extended for adding or removing storage servers from the system.

The scheme is correct if for any set of documents and a keyword index table, the set of index entries and encrypted documents (or blocks) resulting from `Setup` and `Update` would consequently allow `Search` to identify documents associated to the searched keyword exactly as listed in the initial keyword index table.

3.3. Security Model

An SSE scheme aims to protect confidentiality of documents being stored in the storage servers, including the keywords of the documents. Consequently, the entries of the index and search token must be protected so that no information about the stored documents is revealed. Nevertheless some information leak is inevitable for `Search` to be efficient. This characteristic of the scheme is called the leakage profile as was first formalized in [19]. The leakage profile normally consists of the following three functions.

$\mathcal{L}^{setup}$ is information the server obtained by looking at the encrypted objects and index, such as the block size.

$\mathcal{L}^{query}$ is the information gathered by the server in observing messages in the `Search` protocol, such as the number of blocks associated to the input search token.

$\mathcal{L}^{update}$ is the information disclosed to server by the changes to index and set of encrypted objects on the server, including number of keywords of each added or removed object.

Definition 1. *A multi-server SSE scheme with leakage profile $\mathcal{L} = (\mathcal{L}^{setup}, \mathcal{L}^{query}, \mathcal{L}^{update})$ is said to be $\mathcal{L}$-secure against non-adaptive attack by colluding servers, if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that:*

$$|Pr[\mathbf{Ideal}_{SSE,\mathcal{A},\mathcal{S}}(\lambda) = 1] - Pr[\mathbf{Real}_{SSE,\mathcal{A}}(\lambda) = 1]| \leq \mathtt{negl}(\lambda) \tag{1}$$

where the **Real** *and* **Ideal** *games are defined as follows:*

Real *The adversary $\mathcal{A}$ chooses a set of documents D, a keyword index table, a set of servers S and a sequence of search and update queries, and submits them to the challenger. The challenger then runs* `Setup` *with the given input documents and keyword index table. Then the challenger runs* `Search` *on the sequence of search and update queries. All the responses are returned to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit.*

Ideal *The adversary $\mathcal{A}$ chooses a set of documents D, a keyword index table, a set of servers S and a sequence of search and update queries, and submits them to the challenger. The challenger gives the simulator*

$\mathcal{S}$ the leakage $\mathcal{L}^{setup}$,$\mathcal{L}^{query}$,$\mathcal{L}^{update}$ resulting from the given input. The result of the setup, search and update simulation is returned to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit.

For Definition 1, colluding servers means that storage servers share their leakage profile and other information gathered from the execution of the system among themselves. We further remark that for our proposed scheme in the following section (a simplified version compared to the proposal in [3]), the only keyword for each document is its filename and thus there is no keyword index table required, as in the above definition. This provides flexibility to include desired search functionalities (e.g., Boolean, conjunctive) on top of the filename search in our proposed scheme.

4. A Multi-Server SSE Scheme

Now, we describe our scheme implemented in the SDV API. Figure 3 provides detailed steps of the scheme. The scheme consists of three functions. There is a `Setup` algorithm, a `Search` protocol and an `Update` algorithm. In brief, a user submits a document and the `Setup` algorithm processes it, resulting in encrypted blocks $C = \{\mathbf{c}_1, \ldots \mathbf{c}_s\}$, an index table I, and secret keys K that must be stored securely by the users. The index table I stores the links between the filename, the blocks and the servers that store these blocks. In order to search for a document, a user submits a filename of a document D_i to the `Search` protocol, which then generates a filename token tok_i. The protocol checks the index table I to find whether the tok_i matches any of the entries. If yes, then the entry is retrieved. If not then it returns false. The entry contains the links from which servers all the encrypted blocks of the searched document can be retrieved. In our implementation, this entry is used by the SDV controller to retrieve the encrypted blocks, decrypt and combine them to form the document.

The `Update` algorithm provides for adding and removing of documents. Adding a document can be provided by using the processes in the `Setup` algorithm to encrypt the blocks and creating the index entry for the document. In a similar way, removing a document involves using the processes in the `Search` protocol to retrieve the index entry. However, instead of using the entry to retrieve the encrypted blocks, a delete command is issued to remove these blocks from the servers. We note that the mechanisms to submit and retrieve encrypted blocks are not discussed in the scheme. In our implementation we built a submit and a retrieve function in the SDV controller for this purpose.

Extension. The `Update` algorithm may be extended to allow for adding or removing a storage server S_a. A basic approach for server addition is to add the server identifier $id(S_a)$ into the list of existing servers S. The existing storage arrangements and index table I will not be modified. The new server S_a is involved for storing encrypted blocks only when a new document is submitted. It is a bit more complicated for removing a server, since it involves all the encrypted blocks stored in the server. A simplest approach is to retrieve all blocks from the server, and runs `Setup` to redistribute these blocks. Nevertheless, the index table I must be updated by removing all the entries to the server beforehand. This was also discussed in [3].

Expressive Search. The `Setup` algorithm generates index entries that can be queried based on filename, $F_{k_f}(filename(D_i))$. These entries are passed to the SDV controller to create an index table I, as was previously discussed in Figures 1 and 2. While a query based only on filename seems to be a limitation, there are two main benefits in our consideration of using such an approach. Firstly, users normally send and view their documents to and from a cloud storage in a folder mode, by browsing through the documents. This fits well with the query by filename approach. Secondly, and more importantly, by using such an index entry setting, it is thus possible to create a two-level query index. The base level index provides query on filename that pulls the documents from the cloud storage as was presented in our scheme. An upper level index can then be created to allow for any type of expressive search (e.g., range, substring, Boolean, conjunctive) based on keywords that return filenames.

Security Analysis

By referring to Definition 1 and the description of the scheme in Figure 3, we say that $\mathtt{SSE}_{SDV}$ is $\mathcal{L}$-secure against non-adaptive keyword attack by colluding servers, if the underlying pseudorandom function F and the symmetric encryption scheme $\mathcal{E}$ is IND-CPA-secure. In the following, we define the leakage profiles $\mathcal{L}$ based on an adversary $\mathcal{A}$ who controls all storage servers. This means the adversary sees leakages from all storage servers involved in the system.

$\mathtt{SSE}_{SDV}$ = (`Setup`, `Search`, `Update`)

`Setup`
Input: A set of n documents $\{D_1, \ldots, D_n\}$, filenames of the documents (as keyword index), a list of s server identifiers $\{S_1, \ldots, S_s\}$.
Output: A set of secret keys K, a set of encrypted blocks C and an index table I.

