ROSE

Robust Searchable Encryption with Forward and Backward Security

Xu, Peng; Susilo, Willy; Wang, Wei; Chen, Tianyang ; Wu, Qianhong; Liang, Kaitai; Jin, Hai

DOI
10.1109/TIFS.2022.3155977

Publication date
2022

Document Version
Final published version

Published in
IEEE Transactions on Information Forensics and Security

ROSE: Robust Searchable Encryption With Forward and Backward Security

Peng Xu, *Member, IEEE*, Willy Susilo, *Fellow, IEEE*, Wei Wang, *Member, IEEE*, Tianyang Chen, Qianhong Wu, *Member, IEEE*, Kaitai Liang, *Member, IEEE*, and 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 and give the experimental comparisons.**

***Index Terms*— Searchable symmetric encryption, forward and backward security, robustness, key-updatable pseudo-random function.**

Manuscript received October 20, 2021; revised January 10, 2022 and February 4, 2022; accepted February 20, 2022. Date of publication March 2, 2022; date of current version March 22, 2022. The work of Peng Xu was supported in part by the National Key Research and Development Program of China under Grant 2021YFB3101304, in part by the Wuhan Applied Foundational Frontier Project under Grant 2020010601012188, in part by the National Natural Science Foundation of China under Grant 61872412, and in part by the Guangdong Provincial Key Research and Development Plan Project under Grant 2019B010139001. The work of Qianhong Wu was supported in part by the Beijing Natural Science Foundation under Grant M21031; in part by the Natural Science Foundation of China under Grant U21A20467, Grant 61932011, and Grant 61972019; and in part by the Populus euphratica found under Grant CCF-HuaweiBC2021009. The work of Kaitai Liang was supported by the European Union's Horizon 2020 Research and Innovation Programme under Agreement 952697 (ASSURED) and Agreement 101021727 (IRIS). The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Vanesa Daza. *(Corresponding author: Wei Wang.)*

Peng Xu, Tianyang Chen, and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Laboratory, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: xupeng@mail.hust.edu.cn; chentianyang@mail.hust.edu.cn; hjin@mail.hust.edu.cn).

Willy Susilo is with the School of Computing and Information Technology, Institute of Cybersecurity and Cryptology, University of Wollongong, Wollongong, NSW 2522, Australia (e-mail: wsusilo@uow.edu.au).

Wei Wang is with the Cyber-Physical-Social Systems Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: viviawangww@gmail.com).

Qianhong Wu is with the School of Electronic and Information Engineering, Beihang Univerisity, Beijing 100191, China (e-mail: qianhong.wu@buaa.edu.cn).

Kaitai Liang is with the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2628 Delft, The Netherlands (e-mail: kaitai.liang@tudelft.nl).

Digital Object Identifier 10.1109/TIFS.2022.3155977

I. 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 [8]–[11]. Table I 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: Most of these DSSE schemes are unable to handle irrational `update` queries (e.g., adding or deleting the same entry repeatedly and deleting the non-existent entry) issued by the client due to human errors. They will be ineffective or insecure if the client issues the irrational `update`

TABLE I

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, W_{MAX} IS THE SUPPORTED MAXIMUM NUMBER OF KEYWORDS, F IS THE TOTAL NUMBER OF FILES, AND F_{MAX} IS THE SUPPORTED MAXIMUM 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. VI). 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
IM-DSSE$_{\text{II}}$ [15]	✗	$O(F_{\max})$	$O(W_{\max})$	$O(F_{\max})$	2	$O(W_{\max})$	$O(W_{\max} + F_{\max})$	III
IM-DSSE$_{\text{I+II}}$ [15]	✗	$O(F_{\max})$	$O(W_{\max})$	$O(F_{\max})$	2	$O(W_{\max})$	$O(W_{\max} + F_{\max})$	III
ROSE [Sec. VI]	✔	$O((n_w + s'_w + 1)d_w)$	$O(1)$	$O(n_w)$	2	$O(1)$	$O(W \log F)$	III

queries, say, *they are not robust*. System robustness is of significance in practice, and it is easily affected by human errors. In real-world information systems, human error is known as a key factor incurring security breaches. As reported by Verizon, 85% of data breaches involve human mistakes, wherein misdelivery is the top cause of data breaches in the healthcare industry [16]. Further, even if given a DSSE system supporting multiple clients, as described in work [17], we may still suffer from insider attacks. An inside and malicious client can leverage the robustness problem and intentionally issue irrational `update` queries so as to illegally learn plaintext information from the encrypted database or launch a denial-of-service attack. Traditional database systems, like MySQL, seem to be able to get rid of human errors or insider attacks by using integrity constraint mechanisms to prevent irrational `update` operations. But there has not been any DSSE work formally investigating the robustness problem. We also note that a few DSSE schemes are robust (in our definition), but their computation and communication costs are relatively high. Referring to Table I, the details of the above problems are as follows.

