SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

August 2021, Vol. 64 182314:1–182314:18
https://doi.org/10.1007/s11432-020-3039-x

Cetus: an efficient symmetric searchable encryption against file-injection attack with SGX

Yanyu HUANG[1], Siyi LV[1], Zheli LIU[1*], Xiangfu SONG[2], Jin LI[3], Yali YUAN[4] & Changyu Dong[5]

[1]*College of Cyber Science and the College of Computer Science, Tianjin Key Laboratory of Network and Data Security Technology, Nankai University, Tianjin* 300350*, China;*
[2]*School of Computer Science and Technology, Shandong University, Jinan* 250100*, China;*
[3]*School of Computer Science, Guangzhou University, Guangzhou* 510006*, China;*
[4]*Department of Computer Science, University of Göttingen, Göttingen* 37073*, Germany;*
[5]*School of Computing, Newcastle University, Newcastle upon Tyne* NE1 7RU*, UK*

Received 2 April 2020/Revised 18 June 2020/Accepted 7 August 2020/Published online 9 July 2021

Abstract Symmetric searchable encryption (SSE) allows the users to store and query their private data in the encrypted database. Many SSE schemes for different scenarios have been proposed in the past few years, however, most of these schemes still face more or fewer security issues. Using these security leakages, many attacks against the SSE scheme have been proposed, and especially the non-adaptive file injection attack is the most serious. Non-adaptive file injection attack (NAFA) can effectively recover some extremely important private information such as keyword plaintext. As of now, there is no scheme that can effectively defend against such attacks. We first propose the new security attribute called toward privacy to resist non-adaptive file injection attacks. We then present an efficient SSE construction called Cetus to achieve toward privacy. By setting up a buffer and designing the efficient oblivious reading algorithm based on software guard extensions (SGX), we propose the efficient one-time oblivious writing mechanism. Oblivious writing protects the update pattern and allows search operations to be performed directly on the data. The experiment results show that Cetus achieves $O(a_w)$ search time and $O(1)$ update communication. The practical search time, communication, and computation overheads incurred by Cetus are lower than those of state-of-the-art.

Keywords searchable encryption, SGX technique, file injection attack, forward/toward privacy, cloud databases

Citation Huang Y Y, Lv S Y, Liu Z L, et al. Cetus: an efficient symmetric searchable encryption against file-injection attack with SGX. Sci China Inf Sci, 2021, 64(8): 182314, https://doi.org/10.1007/s11432-020-3039-x

1 Introduction

Symmetric searchable encryption (SSE) is a cryptography primitive that allows the client to issue queries in a ciphertext database. In detail, the client can outsource a private database to an untrusted server in the form of ciphertext, and then issue queries such as update and search. The client can add, delete, and search their own data according to their needs. In the whole process, the server should have no idea about the content of the documents under the protection of traditional encryption schemes. Notice that all keywords are encrypted and protected by trapdoors, so the server has no way to infer the contents of the file from enough plaintext of the keyword. In actual application scenarios, the server can obtain some meta-information such as the frequency of keywords to be searched. Considering efficiency and availability, these leakages are usually accepted by default with relaxing privacy requirements. Nonetheless, the abuse of leakages becomes a serious loophole to the security of searchable encryption schemes.

File injection attack and its destructiveness. File injection attacks are typical attacks that exploit the abuse of leakages to recover the plaintext of keywords. These attacks focus on the leakage of file-access patterns that occurred during the process of searching the documents after they are inserted.

* Corresponding author (email: liuzheli@nankai.edu.com)

The server can inject some well-designed files into the encrypted database and each keyword corresponds to a specific set of files. It is common and realistic in some scenarios, such as the mail server which can send mail to the client. When the client issues a search query and retrieves the documents associated with the keyword, the server could observe the query result and learn whether the injection files are contained. If a keyword is included in the injection files of search results but is not included in injection files that are not be searched, then the server can determine the plaintext of the queried keyword.

File injection attacks are extremely destructive for the security requirement because they can recover the plaintext of keywords accurately. The server as an adversary can learn most keywords by injecting a few files. Many studies [1–11] which achieve forward privacy could prevent the server from gaining prior information like previous queries. However, they could only resist adaptive file injection attacks which are weak injection attacks. Non-adaptive file injection attack, as an attack that does not require any prior information, can effectively use a small amount of injected files to recover a large number of plaintext keywords. So far, all SSE solutions have been unable to resist such attacks completely.

Even oblivious mechanisms such as oblivious RAM (ORAM) [1–3] can restrict stricter security privacy to protect the file-access pattern, the cumbersome reading, writing, and obfuscation operations make the computation and communication costs intolerable. Traditional ORAM schemes promise that the reading and writing processes are oblivious and all data blocks are periodically moved to new locations. The oblivious mechanisms ensure confusion by increasing communication bandwidth. Even if there is a method to reduce the communication bandwidth, the computational cost associated with it is huge.

How to defend file injection attack efficiently. Instead of protecting the file access pattern which is an inefficient approach to prevent file injection attacks, we consider the fact that the server can identify the injection file in a query result by default, which was referred to in [12]. The server can obtain the identifier of the injected files through information such as the upload time, the storage position, and the size of files. So that the server can infer the queried keyword by identifying the set of injected files in the search results. Therefore, we simplify the problem of how to resist non-adaptive file injection attacks into how to prevent the server from identifying the injected files.

The core of preventing the server from identifying the file is to ensure that all data is written obliviously. Because the server can identify the injected file from the special file information, we need to protect each type of file information. Regarding file size, we can simply prevent the server from constructing a specialized injection file by adding the file to a fixed length. Our point is to prevent the server from recognizing these injection files by the update time and the storage position of the files. Naturally, the insertion of files will cause the index to be updated in batches. There exists the linkability between files and their index during the update query process. The server can also infer whether the injected file is queried based on the index query. Thus, the update pattern in the index should be protected either.

Some cryptography applications like write-only ORAM (WoORAM) can hide the information about when the data is updated. It only implements the obliviousness of writing operations and is simpler than traditional ORAM because WoORAM only maintains the privacy of part operations (writes) without other cumbersome operation and achieves practical overhead. But WoORAM cannot be applied in SSE construction directly. Because WoORAM can only be performed in a scenario where the reading behavior is oblivious. We leverage a black box such as software guard extensions (SGX) which can perform privacy computations to realize the obliviousness when writing files and index data.

Contribution. In this paper, we propose a brand new SSE scheme named Cetus which first defends a non-adaptive file injection attack. Cetus is the first SSE solution that focuses on the security protection of both file storage and index. Our scheme achieves optimal search and updates complexity, for both computational and communication. Table 1 [2,7,8,13] presents the comparison of the typical SSE schemes and the schemes related to ORAM with our scheme. We can find that the typical SSE schemes cannot achieve forward privacy, and the schemes related to ORAM that can achieve forward security are less efficient than our scheme. In more detail, our contribution is as follows.

(1) We first define toward privacy against file inject attack. Toward privacy allows the leakage of query results but prevents the server from identifying update patterns. That is, the server cannot learn when and where the files are inserted. So that the server cannot associate the inserted file with the file in the search query result afterward. An SSE scheme can defend against file inject attack if it achieves toward privacy. Compared with regular SSE scheme like Fides [8], our write process is still efficient and acceptable.

(2) We propose one-time writing oblivious algorithm to protect the update pattern and adapt it in both file storage and keyword index. The core idea is setting up an isolated buffer during the writing

Table 1 Comparison with prior forward private SSE schemes[a)]

Scheme	Computation		Communication		Client storage	TP
	Search	Update	Search	Update		
Fides [8]	$O(a_w)$	$O(1)$	$O(a_w)$	$O(1)$	$O(K \log D)$	–
Janus [8]	$O(n_w \cdot d_w)$	$O(1)$	$O(n_w)$	$O(1)$	$O(K \log D)$	–
Orion [7]	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(a_w)$	$O(\log^2 N)$	$O(1)$	–
Mitra [7]	$O(a_w)$	$O(1)$	$O(a_w)$	$O(1)$	$O(K \log D)$	–
TWORAM [2]	$\widetilde{O}(a_w \log N + \log^3 N)$	$O(1)$	$O(a_w \cdot \log N \log\log N)$	$O(1)$	$O(\log^2 N)$	√
SPS14 [13]	$O(\min\{^{n_w \log^3 N}_{a_w + \log^2 N}\})$		$n_w + \log N$		$O(K \log D)$	√
Cetus	$O(a_w)$	$O(N/M)$	$O(a_w)$	$O(1)$	$O(K \log D)$	√

a) K is the number of keywords, D is the number of documents in EDB, and N is the number of (keyword, document) pairs. n_w is the size of the search result set matching keyword w, a_w is the total number of entries matching keyword w, d_w is the number of deleted entries matching w. RT denotes round trip, BP denotes backward-private and TP denotes toward privacy.

period. Compared to traditional ORAM, we have abandoned the pattern protection when reading files and indexes.