1. Generate a master key mk and for $j = 1$ to s, $s = |S|$: $k_j \leftarrow \mathcal{E}.\mathtt{Gen}(1^\lambda)$.
2. Set $K = (mk, (k_1, \ldots, k_s))$.
3. Initialize empty index I and a set of list $C = \{\mathbf{c}_1, \ldots \mathbf{c}_s\}$.
4. Create temporary empty list $\mathbf{t}$.
5. For $i = 1$ to n:
   (a) Divide D_i into blocks: $D_i = \{d_{i,1}||1, d_{i,2}||2, \ldots, d_{i,h}||h\}$.
   (b) Generate keyword token $tok_i \leftarrow F(mk, \text{filename}(D_i))$.
   (c) For $x = 1$ to h:
   - Randomly select a cloud server: $j \xleftarrow{R} [s]$, $s = |S|$.
   - Encrypt $d_{i,x}||x$: $e_{d_{i,x}} \leftarrow \mathcal{E}.\mathtt{Enc}(k_j, d_{i,x}||x)$.
   - Store $id(e_{d_{i,x}})$ and $e_{d_{i,x}}$ in $\mathbf{c}_j$.
   - Append $id(e_{d_{i,x}})||j$ in $\mathbf{t}$.

   (d) Set $I[tok_i] = \mathbf{t}$.
6. Output K, C and I.

`Search`
Input: A keyword ($filename$).
Output: A sequence of tuple (block identifier, server identifier) or false.

1. SDV API: $tok_i \leftarrow F(mk, \text{filename}(D_i))$.
2. SDV controller: If tok_i does not match any entry in I, return $\perp$, else return $I[tok_i]$.

`Update`
Input: An operation `op` (add or remove), optionally an input object (a filename), the index table I and the set of server identifiers.
Output: True with the updated index I', or false (for unsuccessful addition or removal).

1. For `op` = `Doc+`, `in` = $filename(D_a)$, execute Step 4 and Step 5 of `Setup` and update I by adding $I[tok_a]$.
2. For `op` = `Doc-`: `in` = $\text{filename}(D_x)$, execute `Search` and update I by deleting $I[tok_x]$.
3. Output I' as the updated index I.

Figure 3. $\mathtt{SSE}_{SDV}$: Description of the scheme.

$\mathcal{L}^{setup} = (|B|, b)$: For setup, the information leaked to the adversary is the number of blocks of a server $|B|$ and the block size of the encrypted blocks, where the block size is denoted as b. A fixed block size means the storage servers cannot even study the difference in size of the encrypted blocks to learn which document a block belongs to. However, in our current implementation, we use a range of different block sizes so that it is efficient in generating the index entries and submission of encrypted blocks of large documents. This means the storage servers learn different sizes of encrypted blocks.

$\mathcal{L}^{query} = |B_q|$: For search, the adversary learns the number of encrypted blocks associated to the input keyword token. This is because the SDV controller sends retrieval requests to the cloud storage servers to retrieve encrypted blocks of a document. Here, we denote the number of retrieved encrypted blocks as $|B_q|$.

$\mathcal{L}^{update} = |B_q|$: Adding and removing a document includes submitting the encrypted blocks, or removing the encrypted blocks of a document from the cloud storage servers. Thus, the leakage is similar to $\mathcal{L}^{query}$, where the adversary learns the number of encrypted blocks associated to the document.

Given the above leakage, we now define the game and how to construct a simulator, $\mathcal{S}$. In a game for non-adaptive adversary, the adversary submits its sequence of queries to the challenger all at once. The sequence must begin with input for setup, and subsequent queries are search query, add document query or remove document query.

For any adversary $\mathcal{A}$, consider the simulator $\mathcal{S}$ defined below. From $\mathcal{L}^{setup} = (|B|, b)$, $\mathcal{S}$ generates $|B|$ random binary strings with length b. These random strings becomes $\mathcal{S}$'s set of unnamed blocks.

Then, by referring to search queries, $\mathcal{S}$ is given $\mathcal{L}^{query} = |B_q|$. $\mathcal{S}$ checks whether the queried block identifiers exists in its set of labeled blocks. If the identifiers have not been used, $\mathcal{S}$ chooses $|B_q|$ unnamed blocks and labels them with the block identifiers submitted by the SDV controller.

Similarly, $\mathcal{S}$ refers to file removal queries and use $\mathcal{L}^{update} = |B_q|$. If there is no encrypted block labeled with the list of block identifiers to be deleted from storage, $\mathcal{S}$ chooses $|B_q|$ unnamed encrypted blocks and labels them accordingly.

For file addition queries with given $\mathcal{L}^{update} = |B_q|$, $\mathcal{S}$ generates $|B_q|$ binary strings of length b and labels them with the submitted block identifiers.

Finally, $\mathcal{S}$ labels all other unnamed encrypted blocks with random binary string of length identical to the length of the block identifiers. Since the encryption scheme $\mathcal{E}$ used in the SDV is IND-CPA, the random binary strings generated by $\mathcal{S}$ are indistinguishable from the encrypted blocks generated by the SDV scheme. Thus, using the set of labeled blocks, the transcript for the query sequence produced by $\mathcal{S}$ is indistinguishable from the transcript produced by the SDV scheme.

We note that the above security analysis is based on the assumption that the SDV controller resides on a trusted query gateway (proxy server). This means the index table I is stored at the gateway and thus leakage due to index is not factored in. In the case where the query gateway is assumed to be semi-honest (honest-but-curious), then the leakage profile of the index table must be considered. It means leakage of $\mathcal{L}^{query} = |B_q|$ even *before* queries, compared to the above analysis where this leakage happens only *after* queries. We further remark that, if an additional query module with more expressive search is deployed, the leakage due to keywords, instead of just leakage due to filename, on the overall SDV system will be at least the leakage profile of the deployed query module.

5. Implementation

In this section, we describe in detail the implementation of the SDV prototype, as well as its performance evaluation. The SDV controller is implemented using Java Servlet on Apache Tomcat version 8. It is hosted in a virtual machine under MIMOS cloud platform. The virtual machine runs on Ubuntu 14.04 as the operating system, with 4GB memory. The SDV API with the implementation of the multi-server SSE scheme, $\mathtt{SSE}_{SDV}$, is developed using Java and C++. The web interface for the SDV App, on the other hand, is based on JSP. MySQL is deployed as the underlying database. We incorporated a local storage server and Google drive as the cloud storage servers for our current deployment. We note that the number of servers is configurable.

5.1. Cryptograhic Building Blocks

We make use of the Advanced Encryption Standard in counter mode (AES-128-CTR) for encryption and decryption, a message authentication code based on SHA256 (HMAC-SHA256) for generating the keyword token, and hash function SHA256 for creating block identifiers.

5.2. Main User Interface

Figure 4 shows the main interface for submitting and retrieving encrypted documents under the SDV system. A user uploads a document through drag-and-drop to the area at the bottom of the page, or click on the button, which pops up a document selection window. Uploaded (or submitted) documents are shown in a list, with a download button for each document being shown on the far right of the filename. Currently the SDV system prototype with this interface is integrated into the MIMOS internal workflow system as a test environment, in which the SDV dashboard is only accessible once the user is authenticated through a unified authentication platform [4].