Diana$_{del}$ [8], ORION [11], and HORUS [11] cannot guarantee forward security when adding or deleting the 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. 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. 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. Although both IM-DSSE$_{\text{II}}$ and IM-DSSE$_{\text{I+II}}$ build oblivious search indexes using encrypted matrices, they will extra leak which file is being updated when issuing the duplicate `add` or `del` queries. When issuing an `add` or a `del` query, both IM-DSSE$_{\text{II}}$ and IM-DSSE$_{\text{I+II}}$ clients inform the server which file is needed to be updated by uploading the corresponding column index. Since the column index is statically calculated with the client's secret key and the file identifier, when giving two duplicate `add` (or `del`) queries, the server knows that the latter query updates the file added (or deleted) by the former query. This extra leakage violates their declared `update` leakage functions.

After deleting a non-existent entry, 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. However, Diana$_{del}$ and Janus explicitly mentioned that they fail to achieve this function. Janus++ has the same problem since it has the same essence as Janus when deleting an entry. 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.

All existing DSSE works miss studying robustness. But, Moneta [8] and Fides [8] unintentionally achieve this feature with a very high cost. 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 [18], 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 I.

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. II) and introducing some background knowledge about searchable encryption and forward and backward security (Sec. III). Our contributions can be summarized as follows.

To the best of our knowledge, this paper is the first one to define robustness of DSSE (Sec. IV). 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.

To construct a robust DSSE scheme, we define a new cryptographic primitive called key-updatable PRF and construct its instance based on an early PRF scheme [19] (Sec. V). 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 [20], 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. V.

Finally, we construct a robust DSSE scheme named ROSE (Sec. VI). ROSE is forward and Type-III backward secure under the adaptive attacks. It has the efficient complexity in most terms of computation, communication and search costs as shown in Table I. Since ROSE applies an ingenious design to make $s'_w \leq \mathrm{Min}(s_w, n_w + 1)$ hold, where $\mathrm{Min}(s_w, n_w + 1)$ 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. VII tests the performance of ROSE comprehensively.

II. DSSE Schemes Revisited

SSE was first introduced by Song *et al.* [1]. Later, Chang *et al.* proposed a forward secure SSE scheme [21]. 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.

Two existing DSSE schemes explicitly assume that the client never issues the irrational `update` queries. Clearly, these schemes are not robust. For example, Cash *et al.* [22] 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], [23], [24], and [9], [15], [25], [26] cannot keep their claimed correctness or security. For example, in [23], [24], and [9], the information leakage caused by the duplicate `add` queries will be beyond the limitation of their security definitions; 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 [26], 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 [25], 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 [23] and [27] leak the relationship of those duplicate `delete` queries, which is beyond their security definitions. The DSSE schemes in [28] and [7], [29], [30] 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 [31] and [32], respectively, that are robust. 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, but it is not backward secure. In addition, DOD-DSSE must set the maximum numbers of keywords and files when initializing it.

Another important research line on DSSE is to leverage the trusted hardware to protect the `update` and `search` queries [33]–[35]. Enclaves (in other words, trusted third party) used in those works can trace the full states of the encrypted database, while in traditional DSSE works, the only trusted party (namely, the client) maintains very limited information about the encrypted database. With the extra security assumption offered by the trusted hardware, the robustness may be achieved. In this work, we attempt to capture robustness without using the trusted hardware assumption.

III. 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 and semantically secure under chosen plaintext attack (SS-CPA) at least.

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, σ is the client's local state, and EDB is an empty encrypted database that is sent to the server.
- $\Sigma.\text{Update}(K, \sigma, op, (w, id); \text{EDB})$: For adding an entry to or deleting an entry from EDB as the client's request, 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, and the server takes EDB as input. After the protocol, the pair (w, id) is added to or deleted from EDB.
- $\Sigma.\text{Search}(K, \sigma, w; \text{EDB})$: For searching EDB as the client's query, the client takes his secret key K, local state σ, and a keyword w as inputs, and 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. 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 security (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 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. Hence, the above three functions are formally constructed from the query list Q as follows.

$$\begin{aligned}\text{sp}(w) &= \{u|(u, w) \in Q\}\\ \text{TimeDB}(w) &= \{(u, id)|(u, add, (w, id)) \in Q\\ &\quad \text{and } \forall u', (u', del, (w, id)) \notin Q\}\\ \text{DelHist}(w) &= \{(u^{add}, u^{del})|\exists id, (u^{add}, add, (w, id)) \in Q\\ &\quad \text{and } (u^{del}, del, (w, id)) \in Q\}\end{aligned}$$

With the above leakage functions, the forward and Type-III backward security is 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 $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op, w)$ and $\mathcal{L}^{Srch}(w) = \mathcal{L}''$(sp(w), TimeDB(w), DelHist(w)), where both $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.*

Note that the definitions of both leakage functions TimeDB(w) and 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 security also have the same problem). Fortunately, this problem does not exist in the definition of forward security.

