ROSE: Robust Searchable Encryption with Forward and Backward Security and Practical Performance

Peng Xu, *Member, IEEE,* Willy Susilo, *Senior Member, IEEE* Wei Wang, *Member, IEEE,* Tianyang Chen, Qianhong Wu, *Member, IEEE,* Hai Jin, *Fellow, IEEE*

Abstract—Dynamic searchable symmetric encryption (DSSE) has been widely recognized as a promising technique to delegate `update` and `search` queries over an outsourced database to an untrusted server while guaranteeing the privacy of data. Many efforts on DSSE have been devoted to obtaining a good tradeoff between security and performance. However, it appears that all existing DSSE works miss studying on what will happen if the DSSE client issues irrational `update` queries carelessly, such as duplicate `update` queries and `delete` queries to remove non-existent entries (that have been considered by many popular database system in the setting of plaintext). In this scenario, we find that (1) most prior works lose their claimed correctness or security, and (2) no single approach can achieve correctness, forward and backward security, and practical performance at the same time. To address this problem, we study for the first time the notion of robustness of DSSE. Generally, we say that a DSSE scheme is robust if it can keep the same correctness and security even in the case of misoperations. Then, we introduce a new cryptographic primitive named key-updatable pseudo-random function and apply this primitive to constructing ROSE, a robust DSSE scheme with forward and backward security. Finally, we demonstrate the efficiency of ROSE by analyzing its computation and communication complexities and testing its search performance. The experimental results show that ROSE has a very efficient search performance over a large dataset.

Index Terms—Searchable Symmetric Encryption; Forward and Backward Security; Robustness; Key-Updatable Pseudo-Random Function

1 Introduction

SEARCHABLE symmetric encryption (SSE) enables a client to outsource his encrypted data to an honest-but-curious server while keeping the ability to issue keyword search queries over these ciphertexts [1], [2]. *Dynamic* SSE (DSSE) adds new capabilities for the client to update the outsourced database, such as adding new entries and deleting some existent entries [3]. In terms of security, a DSSE scheme guarantees that the curious server can infer the information as little as possible about the outsourced database and the issued `search` and `update` queries from the client. This security is described by the notion of *leakage functions* that define the types of leaked information to the server. In practice, the cores of designing a DSSE scheme are tradeoffs between efficiency, such as storage or communication cost or search performance, and the amount of leaked information.

At present, *forward and backward security* [4] is an important property of DSSE to mitigate the devastating leakage-abuse attacks [5], [6] by ensuring that (1) the newly updated entries cannot be linked with the previous `update` and `search` queries (called *forward security* [7]), and (2) the deleted entries cannot be found by the subsequent `search` queries (called *backward security* [8]). Specifically, backward security includes three different types of leakage (Type-I to Type-III ordered from most to least secure). As results, some well-known DSSE schemes were designed in the past three years for obtaining forward-and-backward security while achieving high efficiency as much as possible [9], [8], [10], [11]. Table 1 lists these DSSE schemes and compares their efficiency and forward-and-backward security. It shows that the DSSE schemes with forward and Type-III backward security are much more practical than the others.

Our Motivation. It is unfortunate that most of these DSSE schemes are ineffective or insecure if the client issues the irrational `update` queries carelessly, such as adding or deleting the same entry repeatedly and deleting the non-existent entry. In short, these schemes cannot be robust, since such irrational `update` queries are hard to be avoided in practice, and many popular database systems, like MySQL, have considered them. The remaining DSSE schemes are robust, but their computation and communication costs are very high. Referring to Table 1, the details of the above problems are as follows.

Schemes Diana$_{del}$ [8], ORION [11], and HORUS [11] cannot guarantee forward security when adding or deleting the

- *P. Xu, T. Chen, and H. Jin are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China. P. Xu is also with the Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen, China. Emails: {xupeng, chentianyang, hjin}@mail.hust.edu.cn.*
- *W. Susilo is with the Institute of Cybersecurity and Cryptology, School of Computing and Information Technology, University of Wollongong, Australia. Email: wsusilo@uow.edu.au.*
- *W. Wang is with the Cyber-Physical-Social Systems Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. Emails: viviawangww@gmail.com.*
- *Q. Wu is with the School of Electronic and Information Engineering, Beihang Univerisity, Beijing, China. Email: qianhong.wu@buaa.edu.cn.*

Table 1: Comparisons with prior forward-and-backward secure works. N is the total number of keyword/file-identifier pairs, W is the number of distinct keywords, and F is the total number of files. For keyword w, a_w is the total number of inserted entries, d_w is the number of `delete` queries, $d_{\max}$ is the supported maximum number of `delete` queries, n_w is the number of files currently containing w, s_w is the number of `search` queries that occurred, i_w is the total number of `add` queries, and s'_w is a number having $s'_w \leq s_w$ (s'_w is explained in Sec. 7). All schemes except ROSE and QOS have $a_w = n_w + d_w$. Specifically, ROSE has $a_w = n_w + s'_w + d_w$, and QOS has $a_w = i_w + d_w$. RT is the number of round trips for search until that the server obtains the matching file identifiers. BS stands for backward security. The notation $\widetilde{O}$ hides polylogarithmic factors.

Scheme	Robust	Computation		Communication			Client	BS
		Search	Update	Search	RT	Update	Storage	
Moneta [8]	✔	$\widetilde{O}(a_w \log N + \log^3 N)$	$\widetilde{O}(\log^2 N)$	$\widetilde{O}(a_w \log N + \log^3 N)$	3	$\widetilde{O}(\log^3 N)$	$O(1)$	I
Fides [8]	✔	$O(a_w)$	$O(1)$	$O(a_w)$	2	$O(1)$	$O(W \log F)$	II
Diana$_{del}$ [8]	✗	$O(a_w)$	$O(\log a_w)$	$O(n_w + d_w \log a_w)$	2	$O(1)$	$O(W \log F)$	III
Janus [8]	✗	$O(n_w \cdot d_w)$	$O(1)$	$O(n_w)$	1	$O(1)$	$O(W \log F)$	III
Janus++ [10]	✗	$O(n_w \cdot d_{\max})$	$O(d_{\max})$	$O(n_w)$	1	$O(1)$	$O(W \log F)$	III
MITRA [11]	✗	$O(a_w)$	$O(1)$	$O(a_w)$	2	$O(1)$	$O(W \log F)$	II
ORION [11]	✗	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(n_w \log^2 N)$	$O(\log N)$	$O(\log^2 N)$	$O(1)$	I
HORUS [11]	✗	$O(n_w \log d_w \log N)$	$O(\log^2 N)$	$O(n_w \log d_w \log N)$	$O(\log d_w)$	$O(\log^2 N)$	$O(W \log F)$	III
FB-DSSE [12]	✗	$O(a_w)$	$O(1)$	$O(F)$	$O(1)$	$O(F)$	$O(W \log F)$	I
SD$_a$ [13]	✗	$O(a_w + \log N)$	$O(\log N)$	$O(a_w + \log N)$	2	$O(\log N)$	$O(1)$	II
SD$_d$ [13]	✗	$O(a_w + \log N)$	$O(\log^3 N)$	$O(a_w + \log N)$	2	$O(\log^3 N)$	$O(1)$	II
QOS [13]	✗	$O(n_w \log i_w + \log^2 W)$	$\log^3 N$	$O(n_w \log i_w + \log^2 W)$	$O(\log W)$	$O(\log^3 N)$	$O(1)$	III
Aura [14]	✗	$O(n_w)$	$O(1)$	$O(n_w)$	1	$O(1)$	$O(W \cdot d_{\max})$	II
SEED [Sec. 5]	✗	$O(a_w)$	$O(1)$	$O(n_w)$	2	$O(1)$	$O(W \log F)$	III
ROSE [Sec. 7]	✔	$O((n_w + s'_w + 1) d_w)$	$O(1)$	$O(n_w)$	2	$O(1)$	$O(W \log F)$	III

same entry repeatedly. Suppose that the database is empty; then, the client successively adds the same keyword/file-identifier pair (w, id) twice. In Diana$_{del}$, it is trivial for the server to find that these two `add` queries are for adding the same keyword, which contradicts with forward security. In ORION and HORUS, these two `add` queries have distinctly different procedures in the view of the server. Hence, the server can find the relationship between these two `add` queries, which makes ORION and HORUS fail to guarantee forward security. In addition, these two schemes are also not forward secure when deleting the same entry repeatedly for the same reason. Although both ORION and HORUS try to hide the above mentioned differences by running some dummy operations, their methods fail to achieve the aim.

After deleting a non-existent entry, schemes Diana$_{del}$ and Janus [8], Janus++ [10], SD$_a$ and SD$_d$ [13], Aura[14], and MITRA [11] disallow the client to add the entry in the future. More generally, these seven schemes restrict the client to re-add the already deleted keyword/file-identifier pairs. This restriction contradicts the fact that the keywords of data rely on the data's content, and some keywords of data may be deleted and then recovered along with the constant update of the data's content in practice. Hence, a DSSE scheme will be much more effective in practice if it can break the restriction. However, Diana$_{del}$ and Janus explicitly mentioned that they fail to achieve the breakthrough. Janus++ has the same problem since it has the same essence as Janus when deleting an entry, except that their `delete` queries are based on puncturable encryptions [10], [15] in public-key and symmetric-key settings, respectively. Besides, if the same entry has been duplicately added, then scheme FB-DSSE [12] can not return the correct search result in future. Scheme QOS [13] fails to delete an entry that has been added twice or more, since a `delete` query of QOS only deletes the latest corresponding `add` query. This problem results that the QOS client may receive incorrect search results from the server.

All existing DSSE works miss studying robustness. But, schemes Moneta [8] and Fides [8] unintentionally achieve this feature with a very high cost. Specifically, in terms of communication cost, these two schemes need to return all matching ciphertexts, including the ciphertexts of the deleted entries, to the client from the server, and then the client decrypts the received ciphertexts. Hence, they waste the communication cost to transfer the non-expected ciphertexts. Furthermore, after decrypting those ciphertexts, the client in both Moneta and Fides re-adds the non-removed entries to the server using a new secret key. Moneta applies a two-round ORAM [16], thus it is efficient in terms of round trip. However, it takes much more computation and bandwidth costs than Fides. More details are in Table 1.

Our Contributions. A simple idea to alleviate from the above problems requires the client to query the update history before issuing each `update` query. However, this idea is impractical since it largely increases either the number of `search` queries or the client's storage cost. Moreover, it cannot enable the client to re-add the already deleted entries. Hence, we aim to construct a new DSSE scheme with robustness, forward and backward security, and practical performance. We start by investigating the robustness of some other prior DSSE works (Sec. 2) and introducing some background knowledge about searchable encryption and forward and backward security (Sec. 3). Our contributions can be summarized as follows.

1) To the best of our knowledge, this paper is the first one to define robustness of DSSE (Sec. 4). A robust DSSE scheme means that it can keep the same correctness and security regardless of whether

the client adds or deletes the same keyword/file-identifier pair repeatedly or deletes the non-existent keyword/file-identifier pair or not. Correspondingly, we extend the original definition of backward security to contain the multiple timestamps of the duplicate `update` queries. In contrast, only one timestamp was defined in the original definition, since it implicitly assumes that no duplicate `update` query occurs.

2) To help understanding our ultimate DSSE scheme, we first construct a basic DSSE scheme named SEED (Sec. 5). SEED is forward and Type-III backward secure, practical, but not robust. Table 1 shows that SEED has the optimal complexity in most terms of computation and communication costs, except for the same storage cost as most of the prior works. SEED does not apply any expensive operation, such as the modular exponentiation or bilinear mapping that was used in schemes $\Sigma_{o\phi o\varsigma}$ [7], Janus, and Janus++. Hence, SEED is also of independent interest in achieving high performance.
3) To transform SEED to our ultimate DSSE scheme, we define a new cryptographic primitive called key-updatable PRF and construct its instance based on an early PRF scheme [17] (Sec. 6). This instance is provably secure under the decisional Diffie-Hellman (DDH) assumption in the random oracle (RO) model. It enables a client to outsource his PRF values to a server and then allows the server to update the original secret key of those PRF values to a new one when receiving a key-update token from the client. In concept, key-updatable PRF is distinct from the notion of key-homomorphic PRF [18], even though both of them can update the secret key of PRF values since they use entirely different inputs to achieve the key update. More details about their differences are in Sec. 6.
4) Finally, we construct a robust DSSE scheme named ROSE (Sec. 7). ROSE is forward and Type-III backward secure under the adaptive attacks. It has the same complexity as SEED in most terms of computation and communication costs except for search performance. Since ROSE applies an ingenious design to make $s'_w \leq \text{Min}(s_w, n_w + 1)$ hold, where $\text{Min}(s_w, n_w + 1)$ stands for the minimum one of s_w and $(n_w + 1)$, ROSE has the same search complexity as Janus and Janus++ in the worst case. Moreover, in practice, ROSE takes much less search time than Janus and Janus++, since the number of expensive operations in ROSE, such as modular exponentiation, is linear with $O(s'_w \cdot d_w)$ rather than $O(n_w \cdot d_w)$, and $s'_w < n_w$ usually holds except for some particular case. Hence, ROSE also has practical search performance. Sec. 8 tests the search performance of ROSE comprehensively.

To summarize, this paper investigates many prior DSSE schemes from a new perspective, i.e., robustness, and finds that all prior forward and backward secure DSSE schemes are either not robust or impractical. Hence, we are interested in tackling this problem and constructing ROSE. To achieve this work, we introduce key-updatable PRF for the first time and take it as the critical component to construct ROSE.

2 DSSE Schemes Revisited

SSE was first introduced by Song *et al.* with an instance that has search performance linear with the size of the database [1]. Later, Chang *et al.* proposed a forward secure SSE scheme, but its search performance is still linear [19]. Curtmola *et al.* were the first to formally define information leakage and proposed an SSE scheme in the static setting, and this scheme has the sub-linear search performance [2]. DSSE was first introduced by Kamara *et al.* [3]. In this section, we investigate some other well-known DSSE schemes regarding their robustness and find that only two of them are robust, but no single scheme has robustness, forward and backward security, and practical performance at the same time.

There are two DSSE schemes that explicitly assume the client never issues the irrational `update` queries. Clearly, these schemes are not robust. For example, Cash *et al.* [20] and Stefanov *et al.* [4] proposed two practical DSSE schemes with small leakage, but Cash *et al.* mentioned that *"We assume throughout that the client never tries to add a record/keyword pair that is already present"*, and Stefanov *et al.* assumed that *"deletions happen only after additions"*.

Although the other DSSE schemes do not explicitly state the above assumption, most of them are still not robust. We categorize them into the following three types according to their flaws caused by the irrational `update` queries.