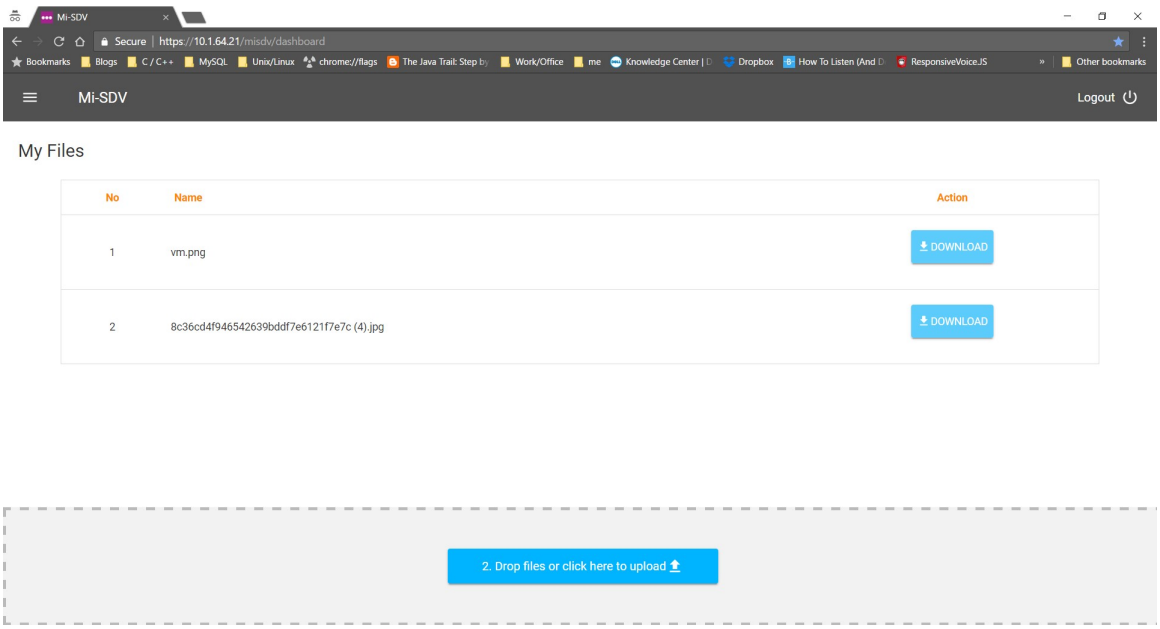


Figure 4. SDV dashboard: Document submission and retrieval.

5.3. Submitting a Document

Figure 5 illustrates the details of the operations conducted by the SDV API for document submission. During `Setup`, a master key is generated using a pseudorandom number generator (PRNG) and a random seed value. This master key is used to derive the server keys for the cloud storage servers. Server keys are derived using HMAC-SHA256 and the master key, with the server identifiers as inputs. When a document is sent in via the SDV App, the API divides them into blocks. The block size is a user configurable parameter, e.g., a 100 MB file is divided into 100 blocks if the block size configured is 1 MB per block.

As mentioned previously, for our prototype only the filename acts as the keyword token. This token is generated through HMAC-SHA256 using the master key and the filename as input. Once the document is divided, every block is randomly assigned to one of the available storage servers. The assignment information is recorded as part of the index entry. The assigned block is then encrypted with the corresponding server key. Finally, the encrypted blocks and the index entry are sent to the SDV controller, which then proceed to process and upload the encrypted blocks to the designated cloud storage servers. All encrypted blocks will then be hashed with SHA256 to produce the block identifier, which will be used in the index to locate the block during retrieval. As an example, Figures 6 and 7 present the execution steps of submitting a 5 MB file to Google Drive and a local storage server. The file is listed (Figure 7a) once it is successfully uploaded (Figure 6b). The submissions to Google drive and a local server through the SDV controller are shown in Figure 7b.

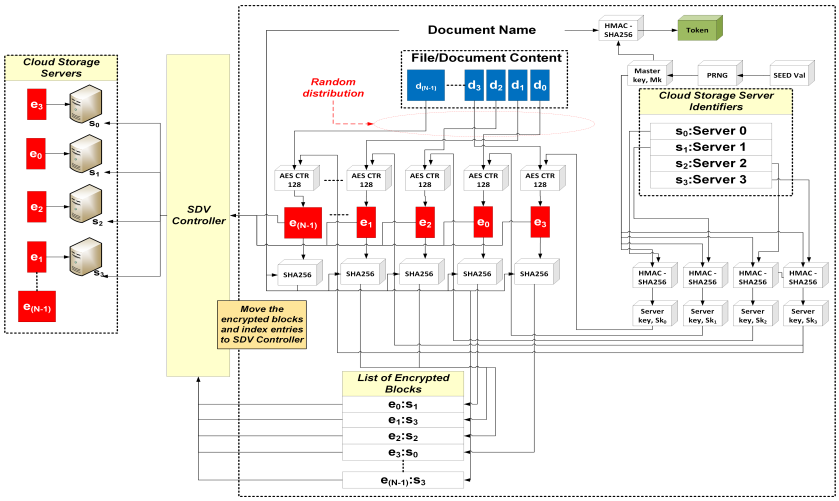
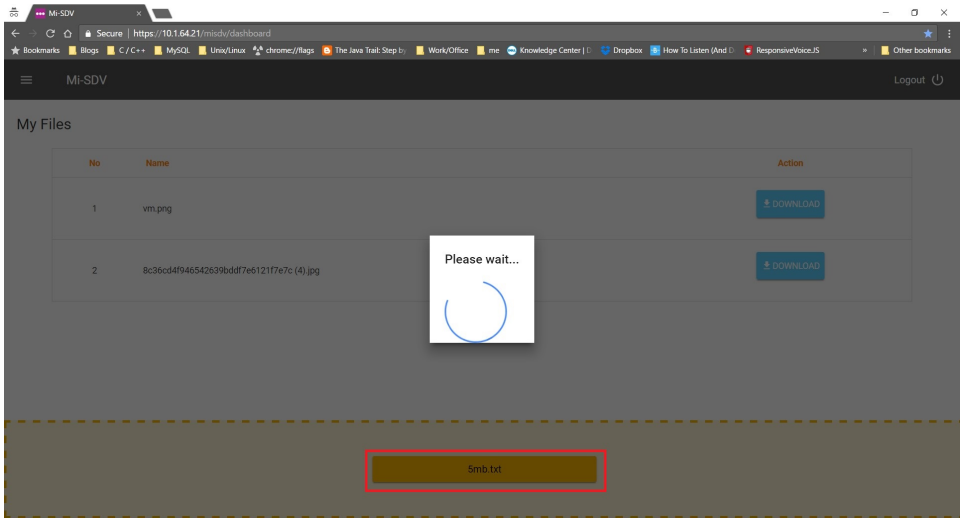
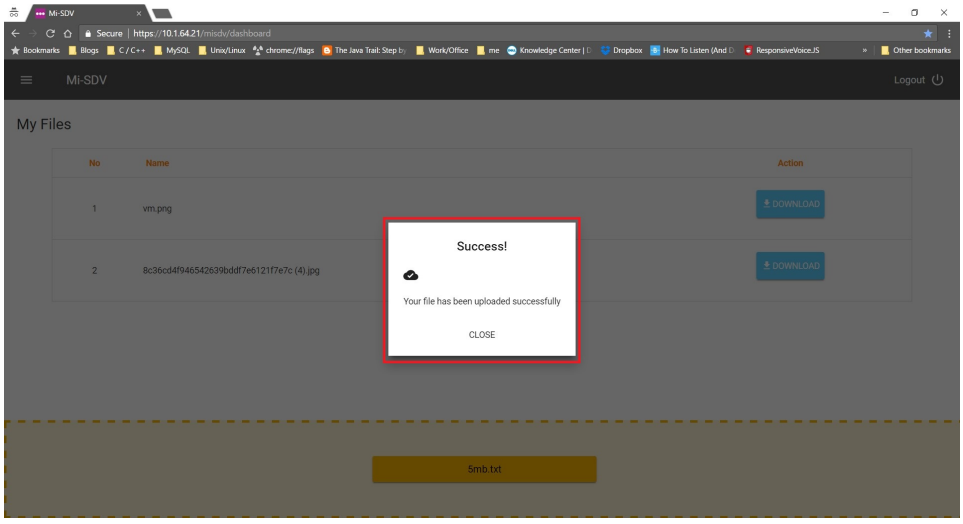