IV. 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 TimeDB(w) and DelHist(w) and name the resulting leakage functions as exTimeDB(w) and exDelHist(w) respectively. Their formal definitions are as follows, where $\mathcal{U}$ denotes the set of all satisfactory timestamps.

$$\begin{aligned}&\text{exTimeDB}(w)\\ &= \{(\mathcal{U}, id)|\forall u \in \mathcal{U}, (u, add, (w, id)) \in Q\\ &\quad \text{and } \forall u' > u, (u', del, (w, id)) \notin Q\}.\\ &\text{exDelHist}(w)\\ &= \{(\mathcal{U}^{add}, \mathcal{U}^{del})|\exists id, \forall u^{add} \in \mathcal{U}^{add} \text{ and } u^{del} \in \mathcal{U}^{del},\\ &\quad (u^{add}, add, (w, id)) \in Q \text{ and } (u^{del}, del, (w, id)) \in Q\}\end{aligned}$$

In addition, both exTimeDB(w) and 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*

$$\begin{aligned}\mathcal{L}^{Updt}(op, w, id) &= \mathcal{L}'(op, w)\\ \mathcal{L}^{Srch}(w) &= \mathcal{L}''(sp(w), exTimeDB(w), exDelHist(w))\end{aligned}$$

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 security can be extended to obtain more generality.

V. 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 1.*

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 $\{x_1, \ldots, x_n\}$; 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.

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

$b \xleftarrow{\$} \{0,1\}$, $(K_1, K_2) \xleftarrow{\$} \mathcal{K}_\text{P} \times \mathcal{K}_\text{P}$, 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.

Fig. 1. 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 [20], 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 [19], 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

$$\begin{aligned}
&\text{P}(K, x) = \text{H}(x)^K \\
&\Delta_{K_1 \to K_2} = \text{P.UpdateToken}(K_1, K_2) = K_1^{-1} \cdot K_2 \\
&\text{P.KeyUpdate}(\Delta_{K_1 \to K_2}, \text{P}(K_1, x)) \\
&\qquad = \text{P}(K_1, x)^{\Delta_{K_1 \to K_2}}
\end{aligned}$$

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 Proof: The security of the above key-updatable PRE instance relies on the DDH assumption in the RO model. Let $\mathbb{G}$ be a finite cyclic and multiplicative group of prime order q where $|q| = \text{poly}(\lambda)$. We say that the DDH assumption holds if for all sufficiently large λ and PPT adversary $\mathcal{A}$, its advantage defined as $\text{Adv}^{\text{DDH}}_{\mathcal{A}} = |\text{Pr}[\mathcal{A}(g^\alpha, g^\beta, g^{\alpha\beta}) = 1] - \text{Pr}[\mathcal{A}(g^\alpha, g^\beta, g^\gamma) = 1]|$ is negligible in λ, where $(\alpha, \beta, \gamma) \xleftarrow{\$} \mathbb{Z}_q^* \times \mathbb{Z}_q^* \times \mathbb{Z}_q^*$. Then, we have the following security theorem.

Algorithm 1 The Contruction of $\mathcal{S}$ in the RO Model

$\underline{\mathcal{S}(g^\alpha, g^\beta, Z)}$ ▷ Note that $Z = g^{\alpha\beta}$ or g^γ

Initialize two empty maps HList and FList
$b \xleftarrow{\$} \{0, 1\}$, $\Delta \xleftarrow{\$} \mathcal{K}_\text{P}$, and $R \xleftarrow{\$} \mathbb{G}$
$(x_1, \cdots, x_n, st) \leftarrow \mathcal{A}^{\text{H}(\cdot),\text{P}(K_1,\cdot),f(\cdot)}(\lambda, \Delta)$
if $b = 1$ **then**
  $b' \leftarrow \mathcal{A}^{\text{H}(\cdot),\text{P}(K_1,\cdot),f(\cdot)}(\{\text{P}(K_1, x_i)|i \in [1, n]\}, st)$
else
  $b' \leftarrow \mathcal{A}^{\text{H}(\cdot),\text{P}(K_1,\cdot),f(\cdot)}(\{f(x_i)|i \in [1, n]\}, st)$
end if
return 1 if $b = b'$; otherwise, **return** 0

$\underline{\text{H}(x)}$

if x has been queried before **then**
  $(r, g^{\beta \cdot r}) \leftarrow \text{HList}[x]$
else
  $r \xleftarrow{\$} \mathbb{Z}_q^*$ and $\text{HList}[x] \leftarrow (r, g^{\beta \cdot r})$
end if
return $g^{\beta \cdot r}$

$\underline{\text{P}(K_1, x)}$

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

$\underline{f(x)}$

if x has been queried before **then**
  $(r, R^r) \leftarrow \text{FList}[x]$
else
  $r \xleftarrow{\$} \mathbb{Z}_q^*$ and $\text{FList}[x] \leftarrow (r, R^r)$
end if
return R^r

Theorem 1: 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.

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

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

According to the definition of the DDH assumption and the above results, we have $|\text{Pr}[\mathcal{S}(g^\alpha, g^\beta, g^{\alpha\beta}) = 1] - \text{Pr}[\mathcal{S}(g^\alpha, g^\beta, g^\gamma) = 1]| = |\text{Pr}[\text{Expt}^{\text{PRF}}_{\mathcal{A},\text{P}}(\lambda) = 1] - \frac{1}{2}| = \text{Adv}^{\text{PRF}}_{\mathcal{A},\text{P}}(\lambda)$. It means that if the DDH assumption holds, 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 $\mathrm{EDB}[L] \leftarrow (R, D, C)$

advantage of $\mathcal{A}$ must be negligible when it breaks the security of the key-updatable PRF instance in the RO model. □

VI. ROSE: Our DSSE Scheme

In this section, we introduce the main ideas to design ROSE, apply key-updatable PRF to constructing ROSE, and finally analyze ROSE.