When issuing the same `add` query repeatedly, the prior DSSE schemes in [3], [21], [22], [23], [9], [24] cannot keep their claimed correctness or security. For example, in [21], [22], [9], the information leakage caused by the duplicate `add` queries will be beyond the limitation of their security definitions, especially the extra leakage about the relationship of those `add` queries; in [3], the duplicate `add` queries will insert several copies of the same data into the database; however, the subsequent `delete` query can just remove one copy of them, in other words, the delete task cannot be achieved completely; in [9], the duplicate `add` queries cause some data to be replaced improperly, such that the subsequent `search` query cannot be achieved correctly; in [24], scheme CLOSE-FB may fail to delete entries or disallow the client to re-add a deleted entry due to it randomly decrypts entries and perform deletion during a search; and in [23], for a keyword, if the client issues a `search` query in the middle of two duplicate `add` queries, the relationship between those two `add` queries will be leaked, which is beyond the leakage amount defined in that work. When issuing the same `delete` query repeatedly, the prior DSSE schemes in [21], [25] leak the relationship of those duplicate `delete` queries, which is beyond their security definitions. The DSSE schemes in [26], [27], [7], [28] disallow the client to re-add an already deleted entry. Otherwise, the re-added entry will be deleted by the server mistakenly. This mistake causes the subsequent `search` query to return incomplete results.

Fortunately, there are two prior DSSE schemes, named DOD-DSSE and FAST in [29], [30], respectively, that are robust. In terms of security, DOD-DSSE is forward and

backward secure, since no relationship among queries is leaked; FAST is forward secure but not backward secure. In terms of performance, when issuing an `update` or `search` query, DOD-DSSE requires the client to fetch all related data (which has a size linear with the maximum number of keywords or files) from the server, update those data locally if the current query is an `update` one, and re-add the updated or original data to the server; thus, this scheme takes very high computation and communication costs; FAST has a practical search and update performance. In addition, DOD-DSSE must set the maximum numbers of keywords and files when initializing it.

3 BACKGROUND

Notations. Let $\lambda \in \mathbb{N}$ be the security parameter. Symbol $x \xleftarrow{\$} \mathcal{X}$ means randomly picking x from the set or space $\mathcal{X}$. Symbol $|x|$ means the binary size of the element x. Symbol $|\mathcal{X}|$ means the number of elements in the set $\mathcal{X}$. Symbol $x||y$ means concatenating strings x and y. Symbol $\{0,1\}^i$ means the $0/1$ strings of length $i \in \mathbb{N}$. Symbol $\{0,1\}^*$ means the $0/1$ strings with an arbitrary length. Symbol $\text{poly}(\lambda)$ means a polynomial in parameter λ. Let $\perp$ be the abort symbol.

Symmetric Encryption (SE). Given a security parameter $\lambda \in \mathbb{N}$, an SE scheme with the key space $\mathcal{K}_{\text{SE}} = \{0,1\}^\lambda$, the plaintext space $\mathcal{M}_{\text{SE}} = \{0,1\}^*$, and the ciphertext space $\mathcal{C}_{\text{SE}}$ consists of two algorithms SE=(SE.Enc, SE.Dec) with the following syntax:

- $\text{SE.Enc}(K, m)$ takes a secret key $K \in \mathcal{K}_{\text{SE}}$ and a plaintext $m \in \mathcal{M}_{\text{SE}}$ as inputs and probabilistically generates a ciphertext ct.
- $\text{SE.Dec}(K, ct)$ takes a secret key K and a ciphertext $ct \in \mathcal{C}_{\text{SE}}$ as inputs and recovers a plaintext $M \in \mathcal{M}_{\text{SE}}$ or returns $\perp$.

An SE scheme must be correct in the sense that given any ciphertext $ct \leftarrow \text{SE.Enc}(K, m)$ of a secret key $K \in \mathcal{K}_{\text{SE}}$ and a plaintext $m \in \mathcal{M}_{\text{SE}}$, algorithm SE.Dec can always recover the plaintext m from ct using the secret key K. In terms of security, a popular requirement of SE is the semantic security under chosen plaintext attack (SS-CPA). Roughly, an SS-CPA secure SE scheme guarantees that without the correct secret key, no probabilistic polynomial time (PPT) adversary can distinguish any two SE ciphertexts of the same binary size.

Pseudorandom Function. Let $\text{F} : \mathcal{K}_{\text{F}} \times \mathcal{X}_{\text{F}} \to \mathcal{Y}_{\text{F}}$ be an efficient function with the key space $\mathcal{K}_{\text{F}}$, the domain $\mathcal{X}_{\text{F}}$, and the range $\mathcal{Y}_{\text{F}}$. It is called a PRF if for all sufficiently large security parameter $\lambda \in \mathbb{N}$ and PPT adversary $\mathcal{A}$, its advantage defined as

$$\text{Adv}^{\text{PRF}}_{\mathcal{A},\text{F}}(\lambda) = |\Pr[\mathcal{A}^{\text{F}(K,\cdot)}(\lambda) = 1] - \Pr[\mathcal{A}^{f(\cdot)}(\lambda) = 1]|$$

is negligible in λ, where $K \xleftarrow{\$} \mathcal{K}_{\text{F}}$ and f is a random function from $\mathcal{X}_{\text{F}}$ to $\mathcal{Y}_{\text{F}}$.

Searchable Encryption. A dynamic searchable symmetric encryption (DSSE) scheme $\Sigma = (\Sigma.\text{Setup}, \Sigma.\text{Update}, \Sigma.\text{Search})$ consists of algorithm $\Sigma.\text{Setup}$ and protocols $\Sigma.\text{Update}$ and $\Sigma.\text{Search}$ both between a client and a server:

- $\Sigma.\text{Setup}(\lambda)$: the client takes a security parameter λ as input and initializes (K, σ, EDB), where K is a secret key for the client, σ is the client's local state, and EDB is an empty encrypted database that is sent to the server.
- $\Sigma.\text{Update}(K, \sigma, op, (w, id); \text{EDB})$: this protocol is for adding an entry to or deleting an entry from EDB as the client's request. In it, the client takes his secret key K, local state σ, query type $op \in \{add, del\}$, and a keyword/file-identifier pair (w, id) as inputs; the server takes EDB as input. After the protocol, the pair (w, id) is added to or deleted from EDB.
- $\Sigma.\text{Search}(K, \sigma, w; \text{EDB})$: this protocol is for searching EDB as the client's query. In it, the client takes his secret key K, local state σ, and a keyword w as inputs; the server takes EDB as input. After the protocol, the results matching keyword w are returned from the server to the client.

Informally, Σ is correct if protocol $\Sigma.\text{Search}$ always returns correct results with an overwhelming probability for each query. We refer readers to [3] for the formal definition. In terms of security, the adaptive security of DSSE is captured by the indistinguishability between a real and an ideal game, and both games allow an adversary to adaptively perform `update` and `search` queries. Intuitively, a secure DSSE scheme guarantees that a PPT adversary should not learn any information about the content of EDB and the queries issued by the client, except some explicit leakage. The adaptive security of a DSSE scheme is parameterized by a (stateful) leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Updt}, \mathcal{L}^{Srch})$ that captures the information learned by the adversarial server throughout the execution of the DSSE scheme, and the components of $\mathcal{L}$ express the information leaked by $\Sigma.\text{Setup}$, $\Sigma.\text{Update}$, and $\Sigma.\text{Search}$ respectively.

Definition 1 (Adaptive Security of DSSE)**.** *A DSSE scheme Σ is said to be $\mathcal{L}-$adaptively secure if for all sufficiently large security parameter $\lambda \in \mathbb{N}$ and PPT adversary $\mathcal{A}$, there is an efficient simulator $\mathcal{S} = (\mathcal{S}.Setup, \mathcal{S}.Update, \mathcal{S}.Search)$ such that*

$$|Pr[Real^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - Pr[Ideal^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1]|$$

is negligible in λ, where games $Real^{\Sigma}_{\mathcal{A}}(\lambda)$ and $Ideal^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$ are defined as:

- *$Real^{\Sigma}_{\mathcal{A}}(\lambda)$: Initially, generate $(K, \sigma, EDB) \leftarrow \Sigma.Setup(\lambda)$ and send EDB to $\mathcal{A}$. Then, $\mathcal{A}$ adaptively issues `update` (resp. `search`) queries with input (op, w, id) (resp. w) and observes the real transcripts generated by these issues. $\mathcal{A}$ eventually outputs a bit.*
- *$Ideal^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$: In this game, $\mathcal{A}$ sees the simulated transcripts instead of the real ones. Initially, $\mathcal{S}$ simulates EDB by running $\mathcal{S}.Setup(\mathcal{L}^{Stp}(\lambda))$. Then, $\mathcal{A}$ adaptively issues `update` (resp. `search`) queries with input (op, w, id) (resp. w) and observes the simulated transcripts generated by $\mathcal{S}.Update(\mathcal{L}^{Updt}(op, w, id))$ (resp. $\mathcal{S}.Search(\mathcal{L}^{Srch}(w))$). Eventually, $\mathcal{A}$ outputs a bit.*

Forward and Type-III Backward Security. We briefly recall the definitions of forward and Type-III backward securities (for more details, please refer to [7], [8], [12].). Let Q be a list of all issued `update` and `search` queries.

Each entry of Q is either an `update` query $(u, op, (w, id))$ or a `search` query (u, w), where $u > 0$ is the timestamp of the corresponding query and gradually increases with the number of queries. Before going ahead, we recall three leakage functions.

For a keyword w, function $\text{sp}(w)$ is to return the timestamps at which keyword w is searched, function $\text{TimeDB}(w)$ is to return all timestamp/file-identifier pairs of keyword w that have been added to but not deleted from EDB, and function $\text{DelHist}(w)$ is to return all insertion/deletion-timestamps pairs of keyword w if there is a file identifier id such that (w, id) has been added to EDB and subsequently deleted from EDB. The above three functions are formally constructed from the query list Q as follows.

$$\text{sp}(w) = \{u | (u, w) \in Q\}$$

$$\text{TimeDB}(w) = \{(u, id) | (u, add, (w, id)) \in Q \text{ and } \forall u', (u', del, (w, id)) \notin Q\}$$

$$\text{DelHist}(w) = \{(u^{add}, u^{del}) | \exists id, (u^{add}, add, (w, id)) \in Q \text{ and } (u^{del}, del, (w, id)) \in Q\}$$

With the above leakage functions, forward and Type-III backward securities are defined as follows.

Definition 2 (Forward Security)**.** *An $\mathcal{L}-$adaptively secure DSSE scheme Σ is forward secure iff the `update` leakage function*

$$\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op, id)$$

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

Definition 3 (Type-III Backward Security)**.** *An $\mathcal{L}-$adaptively secure DSSE scheme Σ is Type-III backward secure iff the `update` and `search` leakage functions $\mathcal{L}^{Updt}$ and $\mathcal{L}^{Srch}$ can be written as*

$$\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op, w)$$

$$\mathcal{L}^{Srch}(w) = \mathcal{L}''(\textit{sp(w), TimeDB(w), DelHist(w)})$$

where both $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.

Note that the definitions of both leakage functions $\text{TimeDB}(w)$ and $\text{DelHist}(w)$ implicitly assume that no duplicate `update` query is issued. The reason is that for each keyword/file-identifier pair, only one insertion/deletion timestamp is returned. Hence, these two leakage functions and the Type-III backward security are not suitable for a robust DSSE scheme (the definitions of Type-I and Type-II backward securities also have the same problem). Fortunately, this problem does not exist in the definition of forward security.

4 Robust Searchable Encryption

Syntax. Roughly, a robust DSSE scheme allows the client to issue duplicate `update` queries and delete non-existent keyword/file-identifier pairs while guaranteeing that (1) the search protocol always returns the correct set of file identifiers, (2) the client can re-add the already deleted keyword/file-identifier pairs, and (3) the security is consistent. Hence, we define robust DSSE as follows.

Definition 4 (Robust DSSE)**.** *Let Σ be a DSSE scheme. Σ is said to be robust if it can keep the same correctness and security even if the client adds or deletes the same keyword/file-identifier pairs repeatedly and re-adds the already deleted keyword/file-identifier pairs.*

General Backward Security Definition. Taking the case of duplicate `update` queries into account, we slightly extend the definitions of leakage functions $\text{TimeDB}(w)$ and $\text{DelHist}(w)$ and name the resulting leakage functions as $\text{exTimeDB}(w)$ and $\text{exDelHist}(w)$ respectively. Their formal definitions are as follows, where $\mathcal{U}$ denotes the set of all satisfactory timestamps.

$$\text{exTimeDB}(w) = \{(\mathcal{U}, id) | \forall u \in \mathcal{U}, (u, add, (w, id)) \in Q \text{ and } \forall u' > u, (u', del, (w, id)) \notin Q\}.$$

$$\text{exDelHist}(w) = \{(\mathcal{U}^{add}, \mathcal{U}^{del}) | \exists id, \forall u^{add} \in \mathcal{U}^{add} \text{ and } u^{del} \in \mathcal{U}^{del}, (u^{add}, add, (w, id)) \in Q \text{ and } (u^{del}, del, (w, id)) \in Q\}$$

In addition, both $\text{exTimeDB}(w)$ and $\text{exDelHist}(w)$ must be the maximal set to contain all possible elements.

With the above new leakage functions, we define the general Type-III backward security as below.

Definition 5 (General Type-III Backward Security)**.** *An $\mathcal{L}-$adaptively secure DSSE scheme Σ is general Type-III backward secure iff the `update` and `search` leakage functions $\mathcal{L}^{Updt}$ and $\mathcal{L}^{Srch}$ can be written as*

$$\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op, w)$$

$$\mathcal{L}^{Srch}(w) = \mathcal{L}''(\textit{sp(w), exTimeDB(w), exDelHist(w)})$$

where both $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.

Clearly, the general Type-III backward security definition implies the traditional Definition 3 if no duplicate `update` query is issued. By the same method, the traditional Type-I and Type-II backward securities can be extended to obtain more generality.

5 SEED: A Basic Scheme

In this section, we show how to construct SEED, a practical forward and Type-III backward secure DSSE scheme. Although SEED is not robust, it is heuristic for constructing ROSE.