Figure 5. SDV API: How document submission is performed (e.g., given four cloud storage servers). AES-CTR: Advanced Encryption Standard in counter mode.

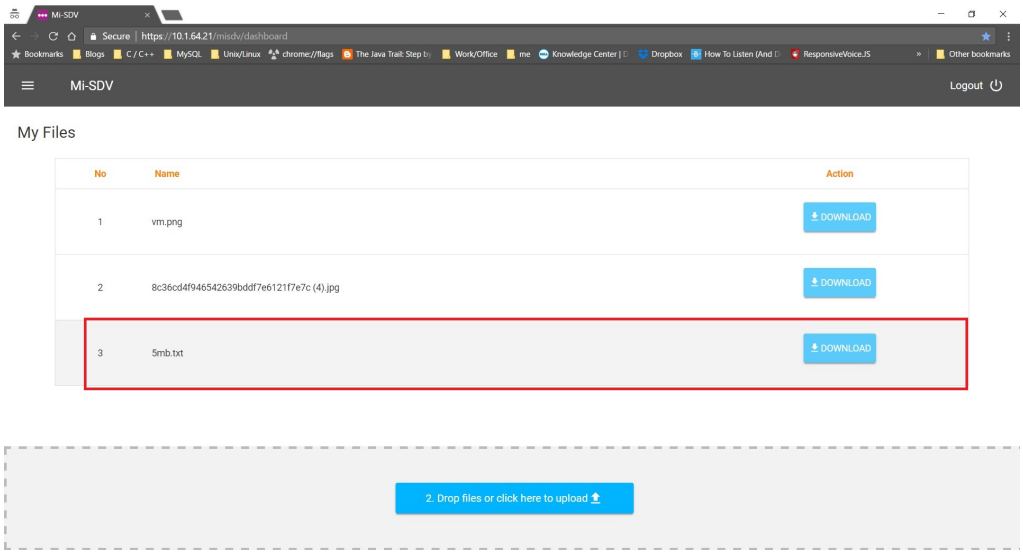


(**a**)

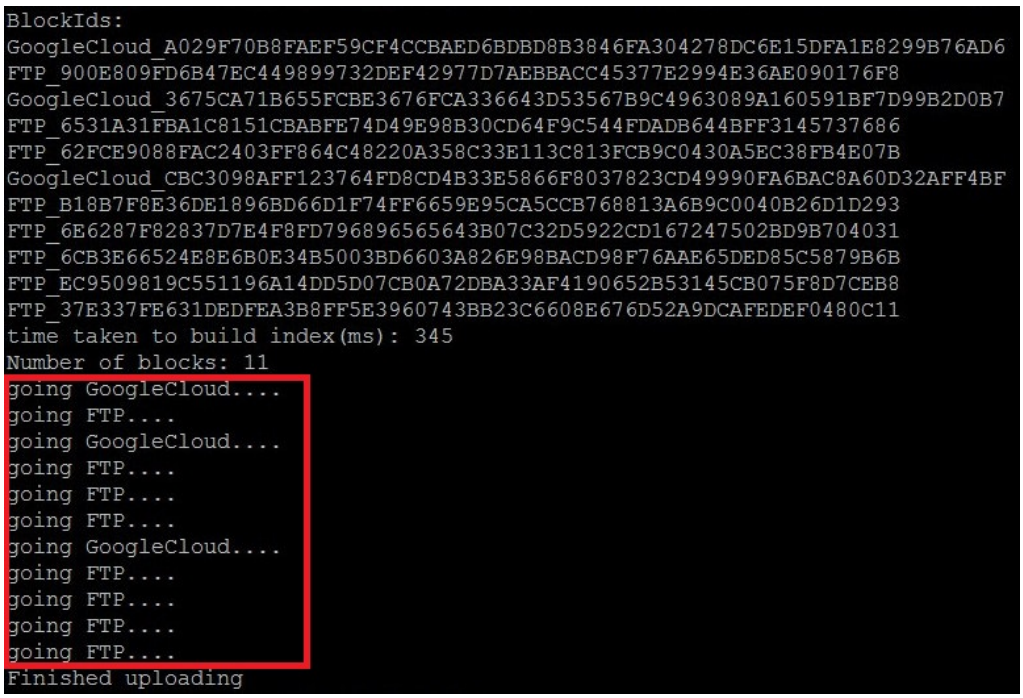


(**b**)

Figure 6. Submitting a document. (**a**) Uploading file; (**b**) Upload successful.



(**a**)



(**b**)

Figure 7. Submitting a document. (**a**) Uploaded file listed; (**b**) Upload: server log showing block identifiers and submission of blocks to Google Drive and local server.

5.4. Searching and Retrieving a Document

The user chooses the document to be retrieved through the SDV App. The filename is forwarded to the API via the SDV controller, where a keyword token is then generated and compared through the index table to check if the uploaded file exists. If a match is found, the block identifiers and corresponding servers where they are located are retrieved. The SDV controller retrieves the encrypted blocks from their servers and instructs the SDV API to decrypt these blocks with the corresponding server keys. The decrypted blocks are reassembled into the document with the filename chosen by the user. Details of file retrieval are likewise illustrated in Figure 8, while the steps for downloading a file are presented in Figures 9 and 10.