(3) We use the secure computing service provided by SGX to solve the problem of one-time oblivious writing that needs to read the buffer data obliviously. In order to achieve efficient data update, we use the method of XOR of random numbers instead of the traditional re-encryption of key replacement. We prove that our algorithms based on SGX can achieve the obliviousness of data and programs.

2 Related work

In this section, we introduce the origin and various schemes of searchable encryption. We also briefly describe the scheme of Write-only ORAM and the research on SGX.

Searchable encryption (SE) was first introduced in [14] which proposed a linear-time search construction. In 2011, Curtmola et al. [15] proposed the modern security definition and a sublinear-time construction. Searchable encryption was modeled as a particular case of structured encryption by Chase and Kamara [16].

Kamara et al. [17] introduced the first sublinear dynamic searchable encryption which supports efficient updates. Compared to [17], the SE scheme in [18] reduced a lot of leakages. For large dataset optimizations, Cash et al. [19] proposed a new dynamic scheme. The schemes in the above work did not satisfy forward security until Ref. [20] first introduced this concept. Since then, most of the work has presented forward security as a basic security property, and many efficient dynamic searchable encryption schemes have been proposed [2, 3, 6, 9–11, 13, 17, 19, 21]. Stefanov et al. [13] first proposed the concept of backward privacy about leakage in deleted entries but without definition. Bost et al. [8] completely defined and ranked backward privacy in 2017. And Ref. [8] also proposed a scheme focused on backward privacy.

ORAM is applied in some studies [2, 3] to achieve a higher level of security. Naveed [22] pointed out that the excessive load imposed by ORAM could not support the operation of the actual application of searchable encryption. Other work on SSE has focused on more efficient search [16, 23–27].

Different from traditional ORAM which maintains complete security attributes, a relaxed security notion of WoORAM was described in [28]. Compared to ORAM, WoORAM only needs to ensure that writing physical access is indistinguishable. Aviv et al. [29] proposed an instance application for synchronization-based cloud storage and backup services. The first proposal of WoORAM tailored for secure processor architectures was by [30]. In 2017, Roche et al. [31] proposed a conceptually simple WoORAM solution that uses a deterministic, sequential writing pattern. Except for the above single-server works, multi-server settings were discussed in [32].

SGX is a set of instructions that increases the security of application code and data, giving them more protection from disclosure or modification. Zheng et al. [33] proposed a distributed data analytics platform supporting a wide range of queries based on SGX. In 2017, Shaon et al. [34] used SGX as a security processor to design a framework to perform data analysis tasks. Some studies focus on an Intel SGX-assisted search platform [35] and SGX-based file system [36]. There was an application for data oblivious genome variants search using Intel SGX [37].

SGX security. SGX can provide private computing services in various security applications. SGX can help to protect selected code and data from disclosure or modification. It partitions applications into

enclaves in memory that increases security. Enclaves have hardware-assisted confidentiality and integrity-added protections to help prevent access from processes at higher privilege levels. Through attestation services, a relying party can receive some verification on the identity of an application enclave before launch. With these capabilities, the applications are prepared for more security.

3 Preliminary

In this section we define the notations involved, in this article and the definition of searchable encryption. At the same time, we introduce the attack method of non-adaptive file injection attacks in detail. Finally, we present our proposed one-time oblivious writing algorithm.

3.1 Notation

In the rest of this paper, we denote $\mathsf{negl}(\lambda)$ as a negligible function where λ is a security parameter. The symmetric keys are λ bits strings and generated uniformly from $\{0,1\}^{\lambda}$ samples. The hash function we used in our construction follows a standard security definition [38]. We assume that an adversary is a probabilistic polynomial time (PPT) algorithm. All algorithms and protocols are run by a time polynomial in the security parameter λ (probabilistic). For a finite set X,

$$x \xleftarrow{\$} X$$

means that x is sampled uniformly from X.

We define a database DB as a tuple of file identifiers and keyword pairs (ind_i, w) where $0 < i < D$ and D is the number of documents. The file identifiers ind_i contain keywords w which are textual keywords from a given alphabet Λ. Let W denote the set of all existing keywords and $|W|$ is the number of keywords. For each independent keyword w, $\mathsf{DB}(w)$ denotes the file identifiers corresponding to w.

3.2 Symmetric searchable encryption

A dynamic SSE scheme consists of three polynomial-time algorithms (Gen, Enc, Dec). And Gen is a probabilistic algorithm that takes as an input a security parameter λ and returns a secret key K_s. Enc is a probabilistic algorithm that takes as inputs a key K_s and a message m and returns a ciphertext c. Dec is a deterministic algorithm that takes as inputs a key K_s and a ciphertext c and returns m if K_s is the key under which c is encrypted.

Setup(DB) is a probabilistic algorithm which takes as input a database DB and output a pair $(\mathsf{EDB}, K, \delta)$ where EDB, K and δ denote the encrypted database, secret key and state of the client, respectively.

$\mathrm{Search}(K, q, \delta;\ \mathsf{EDB}) = (\mathrm{Search}_C(K, q, \delta),\ \mathrm{Search}_S(\mathsf{EDB}))$ is the Search protocol between client and server. Its inputs are key K, state δ and a search query q from the client side and encrypted database EDB from the server side.

$\mathrm{Update}(K, \delta, \mathrm{op}, \mathrm{in};\mathsf{EDB}) = (\mathrm{Update}_C(K, \delta, \mathrm{op}, \mathrm{in}),\ \mathrm{Update}_S(\mathsf{EDB}))$ is the protocol between client and server. Its input is the key K and state δ, operation op and the input in from the client and encrypted database EDB from the server. If set op as add, it means performing add operation, and if set op as del, it means performing deletion operation.

3.3 Non-adaptive file injection attack

Cash et al. [12] summarized the query pattern and the file access pattern into L1 leakage. The query pattern is that search and update queries that use the same keywords will leak duplicates of these keywords. The file-access pattern is the index of documents in search results. Most studies thought that L1 leakage is inconsequential and they did not hide these leakages. However, Ref. [39] proposed adaptive and non-adaptive file injection attacks in 2016 and the successful implementation of such attacks relies on L1 leakage. As mentioned earlier, because adaptive attacks are far weaker than non-adaptive attacks, we ignore adaptive attacks and only discuss non-adaptive attacks as file injection attacks. Because of the perniciousness of file injection attacks, we believe L1 leakage cannot be ignored.

The file injection attack can inject files to be used to break all future queries according to the knowledge of the file-access pattern. Figure 1 shows a sample of the binary-search attack. We consider the server as a semi-honest adversary who knows a set of keyword $K = \{k_0, k_1, \ldots, k_7\}$. The server creates three

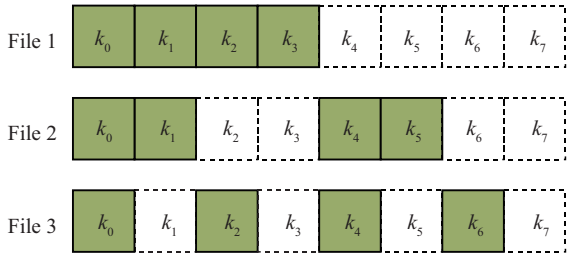

Figure 1 (Color online) Sample of the binary-search attack.

files, each of which contains half of the keywords. All query results can be observed by the server. So that the server can determine the keyword corresponding to the query according to the injected file that is queried. For example, if the query result only has File 1 without Files 2 and 3, then it can be inferred that the keyword of this query is k_3.

3.4 Semantic security definition

We claim that the symmetric encryption involved in the article is semantically secure. The symmetric encryption is used in several algorithms. We follow the definition of semantical secure in [40] as follows.

Definition 1. A private-key encryption scheme Π=(Gen, Enc, Dec) is semantically secure in the presence of an eavesdropper if for every probabilistic polynomial-time algorithm $\mathcal{A}$, there exists a probabilistic polynomial-time algorithm $\mathcal{A}'$ such that for every efficiently-sampleable distribution $X = (X_1, \ldots)$ and all polynomial-time computable functions f and h, there exists a negligible function negl such that

$$|\Pr[\mathcal{A}(1^n, \mathsf{Enc}_k(m), h(m)) = f(m)] - \Pr[\mathcal{A}'(1^n, h(m)) = f(m)]| \leqslant \mathsf{negl}(n), \tag{1}$$

where m is chosen according to distribution X_n, and the probabilities are taken over the choice of m and the key k, and any random coins used by $\mathcal{A}$, $\mathcal{A}'$, and the encryption process.

3.5 Privacy definition

In this subsection, we review the definition of forward privacy and define toward privacy. We first explain the concept of search pattern. In fact, the repetition of token (i.e., searched keywords) can be observed by the server in most SSE schemes. Without a data-oblivious process, if the two searched keywords are the same, then the server will generally return the same search results from the same data storage location. Formally, there exists a list Q for every search query, in which each item is (u, w) where u is the timestamp to indicate the order of each item and w is the searched keyword. Further, we define the search pattern as the repetition of updated keywords. Compared to search pattern, the list Q extra consists of $(u, \mathrm{op}, \mathrm{in})$ where op is an update query with input in. The search pattern $\mathsf{sp}(w)$ for each keyword w is defined as

$$\mathsf{sp}(w) = \{j | (j, w) \in Q\},$$

where $\mathsf{sp}(w)$ only matches search queries.