Algorithm 1 describes algorithm SEED.Setup and two protocols SEED.Update and SEED.Search. Roughly, for each keyword, SEED constructs a hidden chain relationship to connect all keyword-searchable ciphertexts of the keyword in sequence, in which the latest and earliest generated ciphertexts are located at the head and the tail of the chain, respectively. When receiving a keyword search trapdoor from the client, the server determines the first matching ciphertext that is located at the head of the corresponding chain and decrypts out an index that can guide the server to find the next matching ciphertext. By carrying on in the same way, the server can determine all matching ciphertexts. Finally, the server stops the search if the matching ciphertext located at the chain's tail is found and returns all found ciphertexts, and then the client decrypts out the file identifiers. In terms of security, SEED is forward secure since all ciphertexts are independently generated in the view of

Algorithm 1 Algorithm SEED.Setup and Protocols SEED.Update and SEED.Search

Setup(λ)

1: Initialize a PRF function $\mathrm{F} : \mathcal{K}_\mathrm{F} \times \mathcal{X}_\mathrm{F} \to \mathcal{Y}_\mathrm{F}$ with $\mathcal{Y}_\mathrm{F} = \{0,1\}^\lambda$
2: Initialize a cryptographic hash function $\mathrm{H} : \{0,1\}^* \to \{0,1\}^{3\lambda+1}$
3: Initialize two empty maps LastUp and EDB and $\sigma \leftarrow$ LastUp
4: Let $op \in \{add, del\}$ with the binary codes $add = 1$ and $del = 0$
5: $(K_1, K_2) \xleftarrow{\$} \mathcal{K}_\mathrm{F} \times \mathcal{K}_\mathrm{F}$, $K_3 \xleftarrow{\$} \mathcal{K}_\mathrm{SE}$, and $K_\Sigma \leftarrow (K_1, K_2, K_3)$
6: Send EDB to the server

Update($K_\Sigma, \sigma, op, (w, id)$; EDB)

Client:

1: $(id', op') \leftarrow$ LastUp$[w]$
2: $L \leftarrow \mathrm{F}(K_1, w||id||op)$, $R \xleftarrow{\$} \{0,1\}^\lambda$, and $C \leftarrow \mathrm{SE.Enc}(K_3, id)$
3: **if** $(id', op') = (\mathrm{NULL}, \mathrm{NULL})$ **then**
4: $D \leftarrow \mathrm{H}(\mathrm{F}(K_2, w||id||op), R) \oplus (op||0^{3\lambda})$
5: **else**
6: $L' \leftarrow \mathrm{F}(K_1, w||id'||op')$ and $T' \leftarrow \mathrm{F}(K_2, w||id'||op')$
7: **if** $op = add$ **then**
8: $D \leftarrow \mathrm{H}(\mathrm{F}(K_2, w||id||op), R) \oplus (op||0^\lambda||L'||T')$
9: **else**
10: $X \leftarrow \mathrm{F}(K_1, w||id||add)$
11: $D \leftarrow \mathrm{H}(\mathrm{F}(K_2, w||id||op), R) \oplus (op||X||L'||T')$
12: **end if**
13: **end if**
14: Update LastUp$[w] \leftarrow (id, op)$
15: Send ciphertext (L, R, D, C) to the server

Server:

1: Store EDB$[L] \leftarrow (R, D, C)$

Search(K_Σ, σ, w; EDB)

Client:

1: $(id, op) \leftarrow$ LastUp$[w]$
2: **if** $(id, op) = (\mathrm{NULL}, \mathrm{NULL})$ **then**
3: **return** $\perp$
4: **end if**
5: $L \leftarrow \mathrm{F}(K_1, w||id||op)$ and $T \leftarrow \mathrm{F}(K_2, w||id||op)$
6: Send search trapdoor (L, T) to the server

Server:

1: $(L^t, R^t, D^t, C^t, T^t) \leftarrow (\mathrm{NULL}, \mathrm{NULL}, \mathrm{NULL}, \mathrm{NULL}, \mathrm{NULL})$
2: $\mathcal{I} \leftarrow \emptyset$ and $\mathcal{D} \leftarrow \emptyset$
3: **repeat**
4: $(R, D, C) \leftarrow$ EDB$[L]$
5: $op||X||L'||T' \leftarrow D \oplus \mathrm{H}(T, R)$
6: **if** $op = del$ **then**
7: **if** $L^t \neq \mathrm{NULL}$ **then**
8: $D^t \leftarrow \mathrm{H}(T^t, R^t) \oplus (add||0^\lambda||L'||T')$
9: Update EDB$[L^t] \leftarrow (R^t, D^t, C^t)$
10: **end if**
11: Remove ciphertext (L, R, D, C) from EDB
12: $\mathcal{D} \leftarrow \mathcal{D} \cup \{X\}$
13: **end if**
14: **if** $op = add$ and $L \in \mathcal{D}$ **then**
15: **if** $L^t \neq \mathrm{NULL}$ **then**
16: $D^t \leftarrow \mathrm{H}(T^t, R^t) \oplus (add||0^\lambda||L'||T')$
17: Update EDB$[L^t] \leftarrow (R^t, D^t, C^t)$
18: **end if**
19: Remove ciphertext (L, R, D, C) from EDB
20: **end if**
21: **if** $op = add$ and $L \notin \mathcal{D}$ **then**
22: $(L^t, R^t, D^t, C^t, T^t) \leftarrow (L, R, D, C, T)$
23: Orderly add C to $\mathcal{I}$
24: **end if**
25: $L \leftarrow L'$ and $T \leftarrow T'$
26: **until** $(L = 0^\lambda$ and $T = 0^\lambda)$
27: Send $\mathcal{I}$ to the client

Client:

1: **if** $\mathcal{I} = \emptyset$ **then**
2: Update LastUp$[w] \leftarrow (\mathrm{NULL}, \mathrm{NULL})$
3: **return** $\perp$
4: **end if**
5: **for** $i = 1$ to $|\mathcal{I}|$ **do**
6: $id_i \leftarrow \mathrm{SE.Dec}(K_3, \mathcal{I}[i])$
7: **end for**
8: Update LastUp$[w] \leftarrow (id_1, add)$
9: **return** $\{id_i | i \in [1, |\mathcal{I}|]\}$

the adversary, and the newly generated ciphertexts cannot be found by the server using prior search trapdoors. SEED is also Type-III backward secure since the server cannot decrypt out the file identifiers contained in the matching ciphertexts.

Setup. The client runs algorithm SEED.Setup to initialize a PRF function, a hash function, and two empty maps LastUp and EDB, randomly picks three secret keys (K_1, K_2, K_3), and sends EDB to the server whereas LastUp is stored locally.

Update. When updating a keyword/file-identifier pair (w, id) with operation type op, the client generates and sends a keyword-searchable ciphertext (L, R, D, C) to the server, where L is an index, R is a random number, and D and C are two sub-ciphertexts. It is easy to generate the components L, R, and C (line 2). The only somewhat complicated work is to generate D. Before the generation, the client retrieves the last updated file identifier and operation type (id', op') from LastUp according to w (line 1). If they do not exist, namely, it is the first time to issue the `update` query of keyword w, then the client generates D by encrypting $op||0^{3\lambda}$ (lines 3 and 4). Otherwise, the client computes both the index L' and the decryption token T' of the ciphertext generated in the last `update` query of keyword w and then generates D by encrypting $op||X||L'||T'$, where $X = 0^\lambda$ if $op = add$ or $X = \mathrm{F}(K_1, w||id||add)$ otherwise, the index of the ciphertext generated in a prior `update` query for adding (w, id) (lines 6-12). Finally, the client updates LastUp$[w]$ locally to record the newest update information of keyword w (line 14). Note that regardless of $op = add$ or del, the client always sends a ciphertext to the server. Protocol SEED.Update does not remove any ciphertext from EDB. This work will be performed in protocol SEED.Search.

Search. When issuing a `search` query of keyword w, the client computes both the index L and the decryption token T of the ciphertext that was generated in the last `update` query of w, and L and T constitute the search trapdoor (lines 1-5). Upon receiving (L, T), the server initializes $(L^t, R^t, D^t, C^t, T^t)$ and does the following steps:

1) Retrieve ciphertext (R, D, C) from EDB according to L and decrypt out $op||X||L'||T'$ (lines 4 and 5):
2) If $op = del$, it means that ciphertext (L, R, D, C) is used to tell the server that which ciphertext must be removed, then the server updates D^t such that D^t contains $L'||T'$ if (L^t, R^t, D^t, C^t) is existent, removes (L, R, D, C) from EDB, and inserts X into

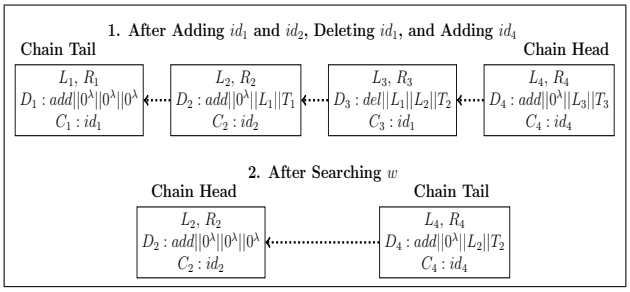

Figure 1: An Example of SEED.

list $\mathcal{D}$, where X is the index of the ciphertext that will be removed later (lines 6-13);

3) If $op = add$ and $L \in \mathcal{D}$, it means that the client wants to remove ciphertext (L, R, D, C), then the server updates D^t such that D^t contains $L'||T'$ if (L^t, R^t, D^t, C^t) is not empty and removes (L, R, D, C) from EDB (lines 14-20);
4) If $op = add$ and $L \notin \mathcal{D}$, it means that ciphertext (L, R, D, C) is valid, the server sets (L^t, R^t, D^t, C^t) to be the ciphertext and T^t to be the decryption token of D^t and inserts C into $\mathcal{I}$ (lines 21-24);
5) Set (L, T) to be (L', T') and repeat the above steps until $L = 0^\lambda$ and $T = 0^\lambda$, namely until the ciphertext located at the tail of the hidden chain of w has been found (lines 25 and 26).

Finally, the server returns $\mathcal{I}$. Then the client decrypts out the expected file identifiers and updates LastUp$[w]$ to record the information about the ciphertext located at the head of the hidden chain of w if such ciphertext is existent; otherwise, it empties LastUp$[w]$ (lines 1-9).

Example. For keyword w, suppose the client adds file identifiers id_1 and id_2, deletes id_1, and adds file identifier id_4 in sequence. Figure 1 shows the four ciphertexts $\{(L_i, R_i, D_i, C_i)|i \in [1, 4]\}$ that are generated by protocol SEED.Update. These ciphertexts contain a hidden chain relationship by letting D_i encrypt L_{i-1} and T_{i-1} for $i \in [2, 4]$. Specifically, D_3 additionally encrypts L_1. Now, we have LastUp$[w] = (id_4, add)$.

When issuing a `search` query of w, the client sends search trapdoor (L_4, T_4) to the server. Upon receiving this trapdoor, the server does the following four steps:

1) In the first step, the server retrieves (R_4, D_4, C_4) from EDB$[L_4]$ and decrypts out $op||X||L'||T' = add||0^\lambda||L_3||T_3$ using T_4; since both $op = add$ and $L_4 \notin \mathcal{D}$ hold, the server sets $(L^t, R^t, D^t, C^t, T^t) \leftarrow (L_4, R_4, D_4, C_4, T_4)$, inserts C_4 into $\mathcal{I}$, and sets $L = L_3$ and $T = T_3$;
2) In the second step, the server retrieves (R_3, D_3, C_3) from EDB$[L_3]$ and decrypts out $op||X||L'||T' = del||L_1||L_2||T_2$; since both $op = del$ and $L^t \neq$ NULL hold, the server updates D_4 such that D_4 contains $add||0^\lambda||L_2||T_2$, removes (L_3, R_3, D_3, C_3) from EDB, inserts L_1 into $\mathcal{D}$, and sets $L = L_2$ and $T = T_2$;
3) In the third step, the server does the similar work as it did in the above first step. After completing this step, we have $(L^t, R^t, D^t, C^t, T^t) = (L_2, R_2, D_2, C_2, T_2)$, $C_2 \in \mathcal{I}$, $L = L_1$, and $T = T_1$;
4) In the fourth step, the server retrieves (R_1, D_1, C_1) from EDB and decrypts out $op||X||L'||T' = add||0^\lambda||0^\lambda||0^\lambda$; since both $op = add$ and $L_1 \in \mathcal{D}$ hold, the server updates D_2 such that D_2 contains $add||0^\lambda||0^\lambda||0^\lambda$, removes (L_1, R_1, D_1, C_1) from EDB, and sets $L = 0^\lambda$ and $T = 0^\lambda$.

Finally, since both $L = 0^\lambda$ and $T = 0^\lambda$ hold, the server stops the search and returns $\mathcal{I} = \{C_4, C_2\}$ to the client. The client decrypts out file identities $\{id_4, id_2\}$ and updates LastUp$[w] = (id_4, add)$. Figure 1 also shows the remainder ciphertexts in EDB and their new hidden chain relationship after the search.

Correctness and Security Analysis. Suppose that the client does not add or delete the same keyword/file-identifier pair repeatedly and re-add the already deleted keyword/file-identifier pair. It is very easy to verify the correctness of SEED, since the indexes of all keyword-searchable ciphertexts are generated by a PRF function with different inputs, and the sub-ciphertexts D and C of each keyword-searchable ciphertext can be correctly decrypted using the corresponding decryption token. In terms of security, SEED achieves forward and Type-III backward security like Diana$_{del}$, Janus, and Janus++ did. Formally, we have the following theorem whose proof is provided in Supplementary Material A.

Theorem 1. *Let the cryptographic hash function H be modelled as a random oracle. Assuming F is a secure PRF, SEED is an adaptively secure DSSE scheme with $\mathcal{L}^{Stp}(\lambda) = \lambda$, $\mathcal{L}^{Updt}(op, w, id) = \emptyset$, and $\mathcal{L}^{Srch}(w) = \{sp(w), TimeDB(w), DelHist(w)\}$.*

Robustness Analysis. In protocol SEED.Update, the index of each keyword-searchable ciphertext is deterministically generated. When adding or deleting the same keyword/file-identifier pair repeatedly, the generated ciphertexts have the same index. This feature leaks the relationship of `update` queries and breaks forward security. It also breaks the correctness since the later generated ciphertext replaces the former one in EDB, which makes the hidden chain relationship among ciphertexts incomplete.

For a keyword w, suppose the client deletes id, searches w, and re-adds id in sequence. Let (L_d, R_d, D_d, C_d) and (L_a, R_a, D_a, C_a) be the two ciphertexts generated by these `delete` and re-`add` queries. According to the line 11 of protocol SEED.Update, D_d contains L_a. Hence, when searching w, the server decrypts D_d and obtains L_a. When re-adding id, the server can link this re-`add` query with the previous `delete` query according to the known L_a. In addition, re-adding an already deleted keyword/file-identifier pair breaks the correctness of SEED due to the same reason as issuing duplicate `update` queries.

To summarize, SEED is not correct and forward secure if the client adds or deletes the same keyword/file-identifier pairs repeatedly or re-adds the already deleted keyword/file-identifier pairs. To address this problem, we will introduce key-updatable PRF and apply it to constructing ROSE.

6 KEY-UPDATABLE PRF

Syntax and Security Definition. Let P be a PRF function. If P is key updatable, then for any two secret keys K_1 and K_2, there is a key-update token that can update the PRF values with K_1 to the PRF values with K_2. In terms of security, key updatable PRF is indistinguishable with a random function under the non-trivial attacks. We define key-updatable PRF and its security as below.