Roughly, for each keyword, ROSE 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.

To guarantee robustness, ROSE designs a probabilistic algorithm to generate each ciphertext's index, such that all ciphertexts have different indexes even if they are generated by the duplicate `update` queries. However, this idea also causes a new challenge 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. ROSE makes the n generated ciphertexts have n different indexes. To remove all those ciphertexts by one `delete` query, ROSE allows the ciphertext generated by the `delete` query to contain a constant-size delete token, which enables the server to find all ciphertexts that were generated by the previous duplicate and relevant `add` queries.

To guarantee forward security when a `search` query occurs in the middle of the relevant `delete` and re-`add` queries, the ROSE client will randomly choose new secret keys to generate the ciphertexts in the subsequent `update` queries (including the subsequent re-`add` queries) after the `search` query. 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, this idea 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 $\mathbf{C}_a$ and $\mathbf{C}_d$ be the two ciphertexts generated by these `add` and `delete` queries. According to the above idea, these two ciphertexts were generated with different secret keys. Hence, the delete token contained in $\mathbf{C}_d$ could be ineffective for deleting the ciphertext with index $\mathbf{C}_a$. Fortunately, key-updatable PRF can overcome this challenge. It can update the secret key of the delete token to the secret key of ciphertext $\mathbf{C}_a$. Hence, it can be used to delete the ciphertext $\mathbf{C}_a$. 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. To generate L, the client

Algorithm 3 Protocol ROSE.Search(K_Σ, σ, w; EDB)

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

Server:
1: Store 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: $(R', D', C') \leftarrow$ EDB[L']
6: $op'||X'||L''||T'' \leftarrow D' \oplus \mathrm{H}(T', R')$
7: **if** $op' = del$ **then**
8: Remove ciphertext (L', R', D', C') from EDB
9: $\mathcal{D} \leftarrow \mathcal{D} \cup \{X'\}$ ▷ Note that $X' \in \mathcal{Y}_\mathrm{P}$ 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 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. $L' = \mathrm{G}(A, R')$ **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 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: **endif**
25: **endif**
26: **if** $op' = srch$ **then**
27: **if** $op^t = srch$ and $\Delta^t \neq$ 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 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 $X' \in \mathcal{K}_\mathrm{P}$ if $op' = srch$
33: **end if**
34: **if** ($op^t = srch$ and $\Delta^t =$ 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: $A \leftarrow$ 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: LastKey[w] $\leftarrow$ (NULL, NULL)
3: LastUp[w] $\leftarrow$ (NULL, NULL, NULL)
4: **return** $\perp$
5: **end if**
6: **for** $i = 1$ to $|\mathcal{I}|$ **do**
7: $id_i \leftarrow$ SE.Dec($K_{\mathrm{SE}}, \mathcal{I}[i]$)
8: **end for**
9: **return** $\{id_i | i \in [1, |\mathcal{I}|]\}$

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). The content of D contains the delete token $X = \mathrm{P}(K', w||id||add)$ 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. 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 can find all matching ciphertexts and delete all expected ciphertexts from them. Namely, only the still-valid ciphertexts will be sent to the client. To delete the matching-but-invalid ciphertexts, (1) the ROSE server applies the delete tokens disclosed from the already found ciphertexts to test if a newly 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 cases (this step is for saving the time cost of the next `search` query). More 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 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 $\Delta_{K\rightarrow K'}$, 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 $op||\Delta_{K\rightarrow K'}||L'||T'$ (lines 7 and 8). Finally, the client updates LastUp[w] and 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')

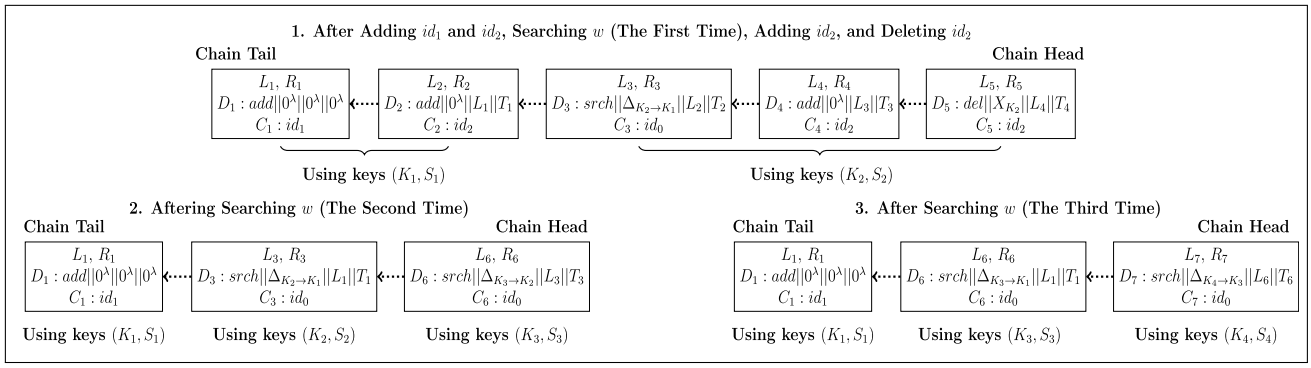


Fig. 2. An example of ROSE.

and (L, R, D, C) from the client, the server inserts (R, D, C) into 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 (lines 4-43). 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).