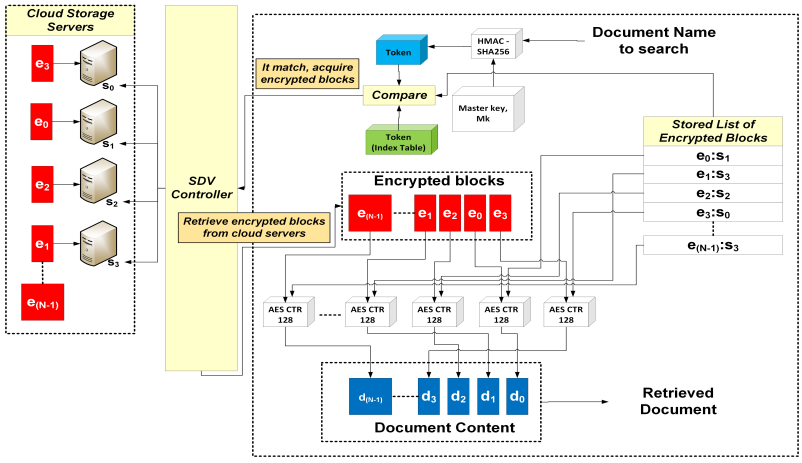
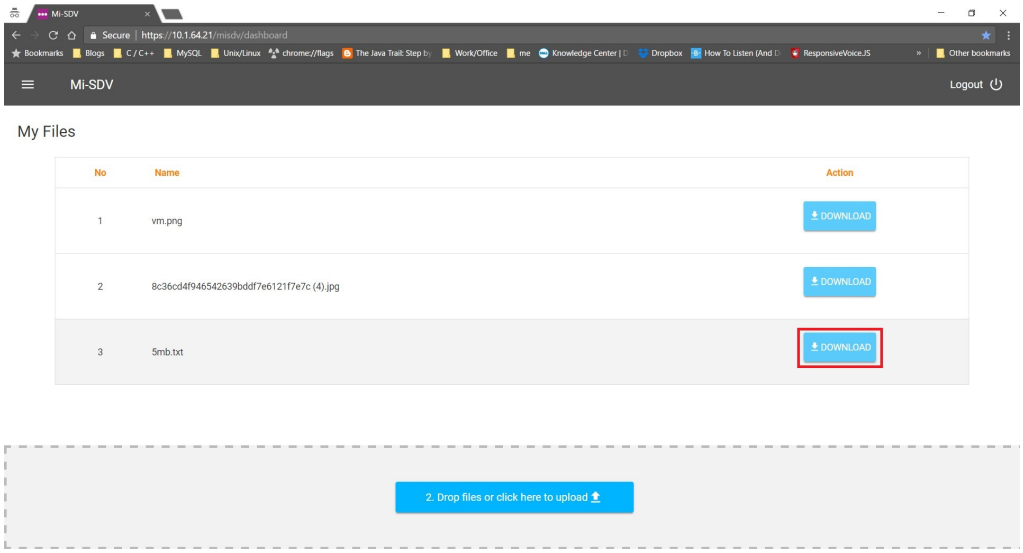
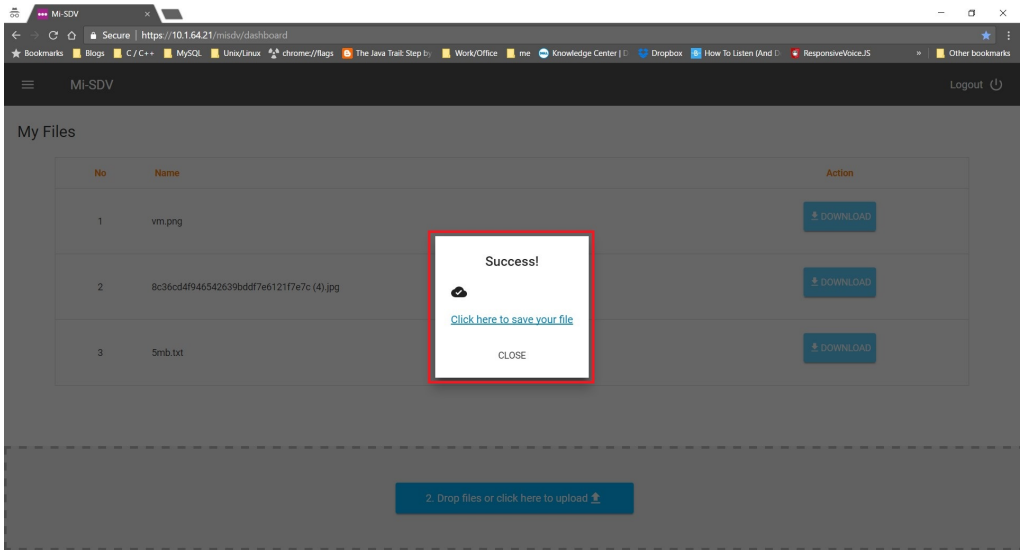


Figure 8. SDV API: How document retrieval is performed (e.g., given four cloud storage servers).

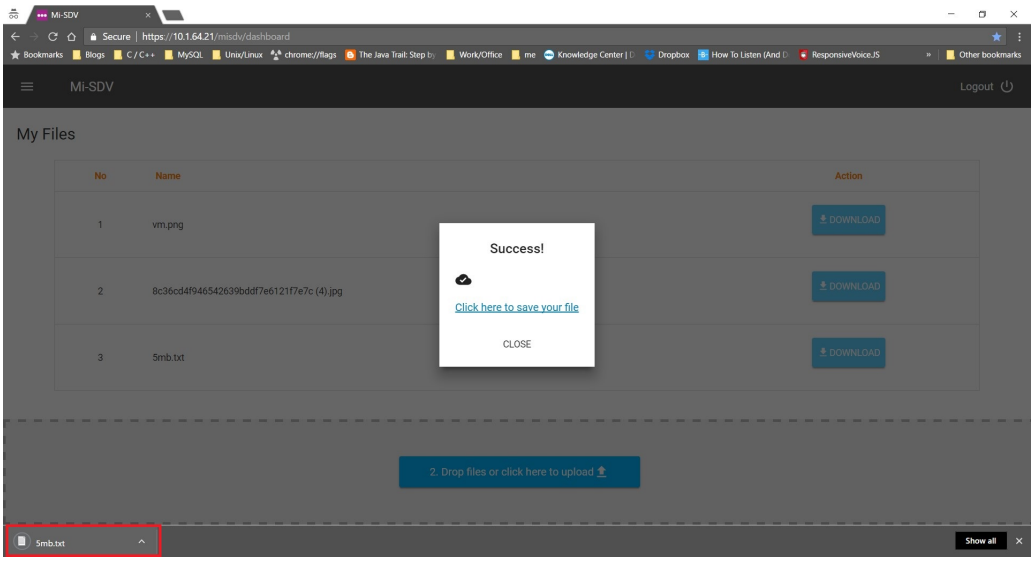


(**a**)



(**b**)

Figure 9. Retrieving a document. (**a**) Select file to download; (**b**) Download successful.