Definition 6 (Key-Updatable PRF)**.** *Let $P : \mathcal{K}_P \times \mathcal{X}_P \to \mathcal{Y}_P$ be an efficient PRF function. We say that P is a secure key-updatable PRF if the following properties hold:*

- *For all keys K_1 and $K_2 \in \mathcal{K}_P$ and $x \in \mathcal{X}_P$, there are two efficient algorithms P.UpdateToken : $\mathcal{K}_P \times \mathcal{K}_P \to \mathcal{K}_P$ and P.KeyUpdate : $\mathcal{K}_P \times \mathcal{Y}_P \to \mathcal{Y}_P$, such that given key-update token $\Delta_{K_1 \to K_2} \leftarrow$ P.UpdateToken(K_1, K_2), equation P.KeyUpdate$(\Delta_{K_1 \to K_2}, P(K_1, x)) = P(K_2, x)$ holds.*
- *For all sufficiently large λ and PPT adversary $\mathcal{A}$, its advantage defined as $Adv^{PRF}_{\mathcal{A},P}(\lambda) = |Pr[Expt^{PRF}_{\mathcal{A},P}(\lambda) = 1] - \frac{1}{2}|$ is negligible in λ, where experiment $Expt^{PRF}_{\mathcal{A},P}(\lambda)$ is defined in Figure 2.*

In addition, we say that the key-update tokens are combinable if there is an efficient operation $\odot$ s.t. $\Delta_{K_1 \to K_2} \odot \Delta_{K_2 \to K_3} = \Delta_{K_1 \to K_3}$ holds for any three keys K_1, K_2, and K_3.

Let (K_1, K_2) be two randomly chosen secret keys. In experiment $\text{Expt}^{\text{PRF}}_{\mathcal{A},\text{P}}(\lambda)$, adversary $\mathcal{A}$ takes key-update token $\Delta_{K_1 \to K_2}$ as input, adaptively issues queries to oracles $\text{P}(K_1, \cdot)$ and $f(\cdot)$, and outputs n number of challenge PRF inputs $\{x_1, ..., x_n\}$; then, randomly giving one of sets $\{\text{P}(K_1, x_i)|i \in [1, n]\}$ and $\{f(x_i)|i \in [1, n]\}$ to $\mathcal{A}$, we say that $\mathcal{A}$ wins in this experiment if he can correctly guess which set is given. During the experiment, $\mathcal{A}$ cannot query oracles $\text{P}(K_1, \cdot)$ or $f(\cdot)$ with these challenge PRF inputs.

$\text{Expt}^{\text{PRF}}_{\mathcal{A},\text{P}}(\lambda)$:

$b \xleftarrow{\$} \{0, 1\}$, $(K_1, K_2) \xleftarrow{\$} \mathcal{K}_\text{P} \times \mathcal{K}_\text{P}$, and $\Delta_{K_1 \to K_2} \leftarrow$ P.UpdateToken(K_1, K_2);

Let $f : \mathcal{X}_\text{P} \to \mathcal{Y}_\text{p}$ be a random function;

$(x_1, \cdots, x_n, st) \leftarrow \mathcal{A}^{\text{P}(K_1,\cdot), f(\cdot)}(\lambda, \Delta_{K_1 \to K_2})$, where $x_i \in \mathcal{X}_\text{P}$ for $i \in [1, n]$ and $n = \text{poly}(\lambda)$;

If $b = 1$, $b' \leftarrow \mathcal{A}^{\text{P}(K_1,\cdot), f(\cdot)}(\{\text{P}(K_1, x_i)|i \in [1, n]\}, st)$;

Else $b' \leftarrow \mathcal{A}^{\text{P}(K_1,\cdot), f(\cdot)}(\{f(x_i)|i \in [1, n]\}, st)$;

Return 1 if $b = b'$; otherwise, return 0;

Note that in the above, $\mathcal{A}$ never queries $\text{P}(K_1, x_i)$ or $f(x_i)$ for $i \in [1, n]$; otherwise, it can trivially guess b.

Figure 2: Experiment on the Security of Key-Updatable PRF.

Key-Updatable PRF vs. Key-Homomorphic PRF. Both key-updatable and key-homomorphic PRFs can alter the original secret key of PRF values to a new secret key. However, they apply different ways to achieve this work. Let $\text{F} : \mathcal{K}_\text{F} \times \mathcal{X}_\text{F} \to \mathcal{Y}_\text{F}$ be a key-homomorphic PRF [18], namely given any two $\text{F}(K_1, x)$ and $\text{F}(K_2, x)$, there is an efficient procedure $\otimes$ such that $\text{F}(K_1 \oplus K_2, x) = \text{F}(K_1, x) \otimes \text{F}(K_2, x)$ holds. Suppose to alter the secret key K_1 of values $\{\text{F}(K_1, x_i)|i \in [1, n]\}$ to any another secret key $K_3 \in \mathcal{K}_\text{F}$, key-homomorphic PRF must take many PRF values $\{\text{F}(K_1 \oplus K_3, x_i)|i \in [1, n]\}$ as inputs and compute $\text{F}(K_3, x_i) = \text{F}(K_1, x_i) \otimes \text{F}(K_1 \oplus K_3, x_i)$ for $i \in [1, n]$, where $n \in \mathbb{N}$. In contrast, to achieve the analogous work, key-updatable PRF takes only one key-update token $\Delta_{K_1 \to K_3}$ as input and computes $\text{P}(K_3, x_i) = \text{P.KeyUpdate}(\Delta_{K_1 \to K_3}, \text{P}(K_1, x_i))$ for $i \in [1, n]$. It is clear that key-updatable PRF is much more efficient than key-homomorphic PRF for updating the secret key of PRF values. Specifically, the generation of $\Delta_{K_1 \to K_3}$ does not rely on $\{x_i|i \in [1, n]\}$.

A Key-Updatable PRF Instance. Referring to an early PRF instance [17], it is easy to construct a key-updatable PRF instance. Let $\mathbb{G}$ be a finite cyclic and multiplicative group of prime order q, where $|q| = \text{poly}(\lambda)$ and $\text{H} : \{0, 1\}^* \to \mathbb{G}$ is a cryptographic hash function. Given $\mathcal{K}_\text{P} = \mathbb{Z}^*_q$, $\mathcal{X}_\text{P} = \{0, 1\}^*$, and $\mathcal{Y}_\text{P} = \mathbb{G}$, a key-updatable PRF instance can be constructed as

$$\text{P}(K, x) = \text{H}(x)^K$$

$$\Delta_{K_1 \to K_2} = \text{P.UpdateToken}(K_1, K_2) = K_1^{-1} \cdot K_2$$

$$\text{P.KeyUpdate}(\Delta_{K_1 \to K_2}, \text{P}(K_1, x)) = \text{P}(K_1, x)^{\Delta_{K_1 \to K_2}}$$

where $(K, K_1, K_2) \in \mathcal{K}_P$ and $x \in \mathcal{X}_\text{P}$. Additionally, this instance has the property of a combinable key-update token when setting operation $\odot$ to be the multiplicative operation of group $\mathbb{Z}^*_q$.

Security Analysis. The security of the above key-updatable PRE instance relies on the DDH assumption in the RO model.

Definition 7 (DDH Assumption)**.** *Let $\mathbb{G}$ be a finite cyclic and multiplicative group of prime order q where $|q| = poly(\lambda)$. We say that the DDH assumption holds if for all sufficiently large λ and PPT adversary $\mathcal{A}$, its advantage defined as $Adv^{DDH}_{\mathcal{A}} = |Pr[\mathcal{A}(g^\alpha, g^\beta, g^{\alpha\beta}) = 1] - Pr[\mathcal{A}(g^\alpha, g^\beta, g^\gamma) = 1]|$ is negligible in λ, where $(\alpha, \beta, \gamma) \xleftarrow{\$} \mathbb{Z}^*_q \times \mathbb{Z}^*_q \times \mathbb{Z}^*_q$.*

Formally, in terms of security, we have the following theorem whose proof is provided in Supplementary Material B.

Theorem 2. *Let the cryptographic hash function H be modelled as a random oracle. The above key-updatable PRF instance is secure under the DDH assumption in the RO model.*

7 ROSE: OUR ULTIMATE SCHEME

In this section, we analyze the fatal design flaws of SEED in the aspect of robustness, outline three key ideas to transform SEED, apply key-updatable PRF to realizing these ideas and constructing ROSE, and finally analyze ROSE.

Two Fatal Design Flaws of SEED. Referring to the previous robustness analysis of SEED, there are two fatal design flaws: one is the deterministic generation of the ciphertexts' indexes; another is that the ciphertext generated by a `delete` query leaks the index of the ciphertext that will be generated by the subsequent re-`add` query if there is a relevant `search` query that occurs in the middle of the `delete` and re-`add` queries.

Three Key Ideas. To tackle the above first design flaw, our first idea is to design a probabilistic algorithm to generate each ciphertext's index, such that all ciphertexts have different indexes even if they are generated by the

Algorithm 2 Algorithm ROSE.Setup and Protocol ROSE.Update.

Setup(λ)

1: Initialize a traditional PRF function $\mathrm{F} : \mathcal{K}_\mathrm{F} \times \mathcal{X}_\mathrm{F} \to \mathcal{Y}_\mathrm{F}$ with $\mathcal{Y}_\mathrm{F} = \{0,1\}^\lambda$ and a key-updatable PRF function $\mathrm{P} : \mathcal{K}_\mathrm{P} \times \mathcal{X}_\mathrm{P} \to \mathcal{Y}_\mathrm{P}$ with the property of a combinable key-update token and $\mathcal{K}_\mathrm{P} = \mathcal{Y}_\mathrm{P} = \{0,1\}^{\lambda'}$, where $\lambda' = \mathrm{poly}(\lambda)$ s.t. F and P have the exact same security in practice
2: Initialize two hash functions $\mathrm{G} : \{0,1\}^* \to \{0,1\}^\lambda$ and $\mathrm{H} : \{0,1\}^* \to \{0,1\}^{2\lambda+\lambda'+2}$
3: Initialize three empty maps LastKey, LastUp, and EDB
4: Let $op \in \{add, del, srch\}$ with the binary codes $add = 01$, $del = 00$, and $srch = 10$
5: Assume the file identifier $id_0 = 0^\lambda$ is an invalid one that never is used by any real file
6: $K_\mathrm{SE} \xleftarrow{\$} \mathcal{K}_\mathrm{SE}$, $K_\Sigma \leftarrow$ (LastKey, K_SE), and $\sigma \leftarrow$ LastUp
7: Send EDB to the server

Update($K_\Sigma, \sigma, op, (w, id)$; EDB)

Client:

1: $(K', S') \leftarrow \mathrm{LastKey}[w]$
2: **if** $(K', S') = (\mathrm{NULL}, \mathrm{NULL})$ **then**
3: $(K', S') \xleftarrow{\$} \mathcal{K}_\mathrm{P} \times \mathcal{K}_\mathrm{F}$ and $\mathrm{LastKey}[w] \leftarrow (K', S')$
4: **end if**
5: $R \xleftarrow{\$} \{0,1\}^\lambda$ and $L \leftarrow \mathrm{G}(\mathrm{P}(K', w||id||op), R)$
6: $C \leftarrow \mathrm{SE.Enc}(K_\mathrm{SE}, id)$
7: $(id', op', R') \leftarrow \mathrm{LastUp}[w]$
8: **if** $(id', op', R') = (\mathrm{NULL}, \mathrm{NULL}, \mathrm{NULL})$ **then**
9: $D \leftarrow \mathrm{H}(\mathrm{F}(S', w||id||op), R) \oplus (op||0^{2\lambda+\lambda'})$
10: **else**
11: $L' \leftarrow \mathrm{G}(\mathrm{P}(K', w||id'||op'), R')$ and $T' \leftarrow \mathrm{F}(S', w||id'||op')$
12: **if** $op = add$ **then**
13: $D \leftarrow \mathrm{H}(\mathrm{F}(S', w||id||op), R) \oplus (op||0^{\lambda'}||L'||T')$
14: **else**
15: $X \leftarrow \mathrm{P}(K', w||id||add)$
16: $D \leftarrow \mathrm{H}(\mathrm{F}(S', w||id||op), R) \oplus (op||X||L'||T')$
17: **end if**
18: **end if**
19: Update $\mathrm{LastUp}[w] \leftarrow (id, op, R)$
20: Send ciphertext (L, R, D, C) to the server

Server:

1: Store $\mathrm{EDB}[L] \leftarrow (R, D, C)$

duplicate `update` queries. However, this idea also causes a new problem that concerns how to delete the ciphertexts generated by the previous duplicate `add` queries. Suppose the client adds the same keyword/file-identifier pair n times and then issues a `delete` query to remove them. Since our first idea makes the n previously generated ciphertexts have n different indexes, the subsequent ciphertext for deleting them must contain these n indexes if applying the original design of protocol SEED.Update. Clearly, it is inefficient. Hence, our second idea is to encrypt a constant-size delete token instead of all relevant indexes when issuing a `delete` query, and this delete token enables the server to find all ciphertexts that were generated by the previous duplicate and relevant `add` queries.

To tackle the above second design flaw, our third idea is that after issuing each `search` query of a keyword, the client will randomly choose new secret keys to generate the ciphertexts in the subsequent `update` queries of the keyword. Suppose the client issues a `search` query at timestamp u_s. This idea is advantageous in guaranteeing that the `update` queries issued before u_s are independent of the `update` queries issued after u_s in the view of the server. However, it also causes a new challenge that concerns how to delete the ciphertexts that were generated before the last `search` query. Without loss of generality, for a keyword w, suppose the client adds id, searches w, and deletes id in sequence. Let (L_a, R_a, D_a, C_a) and (L_d, R_d, D_d, C_d) be the two ciphertexts generated by these `add` and `delete` queries. According to the third idea, these two ciphertexts were generated with different secret keys. Hence, the delete token (introduced in our second idea) contained in D_d is ineffective for deleting the ciphertext with index L_a. Fortunately, key-updatable PRF can overcome this challenge. It can update the secret key of the delete token to the secret key of index L_a, such that the updated delete token has the same secret key with index L_a. Hence, it can be used to delete the ciphertext with index L_a. More details are introduced in the following construction of ROSE.

Setup. Algorithm 2 describes algorithm ROSE.Setup. The client runs this algorithm to initialize a traditional PRF, a key-updatable PRF, two cryptographic hash functions, and three empty maps LastKey, LastUp, and EDB, chooses a secret key K_{SE} of symmetric encryption, generates the secret key K_Σ of ROSE, and sends EDB to the server whereas LastUp is stored locally. Note that LastKey will record the up-to-date secret key of each keyword that is used to generate ciphertexts, and the operation types include the `search` query $srch$ in addition to add and del.