An 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 2 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 2 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, sequentially finds all matching ciphertexts $\{C_5, C_4, C_3, C_2, C_1\}$, deletes $\{C_5, C_4, C_2\}$, and returns $\{C_1\}$ to the client.

Finally, suppose that this 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 2 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. 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 in Appendix. It is easy to summarize and conclude that ROSE is robust.

Theorem 2: 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 I 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 $O((n_w + s'_w + 1)d_w)$.

VII. Implementations

We experimentally compare ROSE with three DSSE schemes, say Fides, HORUS, and IM-DSSE$_{\text{I+II}}$ (All codes are available on https://github.com/HustSecurityLab/ROSE-Experiment). Fides unintentionally achieves robustness. HORUS can be slightly revised to gain robustness. IM-DSSE$_{\text{I+II}}$ has very high run-time efficiency and is evaluated to measure the performance degradation introduced by ROSE to achieve robustness. We do not compare ROSE with Moneta, since Moneta takes a very huge cost due to its underlying cryptographic tool TWORAM. In a nutshell, ROSE usually performs much better than both Fides and HORUS in most of terms, except that as the number of deletions grows, the search time cost of ROSE with a single thread will be slightly more than that of Fides. However, this disadvantage against Fides and HORUS can be relieved by parallelizing the search of ROSE. Compared to IM-DSSE$_{\text{I+II}}$, ROSE, Fides, and HORUS require more time to issue an `update` query and to complete a `search` query. In most cases, ROSE only introduces about an order of magnitude extra time to complete the search, except for the case when there are historical deletions. In terms of client time cost during the search, ROSE outperforms IM-DSSE$_{\text{I+II}}$.

Evaluated Metrics: We evaluate and compare the four schemes in terms of (1) the time cost to issue a single `add`/`delete` query and (2) the time and bandwidth costs when searching a keyword with or without considering historical deletions. When considering historical deletions, the search time cost of ROSE is also affected by the number of historical `search` queries. Hence, we evaluate ROSE's search performance when simultaneously considering historical deletions and `search` queries. In contrast, the historical `search` queries do not affect the search performance of Fides, HORUS, and IM-DSSE$_{\text{I+II}}$. In addition, the two processes, namely ROSE tests whether a `add` ciphertext is deleted during the `search` query (Step 15 in Algorithm 3, Section VI) and updates the delete tokens (Steps 38 to 40 in Algorithm 3, Section VI), can be parallelized to gain higher efficiency during a search when considering historical deletions and `search` queries. Hence, we finally evaluate ROSE by parallelizing the above two processes.

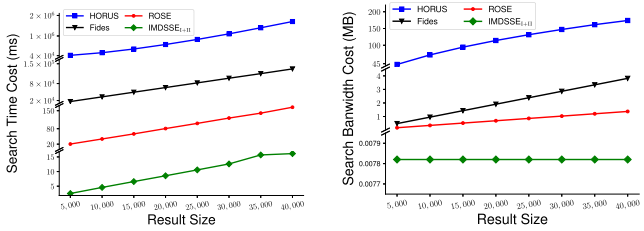
TABLE II
Update Time Cost (μs) Comparisons

	ROSE	Fides	HORUS	IM-DSSE$_{\text{I+II}}$
$op = add$	1,338.73	3,090.51	87,503	17.61
$op = del$	2,087.81	3,092.13	114,692	16.20

System Environment: We develop ROSE in C++ and apply OpenSSL [36] and Relic Toolkit [37] to implement its cryptographic primitives. OpenSSL provides hash functions SHA-2 family and SHA-3 family and symmetric encryption AES-128. Relic Toolkit provides the NIST P-256 elliptic curve to implement key-updatable PRF. We code Fides in C++ and apply OpenSSL to implement its cryptographic primitives as well. We implement the RSA-based trapdoor permutation used in Fides with GMP Library [38]. We adjust the parameters of HORUS's opencode for achieving the same security level as ROSE and Fides do. Moreover, we slightly modify the opencode to fix the bug in the position re-randomization process of OMAP finalization. During the implementations of these schemes, all data are stored in the memory. When evaluating IM-DSSE$_{\text{I+II}}$, we slightly revise the C++ code released by Hoang *et al.* to provide fair comparisons. Those revisions mainly include (1) we use the memory to store the encrypted database and the client-side states, (2) we use in-process variables instead of sockets to convey the communications between the client and the server, (3) we enable the IM-DSSE$_{\text{I+II}}$ code to support incrementally adding or deleting a single pair of keyword and file-identifier, and (4) we enable the client to convert the searched pseudorandom numbers to file identifiers after the search by maintaining client-side state table. Note that the last point enables IM-DSSE$_{\text{I+II}}$ to run in the setting where the file storage service is independent of the encrypted database. Our experiments are performed on a workstation with an Intel Xeon Silver 4216, 64 GB RAM, and Ubuntu Server 20.04. All figures in this section are drawn with Matplotlib [39].