3.5.1 *Forward privacy*

The traditional forward privacy is that the server cannot learn whether the newly updated documents match a previously searched keyword or not.

Definition 2. An $\mathcal{L}$-adaptively-secure SSE scheme is forward-private iff leakage functions $\mathcal{L}^{\mathsf{Update}}$, can be written as

$$\mathcal{L}^{\mathsf{Update}}(\mathsf{op}, w, \mathsf{ind}) = \mathcal{L}'(\mathsf{op}, \mathsf{ind}, |w|),$$

where $\mathcal{L}'$ is stateless.

3.5.2 *Toward privacy*

We propose a new security attribute named toward security. It ensures that the server cannot know when the searched documents were inserted and where these documents are stored. If the server cannot know whether the searched document is a previously specific updated file, an SSE scheme achieves toward security.

Definition 3. An $\mathcal{L}$-adaptively-secure SSE scheme is result-list revealing toward-private iff leakage functions $\mathcal{L}^{\mathsf{Search}}$, $\mathcal{L}^{\mathsf{Update}}$ can be written as

$$\mathcal{L}^{\mathsf{Update}}(\mathsf{op}, w, \mathsf{ind}) = \mathcal{L}'(|w|),$$
$$\mathcal{L}^{\mathsf{Search}}(w) = \mathcal{L}''(\mathsf{sp}(w)),$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless and $|\mathsf{sp}(w)| = a_w$ for a_w is a constant.

The leakage function $\mathcal{L}^{\mathsf{Update}}$ of toward privacy leaks less than the $\mathcal{L}^{\mathsf{Update}}$ of forward privacy, and forward does not restrict $\mathcal{L}^{\mathsf{Search}}$. Therefore, we conclude that an SSE scheme achieving toward privacy is achieving forward privacy.

4 Technique overview

4.1 One-time oblivious writing

As mentioned before, we can defend the non-adaptive file injection attacks efficiently by preventing the server from observing the update pattern of all inserted files and its related index information. In order to achieve this goal, we design a one-time oblivious writing mechanism to protect the access pattern during the write operation. After these data are stored in the final location, they will not change the content and storage location in subsequent queries. The core idea of one-time oblivious writing is setting up a buffer, writing each block to this buffer sequentially, and transferring them into the final storage obliviously. Similar to WoORAM [31], we define the buffer as the holding area and the final storage as the main area.

We first describe the work process of one-time oblivious writing. Assume that the main area Main has N slots and the holding area Hold has M slots. Each slot can store a data block. When writing a block, this block will be written into Hold sequentially. At the same time, the client assigns a uniformly random position in Main for this block. Notice that the write operation of each block will trigger the update operation of the corresponding N/M blocks in Main. As shown in the right part of Figure 2, when the block is written to the i-th slot of Hold, the i-th part of Main (each part contains N/M blocks) will be refreshed. When M blocks are written in succession, all data blocks in Main will be refreshed. After that, we will restart a new round of sequential writing in Hold. Roche et al. [31] had proved that the Hold area does not overflow and the untransferred data is never overwritten by new data.

Now, we present the detail of writing a block into Hold and refreshing the i-th part of Main. For refreshing the i-th part of Main securely, we must update each data block in this part by re-encrypting its data or replacing the new data block. The refresh operation consists of two operations which are oblivious reading and re-encryption. As shown in the left part of Figure 2, the refresh operation takes as inputs all blocks in Hold and one block in i-th part of Main to obliviously output a block. Then, it re-encrypts the result data block and replaces the corresponding data block in Main. The refresh operation should be repeated N/M times until all N/M blocks are traversed. For the oblivious reading operation (introduced in Subsection 4.2), we will read the block from Hold if the target block in Main should be replaced by the block in Hold. Otherwise, we will read the target block from Main. For the re-encryption operation, it takes as input the data block of oblivious reading and generates a new re-encrypted block, so that the attacker cannot know whether the oblivious reading data block comes from the new data block in Hold or the old data block in Main. Through the above two main operations, N/M blocks in Main will be refreshed obliviously when the client writes a block to Hold.

The above operation can ensure that the data block is obliviously written to Main. Meanwhile, the attacker cannot determine where the newly added data block is written. The subsequent query operations will be performed directly from Main. Therefore, this mechanism is called "one-time oblivious writing".

Challenge. In the work related to write-only ORAM [31], it has been proved that the above-mentioned work can achieve oblivious writing. However, when one-time oblivious writing is actually applied to the

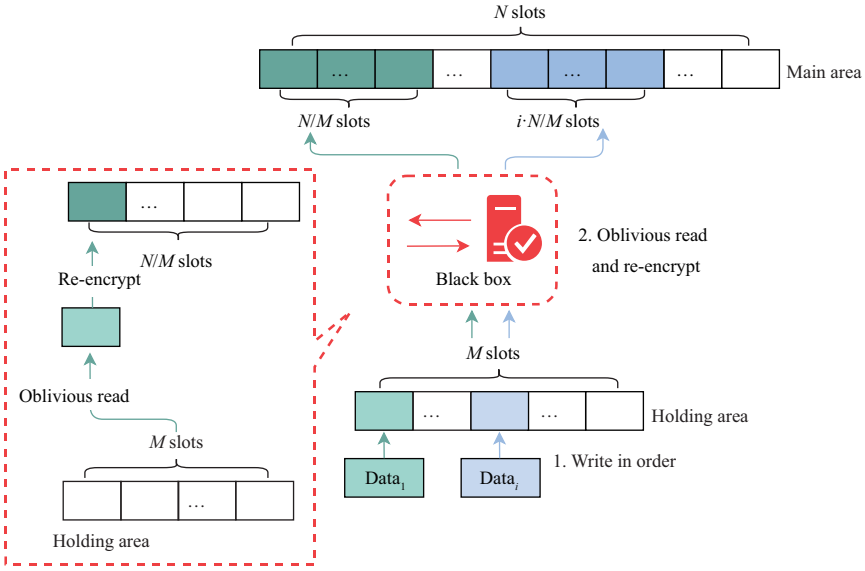

Figure 2 (Color online) Sample process of one-time oblivious writing.

SSE scheme, there exist many challenges. First, oblivious reading must be efficient and acceptable in actual application scenarios. Each write operation brings N/M oblivious reading operations. How to design an efficient oblivious reading algorithm becomes a major challenge for the consideration of efficiency. Second, in order to resist non-adaptive file injection attacks, the SSE scheme should achieve forward security, and the re-encryption operation cannot destroy the relations of the blocks. These challenges put forward new design requirements for oblivious reading and re-encryption algorithms.

4.2 Oblivious reading

As mentioned in Subsection 4.1, oblivious reading is the core operation and its performance becomes the bottleneck of the one-time oblivious writing mechanism. Considering the performance and security, we propose the ObliviousRead algorithm based on SGX. Because the client knows whether the block in Main should be updated is in Hold and the corresponding position of Hold, the client will generate a vector to guide SGX to complete the ObliviousRead program. Specifically, the client generates a vector of length $m+1$, where the position of the target block is 1 and the other positions are 0.

Our ObliviousRead function takes as inputs all data blocks in Hold, the current block in Main, and the select vector of the client. The specific execution process is shown in Algorithm 1 and Figure 3. The ObliviousRead function will perform an AND operation on each bit of input data block and the corresponding item in the select vector, and then, perform an OR operation on the outputs of the AND operations to generate a unique data block as output. The data is either the data block in Hold that is expected to be read into the target slot of Main, or the original data block in Main.

Algorithm 1 ObliviousRead (Array, vec)

```
1: Enclave:
2: result ← null;
3: for i from 1 to |vec| do
4:     for i from 1 to |Array[i]| do
5:         each bit of Array[i] ← each bit of Array[i] & vec[i]; //Each item in Array AND with vec one-one correspondence.
6:     end for
7:     result ← result | Array[i]; //The result OR with each other.
8: end for
9: Return result.
```

Security analysis. The design based on SGX needs to satisfy data oblivious and program oblivious [41]. That is, the server cannot determine which block is read in ObliviousRead procedure by observing the content of blocks and the process of algorithm execution. We prove that ObliviousRead achieves these security goals. Obviously, Algorithm 1 accesses each input data block in turn and executes the same procedure for each input block. The specific security analysis is shown below.