Update. Algorithm 2 also describes protocol ROSE.Update. When updating a keyword/file-identifier pair (w, id) with operation type $op = add$ or del, the client randomly picks the secret keys (K', S') of w and inserts them into LastKey$[w]$ if it is for the first time to issue the `update` query of w; otherwise, it retrieves the keys from LastKey$[w]$ (lines 1-4). Then, the client generates and sends a keyword-searchable ciphertext (L, R, D, C) to the server. Although the ciphertext of ROSE has the same components as SEED, ROSE novelly applies the above first and second ideas to generate L and D. To generate L, the client computes the key-updatable PRF value $\mathrm{P}(K', w||id||op)$ and takes this value and random number R as inputs to compute $L = \mathrm{G}(\mathrm{P}(K', w||id||op), R)$ (line 5). To generate D, ROSE has a procedure similar to SEED except that in ROSE, D contains the delete token $X = \mathrm{P}(K', w||id||add)$ rather than some ciphertext's index if $op = del$ (lines 15 and 16). Finally, the client updates LastUp$[w]$ locally to record the newest update information (id, op, R) of keyword w (line 19).

Search. Algorithm 3 describes protocol ROSE.Search. In contrast to SEED, protocol ROSE.Search has the following two key differences. When issuing a `search` query, the ROSE client generates and sends not only a search trapdoor but also a keyword-searchable ciphertext with operation type $op = srch$ to the server, and the ciphertext contains a key-update token of key-updatable PRF, which will be disclosed to the server in due course for updating the secret key of delete tokens. When receiving a search trapdoor, ROSE has a search procedure similar to SEED except for

Algorithm 3 Protocol ROSE.Search$(K_\Sigma, \sigma, w; \text{EDB})$.

Client:

1: $(id', op', R') \leftarrow \text{LastUp}[w]$
2: **if** $(id', op', R') = (\text{NULL}, \text{NULL}, \text{NULL})$ **then**
3: **return** $\perp$
4: **end if**
5: $(K', S') \leftarrow \text{LastKey}[w]$
6: $L' \leftarrow \text{G}(\text{P}(K', w||id'||op'), R')$ and $T' \leftarrow \text{F}(S', w||id'||op')$
7: $(K, S) \xleftarrow{\$} \mathcal{K}_\text{P} \times \mathcal{K}_\text{F}$, $op \leftarrow srch$, $R \xleftarrow{\$} \{0,1\}^\lambda$, and $\Delta_{K \to K'} \leftarrow \text{P.UpdateToken}(K, K')$
8: $L \leftarrow \text{G}(\text{P}(K, w||id_0||op), R)$, $D \leftarrow \text{H}(\text{F}(S, w||id_0||op), R) \oplus (op||\Delta_{K \to K'}||L'||T')$, and $C \leftarrow \text{SE.Enc}(K_\text{SE}, id_0)$
9: Update $\text{LastUp}[w] \leftarrow (id_0, op, R)$ and $\text{LastKey}[w] \leftarrow (K, S)$
10: Send search trapdoor (L', T') and ciphertext (L, R, D, C) to the server

Server:

1: Store $\text{EDB}[L] \leftarrow (R, D, C)$
2: $(L^t, R^t, D^t, C^t) \leftarrow (L, R, D, C)$, $(op^t, \Delta^t) \leftarrow (srch, \text{NULL})$, and $(L'^t, T'^t) \leftarrow (L', T')$
3: $\mathcal{I} \leftarrow \emptyset$ and $\mathcal{D} \leftarrow \emptyset$
4: **repeat**
5: $(R', D', C') \leftarrow \text{EDB}[L']$
6: $op'||X'||L''||T'' \leftarrow D' \oplus \text{H}(T', R')$
7: **if** $op' = del$ **then**
8: Remove ciphertext (L', R', D', C') from EDB
9: $\mathcal{D} \leftarrow \mathcal{D} \cup \{X'\}$ ▷ Note that $X' \in \mathcal{Y}_\text{P}$ if $op' = del$
10: $D^t \leftarrow D^t \oplus (0^{\lambda'+2}||L'^t \oplus L''||T'^t \oplus T'')$ ▷ Note that operation $\oplus$ is handled before operation $||$
11: Update $\text{EDB}[L^t] \leftarrow (R^t, D^t, C^t)$
12: $(L'^t, T'^t) \leftarrow (L'', T'')$
13: **end if**
14: **if** $op' = add$ **then**
15: **if** $\exists A \in \mathcal{D}$ s.t. $L' = \text{G}(A, R')$ **then**
16: Remove ciphertext (L', R', D', C') from EDB
17: $D^t \leftarrow D^t \oplus (0^{\lambda'+2}||L'^t \oplus L''||T'^t \oplus T'')$
18: Update $\text{EDB}[L^t] \leftarrow (R^t, D^t, C^t)$
19: $(L'^t, T'^t) \leftarrow (L'', T'')$
20: **else**
21: $(L^t, R^t, D^t, C^t) \leftarrow (L', R', D', C')$
22: $(L'^t, T'^t) \leftarrow (L'', T'')$ and $op^t \leftarrow op'$
23: $\mathcal{I} \leftarrow \mathcal{I} \cup \{C'\}$
24: **end if**
25: **end if**
26: **if** $op' = srch$ **then**
27: **if** $op^t = srch$ and $\Delta^t \neq \text{NULL}$ **then**
28: Remove ciphertext (L', R', D', C') from EDB
29: $D^t \leftarrow D^t \oplus (00||\Delta^t \oplus (\Delta^t \odot X')||L'^t \oplus L''||T'^t \oplus T'')$
30: Update $\text{EDB}[L^t] \leftarrow (R^t, D^t, C^t)$
31: $(L'^t, T'^t) \leftarrow (L'', T'')$
32: $\Delta^t \leftarrow \Delta^t \odot X'$ ▷ Note that $X' \in \mathcal{K}_\text{P}$ if $op' = srch$
33: **end if**
34: **if** ($op^t = srch$ and $\Delta^t = \text{NULL}$) or ($op^t \neq srch$) **then**
35: $(L^t, R^t, D^t, C^t) \leftarrow (L', R', D', C')$
36: $(L'^t, T'^t) \leftarrow (L'', T'')$ and $(op^t, \Delta^t) \leftarrow (op', X')$
37: **end if**
38: **for** all $A \in \mathcal{D}$ **do**
39: $A \leftarrow \text{P.KeyUpdate}(X', A)$
40: **end for**
41: **end if**
42: $L' \leftarrow L''$ and $T' \leftarrow T''$
43: **until** $(L' = 0^\lambda$ and $T' = 0^\lambda)$
44: **if** $\mathcal{I} = \emptyset$ **then**
45: Remove all previously found ciphertexts
46: **end if**
47: Send $\mathcal{I}$ to the client

Client:

1: **if** $\mathcal{I} = \emptyset$ **then**
2: $\text{LastKey}[w] \leftarrow (\text{NULL}, \text{NULL})$
3: $\text{LastUp}[w] \leftarrow (\text{NULL}, \text{NULL}, \text{NULL})$
4: **return** $\perp$
5: **end if**
6: **for** $i = 1$ to $|\mathcal{I}|$ **do**
7: $id_i \leftarrow \text{SE.Dec}(K_\text{SE}, \mathcal{I}[i])$
8: **end for**
9: **return** $\{id_i | i \in [1, |\mathcal{I}|]\}$

the following: (1) the ROSE server applies the delete tokens disclosed from the already found ciphertexts to test if a new found ciphertext can be removed; and (2) when finding a matching ciphertext with operation type $op = srch$, the ROSE server decrypts out a key-update token, applies the token to update the secret key of all previously disclosed delete tokens, and removes this ciphertext in some case (this step is for saving the time cost of the next `search` query). The details of protocol ROSE.Search are as below.

When issuing a `search` query of keyword w, the client retrieves the last secret keys (K', S') from $\text{LastKey}[w]$ and takes them as inputs to compute both the index L' and the decryption token T' of the ciphertext that was generated in the last `update` or `search` query of w, and L' and T' constitute the search trapdoor (lines 1-6). Then, the client picks two new random secret keys (K, S), computes the key-update token $\Delta_{K \to K'}$, and takes (K, S) and $(op = srch, w, id_0)$ as inputs to generate a keyword-searchable ciphertext (L, R, D, C), where $id_0 = 0^\lambda$ does not stand for any real file, and D contains $op||\Delta_{K \to K'}||L'||T'$ (lines 7 and 8). Finally, the client updates $\text{LastUp}[w]$ and $\text{LastKey}[w]$ to the information about the current `search` query and these two new secret keys, respectively, and sends (L', T') and (L, R, D, C) to the server.

Recall that all ciphertexts of each keyword are connected by a hidden chain relationship. Thus, in the chain of keyword w, let (L^t, R^t, D^t, C^t) and (L', R', D', C') be two adjacent ciphertexts and (L^t, R^t, D^t, C^t) be in front of (L', R', D', C'). Let $(op^t, \Delta^t, L'^t, T'^t)$ be the operation type, the key-update token, the ciphertext's index, and the decryption token that are contained in D^t. Upon receiving (L', T') and (L, R, D, C) from the client, the server inserts (R, D, C) into $\text{EDB}[L]$ and sets $(L^t, R^t, D^t, C^t) \leftarrow (L, R, D, C)$ and $(op^t, \Delta^t, L'^t, T'^t) \leftarrow (srch, \text{NULL}, L', T')$ at the beginning (lines 1 and 2).

Next, the server will repeat finding a new matching ciphertext and then handle this ciphertext in different ways until all matching ciphertexts are found. The specific steps are as follows:

1) The server retrieves the ciphertext (R', D', C') from $\text{EDB}[L']$ and decrypts out $op'||X'||L''||T''$ (lines 5 and 6);
2) If $op' = del$, it means that the retrieved ciphertext is only for deleting some previous ciphertexts, and X' is the delete token; then, the server removes the ciphertext (L', R', D', C') from EDB, inserts the delete token X' into $\mathcal{D}$, updates D^t such that D^t

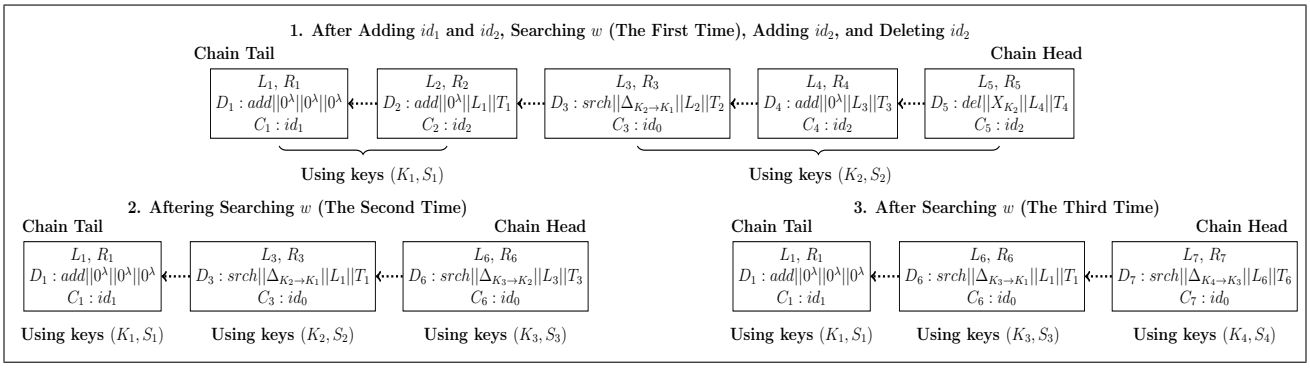

Figure 3: An Example of ROSE.

contains $L''||T''$ rather than $L'^t||T'^t$, and resets $(L'^t, T'^t) \leftarrow (L'', T'')$ (lines 7-13);

3) If $op' = add$, the steps are as follows:
   a) If $\exists A \in \mathcal{D}$ s.t. $L' = \mathrm{G}(A, R')$, it means that the retrieved ciphertext matches up to a known delete token; then, the server removes the ciphertext (L', R', D', C') from EDB and applies the same way as the server does in the lines 10-12 to update D^t and reset (L'^t, T'^t) (lines 15-19);
   b) Otherwise, the retrieved ciphertext is a valid one; then, the server inserts the sub-ciphertext C' into $\mathcal{I}$ and resets (L^t, R^t, D^t, C^t), (L'^t, T'^t), and op^t (lines 20-24);
4) If $op' = srch$, the steps are as follows:
   a) If $op^t = srch$ and $\Delta^t \neq$ NULL, it means that the two adjacent ciphertexts (L^t, R^t, D^t, C^t) and (L', R', D', C') were generated by two previous `search` queries, and the server has known the key-update tokens contained in D^t and D' respectively; then, the server can remove one of these ciphertexts to save the time cost of the next `search` query while maintaining the search correctness. Specifically, in this case, the server removes the ciphertext (L', R', D', C') from EDB, combines the key-update tokens Δ^t and X' to the new token $\Delta^t \odot X'$, updates D^t such that D^t contains $(\Delta^t \odot X')||L''||T''$, and resets (L'^t, T'^t) and Δ^t (lines 27-33);
   b) Otherwise, the server resets (L^t, R^t, D^t, C^t), (L'^t, T'^t), and (op^t, Δ^t) (lines 34-37);
   c) Finally, the server updates the secret key of all known delete tokens using the key-update token X' (lines 38-40);
5) The server resets $(L', T') \leftarrow (L'', T'')$ and repeats the above steps for finding the next matching ciphertext until $L' = 0^\lambda$ and $T' = 0^\lambda$, namely until all matching ciphertexts are found (lines 42 and 43).

Finally, the server returns $\mathcal{I}$. Then the client empties LastKey$[w]$ and LastUp$[w]$ if $\mathcal{I}$ is empty and decrypts out the expected file identifiers (lines 1-9).

Example. For keyword w, suppose the client adds file identifiers id_1 and id_2, searches w, duplicately adds id_2, and deletes id_2 in sequence. According to protocols ROSE.Update and ROSE.Search, the client orderly uploads five ciphertexts to the server, these ciphertexts contain a hidden chain relationship as the first part of Figure 3 shows, and we assume that ciphertexts with indexes L_1 and L_2 are generated using secret keys (K_1, S_1), and the other ciphertexts are generated using secret keys (K_2, S_2).