Dataset: Since both Fides and HORUS take too much time costs when running them over a large-scale dataset, we simulate a moderate-scale dataset. This dataset is a fair one and similar as the dataset used by the work [11] to test HORUS. It contains 180,000 entries, where the result size of a search query is between 5,000 and 40,000. Each entry consists of a two-bytes keyword and a four-bytes file identifier.

Update Performance Comparisons: We compare the time costs of issuing a single `add` and a `delete` query of ROSE, Fides, HORUS, and IM-DSSE$_{\text{I+II}}$. For each scheme, we generate `add` queries to all 180,000 entries at first, then generate `delete` queries to delete all these entries, and test the average time costs to issue a `add` query and a `delete` query, respectively. The average update time cost includes the client-side and server-side costs. Table II shows that



(a) Search Time Cost. (b) Search Bandwidth Cost.

Fig. 3. Search performance comparisons without the historical deletion.

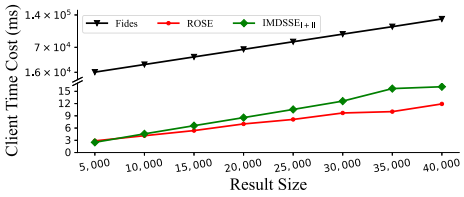
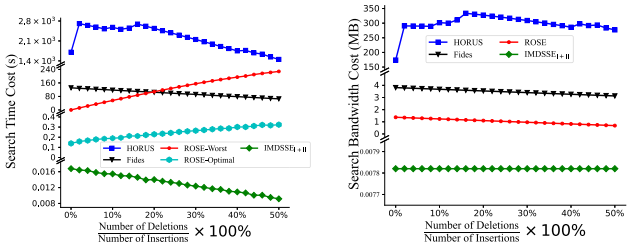


Fig. 4. Client time cost comparisons during a search without the historical deletion.



(a) Search Time Cost. (b) Search Bandwidth Cost.

Fig. 5. Search performance comparisons with historical deletions but without the historical search.

ROSE takes on average 1,338.73 and 2,087.81 microseconds to add and delete an entry, respectively, about two orders of magnitude more than those of IM-DSSE$_{I+II}$. Although, the update performance of ROSE (no more than 2.1 milliseconds) is practical in real applications. Compared with Fides, ROSE is approximately 2.31× and 1.48× faster when adding and deleting an entry, respectively. Compared with HORUS, ROSE is clearly too much faster.

Search Performance Comparisons Without the Historical Deletion: This part compares the time and bandwidth costs when searching keywords with ROSE, Fides, HORUS, and IM-DSSE$_{I+II}$. Note that the experiment here assumes that there is no `delete` query before searching keywords. The search time cost includes the client-side and server-side time costs. The bandwidth cost means the transferred data size between the client and the server. Figure 3 shows that ROSE outperforms Fides and HORUS but is less efficient than IM-DSSE$_{I+II}$. Specifically, ROSE takes about 4.04 microseconds to find one matching ciphertext on average. It is approximately 814× and 8,560× faster than Fides and HORUS, respectively. Compared to Fides and HORUS, ROSE achieves a significant advantage in bandwidth cost. Suppose the search results contain 40,000 file identifiers. The bandwidth cost of ROSE is approximately 1.37 MB, which saves approximate 64% and 99.21% bandwidth compared with Fides and HORUS, respectively.ROSE costs about more 3.61 microseconds than IM-DSSE$_{I+II}$ to find one matching ciphertext on average. In terms of bandwidth cost, IM-DSSE$_{I+II}$ is more efficient than the rest, and its bandwidth does not change along with the result size. This is because the size of its encrypted index is fixed during the setup phase.

To clearly show that ROSE is more friendly to the client, we explicitly list the client time costs of ROSE, Fides, and IM-DSSE$_{I+II}$ during a search. We omit HORUS here, since the HORUS client undertakes almost all the computational tasks during a search. Hence, the client time cost of HORUS is much more than that of ROSE, Fides, and IM-DSSE$_{I+II}$. Figure 4 shows that ROSE is at least 5,713× faster than Fides and gains a little advantage (about 1.3× faster on average) against IM-DSSE$_{I+II}$. For example, when the search result size is 40,000, the ROSE client uses approximately 11.91 milliseconds, while the Fides and the IM-DSSE$_{I+II}$ client take approximately 2.19 minutes and 16.12 milliseconds, respectively.

Search Performance Comparisons With Historical Deletions: In this part, we test how the historical `delete` and `search` queries affect the performance of a freshly issued `search` query. Hence, before testing the search performance, we suppose two cases: (1) the client has issued some `delete` queries but no `search` query; (2) the client has issued not only some `delete` queries but also some `search` queries, where all `search` queries were issued ahead of the first `delete` query. We compare the search performance of ROSE, Fides, HORUS, and IM-DSSE$_{I+II}$ in the first case. In the second case, we only evaluate the search performance of ROSE, since the historical `search` queries do not affect the search performance of Fides, HORUS, and IM-DSSE$_{I+II}$.