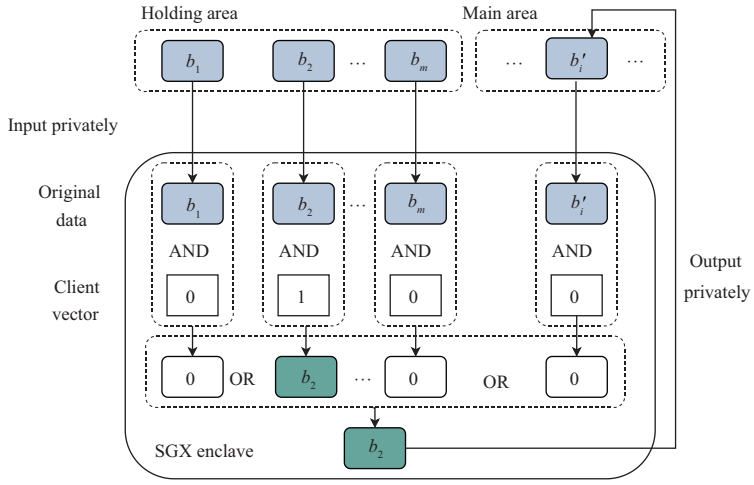

Figure 3 (Color online) Sample process of ObliviousRead algorithm.

Definition 4. Let $d := \{d_1, d_2, \ldots, d_n\}$ denote a block reading sequence of length n where d_i denotes a block which is the output of ObliviousRead. Let $A(d)$ denote the sequence of accesses on the enclave of SGX given the block reading sequence d. The ObliviousRead is said to be secure if for any two block reading sequences d and z such that $|d| = |z|$, their accesses pattern $A(d)$ and $A(z)$ are computationally indistinguishable by anyone but the SGX.

Theorem 1. ObliviousRead is oblivious in the sense of Definition 4.

Proof. Let $y := \{y_1, y_2, \ldots, y_m\}$ denote the block sequence of length m as the input of ObliviousRead and res denote the output block. Let $\text{vec} := \{\text{vec}_1, \text{vec}_2, \ldots, \text{vec}_m\}$ denote the select vector of length m from the client where the length of each vec_i is 1 bit.

Data oblivious. Each block y_i is used as the input of ObliviousRead in ciphertext to perform a series of computation operations in the enclave of SGX. Definition 1 ensures that all blocks that the server can observe are semantically secure. So that there is no plaintext of data leaked to the server during the transmission process. As mentioned in Section 2, the enclave of SGX can protect running code from being disclosed. All blocks which are handled in the enclave cannot be observed by the server. Even the blocks should be decrypted and re-encrypted when run with ObliviousRead function, the content of plaintext will not expose to the server. The ciphertext of output r is indistinguishable from the ciphertext of all input blocks. The server cannot determine which block in the input is read from the input data sequence. Therefore, ObliviousRead achieves data oblivious.

Program oblivious. For all data sent to the enclave, SGX will perform the same operation with each block to ensure the obliviousness of the program. The analysis of data oblivious and program oblivious is as follows.

During the data transfer phase, the input data sequence y is transferred to the SGX's memory indiscriminately. At the same time, the client's select vector vec will also be sent to SGX privately. In the data processing phase, each item of y is ANDed with each item of vec one-to-one without any difference. That is, the AND results are $\text{res} := \{\text{res}_1, \text{res}_2, \ldots, \text{res}_n\}$ where $\text{res}_i = y_i$ AND res_i. Ultimately, all data in the input data sequence are computed with the select vector of the client indiscriminately. Therefore, ObliviousRead achieves program oblivious.

4.3 Lightweight re-encryption

Lightweight re-encryption is another core operation of one-time oblivious writing. It is still implemented based on SGX like ObliviousRead. It takes as input the output of the ObliviousRead function and outputs a re-encrypted data block. All blocks are encrypted by default using traditional encryption methods. On the basis of the data blocks existing in the form of ciphertext, we implement the re-encryption of the blocks by XORing different masks. Without changing and exposing the original plaintext of the data block, the relations between the data block can be maintained and the forward security is not broken. Because our re-encryption operation mainly relies on SGX and XOR operations, we call it "lightweight re-encryption". Experiments show that it is very efficient in Subsection 7.3.

In the re-encryption algorithm, the client generates uniformly random masks and stores these masks locally. For example, when writing a block to Hold, the block should be XORed with an original mask in the client. When this block is transferred to Main during one-time oblivious writing, the client will send the original mask and a new mask to the enclave of SGX privately. This block will also be sent to the enclave. In the enclave, this block will remove the original mask and add a new mask by XOR operation. Notice that all the masks are stored in the client.

For security and efficiency considerations, each re-encrypted block ideally requires a different XOR mask. However, the storage capacity of Main is N and Hold is M. It means that the client has $O(N+M)$ mask storage. To reducing the mask storage to a constant level, we leverage the hash algorithm to generate the masks. Specifically, for the mask ofMain, we use the hash algorithm with the key. The client generates a random number r as the hash key and the index of the current block in Main as the hash input. The hash result is H_r(index) which acts as a mask. For the mask of Hold, we use a keyless hash which takes as input a random number generated by the client to generate the masks, because all blocks in Hold are directly overwritten but not be refreshed.

The client can support the normal update of the mask by storing at least 4 random numbers. Take Hold as an example, we define a traversal write operation of Hold as a round of write operations. After a round of write operation, all slots Hold are full. In a round of write operation, all blocks in Hold are XORed with the same mask. That is, the client only generates a new mask for Hold at the beginning of a round. When the i-th round begins, all blocks in Hold are XORed with the mask of the $(i-1)$-th round. The mask of $(i-2)$-th round has been completely covered by the mask of the $(i-1)$-th round. Therefore, the client does not need to store $O(M)$ mask but privately stores 2 random numbers for Hold. It turns out that the client always only needs 4 random numbers for Hold and Main. These two areas maintain two random numbers respectively. Because Main is updated with Hold, that is, Main and Hold have the same update frequency, Main only needs to store at most two random numbers. The details of the implementation of the above algorithm in the actual scheme will be described in Section 5.

5 The proposed SSE scheme

In this section, we propose a new SSE scheme called Cetus, which can defend against file-injection attacks. It achieves forward privacy and toward privacy. We leverage one-time oblivious writing algorithm in both index and file storage to achieve toward security.

5.1 Storage structure

The server has two arrays MapW and MapF. MapW stores the relationship of keywords and files, which is divided into the main area Main with N blocks and holding area Hold with M blocks. Each block contains w, id, and pre where w is the keyword contained in the file whose identifier is id and pre is the previous block corresponding to w. MapF stores file identifier as key and file data as value. Similarly, file storage is also divided into the main area and holding area. The former has N' blocks and the latter has M' blocks. Each block will not be stored in the main area directly but through the holding area.

For index storage, the client maintains two status maps sM and sH to indicate whether the slot in Main and Hold is full or empty. Additionally, sH.des stores the destination of blocks in Hold and sH.sta stores the operation type to indicate whether the operation is add or delete. In addition, the client stores target map Cur for each keyword. The client should store four random numbers r_h, r'_h, r_m, and r'_m for data re-encryption. Similar to index storage, the client also needs to store the same data for file storage.

5.2 Basic algorithm

The typical SSE scheme consists of three algorithms. Setup, Update and Search, to where Update includes both add and delete operations. As the operation in file storage is similar to keyword index, we will not describe the algorithm in detail but point the difference of algorithm between index and file storage.

5.2.1 *Setup*

In Setup procedure (Algorithm 2), the client initializes an encrypt key k for symmetric encryption and two dictionary arrays Main and Hold for the main area and holding area respectively. The global variable cyc is initialized to 0 as the currently written position in the holding area. Status maps sM and sH are

initialized as 0. For each keyword, the client initializes target map $\mathsf{Cur}[W]$.pos and $\mathsf{Cur}[W]$.key to store the starting point of the result list and a random number respectively. The Main and Hold are sent to the server in ciphertext.

Algorithm 2 Setup(λ)

1: $k \xleftarrow{\$} \{0,1\}^{\lambda}$, cyc$\leftarrow 0$;
2: $r'_h, r'_m \xleftarrow{\$} \{0,1\}^{\lambda}$; //Input of random mask for lightweight re-encryption in SGX.
3: $\mathsf{Main}[N]$, $\mathsf{Hold}[M]$ $\leftarrow$ empty map; //Initialize main area and holding area.
4: $\sigma \leftarrow \mathsf{Cur}[W]$, $\mathsf{sM}[N]$, $\mathsf{sH}[M]$;
5: $\mathsf{EDB} \leftarrow \mathsf{Main}, \mathsf{Hold}$; //$N$ blocks in Main and M blocks in Hold together form encryption database.
6: $L \leftarrow N/M$;
7: Send EDB to the server.

5.2.2 *Update*

Algorithm 3 describes the update procedure. When the client updates a document, it generates a uniform identifier and executes an update operation for each keyword of this document. In an update operation, the client takes as inputs a keyword w, an operation type op and a file identifier id. First, the client chooses a random position fnl which is empty from status map sM and this position will be adopted by the newly w-id pair. If w is added for the first time, the client initializes the key and pos with empty value where the key is the encrypt key and pos is the current position of the latest pair of keyword w. Then the current key and pos are combined as the previous pointer and stored in the following block. Next, the key is refreshed with a new random number and pos is set as fnl. The value which will be stored in the server is computed by the client with a file identifier id, pre-pointer, and a mask. The mask is generated by the hash function that takes as input a random number. And the client also marks the next location of the holding area as existing and records the final location of the block in main. Finally, the client sends this block to the server and the server stores it in the next position in the holding area.