(**a**)

```
Downloading...
Number of blocks: 11
BlockIds:
GoogleCloud_A029F70B8FAEF59CF4CCBAED6BDBD8B3846FA304278DC6E15DFA1E8299B76AD6
FTP_900E809FD6B47EC449899732DEF42977D7AEBBACC45377E2994E36AE090176F8
GoogleCloud_3675CA71B655FCBE3676FCA336643D53567B9C4963089A160591BF7D99B2D0B7
FTP_6531A31FBA1C8151CBABFE74D49E98B30CD64F9C544FDADB644BFF3145737686
FTP_62FCE9088FAC2403FF864C48220A358C33E113C813FCB9C0430A5EC38FB4E07B
GoogleCloud_CBC3098AFF123764FD8CD4B33E5866F8037823CD49990FA6BAC8A60D32AFF4BF
FTP_B18B7F8E36DE1896BD66D1F74FF6659E95CA5CCB768813A6B9C0040B26D1D293
FTP_6E6287F82837D7E4F8FD796896565643B07C32D5922CD167247502BD9B704031
FTP_6CB3E66524E8E6B0E34B5003BD6603A826E98BACD98F76AAE65DED85C5879B6B
FTP_EC9509819C551196A14DD5D07CB0A72DBA33AF4190652B53145CB075F8D7CEB8
FTP_37E337FE631DEDFEA3B8FF5E3960743BB23C6608E676D52A9DCAFEDEF0480C11
going GoogleCloud....
going FTP....
going GoogleCloud....
going FTP....
going FTP....
going GoogleCloud....
going FTP....
going FTP....
going FTP....
going FTP....
going FTP....
Finished downloading
```

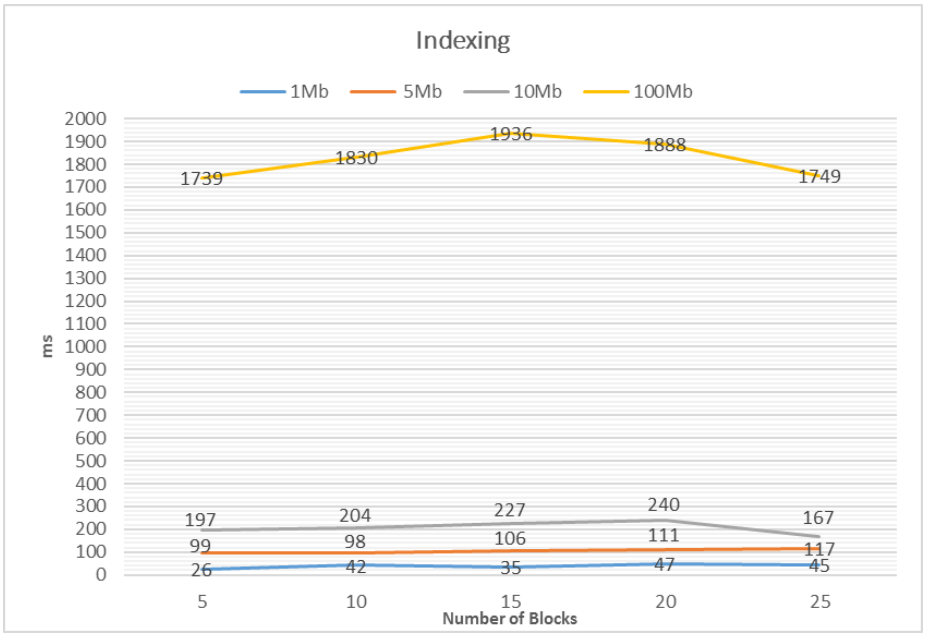
(**b**)

Figure 10. Retrieving a document. (**a**) Downloaded file saved; (**b**) Download: Server log showing block identifiers and retrieval of blocks from Google Drive and local server.

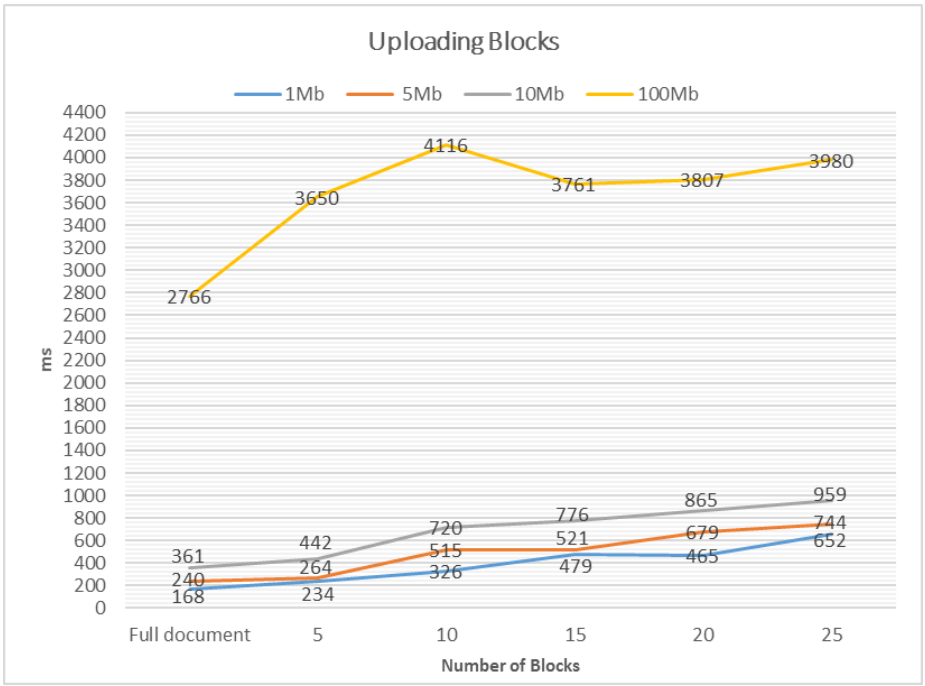
5.5. Performance

For performance, we measure the computation time for building the index table (Figure 11a) and for uploading the encrypted blocks (Figure 11b), as compared to directly uploading the original document unencrypted. This gives us the computation overhead due to the SDV system, and how practical it is. The uploading process is simulated by submitting blocks and documents through a FTP server, so that we do not have to take into account Internet connection delays. However, we note that the current deployed prototype does distribute encrypted blocks to both local storage and third-party cloud servers (i.e., Google Drive) and can be configured to add more servers. We divided a document

into 5, 10, 15, 20, or 25 blocks, and applied them to four different text documents with increasing file sizes: 1 MB, 5 MB, 10 MB and 100 MB. Table 2 shows the test data for 5 blocks and 25 blocks, while the detailed results are displayed in the graphs in Figure 11. The time measured is in milliseconds (ms).