Next, suppose the client searches w again. The second part of Figure 3 shows the remainder ciphertexts and their relationships after the search. Specifically, according to protocol ROSE.Search, the client sends search trapdoor (L_5, T_6) and ciphertext (L_6, R_6, D_6, C_6) to the server, where ciphertext (L_6, R_6, D_6, C_6) is generated using secret keys (K_3, S_3). Upon receiving those messages, the server inserts ciphertext (L_6, R_6, D_6, C_6) into EDB and performs the following steps:

1) The server retrieves ciphertext (L_5, R_5, D_5, C_5) and decrypts out $del||X_{K_2}||L_4||T_4$; since this ciphertext has operation type $op = del$, the server removes it, updates D_6 such that D_6 contains $L_4||T_4$ instead of $L_5||T_5$, and inserts the delete token X_{K_2} into $\mathcal{D}$ (lines 7-13 of protocol ROSE.Search);
2) The server retrieves ciphertext (L_4, R_4, D_4, C_4) and decrypts out $add||0'||L_3||T_3$; since this ciphertext has operation type $op = add$, and $L_4 = \mathrm{G}(X_{K_2}, R_4)$ holds, the server removes this ciphertext and updates D_6 again such that D_6 contains $L_3||T_3$ instead of $L_4||T_4$ (lines 15-19 of protocol ROSE.Search);
3) The server retrieves ciphertext (L_3, R_3, D_3, C_3), decrypts out $srch||\Delta_{K_2\to K_1}||L_2||T_2$, and updates the secret key K_2 of the delete token X_{K_2} to the secret key K_1 using the key-update token $\Delta_{K_2\to K_1}$ (lines 38-40 of protocol ROSE.Search); let X_{K_1} be the updated delete token;
4) The server retrieves ciphertext (L_2, R_2, D_2, C_2) and decrypts out $add||0'||L_1||T_1$; since this ciphertext has operation type $op = add$, and $L_2 = \mathrm{G}(X_{K_1}, R_2)$ holds, the server removes this ciphertext and updates D_3 such that D_3 contains $L_1||T_1$ instead of $L_2||T_2$ (lines 15-19 of protocol ROSE.Search);
5) The server retrieve the final ciphertext (L_1, R_1, D_1, C_1) and sends C_1 to the client according to the lines 21-23 and 47 of protocol ROSE.Search.

Finally, suppose that it is the third time the client searches w. The client sends search trapdoor (L_6, T_6) and ciphertext (L_7, R_7, D_7, C_7) to the server. The third part of Figure 3 shows the remainder ciphertexts and their relationships after the search. This part mainly shows that in the special case that the conditions in the lines 26 and 27 of protocol ROSE.Search hold, ciphertext (L_3, R_3, D_3, C_3) is removed, the key-update tokens $\Delta_{K_2 \to K_1}$ and $\Delta_{K_3 \to K_2}$ are combined to the key-update token $\Delta_{K_3 \to K_1}$, and D_6 is updated to contain $\Delta_{K_3 \to K_1} || L_1 || T_1$ instead of $\Delta_{K_3 \to K_2} || L_3 || T_3$ (lines 28 and 32 of protocol ROSE.Search).

Analysis on Correctness, Security, and Robustness. Suppose that the client does not add or delete the same keyword/file-identifier pairs repeatedly or re-add the already deleted keyword/file-identifier pairs. It is very easy to verify the correctness of ROSE due to reasons similar to SEED. Without the assumption, it is also easy to verify the correctness of ROSE, since for a keyword, (1) all ciphertexts generated by the `update` and `search` queries, including by the duplicate `update` queries, have independent storage addresses and construct a hidden chain relationship well, (2) this chain can guide the server to find all matching ciphertexts in an orderly manner from chain head to chain tail, (3) the key-updatable feature of PRF P enables `delete` queries to remove all previously added and relevant ciphertexts, (4) the duplicate `delete` queries do not affect the search results, and (5) the re-added ciphertexts cannot be removed by the previously issued `delete` queries. Hence, ROSE is robust in terms of correctness.

In terms of security, we prove that without the above assumption, ROSE is forward and general Type-III backward secure. It also implies that with the above assumption, ROSE is forward and Type-III backward secure, since the general Type-III backward security implies the traditional Type-III backward security. Formally, we have the following theorem whose proof is provided in Supplementary Material C. It is easy to summarize and conclude that ROSE is robust.

Theorem 3. *Let the cryptographic hash functions H and G be modelled as two random oracles. Suppose that the client can add or delete the same keyword/file-identifier pairs repeatedly and re-add the already deleted keyword/file-identifier pairs, and F and P are two secure PRF functions, then ROSE is an adaptively secure DSSE scheme with* $\mathcal{L}^{Stp}(\lambda) = \lambda$, $\mathcal{L}^{Updt}(op, w, id) = \emptyset$, *and* $\mathcal{L}^{Srch}(w) = \{sp(w), exTimeDB(w), exDelHist(w)\}$.

Performance Analysis. Table 1 has listed the computation and communication complexities of ROSE. When issuing an `update` query, the client takes a constant computation complexity to generate a constant-size ciphertext and send this ciphertext to the server by an one-pass communication; finally, the server stores this ciphertext in EDB. When issuing a `search` query, the client takes a constant computation complexity to generate a constant-size search trapdoor and a constant-size ciphertext and sends them to the server; then, the server finds all matching ciphertexts and sends the valid ones of them to the client; finally, the client takes the computation complexity linear with the number of the received ciphertexts to decrypt out file identifiers in the symmetric-key setting. For each keyword, the client stores a pair of PRF's secret keys and a triple *(file identity, operation type, random number)* locally. Hence, ROSE is very practical in most aspects.

The only one slightly complicated work in ROSE is to find all matching ciphertexts. For keyword w, suppose that the client has issued the `add` queries n_w times, the `delete` queries d_w times, and the `search` queries s_w times. Thus, for keyword w, database EDB has stored n_w ciphertexts with operation type $op = add$, s'_w ciphertexts with operation type $op = srch$, and d_w ciphertexts with operation type $op = del$ at most, where $s'_w \leq \text{Min}(s_w, n_w + 1)$ holds since the redundant ciphertexts with operation type $op = srch$ are removed according to the lines 28-32 of protocol ROSE.Search. Under the above assumption, the complexity of searching w over EDB mainly consists of the complexity $O(d_w)$ for removing Type-del ciphertexts, the complexity $O(n_w \cdot d_w)$ for finding or removing Type-add ciphertexts, and the complexity $O(s'_w \cdot d_w)$ for updating the secret keys of the delete tokens. To summarize, the search complexity of protocol ROSE.Search is $O((n_w + s'_w + 1)d_w)$.

8 Implementations

System Environment. We develop SEED and ROSE both in C++ and apply libraries Crypto++ [31] and Relic Toolkit [32] to implement their cryptographic primitives, where Crypto++ provides hash functions SHA-3 family and symmetric encryption AES-128, and Relic Toolkit provides the calculations on the NIST P-256 elliptic curve to implement key-updatable PRF. Hash function SHA-3 is also used to implement the traditional PRF. Both the developed SEED and ROSE can achieve the 128-bit security level in practice. Our experiments are performed on a desktop computer with an Intel Core I5-8259U 2.3 GHz CPU (four cores), 32 GB of DDR4 RAM, and Ubuntu Server 18.04.

Evaluation Methodology. Both SEED and ROSE clearly have good performances in most aspects. Thus, our experiments focus on the computation performances of protocols Update and Search both of SEED and ROSE, i.e., the average time costs of the client to issue an `update` query and a `search` query, and the average time costs of the server to find one matching ciphertext over different datasets. Let d_w and s'_w be the numbers of the Type-del and Type-$srch$ ciphertexts, respectively.

For testing the search performance of SEED, we set up 12 datasets. Each dataset has in total 10^5 ciphertexts of the same keyword, including the Type-del ciphertexts. These datasets have different values of d_w, where the maximum value of d_w is 600. By testing the search performances over these datasets, we can find the slight influence of the different values of d_w on the time cost. We do not set d_w to be a very large value, since the Type-del ciphertexts appear only after the last `search` query in practice (protocol SEED.Search has removed the Type-del ciphertexts that were generated before the last `search` query).

For testing the search performance of ROSE, we set up 48 datasets. All datasets have the same total number of ciphertexts of the same keyword but different numbers of Type-del or Type-$srch$ ciphertexts. The total number is also 10^5. The maximum values of d_w and s'_w are 600 and 400, respectively. We also do not set d_w to be a very large value due to the same reason as SEED. We do not consider the number of ciphertexts generated by the duplicate `update`

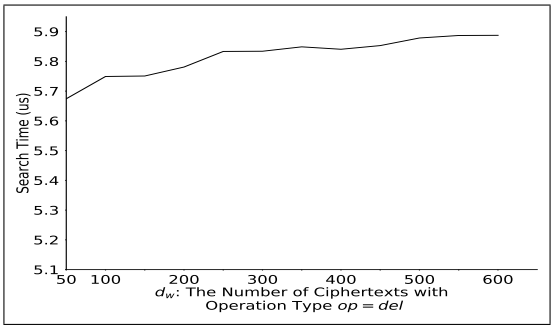

Figure 4: Search Performance of SEED.

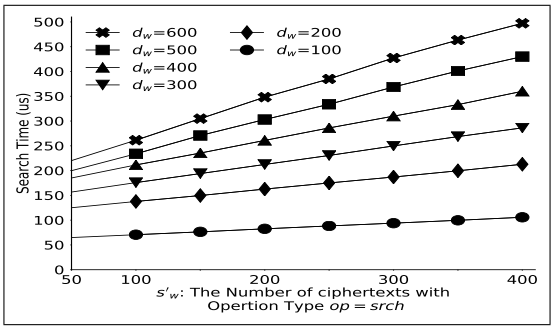

Figure 5: Search Performance of ROSE.

queries, including the queries to re-add the already deleted keyword/file-identifier pairs, since such kinds of `update` queries do not increase the extra computation and communication costs.

Update Evaluation. In protocol SEED.Update, the client takes an average time of approximately 10.53 microseconds (resp. 11.08 microseconds) to generate a ciphertext for issuing an `add` query (resp. a `delete` query). To achieve the same work in protocol ROSE.Update, the average time costs of the client are 819.78 microseconds and 1205.60 microseconds, respectively.

Search Evaluation. In protocol SEED.Search, the client takes an average time of approximately 1.86 microseconds to generate a search trapdoor. Figure 4 shows the average time costs of the sever to find one matching ciphertext respectively over these 12 datasets, which have $d_w \in \{50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600\}$ respectively. When $d_w = 50$, the average time cost is approximately 5.75 microseconds; with the increase in d_w, the search performance has slight degradation; when $d_w = 600$, the average time cost is approximately 6.20 microseconds. It is clear that SEED has very practical search performance, and the number of Type-del ciphertexts affect the search performance insignificantly.

In protocol ROSE.Search, the client takes an average time of approximately 876.32 microseconds to generate a search trapdoor and a Type-$srch$ ciphertext. Let $\mathbb{D}_{d_w,s'_w}$ denote these 48 datasets, where $d_w \in \{100, 200, 300, 400, 500, 600\}$ and $s'_w \in \{50, 100, 150, 200, 250, 300, 350, 400\}$. Figure 5 shows the average time costs of the server to find one matching ciphertext over these 48 datasets. For example, the average time cost to find one matching ciphertext over dataset $\mathbb{D}_{100,50}$ is approximately 67.57 microseconds; over dataset $\mathbb{D}_{600,400}$, the average time cost is approximately 558.33 microseconds. With the increase in d_w and s'_w, although the average time cost increases significantly, the performance is still very efficient, especially for the dataset with smaller d_w but larger s'_w.

In addition, suppose that there are many matching ciphertexts of a `search` query. A reasonable concern might be the time cost of the client to decrypt out file identifiers using AES after receiving many ciphertexts from the server. However, it is not a major undertaking. Now, an Advanced Encryption Standard instruction set (AES-NI) is integrated into many processors, including the CPU we deploy. With the help of AES-NI, the performance of AES encryption/decryption can be very high, such as the performance shown by Intel[1]. Hence, we do not pay attention to the decryption performance of the client.

9 Conclusions

To the best of our knowledge, this work is for the first time to study the correctness, security and performance of DSSE under the assumption that the client can issue the irrational `update` queries. Under this assumption, we find that (1) most prior DSSE schemes cannot guarantee their claimed correctnexss or security, and (2) while a few prior DSSE schemes seem robust, they do not formally study the robustness, and they either entail very high computation and communication costs or obtain the weak security. To tackle all the above problems, we formally define the robustness of DSSE and propose ROSE, the first robust DSSE scheme with forward and backward security and practical performance. To construct ROSE, we introduce SEED as a stepping stone, analyze the design flaws of SEED in achieving robustness, introduce for the first time key-updatable PRF, and transform SEED to ROSE by applying this new kind of PRF. In terms of performance, we analyze the performance of ROSE comprehensively and test the search performances of both SEED and ROSE on large datasets. The experimental results show that both SEED and ROSE are very efficient.

References

[1] D. X. Song, D. A. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, 2000, pp. 44–55. [Online]. Available: https://doi.org/10.1109/SECPRI.2000.848445

[2] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, 2006, pp. 79–88. [Online]. Available: https://doi.org/10.1145/1180405.1180417

[3] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 965–976. [Online]. Available: https://doi.org/10.1145/2382196.2382298

[4] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/practical-dynamic-searchable-encryption-small-leakage

1. Intel AES-NI Performance Testing: online at https://software.intel.com/en-us/articles/intel-aes-ni-performance-testing-on-linuxjava-stack/.

[5] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, 2015, pp. 668–679. [Online]. Available: https://doi.org/10.1145/2810103.2813700

[6] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 707–720. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang

[7] R. Bost, "$\sigma_{o\phi o\varsigma}$: Forward secure searchable encryption," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 1143–1154. [Online]. Available: https://doi.org/10.1145/2976749.2978303

[8] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1465–1482. [Online]. Available: https://doi.org/10.1145/3133956.3133980

[9] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1449–1463. [Online]. Available: https://doi.org/10.1145/3133956.3133970

[10] S. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 763–780. [Online]. Available: https://doi.org/10.1145/3243734.3243782

[11] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 1038–1055. [Online]. Available: https://doi.org/10.1145/3243734.3243833

[12] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, 2019, pp. 283–303. [Online]. Available: https://doi.org/10.1007/978-3-030-29962-0_14

[13] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/dynamic-searchable-encryption-with-small-client-storage/

[14] S.-F. Sun, R. Steinfeld, S. Lai, X. Yuan, A. Sakzad, J. K. Liu, S. Nepal, and D. Gu, "Practical non-interactive searchable encryption with forward and backward privacy," 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/practical-non-interactive-searchable-encryption-with-forward-and-backward-privacy/

[15] M. D. Green and I. Miers, "Forward secure asynchronous messaging from puncturable encryption," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 305–320. [Online]. Available: https://doi.org/10.1109/SP.2015.26

[16] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption," in *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, 2016, pp. 563–592. [Online]. Available: https://doi.org/10.1007/978-3-662-53015-3_20

[17] M. Naor, B. Pinkas, and O. Reingold, "Distributed pseudo-random functions and kdcs," in *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, 1999, pp. 327–346. [Online]. Available: https://doi.org/10.1007/3-540-48910-X_23

[18] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan, "Key homomorphic prfs and their applications," in *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, 2013, pp. 410–428. [Online]. Available: https://doi.org/10.1007/978-3-642-40041-4_23

[19] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, 2005, pp. 442–455. [Online]. Available: https://doi.org/10.1007/11496137_30

[20] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014

[21] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, 2013, pp. 258–274. [Online]. Available: https://doi.org/10.1007/978-3-642-39884-1_22

[22] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 310–320. [Online]. Available: https://doi.org/10.1145/2660267.2660297

[23] A. A. Yavuz and J. Guajardo, "Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware," in *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, 2015, pp. 241–259. [Online]. Available: https://doi.org/10.1007/978-3-319-31301-6_15

[24] K. He, J. Chen, Q. Zhou, R. Du, and Y. Xiang, "Secure dynamic searchable symmetric encryption with constant client storage cost," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 1538–1549, 2021. [Online]. Available: https://doi.org/10.1109/TIFS.2020.3033412

[25] J. Li, Y. Huang, Y. Wei, S. Lv, Z. Liu, C. Dong, and W. Lou, "Searchable symmetric encryption with forward search privacy," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 1, pp. 460–474, 2021. [Online]. Available: https://doi.org/10.1109/TDSC.2019.2894411

[26] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. N. Garofalakis, "Practical private range search revisited," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 185–198. [Online]. Available: https://doi.org/10.1145/2882903.2882911

[27] I. Demertzis and C. Papamanthou, "Fast searchable encryption with tunable locality," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017, pp. 1053–1067. [Online]. Available: https://doi.org/10.1145/3035918.3064057

[28] I. Demertzis, C. Papamanthou, and R. Talapatra, "Efficient searchable encryption through compression," *PVLDB*, vol. 11, no. 11, pp. 1729–1741, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1729-demertzis.pdf

[29] T. Hoang, A. A. Yavuz, and J. Guajardo, "Practical and secure dynamic searchable encryption via oblivious access on distributed data structure," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, 2016, pp. 302–313. [Online]. Available: http://dl.acm.org/citation.cfm?id=2991088

[30] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward private searchable symmetric encryption with optimized I/O efficiency," *IEEE Trans. Dependable Sec. Comput.*, 2018. [Online]. Available: https://doi.org/10.1109/TDSC.2018.2822294

[31] C. L. 8.1.0, https://www.cryptopp.com, 2019, last accessed 22 April 2019.