The file storage update algorithm deletes lines 4–8 in Algorithm 3, that is, it is not necessary to save the file location and key in the client. At the same time, the encrypted part of pre was deleted from the calculation of val in line 9, where val is $E_k(\text{ind}) \oplus H(r'_h)$ and ind is the identifier of the file.

Algorithm 3 Update($k, \text{op}, \sigma, \text{id}, w$; EDB)

1: //update index.
2: Client:
3: fnl $\xleftarrow{\$} \{x | x < N, x \in \mathbb{N}, \mathsf{sM}[x].\text{sta} = 0\}$; //Choose an empty slot in main area uniform randomly.
4: **if** $\mathsf{Cur}[w] =$ null **then**
5: $\quad \mathsf{Cur}[w].\text{key}||\mathsf{Cur}[w].\text{pos} \leftarrow$ null; //w is first add.
6: **end if**
7: pre $\leftarrow \mathsf{Cur}[w].\text{key}||\mathsf{Cur}[w].\text{pos}$; //Record the previous item of w.
8: $\mathsf{Cur}[w].\text{key} \xleftarrow{\$} \{0,1\}^{\lambda}$;
9: $\mathsf{Cur}[w].\text{pos} \leftarrow$ fnl; //Initialize encrypt key and position of newly add item.
10: val$\leftarrow (E_k(\text{id})||E_{\mathsf{Cur}[w].\text{key}}(pre)) \oplus H(r'_h)$;
11: $\mathsf{sH}[\text{cyc}].\text{sta} \leftarrow 1$; //cyc-th slot in holding area is full.
12: $\mathsf{sH}[\text{cyc}].\text{des} \leftarrow$ fnl;
13: Send val to the server;
14: Server:
15: $\mathsf{Hold}[\text{cyc}] \leftarrow$ val; //Store val to cyc-th slot in holding area;
16: OneTimeWrite(Main,Hold).

Each update operation (that is, writing a block) triggers the one-time oblivious writing algorithm which is introduced roughly in Subsection 4.1. In OneTimeWrite which is described in Algorithm 4, L blocks in the main area are re-encrypted sequentially where $L = N/M$. The client generates a vector vec according to the status map sH where vec_i indicates the destination of i-th blocks in L target position. Then the client sends vec and four random numbers used to refresh the ciphertext to the server. The server constructs a permutation matrix $\mathbf{I}$ of size $L \times M$ and clean vector CleanVec on the basis of vec. For each block j whose destination is i, $\mathbf{I}(i,j)$ is set as 1. In CleanVec, the position in the main area that will be covered by the block in the holding area is set to be empty (that is, 0). The blocks in the holding area as a vector with size M perform ObliviousRead with each column of $\mathbf{I}$ to generate one block. After L ObliviousRead operation, there are L blocks that are read and permuted privately. Next, the server performs an OR operation on the corresponding L blocks in the main area and each element in

Algorithm 4 OneTimeWrite(Main,Hold)

1: Client:
2: vec$[M] \leftarrow \emptyset$;
3: **for** $i = 0$ to $M - 1$ **do**
4: **if** cyc$\cdot L \leqslant$ sH$[i]$.des $<$ (cyc+1) $\cdot\, L - 1$ **then**
5: vec$[i]$ = sH$[i]$.des; //Record all the final position of blocks in Hold which is in L position of Main.
6: **else**
7: vec$[i] = 0$;
8: **end if**
9: **end for**
10: Send vec, r_m, r'_m, r_h, r'_h to Enclave;
11: Server:
12: Send Main[cyc$\cdot L \sim (\text{cyc} + 1) \cdot L - 1$] and Hold to Enclave;
13: Enclave:
14: $\mathbf{I}(L, M) \leftarrow \{0\}$; //Initialize a $L \times M$ permutation matrix $\mathbf{I}$.
15: CleanVec$[L] \leftarrow \{1\}$; //Initialize a vector of length L where each term is 1.
16: **for** $i = 0$ to $M - 1$ **do**
17: fnl = vec$[i] - L \cdot$ cyc
18: $\mathbf{I}(\text{fnl}, i) = 1$; //Constructing $\mathbf{I}$ according to vec.
19: CleanVec[fnl] $\leftarrow 0$; //Clean the blocks in Main which will be replaced by the blocks in Hold.
20: **end for**
21: array$_h \leftarrow$ Hold;
22: **for** $i = 0$ to $M - 1$ **do**
23: **if** $i \leqslant$ cyc **then**
24: array$_h[i] \leftarrow$ array$_h[i] \oplus H(r'_h)$;
25: **else**
26: array$_h[i] \leftarrow$ array$_h[i] \oplus H(r_h)$; //cyc is public so that the IF sentence leaks no information.
27: **end if**
28: **end for**
29: array$_m \leftarrow$ Main[cyc$\cdot L \sim (\text{cyc} + 1) \cdot L - 1$]
30: **for** $i = 0$ to $L - 1$ **do**
31: array$_m[i] \leftarrow$ array$_m[i] \oplus H_{r_m}(\text{cyc} \cdot L + i)$;//Remove the previous mask.
32: **end for**
33: res$[L]$, res$'[L] \leftarrow \{0\}$;
34: **for** $i = 0$ to $L - 1$ **do**
35: res$[i]$ = ObliviousRead(array$_h$, $\mathbf{I}(i, *)$); //If there has a block of Hold which should be placed in i slot, the result is this block, otherwise 0.
36: res$'[i] = \{\text{res}[i] \mid (\text{array}_m[i]$ & CleanVec$[i])\} \oplus H_{r'_m}(\text{cyc} \cdot L + i)$; //| is OR and & is AND and the result is either the original block in the main area or the block transferred from the holding area.
37: **end for**
38: Send res$'[i]$ to the server.
39: Server:
40: Main[cyc$\cdot L \sim (\text{cyc} + 1) \cdot L - 1] \leftarrow$ res$'$;
41: cyc $:= (\text{cyc} + 1) \mod M$; //Increment counter.
42: **if** cyc $= 0$ **then**
43: Ser $\leftarrow r'_m$;
44: $r_m \leftarrow r'_m, r_h \leftarrow r'_h$;
45: $r'_m, r'_h \xleftarrow{\$} \{0, 1\}^\lambda$;
46: **end if**

the CleanVec. Finally, the results of permutation and cleaning perform AND operation and these results are stored in origin L positions in the main area. After M OneTimeWrite operation, the main area is totally refreshed by the blocks in the holding area.

5.2.3 *Search*

In Search procedure (Algorithm 5), the client looks up Cur$[w]$ to ensure the position pos and key of the latest block according to keyword w. The pos and key combined as Token and Ser are sent to the server where Ser is the latest r'_m which can generate a mask. Then the server determines the first position of keyword w according to pos and removes the mask by performing a hash function taking as input Ser and its index. The results include the ciphertext of the file identifier and the pointer of the next block. And the server decrypts the pointer with the key of Token to retrieve the pos and key of the next block. Next, the server repeats this process until the pos and key is empty. All of the file identifiers are sent to the client and decrypted by the private key. Finally, the server retrieves files from file storage according to the identifiers.

Notice that in lines 2–4 of Algorithm 5, we need to determine whether all blocks in the holding area have been transferred to the main area. If there still exist blocks in the holding area that have not been moved to the main area, we should perform OneTimeWrite algorithm to ensure that the complete query

Algorithm 5 Search(K, w, σ; EDB)

```
1: //Search in index and file storage.
2: Client:
3: while cyc ≠ 0 do
4:     OneTimeWrite(Main,Hold); //Before each search, you must ensure that the main area has been updated once, that is, the
       blocks in the holding area have been transferred to the main area for a round.
5: end while
6: key, pos ←Cur[w]; //Record the encrypt key and position of the last item with w.
7: Token ← key, pos;
8: Send Token, Ser to the server;
9: Server:
10: ResVal ← ∅;
11: while pre is full do
12:     val ← Main[pos];
13:     E_k(id), E_key(pre) ← val ⊕ H_Ser(pos); //Remove the mask and retrieve the file identifier and the pointer of previous item.
14:     ResVal = ResVal ∪ E_k(id);
15:     pre = D_key(E_Cur[w].key(pre)); //Decrypt the pointer information of previous item.
16:     key, pos ← pre;
17: end while
18: Send ResVal to the client;
19: Client:
20: IdSet ← ∅; //Indentifier results set.
21: IdSet ← IdSet ∪ D_k(E_k(id));
22: Send IdSet to the server;
23: Server:
24: FileSet ← ∅; //File results set.
25: FileSet ← FileSet ∪ Main.get(id).
```

results are returned only by searching in the main area.

Storage capacity. In the client, the position map of keywords and position is stored whose storage is $O(K)$ where K denotes the number of keywords. And the update asymptotic performance is $O(1)$ for a constant number of operations in the client. For the server, the performance of update is $O(N/M)$ where N is the size of the main area and M is the size of the holding area. The communication complexity in update is $O(1)$ because a pair of keywords and id will be transferred each time. In search operation, the communication and computation complexity are both $O(a_w)$ where a_w is the total number of entries matching keyword w. There is only one round trip to retrieve the identifier of files and another round trip for files.