(**a**)



(**b**)

Figure 11. Performance results. (**a**) Indexing; (**b**) Uploading Blocks.

From Figure 11a, we may summarize that computation time of index generation grows linearly with the size of the document, which is an expected behavior. However, the number of blocks seems to have a marginal effect on computation to generating indexes, at least in the range of 5–25 blocks. In contrast, the number of blocks affects the efficiency of uploading a document to the storage server,

as can be observed in Figure 11b, where "Full document" denotes an original unencrypted document. Given the time required grows linearly with the number of blocks when uploading these blocks, a smaller number of blocks per document is desirable. However, there is a question as to whether such a configuration leaks more information as compared to having a large number of blocks.

On the other hand, if we fix the block size, then the number of blocks grows linearly with the size of the document. Uploading blocks of a large document will thus incur larger overhead. For instance, we set the block size to 500 KB in our experiment. This means for a 100 MB document, 200 blocks were generated and it takes 2 s to build the index and 7.2 s to upload the document.

Table 2. Performance data for 5 blocks and (25 blocks) in milliseconds.

	1 MB [1 mb.txt]	**5 MB [5 mb.txt]**	**10 MB [10 mb.txt]**	**100 MB [100 mb.txt]**
Indexing	26 (45)	99 (117)	197 (167)	1739 (1749)
Uploading blocks	234 (465)	264 (679)	442 (865)	3650 (3807)
Uploading full document	168	240	361	2766

6. Discussions

We further discuss implementation issues and how some of these issues present a challenge in terms of balancing the requirements between security and efficiency.

6.1. Number of Servers

The SDV system allows one to configure the number of cloud storage servers. For the current deployment, a local storage server and Google Drive are used. In the near future we may add Dropbox, Amazon S3, etc. Each of these services has different development interface. For SDV to be able to support more cloud storage servers, the challenge is to define a common functional interface that is able to interact with these service providers.

6.2. Generating Block Identifiers

As discussed in Sections 4 and 5, a main contribution of SDV is that documents are broken into many blocks, and the encrypted blocks are submitted to the cloud storage servers. In order to be able to identify each single block, an identifier (or a filename) must be assigned to the block. Different blocks may contain identical content and we must avoid name collision and ensure uniqueness of each identifier. Hence we derive the name of the block by hashing the first few bytes together with a timestamp using SHA256. The reason we do not hash the entire block is due to computation efficiency, especially if a block is configured to be of large size (e.g., 50 MB).

6.3. Block Size

In terms of information leakage at the storage servers, a fixed block size for all documents would be ideal, since storage servers would not be able to infer any information based on the length of a block. However, for better performance (as discussed in Section 5.5), different block sizes are desirable, so as to cater for efficient indexing and uploading especially for large documents. The optimal range of block sizes for documents with different sizes, and how this will affect the data privacy assurance of the underlying multi-server SSE scheme in terms of leakage remain to be studied. A potential direction would be to examine existing block size mechanisms for storage allocation in a large storage system such as Ceph [36].

6.4. Limitation

The current SDV system prototype provides a user to upload and download documents in a privacy-preserving manner, through an interface that allows the user to browse through the list of documents. In other words, query (or search) is by filename. This somehow limits the capability of the system. The next development iteration would thus be adding a query module with various search functionalities on top of the current query index table.

7. Conclusions

In this work, we have developed and implemented the SDV system prototype that runs on a multi-server SSE scheme, and showed that encrypting documents into blocks and uploading to several cloud storage providers entails acceptable cost overhead compared to uploading original files unencrypted. Our system is easy to use and robust, with an intuitive web interface that is compatible with any browser. It solves the problem of privacy and searchability of encrypted documents with minimal leakage as no single cloud service provider will have the complete document block set.

For future work, we plan to improve the robustness and scalability of the system by allowing users to remove cloud service providers easily. In addition, we will continue to study the issue of block size and its implications for information leakage. Also, the current system setup is more suitable for use in an organization environment, where an organization maintains a gateway containing the SDV controller. In view of this, we want to examine a client-side implementation that potentially stores the keys and index table on the client side, therefore providing another approach that is user-centric. This means that submitting and retrieving documents is performed through direct interaction between the user devices and the cloud storage servers without any intermediary.

Acknowledgments: The authors thank the anonymous reviewers for their valuable feedback.

Author Contributions: Geong Sen Poh and Moesfa Soeheila Mohamad conceived and designed the underlying scheme; Ji-Jian Chin, Vishnu Monn Baskaran and Kay Win Lee implemented the scheme; Kay Win Lee and Dharmardharshni Maniam designed and implemented the system; Geong Sen Poh and Muhammad Reza Z'aba wrote most parts of the paper. All authors have read and approved the final manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Netwrix. 2016 Cloud Security Report, 2016. Available online: https://www.netwrix.com/2016cloud_security_report.html (accessed on 6 May 2017).
2. Cisco Systems, Inc. *Cisco Global Cloud Index: Forecast and Methodology, 2015–2020*; Cisco Systems, Inc.: San José, CA, USA, 2016.
3. Poh, G.S.; Mohamad, M.S.; Chin, J.J. Searchable Symmetric Encryption over Multiple Servers. In Proceedings of the Arctic Crypt, Longyearbyen, Norway, 17–22 July 2016.
4. Seak, S.C.; Siong, N.K.; Loon, W.H.; Haron, G.R. A Centralized Multimodal Unified Authentication Platform for Web-based Application. In Proceedings of the World Congress on Engineering and Computer Science (WCECS) 2014, San Francisco, CA, USA, 22–24 October 2014; pp. 157–172.
5. Cash, D.; Jarecki, S.; Jutla, C.S.; Krawczyk, H.; Rosu, M.C.; Steiner, M. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO 2013*; Canetti, R., Garay, J.A., Eds.; Springer: New York, NY, USA, 2013; Volume 8042, pp. 353–373.
6. Cao, N.; Wang, C.; Li, M.; Ren, K.; Lou, W. Privacy-Preserving Multi-Keyword Ranked Search over Encrypted Cloud Data. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *25*, 222–233.
7. Popa, R.A.; Redfield, C.M.S.; Zeldovich, N.; Balakrishnan, H. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP 2011*; Wobber, T., Druschel, P., Eds.; ACM: New York, NY, USA, 2011; pp. 85–100.
8. Song, D.X.; Wagner, D.; Perrig, A. Practical Techniques for Searches on Encrypted Data. In Proceedings of the IEEE Symposium on Security and Privacy (S&P 2000), Berkeley, CA, USA, 14–17 May 2000; p. 44.