[32] D. F. Aranha and C. P. L. Gouvêa, "RELIC is an Efficient LIbrary for Cryptography," https://github.com/relic-toolkit/relic, 2014, last accessed 22 April 2019.

APPENDIX A
SECURITY PROOF OF SEED

To prove the forward and Type-III backward security of SEED, we construct a simulator $\mathcal{S}$, which takes leakage functions $\mathcal{L}^{Stp}(\lambda) = \lambda$, $\mathcal{L}^{Updt}(op, w, id) = \emptyset$, and $\mathcal{L}^{Srch}(w) = \{\text{sp}(w), \text{TimeDB}(w), \text{DelHist}(w)\}$ as inputs to simulate algorithm SEED.Setup and protocols SEED.Update and SEED.Search respectively, and demonstrate that the simulated SEED is indistinguishable from the real SEED under the adaptive attacks. Algorithm 4 describes the simulator $\mathcal{S}$. Specifically, the simulator $\mathcal{S}$ consists of the following three phases.

Setup Phase. In this phase, simulator $\mathcal{S}$ takes leakage function $\mathcal{L}^{Stp}(\lambda) = \lambda$ as input, initializes an empty database EDB for the server, an empty map UpdateList for recording the updated ciphertexts and their decryption tokens, and a variable u of timestamp, and sends EDB to the server. It is clear that $\mathcal{S}.\text{Setup}(\mathcal{L}^{Stp}(\lambda))$ is indistinguishable from the real algorithm SEED.Setup, since the simulated database EDB is the same as a real database.

Update Phase. When adversary $\mathcal{A}$ issues an `update` query (op, w, id), simulator $\mathcal{S}$ takes leakage function $\mathcal{L}^{Updt}(op, w, id)$ as input, computes the timestamp u of this `update` query, randomly picks a ciphertext and its decryption token (L, R, D, C, T), inserts (L, R, D, C, T) into UpdateList$[u]$, and sends (L, R, D, C) to the server. According to the security of PRF F, the randomness of oracle H, and the security of symmetric encryption, the simulated ciphertext (L, R, D, C) has the same distribution as the real ciphertext that is generated by protocol SEED.Update in the RO model. Hence, $\mathcal{S}.\text{Update}(\mathcal{L}^{Updt}(op, w, id))$ is indistinguishable from the real protocol SEED.Update.

Search Phase. When adversary $\mathcal{A}$ issues a `search` query w, simulator $\mathcal{S}$ takes leakage function $\mathcal{L}^{Srch}(w)$ as input and performs the following steps:

1) $\mathcal{S}$ computes the timestamp u of this `search` query and extracts the timestamp u'_s of the last `search` query of w if w has been searched before; otherwise, it sets $u'_s = 0$, and extracts the timestamps of all `update` queries of w that have occurred after the timestamp u'_s (lines 1-3);
2) If both $n = 0$ and $u'_s = 0$ hold, it means that adversary $\mathcal{A}$ never issue a `update` query of w; then, $\mathcal{S}$ returns $\perp$ (lines 4-6);
3) $\mathcal{S}$ extracts the timestamp u_0 of the last `add` query having $u_0 < u'_s$ and $u_0 \in$ TimeDB(w) and sets $u_0 = 0$ if there is no such kind of `add` query (line 7); if both $n = 0$ and $u_0 = 0$ hold, it means that either adversary $\mathcal{A}$ never issue an `add` query before timestamp u'_s or all added ciphertexts of w before timestamp u'_s have been removed by the adversary $\mathcal{A}$'s `delete` queries; then, $\mathcal{S}$ returns $\perp$ (lines 8-10);
4) $\mathcal{S}$ retrieves $(L_0, R_0, D_0, C_0, T_0)$ from UpdateList$[u_0]$; if both $n = 0$ and $u_0 \neq 0$ hold, it means that adversary $\mathcal{A}$ never issued any `update` query after timestamp u'_s, and there are still some matching ciphertexts that were added before timestamp u'_s; then, $\mathcal{S}$ sends search trapdoor (L_0, T_0) to the server and returns all file identifiers in TimeDB(w) after receiving the response from the server (lines 11-15);
5) $\mathcal{S}$ retrieves $(L_n, R_n, D_n, C_n, T_n)$ from UpdateList$[u_n]$, and programs random oracle H such that all ciphertexts updated after timestamp u'_s contain the correct information and construct a hidden chain relationship with the ciphertexts updated before timestamp u'_s (lines 16-26);
6) Finally, $\mathcal{S}$ sends search trapdoor (L_n, T_n) to the server and returns all file identifiers in TimeDB(w) after receiving the response from the server (lines 27 and 28).

Upon receiving the search trapdoor from simulator $\mathcal{S}$ in the above steps, the server implements the real search procedure to find all matching ciphertexts, except that the hash function H works as a random oracle. Moreover, the search results are correct since random oracle H has been programmed well. Hence, $\mathcal{S}.Search(\mathcal{L}^{Srch}(w))$ is indistinguishable from the real protocol SEED.Search in the RO model.

To summarize, the above simulator $\mathcal{S}$ takes leakage functions $\mathcal{L}^{Stp}(\lambda) = \lambda$, $\mathcal{L}^{Updt}(op, w, id) = \emptyset$, and $\mathcal{L}^{Srch}(w) = \{\text{sp}(w), \text{TimeDB}(w), \text{DelHist}(w)\}$ as inputs and simulates an ideal game of SEED that is indistinguishable from an real game of SEED under the adaptive attacks in the RO model.

APPENDIX B
SECURITY PROOF OF KEY-UPDATABLE PRF

Suppose $\mathcal{A}$ is a PPT adversary to break the key-updatable PRF instance in the experiment $\text{Expt}^{\text{PRF}}_{\mathcal{A},\text{P}}$. Algorithm 5 constructs a simulator $\mathcal{S}$ that can simulate the experiment $\text{Expt}^{\text{PRF}}_{\mathcal{A},\text{P}}$ in the RO model and leverage the capability of the adversary $\mathcal{A}$ to break the DDH assumption.

Next, we will prove that $\mathcal{S}$ correctly simulates the experiment $\text{Expt}^{\text{PRF}}_{\mathcal{A},\text{P}}$ in the view of $\mathcal{A}$ if $Z = g^{\alpha\beta}$; otherwise, $\mathcal{A}$ has no advantage to correctly guess b.

When $Z = g^{\alpha\beta}$, for any x, we have $g^{\beta \cdot r} = \text{H}(x)$ and $\text{P}(K_1, x) = g^{\alpha\beta r} = g^{K_1 \cdot \beta r} = \text{H}(x)^{K_1}$. It means that $\mathcal{S}$ correctly simulates function $\text{P}(K_1, \cdot)$ even if it does not know K_1. In addition, it is clear that the constructed function $f(\cdot)$ is a random one. Hence, we have that $\mathcal{S}$ is indistinguishable from the experiment $\text{Expt}^{\text{PRF}}_{\mathcal{A},\text{P}}$ in the view of $\mathcal{A}$ if $Z = g^{\alpha\beta}$. Formally, we have

$$\Pr[\text{Expt}^{\text{PRF}}_{\mathcal{A},\text{P}}(\lambda) = 1] = \Pr[\mathcal{S}(g^{\alpha}, g^{\beta}, g^{\alpha\beta}) = 1].$$

When $Z = g^{\gamma}$, for any x, we have $\text{P}(K_1, x) = g^{\gamma r}$ where r is random. It implies that for $i \in [1, n]$, $\text{P}(K_1, x_i)$ is indistinguishable from $f(x_i)$, since g^{γ} has the same random distribution with R, and $\mathcal{A}$ cannot query $\text{P}(K_1, x_i)$ or $f(x_i)$. Summarily, $\mathcal{A}$ has no advantage to correctly guess b if $Z = g^{\gamma}$. Formally, we have

$$\Pr[\mathcal{S}(g^{\alpha}, g^{\beta}, g^{\gamma}) = 1] = \frac{1}{2}.$$

According to the definition of the DDH assumption and the above results, we have

$$\begin{aligned} &|\Pr[\mathcal{S}(g^{\alpha}, g^{\beta}, g^{\alpha\beta}) = 1] - \Pr[\mathcal{S}(g^{\alpha}, g^{\beta}, g^{\gamma}) = 1]| \\ &= |\Pr[\text{Expt}^{\text{PRF}}_{\mathcal{A},\text{P}}(\lambda) = 1] - \frac{1}{2}| = \text{Adv}^{\text{PRF}}_{\mathcal{A},\text{P}}(\lambda). \end{aligned}$$

Algorithm 4 The Construction of $\mathcal{S}$ in the Ideal Game of SEED

Setup($\mathcal{L}^{Stp}(\lambda)$)
1: Initialize EDB $\leftarrow \emptyset$, an empty map UpdateList, and a global variable $u \leftarrow 0$
2: UpdateList[0] $\leftarrow (0^\lambda, \text{NULL}, \text{NULL}, \text{NULL}, 0^\lambda)$
3: Send EDB to the server

Update($\mathcal{L}^{Updt}(op, w, id)$)
1: $u \leftarrow u + 1$ ▷ the timestamp of the current `update` query
2: $(L, R, D, C, T) \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^{3\lambda+1} \times \mathcal{C}_{\text{SE}} \times \{0,1\}^\lambda$
3: UpdateList[u] $\leftarrow (L, R, D, C, T)$
4: Send ciphertext (L, R, D, C) to the server

Search($\mathcal{L}^{Srch}(w)$)
1: $u \leftarrow u + 1$ ▷ the timestamp of the current `search` query
2: Extract the timestamp u'_s of the last `search` query from sp(w) where $u'_s = 0$ if sp(w) $= \emptyset$
3: Extract all timestamps $\{u_1, \cdots, u_n\}$ between u'_s and u from both TimeDB(w) and DelHist(w) where $u_i < u_j$ if $i < j$
4: **if** $n = 0$ and $u'_s = 0$ **then**
5: **return** $\perp$
6: **end if**
7: Extract the maximum timestamp u_0 less than u'_s from TimeDB(w), where $u_0 = 0$ if $u'_s = 0$ or no such kind of u_0 exists
8: **if** $n = 0$ and $u_0 = 0$ **then**
9: **return** $\perp$
10: **end if**
11: $(L_0, R_0, D_0, C_0, T_0) \leftarrow$ UpdateList[u_0]
12: **if** $n = 0$ and $u_0 \neq 0$ **then**
13: Send search trapdoor (L_0, T_0) to the server
14: **return** all file identifiers in TimeDB(w) after receiving the response from the server
15: **end if**
16: $(L_n, R_n, D_n, C_n, T_n) \leftarrow$ UpdateList[u_n]
17: **for** $i = n$ to 1 **do**
18: $(L_{i-1}, R_{i-1}, D_{i-1}, C_{i-1}, T_{i-1}) \leftarrow$ UpdateList[u_{i-1}]
19: **if** u_i is the timestamp of an `add` query **then**
20: Program H s.t. $\text{H}(T_i, R_i) = D_i \oplus (add || 0^\lambda || L_{i-1} || T_{i-1})$
21: **else**
22: Let u^{add} be the timestamp s.t. $(u^{add}, u_i) \in$ DelHist(w)
23: $(L^{add}, R^{add}, D^{add}, C^{add}, T^{add}) \leftarrow$ UpdateList[u^{add}]
24: Program H s.t. $\text{H}(T_i, R_i) = D_i \oplus (del || L^{add} || L_{i-1} || T_{i-1})$
25: **end if**
26: **end for**
27: Send search trapdoor (L_n, T_n) to the server
28: **return** all file identifiers in TimeDB(w) after receiving the response from the server

Algorithm 5 The Contruction of $\mathcal{S}$ in the RO Model

$\mathcal{S}(g^\alpha, g^\beta, Z)$ ▷ Note that $Z = g^{\alpha\beta}$ or g^γ
1: Initialize two empty maps HList and FList
2: $b \xleftarrow{\$} \{0,1\}$, $\Delta \xleftarrow{\$} \mathcal{K}_{\text{P}}$, and $R \xleftarrow{\$} \mathbb{G}$
3: $(x_1, \cdots, x_n, st) \leftarrow \mathcal{A}^{\text{H}(\cdot), \text{P}(K_1, \cdot), f(\cdot)}(\lambda, \Delta)$
4: **if** $b = 1$ **then**
5: $b' \leftarrow \mathcal{A}^{\text{H}(\cdot), \text{P}(K_1, \cdot), f(\cdot)}(\{\text{P}(K_1, x_i) | i \in [1, n]\}, st)$
6: **else**
7: $b' \leftarrow \mathcal{A}^{\text{H}(\cdot), \text{P}(K_1, \cdot), f(\cdot)}(\{f(x_i) | i \in [1, n]\}, st)$
8: **end if**
9: **return** 1 if $b = b'$; otherwise, **return** 0

H(x)
1: **if** x has been queried before **then**
2: $(r, g^{\beta \cdot r}) \leftarrow$ HList[x]
3: **else**
4: $r \xleftarrow{\$} \mathbb{Z}_q^*$ and HList[x] $\leftarrow (r, g^{\beta \cdot r})$
5: **end if**
6: **return** $g^{\beta \cdot r}$

P(K_1, x)
1: Query H(x) if x was never queried before
2: $(r, g^{\beta \cdot r}) \leftarrow$ HList[x]
3: **return** Z^r ▷ It implies that $K_1 = \alpha$ and $K_2 = \alpha \cdot \Delta$

$f(x)$
1: **if** x has been queried before **then**
2: $(r, R^r) \leftarrow$ FList[x]
3: **else**
4: $r \xleftarrow{\$} \mathbb{Z}_q^*$ and FList[x] $\leftarrow (r, R^r)$
5: **end if**
6: **return** R^r

It means that if the DDH assumption holds, the advantage of $\mathcal{A}$ must be negligible when it breaks the security of the key-updatable PRF instance in the RO model.