We select the most frequently used keyword whose search result size is 40,000 as the dataset used in the above two cases. Specifically, in the first case, for the selected keyword, we issue 40,000 `add` queries, then issue a variable number of `delete` queries, and finally test the search performance of the keyword over these historical ciphertexts. Note that all `add` and `delete` queries randomly chooses their corresponding file identifiers, and each file identifier chosen by a `delete` query must be also chosen by a previous `add` query. The second case has the same setting as the first case, except that to simulate the historical searches, we additionally add some ciphertexts with operation $op = srch$, and these ciphertexts appear after the 40,000 `add` queries but before all `delete` queries.

Figure 5 presents the experimental results in the first case. In this part, IM-DSSE$_{I+II}$ keeps its significant advantage in both terms of bandwidth cost and time cost against ROSE, Fides, and HORUS. In terms of bandwidth cost, ROSE saves at least 64% and 99.21% overhead compared with Fides and HORUS, respectively, since ROSE avoids re-encrypting and re-uploading ciphertexts to the server during a search. In terms of time cost (including the server and client's time costs), ROSE always outperforms HORUS and conditionally

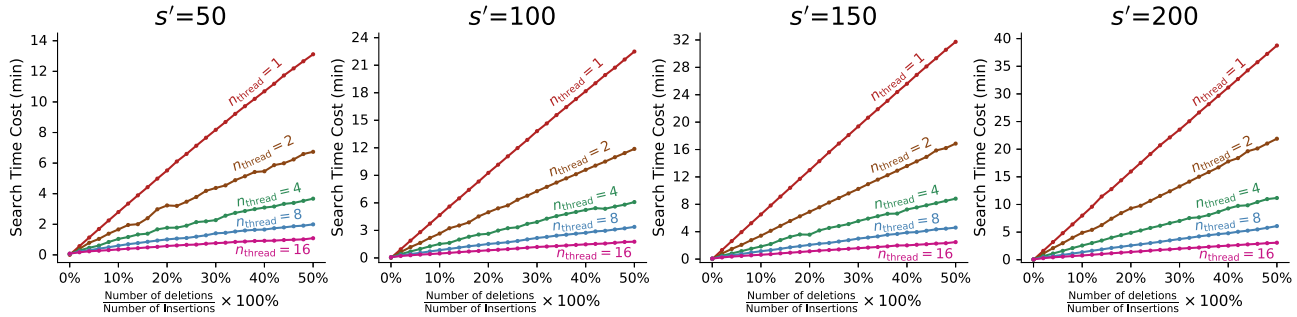


Fig. 6. Search time cost of ROSE with historical deletions and searches using different number of threads. symbol s' denotes the number of ciphertexts with $op = srch$ before the first `delete` query. Symbol n_{thread} means the number of threads.

outperforms Fides when the ratio of historical `delete` queries to historical `add` queries is less than about 20% (shown as *ROSE-Worst* in Figure 5(a)). For example, when the ratio is 10%, ROSE takes approximately 56.21 seconds to complete the search, which is about 2.09× faster than Fides. When the ratio is greater than 20%, ROSE takes more time cost than Fides. However, the first case we defined, indeed, is also the worst-case for ROSE, since all historical `delete` queries appear after all historical `add` queries, which causes ROSE must test each added ciphertexts with most of `delete` tokens. In contrast, in the optimal case that each added ciphertexts just be tested with one `delete` token at most, ROSE always outperforms Fides greatly (shown as *ROSE-Optimal* in Figure 5(a)). For example, given $\frac{\text{Number of Deletions}}{\text{Number of Insertions}} \times 100\% = 50\%$, the optimal search performance is approximately 0.32 seconds, about 704× less than the worst one (say, 3.80 minutes). Hence, in real applications, the search performance of ROSE should be between the worst one and the optimal one.

In the second case, given the different number of historical deletions and searches, Figure 6 shows the parallel search performance of ROSE. The numerical results demonstrate that ROSE has the powerful capability to parallelize a search process. For example, given 16 threads, $\frac{\text{Number of Deletions}}{\text{Number of Insertions}} \times 100\% = 50\%$, and the historical search number $s' = 50$ (respectively 100, 150, and 200), ROSE takes approximately 1.09 minutes (respectively 1.77, 2.48, and 3.08 minutes). Compared with the time cost of a single thread, the search performance improves about 12.05 times (respectively 12.73, 12.78, and 12.57 times).

According to the above extreme experiments, the historical `delete` and `search` queries will affect the search performance of ROSE. However, ROSE is still efficient in practice due to the following reasons. First, all `delete` queries just affect the search performance once. It means that such negative effect will be significant if the client issues too many `delete` queries between two successive `search` queries. However, it is hard to find such kind of scenario in the real application. Secondly, ROSE allows the server to combine the ciphertexts with operation $op = srch$ if they are adjacent. This feature will reduce the negative effect of historical `search` queries to the search performance. We omit this advantage in the above experiments, since it is hard to find a real dataset including the client' search history. Thirdly, ROSE has the powerful capability to parallelize a search process. Hence, ROSE can sufficiently use the abundant computing power of the server, like Cloud.

VIII. Conclusion and Future Works

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 correctness 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 for the first time key-updatable PRF and then achieve ROSE. Finally, we experimentally compare ROSE with two state-of-the-art DSSE schemes. It shows that ROSE is very efficient in both updating data and performing keyword searches.