6 Security proof

In this section, we prove that Cetus achieves forward and toward privacy.

Theorem 2. Let λ denote the security parameter. Assume $\mathcal{L}'$ is stateless. Define $\mathcal{L}_{Cetus} = (\mathcal{L}^{\mathsf{Search}}_{Cetus}, \mathcal{L}^{\mathsf{Update}}_{Cetus})$, where

$$\mathcal{L}^{\mathsf{Update}}_{Cetus}(\mathsf{op}, w, \mathsf{ind}) = (|w|),$$
$$\mathcal{L}^{\mathsf{Search}}_{Cetus}(w) = (\mathsf{sp}(w)),$$

where $|\mathsf{sp}(w)| = a_w$ for a_w is a constant.

Then Cetus is $\mathcal{L}_{Cetus}$-adaptively-secure with forward privacy and toward privacy in the (programmable) random oracle model. Hence, Cetus thwarts the devastating non-adaptive file-injection attacks of [39].

Proof. We build the simulation $\mathcal{S}$ as follows.

Simulation of setup. This simulator $\mathcal{S}$ is almost the same as setup algorithm except for $\mathcal{S}$.setup without generating k, r_1, and r_2.

Simulation of update. Algorithm 6 describes the simulation of update. Let hash table $H_{\text{table}} = (\text{in}, \text{out})$ denote the random oracle where for each existing parameter in, returning corresponding out. The simulator $\mathcal{S}$ selects mask and r uniformly at random, and initializes pkey, ppos as $\perp$. H_{table} takes as input mask and output r. Then $\mathcal{S}$ chooses current position cPos from the main area of size N and computes val with id, pkey, ppos and mask. The val is stored in the holding area and L slots in the main area are refreshed both sequentially. Since $\mathcal{S}$ does not store the cPos of each block in the holding area, it flips a coin to determine whether the block should be moved to the main area. The cyc as the current writing position in the holding area needs to be incremented by one.

Algorithm 6 Simulation of update($|w|$)

id, mask, mask′,$r, r' \xleftarrow{\$} \{0,1\}^{\lambda}$;
pkey, ppos $\leftarrow \perp$;
$H_{\text{table}} \leftarrow H_{\text{table}} \cup (r, \text{mask}), H_{\text{table}} \leftarrow H_{\text{table}} \cup (r', \text{mask}')$;
for $i = 0$ to $|w|$ **do**
  id, ckey, key$\xleftarrow{\$} \{0,1\}^{\lambda}$;
  cPos[cyc]$\xleftarrow{\$} \{x | x < N, x \in \mathbb{N}\}$;
  val$\leftarrow$ (id$||E_{\text{ckey}}$(pkey$||$ppos)) $\oplus$ mask;
  Hold[cyc] $\leftarrow$ val;
  mask′ $\xleftarrow{\$} \{0,1\}^{\lambda}$;
  for $i = 0$ to $L - 1$ **do**
    if cPos[j] $= i$ **then**
      Main[cyc$\cdot L + i$] = Hold[j] $\oplus$mask $\oplus$ mask′
    else
      Main[cyc$\cdot L + i$] = Main[cyc$\cdot L + i$] $\oplus$mask $\oplus$ mask′;
    end if
  end for
  pkey $\leftarrow$ ckey;
  ppos $\leftarrow$ cPos[cyc];
end forcyc := (cyc + 1) mod M;
if cyc is 0 **then**
  Ser$\leftarrow r$;
  $r, r' \xleftarrow{\$} \{0,1\}^{\lambda}$;
end if

Algorithm 7 Simulation of search(Result(w))

$\bar{w} \leftarrow$ min(Result(w));
while cyc $\neq 0$ **do**
  for $i = 0$ to $L - 1$ **do**
    if cPos[j] $= i$ **then**
      Main[cyc$\cdot L + i$] = Hold[j] $\oplus$mask $\oplus$ mask′;
    else
      Main[cyc$\cdot L + i$] = Main[cyc$\cdot L + i$] $\oplus$mask $\oplus$ mask′;
    end if
  end for
  cyc := (cyc + 1) mod M;
end while
Ser$\leftarrow r'$;
if (Ser, out)$\in H_{\text{table}}$ **then**
  mask′ $\leftarrow$ out;
end if
if Cur[$\bar{w}$] $= \perp$ **then**
  return $\emptyset$;
else
  Token$\leftarrow$ (Cur[$\bar{w}$].key, Cur[$\bar{w}$].pos);
end if
key, pos $\leftarrow$ Token;
while pre is full **do**
  val $\leftarrow$ Main[pos];
  id, E_{key}(pre) $\leftarrow$ val $\oplus$ mask′;
  ResVal = ResVal $\cup$ E_k(id);
  pre = $D_{\text{key}}(E_{\text{key}}(\text{pre}))$;
  key, pos $\leftarrow$ pre;
end while
IdSet $\leftarrow \emptyset$;
IdSet $\leftarrow$ IdSet $\cup$ $D_k(E_k(\text{id}))$;
FileSet $\leftarrow \emptyset$;
FileSet $\leftarrow$ FileSet $\cup$ Main.get(id);
return Token, FileSet.

Simulation of search. Algorithm 7 describes the simulation of search. The simulator $\mathcal{S}$ maintains a position map Cur[W] which consists of $(w, \text{key}, \text{pos})$ pairs. At first, $\mathcal{S}$ computes $\bar{w} \leftarrow \min(\text{Result}(w))$ to distinguish different keywords. Notice that Result(w) consists of the position and content of each item in query results. If there exist blocks in the holding area that have not been moved to the main area, $\mathcal{S}$ should refresh these transfer blocks to the main area. Standard encryption algorithms ensure the semantic security of the ciphertext. Given a security parameter λ, the probability that the adversary attempts to guess the value of ciphertext is $\mathsf{negl}(\lambda)$. The search key Ser is from r' which is the input of H_{table} and output is mask′. When Cur[$\bar{w}$] is empty, it means that $\bar{w}$ has not been added to the database.

If $\bar{w}$ exists, a token is returned in form of (key, pos). According to pos, $\mathcal{S}$ can retrieve value val which contains the file identifier id and the previous pointer pre. $\mathcal{S}$ finds all data blocks from pre successively to collect all the 0-file identifiers. Finally, $\mathcal{S}$ retrieves files from these file identifiers.

Conclusion. By combining all the contributions from all the games, there exists an adversary $\mathcal{A}$ such that

$$|\mathsf{Pr}[\mathrm{SSEREAL}_{\mathcal{A}}^{Cetus}(\lambda) = 1] - \mathsf{Pr}[\mathrm{SSEIDEAL}_{\mathcal{A},\mathcal{S},L_{Cetus}}^{Cetus}(\lambda)=1]|$$
$$\leqslant \mathsf{Adv}_{\mathcal{A}}(\lambda) + \mathrm{negl}(\lambda).$$

We conclude that the probability of result is $\mathrm{negl}(\lambda)$ by assuming standard encryption is secure.

7 Experiment and evaluation

The evaluation focuses on the performance of storage occupancy and response time. We implement Cetus and FIDES in C++ using the PolarSSL libraries for cryptographic operations. We build a test framework that performs cryptographic calculations on a set of files and their keywords. The codes run as a single program without remote procedure call (RPC) because we prefer to measure the execution time. All experiments run single-threaded on the processors. The network latency and the cost of generating index keywords are ignored.

We use 128-bits AES-CBC as a key-based pseudo-random generator and symmetric encryption. We choose SHA-256 and SHA-512 as the underlying hash functions. The source code can be found in GitHub[1)]. We use RocksDB as a data structure to store all files previously. RocksDB is a persistent key-value database. The experiments were performed in a computer device with Intel Core i7-9700 3.0 GHz CPU, 16 GB memory and 256 GB disk storage. The SGX department runs in simulation mode. Because one-time oblivious writing algorithm does not involve data encryption and decryption operations, we can treat the encrypted data in the server as "plaintext" and do not move arrays into the enclave's memory which is achieved by declaring the array with the "user_check" option in the edl file.

7.1 Dataset

We used the public Enron email dataset[2)] as an EDB which was collected and prepared by the CALO Project (A Cognitive Assistant that Learns and Organizes). It contains data from about 150 users, mostly senior management of Enron, organized into folders. There are 517401 files and storage occupancy is 1.32 GB. We used PorterStemmer provided by NLTK library[3)] to extract keywords from the original dataset and collected 0.4 million keywords and 62 million keyword/document pairs.

7.2 EDB creation

The EDB creation implements the initialization of Enron mail dataset in the database. We implement our scheme in two hash functions which are SHA-256 and SHA-512. When using SHA-256, the size of the data block is 32 bytes. We use AES-128 as the encryption algorithm, so the ciphertext of id and the ciphertext of the location information pre of the previous block are both 16 bytes. The location information pre contains the encryption key and the index. We compress the encryption key to 8 bytes and fill the index to 8 bytes to ensure that the size of the plaintext is 16 bytes. When using SHA-512, we do not need to compress the encryption key in pre but encrypt the key and index in pre separately to get 32 bytes ciphertext. At the same time, we fill the 16 bytes dummy data with the 16 bytes id ciphertext to get the 64 bytes original data. This 64 bytes raw data is XORed with the hashed result to get the final 64 bytes data block.