APPENDIX C
SECURITY PROOF OF ROSE

The security proof of ROSE is analogous to the proof of SEED. To prove the forward and general Type-III backward security, we construct a simulator $\mathcal{S}$, which takes leakage functions $\mathcal{L}^{Stp}(\lambda) = \lambda$, $\mathcal{L}^{Updt}(op, w, id) = \emptyset$, and $\mathcal{L}^{Srch}(w) = \{\text{sp}(w), \text{exTimeDB}(w), \text{exDelHist}(w)\}$ as inputs to simulate algorithm ROSE.Setup and protocols ROSE.Update and ROSE.Search respectively, and demonstrate that the simulated ROSE is indistinguishable from the real ROSE under the adaptive attacks. Algorithm 6 describes the simulator $\mathcal{S}$. Specifically, the simulator $\mathcal{S}$ consists of the following three phases.

Setup Phase. In this phase, simulator $\mathcal{S}$ takes leakage function $\mathcal{L}^{Stp}(\lambda) = \lambda$ as input, initializes an empty database EDB, four empty maps CipherList, UList, PList and TList, and a variable u of the timestamp, and sends EDB to the server. Taking a timestamp as input, maps CipherList and TList return a ciphertext and the corresponding decryption token, respectively, both of which are generated by the `update` or `search` query at the input timestamp; map UList returns a key-update token that is generated by the `search` query at the input timestamp; map PList returns a pair of PRF P values, where one of them is used by the `update` query at the input timestamp to generate a ciphertext. It is clear that $\mathcal{S}$.Setup($\mathcal{L}^{Stp}(\lambda)$) is indistinguishable from the real algorithm ROSE.Setup, since the simulated database EDB is the same as a real database.

Update Phase. When adversary $\mathcal{A}$ issues an `update` query (op, w, id), simulator $\mathcal{S}$ takes leakage function $\mathcal{L}^{Updt}(op, w, id)$ as input, computes the timestamp u of this `update` query, randomly picks a ciphertext (L, R, D, C), inserts this ciphertext into CipherList[u], and sends this ciphertext to the server. According to randomness of oracles H and G and the security of symmetric encryption, the simulated ciphertext (L, R, D, C) has the same distribution as the real ciphertext that is generated by protocol ROSE.Update in the RO model. Hence, $\mathcal{S}$.Update($\mathcal{L}^{Updt}(op, w, id)$) is indis-

Algorithm 6 The Construction of $\mathcal{S}$ in the Ideal Game of ROSE

Setup($\mathcal{L}^{Stp}(\lambda)$)

1: Initialize EDB $\leftarrow \emptyset$, four empty maps CipherList, UList, PList and TList, and a global variable $u \leftarrow 0$
2: CipherList[0] $\leftarrow (0^\lambda, \text{NULL}, \text{NULL}, \text{NULL})$ and TList[0] $\leftarrow 0^\lambda$
3: Send EDB to the server

Update($\mathcal{L}^{Updt}(op, (w, id))$)

1: $u \leftarrow u + 1$ ▷ the timestamp of the current `update` operation
2: $(L, R, D, C) \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^{2\lambda+\lambda'+2} \times \mathcal{C}_{\text{SE}}$
3: CipherList[u] $\leftarrow (L, R, D, C)$
4: Send ciphertext (L, R, D, C) to the server

Search($\mathcal{L}^{Srch}(w)$)

1: $u \leftarrow u + 1$ ▷ the timestamp of the current `search` query
2: Extract the timestamp u_0 of the last search query from sp(w) where $u_0 = 0$ if sp$(w) = \emptyset$
3: Extract all timestamps $\{u_1, \cdots, u_n\}$ between u_0 and u from both exTimeDB(w) and exDelHist(w) and exclude the repeated timestamps, where $u_i < u_j$ if $i < j$
4: **if** $n = 0$ and $u_0 = 0$ **then**
5: **return** $\perp$
6: **end if**
7: Extract the maximum timestamp u_{max} less than u_0 from exTimeDB(w), where $u_{max} = 0$ if $u_0 = 0$ or no such kind of u_{max} exists
8: **if** $n = 0$ and $u_{max} = 0$ **then**
9: **return** $\perp$
10: **end if**
11: $(L_u, R_u, D_u, C_u) \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^{2\lambda+\lambda'+2} \times \mathcal{C}_{\text{SE}}$
12: CipherList[u] $\leftarrow (L_u, R_u, D_u, C_u)$
13: $P_u \xleftarrow{\$} \mathcal{Y}_{\text{P}}$ and TList[u] $\leftarrow T_u \xleftarrow{\$} \mathcal{Y}_{\text{F}}$
14: UList[u] $\leftarrow \Delta_u \xleftarrow{\$} \mathcal{K}_{\text{P}}$
15: Program G s.t. G$(P_u, R_u) = L_u$
16: **if** $n = 0$ and $u_{max} \neq 0$ **then**
17: $(L_{u_0}, R_{u_0}, D_{u_0}, C_{u_0}) \leftarrow$ CipherList[u_0] and $T_{u_0} \leftarrow$ TList[u_0]
18: Program H s.t. H$(T_u, R_u) = D_u \oplus (srch||\Delta_u||L_{u_0}||T_{u_0})$
19: Send search trapdoor (L_{u_0}, T_{u_0}) and ciphertext (L_u, R_u, D_u, C_u) to the server
20: **return** all file identifiers in exTimeDB(w) after receiving the response from the server
21: **end if**
22: **for** each $(\mathcal{U}^{add}, \mathcal{U}^{del}) \in$ exDelHist(w) and $\mathcal{U} \in$ exTimeDB(w) **do**
23: $u_{min} \leftarrow$ Min$(\mathcal{U}^{add} \cup \mathcal{U}^{del})$ or Min$(\mathcal{U})$
24: **if** PList[u_{min}] = (NULL, NULL) **then**
25: PList[u_{min}] $\leftarrow (P_{add}, P_{del}) \xleftarrow{\$} \mathcal{Y}_{\text{P}} \times \mathcal{Y}_{\text{P}}$
26: **else**
27: $(P_{add}, P_{del}) \leftarrow$ PList[u_{min}]
28: **end if**
29: Orderly extract all key-update tokens $\{\Delta_1, \cdots, \Delta_m\}$ with their timestamps between u_{min} and u from UList
30: **for** all $v \in \mathcal{U}^{add} \cup \mathcal{U}^{del}$ or $\mathcal{U}$, PList[v] $\neq$ (NULL, NULL), TList[v] $\neq$ NULL, and $u_{min} < v < u$ **do**
31: **if** $m \neq 0$ **then**
32: PList[v] $\leftarrow (P_{add}^{\prod_{i=1}^{m} \Delta_i^{-1}}, P_{del}^{\prod_{i=1}^{m} \Delta_i^{-1}})$
33: **else**
34: PList[v] $\leftarrow (P_{add}, P_{del})$
35: **end if**
36: Set TList[v] $\leftarrow T_v \xleftarrow{\$} \mathcal{Y}_{\text{F}}$
37: **end for**
38: **end for**
39: $(L_{u_n}, R_{u_n}, D_{u_n}, C_{u_n}) \leftarrow$ CipherList[u_n] and $T_{u_n} \leftarrow$ TList[u_n]
40: **for** $i = n$ to 1 **do**
41: $(P_{add}, P_{del}) \leftarrow$ PList[u_i]
42: $(L_{u_{i-1}}, R_{u_{i-1}}, D_{u_{i-1}}, C_{u_{i-1}}) \leftarrow$ CipherList[u_{i-1}]
43: $T_{u_{i-1}} \leftarrow$ TList[u_{i-1}]
44: **if** u_i is corresponding to an `add` query **then**
45: Program G s.t. G$(P_{add}, R_{u_i}) = L_{u_i}$
46: Program H s.t. H$(T_{u_i}, R_{u_i}) = D_{u_i} \oplus (add||0^\lambda||L_{u_{i-1}}||T_{u_{i-1}})$
47: **else**
48: Program G s.t. G$(P_{del}, R_{u_i}) = L_{u_i}$
49: Program H s.t. H$(T_{u_i}, R_{u_i}) = D_{u_i} \oplus (del||P_{add}||L_{u_{i-1}}||T_{u_{i-1}})$
50: **end if**
51: **end for**
52: Program H s.t. H$(T_u, R_u) = D_u \oplus (srch||\Delta_u||L_{u_n}||T_{u_n})$
53: Send search trapdoor (L_{u_n}, T_{u_n}) and ciphertext (L_u, R_u, D_u, C_u) to the server
54: **return** all file identifiers in exTimeDB(w) after receiving the response from the server

tinguishable from the real protocol ROSE.Update.

Search Phase. When adversary $\mathcal{A}$ issues a `search` query w, simulator $\mathcal{S}$ takes leakage function $\mathcal{L}^{Srch}(w)$ as input and performs the following steps:

1) $\mathcal{S}$ computes the timestamp u of this `search` query and extracts the timestamp u_0 of the last `search` query of w if w has been searched before; otherwise, it sets $u_0 = 0$ and extracts the timestamps of all `update` queries of w that occurred after timestamp u_0 (lines 1-3);
2) If both $n = 0$ and $u_0 = 0$ hold, it means that adversary $\mathcal{A}$ never issued an `update` query of w; then, $\mathcal{S}$ returns $\perp$ (lines 4-6);
3) $\mathcal{S}$ extracts the timestamp u_{max} of the last `add` query of w having $u_{max} < u_0$ and sets $u_{max} = 0$ if $u_0 = 0$ or there is no such kind of `add` query (line 7); if both $n = 0$ and $u_{max} = 0$ hold, it means that either adversary $\mathcal{A}$ never issued an `add` query of w before timestamp u_0 or all added ciphertexts of w before timestamp u_0 have been removed by the adversary $\mathcal{A}$'s `delete` queries; then, $\mathcal{S}$ returns $\perp$ (lines 8-10);
4) $\mathcal{S}$ randomly picks a ciphertext (L_u, R_u, D_u, C_u), inserts this ciphertext into CipherList[u], randomly picks P_u from $\mathcal{Y}_{\text{P}}$, a decryption token T_u from $\mathcal{Y}_{\text{F}}$ and a key-update token Δ_u from $\mathcal{K}_{\text{P}}$, inserts T_u and Δ_u into TList[u] and UList[u] respectively, and programs oracle G such that L_u can be correctly generated by taking P_u and R_u as inputs (lines11-15);
5) If both $n = 0$ and $u_{max} \neq 0$ hold, it means that $\mathcal{A}$ does not issue any `update` query of w after the last `search` query (or timestamp u_0), and there are still some matching ciphertexts that were added before the last `search` query (or timestamp u_0); then, $\mathcal{S}$ retrieves the ciphertext $(L_{u_0}, R_{u_0}, D_{u_0}, C_{u_0})$ from CipherList[u_0] and the corresponding decryption token T_{u_0} from TList[u_0], programs oracle H such that ciphertext (L_u, R_u, D_u, C_u) contains the correct information and constructs a hidden chain

relationship with ciphertext $(L_{u_0}, R_{u_o}, D_{u_0}, C_{u_0})$, sends search trapdoor (L_{u_0}, T_{u_0}) and ciphertext (L_u, R_u, D_u, C_u) to the server, and returns all file identifiers in exTimeDB(w) after receiving the response from the server (lines 16-21); (Note that if $n \neq 0$ holds, it means that $\mathcal{A}$ issued some `update` queries of w after the last `search` query (or timestamp u_0); then, $\mathcal{S}$ will compute some values for the ciphertexts generated by those `update` queries and program oracles G and H such that these ciphertexts can be correctly found or removed by the server when receiving a correct search trapdoor.)

6) For each $(\mathcal{U}^{add}, \mathcal{U}^{del}) \in$ exDelHist(w) and $\mathcal{U} \in$ exTimeDB(w), $\mathcal{S}$ performs the following steps (lines 22-38):
   a) Extracts the minimum value u_{min} from $\mathcal{U}^{add} \cup \mathcal{U}^{del}$ or $\mathcal{U}$ (line 23);
   b) Randomly picks (P_{add}, P_{del}) from $\mathcal{Y}_{\mathrm{P}} \times \mathcal{Y}_{\mathrm{P}}$ if PList$[u_{min}]$ is empty; otherwise, retrieve (P_{add}, P_{del}) from PList$[u_{min}]$ (lines 24-28);
   c) Extracts all key-update tokens with timestamps between u_{min} and u (line 29);
   d) Finally, for all $v \in \mathcal{U}^{add} \cup \mathcal{U}^{del}$ or $\mathcal{U}$, PList$[v] \neq$ (NULL, NULL), TList$[v] \neq$ NULL, and $u_{min} < v < u$, computes a pair of PRF P values according to the above extracted key-update tokens, picks a random delete token, and inserts them into the corresponding maps (lines 31-38);
7) For the ciphertexts stored in CipherList with timestamps $\{u_1, ..., u_n\}$, programs oracles G and H according to their operation types and precomputed values stored in maps PList and TList, such that these ciphertexts can be correctly found or removed by the server in the future (lines 39-51);
8) Programs oracle H such that the ciphertext (L_u, R_u, D_u, C_u) can be connected with ciphertext CipherList$[u_n]$ by a hidden chain relationship, sends search trapdoor (L_{u_n}, T_{u_n}) and ciphertext (L_u, R_u, D_u, C_u) to the server, and returns all file identifiers in exTimeDB(w) after receiving the response from the server (lines 52-54).

In the above steps, upon receiving the search trapdoor from simulator $\mathcal{S}$, the server implements the real search procedure to find all matching ciphertexts, except that the hash functions G and H work as two random oracles. Moreover, the search results are correct since oracles G and H have been programmed well. Hence, $\mathcal{S}.Search(\mathcal{L}^{Srch}(w))$ is indistinguishable from the real protocol ROSE.Search in the RO model.

To summarize, the above simulator $\mathcal{S}$ takes leakage functions $\mathcal{L}^{Stp}(\lambda) = \lambda$, $\mathcal{L}^{Updt}(op, w, id) = \emptyset$, and $\mathcal{L}^{Srch}(w) = \{\mathrm{sp}(w), \text{exTimeDB}(w), \text{exDelHist}(w)\}$ as inputs and simulates an ideal game of ROSE that is indistinguishable from an real game of ROSE under the adaptive attacks in the RO model.