9. Lau, B.; Chung, S.P.; Song, C.; Jang, Y.; Lee, W.; Boldyreva, A. Mimesis Aegis: A Mimicry Privacy Shield-A System's Approach to Data Privacy on Public Cloud. In Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 33–48.
10. Pappas, V.; Krell, F.; Vo, B.; Kolesnikov, V.; Malkin, T.; Choi, S.G.; George, W.; Keromytis, A.D.; Bellovin, S. Blind Seer: A Scalable Private DBMS. In Proceedings of the IEEE Symposium on Security and Privacy (S&P 2014), San Jose, CA, USA, 18–21 May 2014; pp. 359–374.
11. Fisch, B.A.; Vo, B.; Krell, F.; Kumarasubramanian, A.; Kolesnikov, V.; Malkin, T.; Bellovin, S.M. Malicious-Client Security in Blind Seer: A Scalable Private DBMS. In Proceedings of the IEEE Symposium on Security and Privacy (S&P 2015), San Jose, CA, USA, 17–21 May 2015; pp. 395–410.
12. Popa, R.A.; Stark, E.; Valdez, S.; Helfer, J.; Zeldovich, N.; Balakrishnan, H. Building Web Applications on Top of Encrypted Data Using Mylar. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, Seattle, WA, USA, 2–4 April 2014; pp. 157–172.
13. Networks, S. Skyhigh Networks: Cloud Security Software, 2016. Available online: https://www.skyhighnetworks.com/ (accessed on 6 May 2016).
14. CipherCloud. CipherCloud: Enterprise Cloud Security, 2016. Available online: https://www.ciphercloud.com/ (accessed on 6 May 2016).
15. Bitglass. Bitglass: Cloud Access Security Broker, 2016. Available online: http://www.bitglass.com/ (accessed on 6 May 2016).
16. Cash, D.; Grubbs, P.; Perry, J.; Ristenpart, T. Leakage-Abuse Attacks Against Searchable Encryption. In *ACM CCS 2015*; Ray, I., Li, N., Kruegel, C., Eds.; ACM: New York, NY, USA, 2015; pp. 668–679.
17. Goh, E.J. Secure Indexes. IACR Cryptology ePrint Archive, Report 2003/216, 2003. Available online: http://eprint.iacr.org/2003/216/ (accessed on 6 May 2016).
18. Chang, Y.C.; Mitzenmacher, M. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS 2005*; Ioannidis, J., Keromytis, A.D., Yung, M., Eds.; Springer: New York, NY, USA, 2005; Volume 3531, pp. 442–455.
19. Curtmola, R.; Garay, J.A.; Kamara, S.; Ostrovsky, R. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *ACM CCS 2006*; Juels, A., Wright, R.N., di Vimercati, S.D.C., Eds.; ACM: New York, NY, USA, 2006; pp. 79–88.
20. Chase, M.; Kamara, S. Structured Encryption and Controlled Disclosure. In *ASIACRYPT 2010*; Abe, M., Ed.; Springer: New York, NY, USA, 2010; Volume 6477, pp. 577–594.
21. Kamara, S.; Papamanthou, C. Parallel and Dynamic Searchable Symmetric Encryption. In *FC'13*; Sadeghi, A.R., Ed.; Springer: New York, NY, USA, 2013; Volume 7859, pp. 258–274.
22. Kamara, S.; Papamanthou, C.; Roeder, T. Dynamic Searchable Symmetric Encryption. In *ACM CCS'12*; Yu, T., Danezis, G., Gligor, V.D., Eds.; ACM: New York, NY, USA, 2012; pp. 965–976.
23. Cash, D.; Jaeger, J.; Jarecki, S.; Jutla, C.S.; Krawczyk, H.; Rosu, M.C.; Steiner, M. Dynamic Searchable Encryption in Very Large Databases: Data Structures and Implementation. In Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA, 23–26 February 2014.
24. Kurosawa, K.; Ohtaki, Y. How to Construct UC-Secure Searchable Symmetric Encryption Scheme. 2015. Available online: https://pdfs.semanticscholar.org/bdbb/d27c0cda8f05419565cfc20b8ce953515047.pdf (accessed on 9 May 2017).
25. Mohamad, M.S.; Poh, G.S. Verifiable Structured Encryption. In *Inscrypt'12*; Kutylowski, M., Yung, M., Eds.; Springer: New York, NY, USA, 2012; Volume 7763, pp. 137–156.
26. Naveed, M.; Prabhakaran, M.; Gunter, C.A. Dynamic Searchable Encryption via Blind Storage. In Proceedings of the IEEE Symposium on Security and Privacy (S&P 2014), San Jose, CA, USA, 18–21 May 2014; pp. 639–654.
27. Stefanov, E.; Papamanthou, C.; Shi, E. Practical Dynamic Searchable Encryption with Small Leakage. In Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA, 23–26 February 2014.
28. Bösch, C.; Peter, A.; Leenders, B.; Lim, H.W.; Tang, Q.; Wang, H.; Hartel, P.H.; Jonker, W. Distributed Searchable Symmetric Encryption. In Proceedings of the Twelfth Annual Conference on Privacy, Security and Trust (PST), Toronto, ON, Canada, 23–24 July 2014; pp. 330–337.
29. Ishai, Y.; Kushilevitz, E.; Lu, S.; Ostrovsky, R. Private Large-Scale Databases with Distributed Searchable Symmetric Encryption. *IACR Cryptol. ePrint Arch.* **2015**, *2015*, 1190.

30. Kuzu, M.; Islam, M.S.; Kantarcioglu, M. Distributed Search over Encrypted Big Data. In *ACM CODASPY 2015*; Park, J., Squicciarini, A.C., Eds.; ACM: New York, NY, USA, 2015; pp. 271–278.
31. Boneh, D.; Crescenzo, G.D.; Ostrovsky, R.; Persiano, G. Public Key Encryption with Keyword Search. In *EUROCRYPT 2004*; Cachin, C., Camenisch, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3027, pp. 506–522.
32. Gentry, C. A Fully Homomorphic Encryption Scheme. Ph.D. Thesis, Stanford University, Stanford, CA, USA, 2009.
33. Goldreich, O.; Ostrovsky, R. Software Protection and Simulation on Oblivious RAMs. *J. ACM* **1996**, *43*, 431–473.
34. Stefanov, E.; Shi, E. ObliviStore: High Performance Oblivious Cloud Storage. In Proceedings of the IEEE Symposium on Security and Privacy (S&P 2013), San Francisco, CA, USA, 19–22 May 2013; pp. 253–267.
35. Bösch, C.; Hartel, P.; Jonker, W.; Peter, A. A Survey of Provably Secure Searchable Encryption. *ACM Comput. Surv.* **2014**, *47*, 18.
36. Storage, I. Ceph. Available online: http://ceph.com/ (accessed on 6 May 2017).