Table 2 presents the comparison of the performance about creation time and server storage. We notice that different hash functions have effects on the speed and storage. Our scheme with SHA-256 saves 50% of server storage and is 40% faster than SHA-512. The server storage is related to the size of a block. Given that the hash value of SHA-512 is twice that of SHA-256, the server storage has also doubled accordingly. In execution time, our scheme with SHA-256 is 20 × faster than Fides and the client storage of our scheme with SHA-256 is almost the same as Fides. The server storage of our scheme is twice that of Fides.

1) https://github.com/emigrantMuse/cetus.
2) Enron Email Dataset. Accessed December 15, 2019, 2019,https://www.cs.cmu.edu/~./enron.
3) NLTK Project. Natural Language Toolkit. Accessed December 15, 2019, 2016, http://www.nltk.org.

Table 2 Comparison with EDB creation

Implementation	Time (s)	Pairs (s)	Storage (MB)	
			Client	Server
Our scheme (with SHA-256)	1961	31626	17.99	1892
Our scheme (with SHA-512)	2745	22593	18.00	3783
Fides	39653	1564	16	803

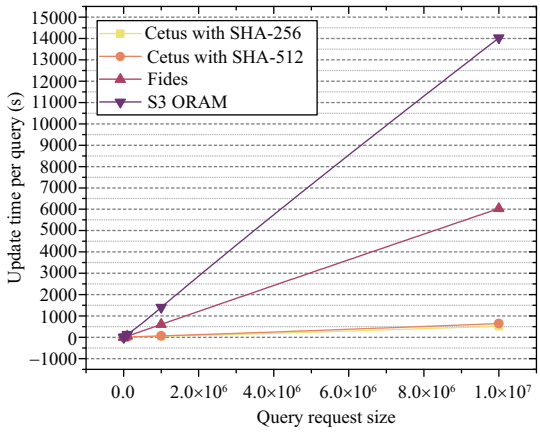

Figure 4 (Color online) Search time with variable query result size.

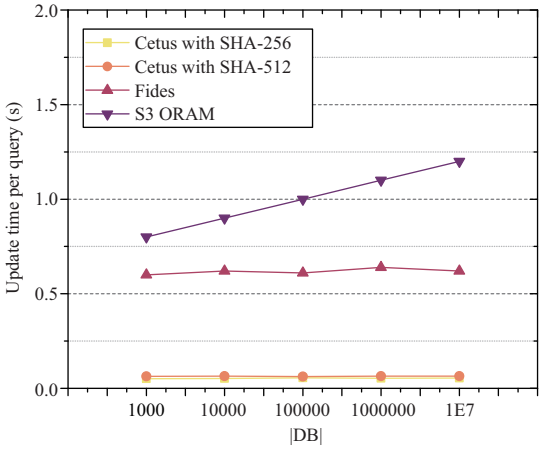

Figure 5 (Color online) Update communication with variable document number.

7.3 EDB update

Different from other SSE schemes, our scheme leverages SGX as a black box to perform the update operation, and thus we extra evaluate the performance of one-time oblivious writing with SGX. In our experiment, we test the influence of holding area size on writing speed. We evaluate the total performance and compare with Fides which is an typical SSE scheme and S^3 ORAM which is an oblivious RAM scheme. Figures 4 and 5 show the comparison of update time among Cetus, Fides and S^3 ORAM. The results show that our scheme is approximately 10 times faster than Fides and 26 times faster than ORAM. The update speed of our scheme and Fides is not affected by the database capacity, and the update speed of S^3 ORAM will increase as the database capacity increases.

Figure 6 shows the response time of the SGX container under different ratios of the holding area to the main area. We found that the run-up time of SGX was not affected by the capacity of the holding area.

Figure 7 presents the execution time of one-time oblivious writing under different ratios of holding area and main area. As we expected before, the larger the ratio of the holding area to the total server capacity, the less execution time for a single writing. Theoretically we want the size of the main area to be a constant multiple of the holding area. In an actual application scenario, a larger holding area can obtain higher update efficiency. Correspondingly, the increase in the capacity of the holding area will also increase the storage overhead of the server. The capacity of the holding area and the efficiency of the update operation are an important trade-off.

7.4 EDB search

We compare the search speed of our scheme and Fides to evaluate the performance in the search procedure. We searched all keywords extracted from Enron mail dataset and computed the average time as the total time spent searching divided by the number of matching documents. We implemented our scheme with SHA-256 and SHA-512 to observe the influence of different hash functions. Figure 8 shows the average time for the number of documents returned in the search results. Notice that Fides need to delete previous nodes and add new nodes in search procedure but our scheme does not require any extra operation.

We also evaluate the search speed in variable database and compute average response time when the number of search time is 256. Figure 9 shows that under different database sizes, that is, server storage,

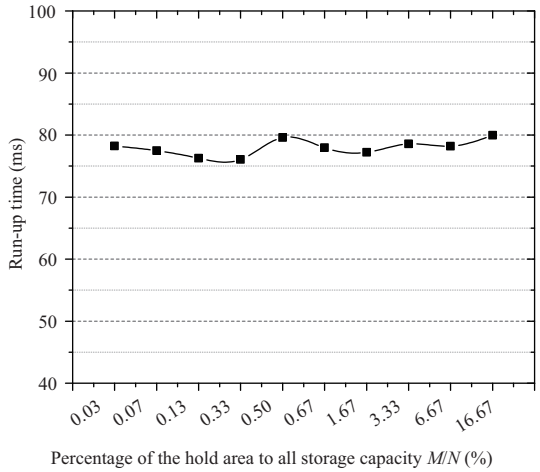

Figure 6 Run-up time of SGX with different holding area sizes.

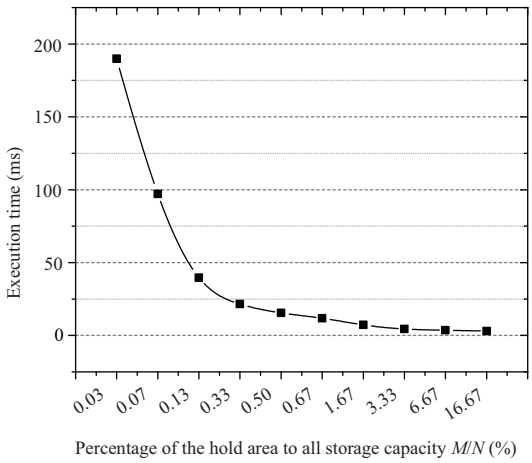

Figure 7 Execution time of SGX with different holding area sizes.

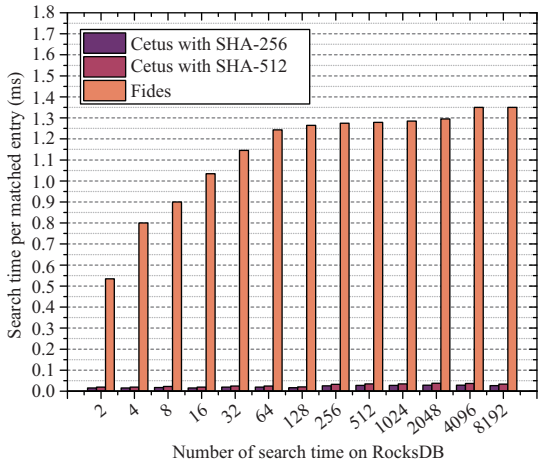

Figure 8 (Color online) Comparison with search time per matched document.

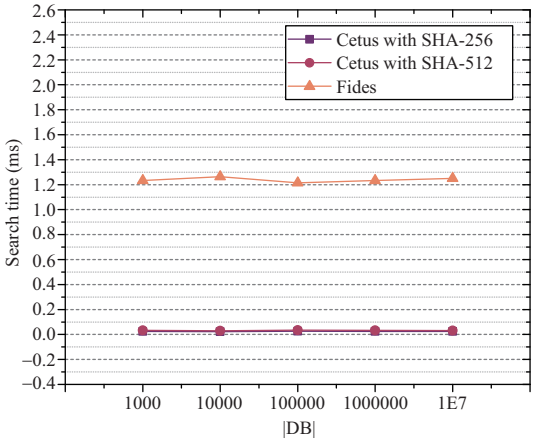

Figure 9 (Color online) Search time with variable document numbers.

the speed of search operations has not changed significantly. Our scheme with SHA-256 is about 30% faster than SHA-512 and about 5 times faster than Fides. This is because the search operation only performs a query on data related to all keywords. The result of the query is only related to the number of files containing keywords.

We think that the communication bandwidth between the server and the client is also an important evaluation index of the efficiency of an SSE scheme. So that we calculate the communication overhead under variable query result size and database size. Figure 10 presents that the communication load increases linearly by query results. And our scheme with SHA-256 saves half of the bandwidth compared with SHA-512 and the bandwidth of the SHA-256 is 1/3 of Fides. Figure 11 shows that the size of the database does not affect the communication bandwidth during the search process.

8 Conclusion and future work

In this work, we describe Cetus, a first toward privacy SSE scheme which also satisfies forward privacy. After learning deeply about file injection attacks, we defend against such attacks in our scheme by protecting update pattern. And Cetus has $O(a_w)$ search time and $O(1)$ update communication. However, our current work left a lot of unresolved issues, like we designed different protection modes for different storage types like index and file storage. And one worthwhile question is to study how to take advantage

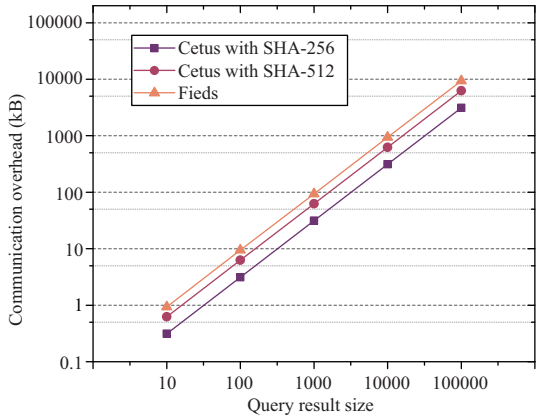

Figure 10 (Color online) Communication with variable document numbers.

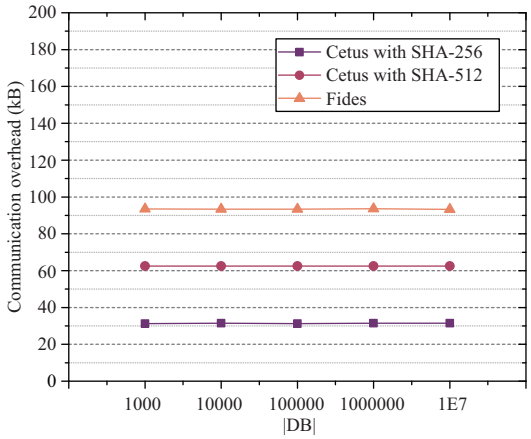

Figure 11 (Color online) Search communication overhead with variable document numbers.

of multiple servers to make our construction more concise. In addition, a more delicate balance of security and efficiency is also a very interesting issue.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant No. 61672300), National Natural Science Foundation of Tianjin (Grant No. 18ZXZNGX00140), and National Natural Science Foundation for Outstanding Youth Foundation (Grant No. 61722203).

References

1 Stefanov E, van Dijk M, Shi E, et al. Path ORAM: an extremely simple oblivious RAM protocol. In: Proceedings of 2013 ACM SIGSAC Conference on Computer & Communications Security. 2013. 299–310
2 Garg S, Mohassel P, Papamanthou C. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In: Proceedings of Annual International Cryptology Conference. Berlin: Springer, 2016. 563–592
3 Naveed M, Prabhakaran M, Gunter C A. Dynamic searchable encryption via blind storage. In: Proceedings of 2014 IEEE Symposium on Security and Privacy, 2014. 639–654
4 Song X, Dong C, Yuan D, et al. Forward private searchable symmetric encryption with optimized I/O efficiency. IEEE Trans Dependable Secure Comput, 2020, 17: 912–927
5 Kim K S, Kim M, Lee D, et al. Forward secure dynamic searchable symmetric encryption with efficient updates. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017. 1449–1463
6 Liu Z, Lv S, Wei Y, et al. FFSSE: flexible forward secure searchable encryption with efficient performance. IACR Cryptol ePrint Arch, 2017, 2017: 1105
7 Ghareh C J, Papadopoulos D, Papamanthou C, et al. New constructions for forward and backward private symmetric searchable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018. 1038–1055
8 Bost R, Minaud B, Ohrimenko O. Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017. 1465–1482
9 Etemad M, Küpçü A, Papamanthou C, et al. Efficient dynamic searchable encryption with forward privacy. Proc Privacy Enhancing Technol, 2018, 2018: 5–20
10 Li J, Huang Y, Wei Y, et al. Searchable symmetric encryption with forward search privacy. IEEE Trans Dependable Secure Comput, 2019. doi: 10.1109/TDSC.2019.2894411
11 Bost R. Σοφος: forward secure searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016. 1143–1154
12 Cash D, Grubbs P, Perry J, et al. Leakage-abuse attacks against searchable encryption. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015. 668–679
13 Stefanov E, Papamanthou C, Shi E. Practical dynamic searchable encryption with small leakage. In: Proceedings of Network and Distributed System Security Symposium, 2014. 71: 72–75
14 Song D X, Wagner D, Perrig A. Practical techniques for searches on encrypted data. In: Proceedings of 2000 IEEE Symposium on Security and Privacy, 2000. 44–55
15 Curtmola R, Garay J, Kamara S, et al. Searchable symmetric encryption: improved definitions and efficient constructions. J Comput Secur, 2011, 19: 895–934
16 Chase M, Kamara S. Structured encryption and controlled disclosure. In: Proceedings of International Conference on the Theory and Application of Cryptology and Information Security. Berlin: Springer, 2010. 577–594
17 Kamara S, Papamanthou C, Roeder T. Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, 2012. 965–976
18 Kamara S, Papamanthou C. Parallel and dynamic searchable symmetric encryption. In: Proceedings of International Conference on Financial Cryptography and Data Security. Berlin: Springer, 2013. 258–274
19 Cash D, Jaeger J, Jarecki S, et al. Dynamic searchable encryption in very-large databases: data structures and implementation. In: Proceedings of Network and Distributed System Security Symposium, 2014. 14: 23–26

20 Chang Y C, Mitzenmacher M. Privacy preserving keyword searches on remote encrypted data. In: Proceedings of International Conference on Applied Cryptography and Network Security. Berlin: Springer, 2005. 442–455
21 Hahn F, Kerschbaum F. Searchable encryption with secure and efficient updates. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014. 310–320
22 Naveed M. The fallacy of composition of oblivious RAM and searchable encryption. IACR Cryptol ePrint Arch, 2015, 2015, 668
23 Cash D, Jarecki S, Jutla C, et al. Highly-scalable searchable symmetric encryption with support for boolean queries. In: Proceedings of Annual Cryptology Conference. Berlin: Springer, 2013. 353–373
24 Demertzis I, Papadopoulos S, Papapetrou O, et al. Practical private range search revisited. In: Proceedings of 2016 International Conference on Management of Data, 2016. 185–198
25 Kamara S, Moataz T. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques. Cham: Springer, 2017. 94–124
26 Meng X, Kamara S, Nissim K, et al. Grecs: graph encryption for approximate shortest distance queries. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015. 504–517
27 Kamara S, Moataz T. SQL on structurally-encrypted databases. In: Proceedings of International Conference on the Theory and Application of Cryptology and Information Security. Cham: Springer, 2018. 149–180
28 Blass E O, Mayberry T, Noubir G, et al. Toward robust hidden volumes using write-only oblivious RAM. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014. 203–214
29 Aviv A J, Choi S G, Mayberry T, et al. Oblivisync: practical oblivious file backup and synchronization. 2016. ArXiv: 1605.09779
30 Haider S K, van Dijk M. Flat ORAM: a simplified write-only oblivious RAM construction for secure processors. Cryptography, 2019, 3: 10
31 Roche D S, Aviv A, Choi S G, et al. Deterministic, stash-free write-only ORAM. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017. 507–521
32 Li L, Datta A. Write-only oblivious RAM-based privacy-preserved access of outsourced data. Int J Inf Secur, 2017, 16: 23–42
33 Zheng W, Dave A, Beekman J G, et al. Opaque: an oblivious and encrypted distributed analytics platform. In: Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), 2017. 283–298
34 Shaon F, Kantarcioglu M, Lin Z, et al. SGX-bigmatrix: a practical encrypted data analytic framework with trusted processors. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017. 1211–1228
35 Hoang T, Ozmen M O, Jang Y, et al. Hardware-supported ORAM in effect: practical oblivious search and update on very large dataset. Proc Privacy Enhancing Technol, 2019, 2019: 172–191
36 Ahmad A, Kim K, Sarfaraz M I, et al. OBLIVIATE: a data oblivious filesystem for intel SGX. In: Proceedings of Network and Distributed System Security Symposium, 2018
37 Mandal A, Mitchell J C, Montgomery H, et al. Data oblivious genome variants search on Intel SGX. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology. Cham: Springer, 2018. 296–310
38 Goldreich O. Foundations of Cryptography: Volume 2, Basic Applications. Cambridge: Cambridge University Press, 2009
39 Zhang Y, Katz J, Papamanthou C. All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), 2016. 707–720
40 Katz J, Lindell Y. Introduction to Modern Cryptography. Boca Raton: CRC Press, 2014
41 Costan V, Devadas S. Intel SGX explained. IACR Cryptol ePrint Arch, 2016, 2016: 1–118