Security and efficiency are two important indicators for evaluating DSSE schemes. In terms of efficiency, we believe that an intuitive future work is to achieve higher performance without decreasing security. That expands robust DSSE's application scope to capture more high timeliness requirement scenarios. Besides, supporting smaller client storage will suit practical emerging ICT scenarios (e.g., the internet of things). In terms of security, we state that extending this work to support higher security with minimum efficiency degradation may be meaningful. That may enable DSSE to explore in those applications which care more about data confidentiality than performance. There is a tradeoff between security and efficiency. An open and challenging problem is to design securer and more efficient robust DSSE with properties of the highest backward security level, the smallest client storage, and more.

Appendix
Security Proof of ROSE

To prove the forward and general Type-III backward security, we construct a simulator $\mathcal{S}$, which takes leakage functions

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

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

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

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

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

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

1: $u \leftarrow u + 1$ ▷ the timestamp of the current `search` query
2: Extract the timestamp u_0 of the last search query from $\text{sp}(w)$ where $u_0 = 0$ if $\text{sp}(w) = \emptyset$
3: Extract all timestamps $\{u_1, \cdots, u_n\}$ between u_0 and u from both $\text{exTimeDB}(w)$ and $\text{exDelHist}(w)$ 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 $\text{exTimeDB}(w)$, 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. $\text{G}(P_u, R_u) = L_u$
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. $\text{H}(T_u, R_u) = D_u \oplus (srch||\Delta_u||L_{u_0}||T_{u_0})$
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 $\text{exTimeDB}(w)$ after receiving the response from the server
21: **end if**
22: **for** each $(\mathcal{U}^{add}, \mathcal{U}^{del}) \in \text{exDelHist}(w)$ and $\mathcal{U} \in \text{exTimeDB}(w)$ **do**
23: $u_{min} \leftarrow \text{Min}(\mathcal{U}^{add} \cup \mathcal{U}^{del})$ or $\text{Min}(\mathcal{U})$
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. $\text{G}(P_{add}, R_{u_i}) = L_{u_i}$
46: Program H s.t. $\text{H}(T_{u_i}, R_{u_i}) = D_{u_i} \oplus (add||0^{\lambda}||L_{u_{i-1}}||T_{u_{i-1}})$
47: **else**
48: Program G s.t. $\text{G}(P_{del}, R_{u_i}) = L_{u_i}$
49: Program H s.t. $\text{H}(T_{u_i}, R_{u_i}) = D_{u_i} \oplus (del||P_{add}||L_{u_{i-1}}||T_{u_{i-1}})$
50: **end if**
51: **end for**
52: Program H s.t. $\text{H}(T_u, R_u) = D_u \oplus (srch||\Delta_u||L_{u_n}||T_{u_n})$
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 $\text{exTimeDB}(w)$ after receiving the response from the server

$\mathcal{L}^{Stp}(\lambda) = \lambda$, $\mathcal{L}^{Updt}(op, w, id) = \emptyset$, 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 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, 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 $\mathcal{S}.\text{Setup}(\mathcal{L}^{Stp}(\lambda))$ 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, $\mathcal{S}.\text{Update}(\mathcal{L}^{Updt}(op, w, id))$ is indistinguishable 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 $\mathcal{Y}_{\mathrm{P}}$, a decryption token T_u from $\mathcal{Y}_{\mathrm{F}}$ and a key-update token Δ_u from $\mathcal{K}_{\mathrm{P}}$, 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 $\{u_1, \ldots, u_n\}$, 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 $\mathcal{L}^{Srch}(w) = \{\mathrm{sp}(w), \mathrm{exTimeDB}(w), \mathrm{exDelHist}(w)\}$ 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.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their valuable suggestions that helped to improve the article greatly.

REFERENCES

[1] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, May 2000, pp. 44–55.
[2] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proc. 13th ACM Conf. Comput. Commun. Secur. (CCS)*, Alexandria, VA, USA, Oct. 2006, pp. 79–88.
[3] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, Raleigh, NC, USA, Oct. 2012, pp. 965–976.
[4] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2014, pp. 1–15.
[5] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Denver, CO, USA, Oct. 2015, pp. 668–679.
[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 *Proc. 25th USENIX Secur. Symp.*, Austin, TX, USA, Aug. 2016, pp. 707–720.
[7] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Vienna, Austria, Oct. 2016, pp. 1143–1154.
[8] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Dallas, TX, USA, Oct. 2017, pp. 1465–1482.
[9] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Dallas, TX, USA, Oct. 2017, pp. 1449–1463.
[10] S.-F. Sun *et al.*, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Toronto, ON, Canada, Oct. 2018, pp. 763–780.
[11] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Toronto, ON, Canada, Oct. 2018, pp. 1038–1055.

[12] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *Proc. 24th Eur. Symp. Res. Comput. Secur.*, Luxembourg, U.K., Sep. 2019, pp. 283–303.

[13] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2020, pp. 1–18.

[14] S.-F. Sun *et al.*, "Practical non-interactive searchable encryption with forward and backward privacy," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2021, pp. 1–18.

[15] T. Hoang, A. A. Yavuz, and J. Guajardo, "A secure searchable encryption framework for privacy-critical cloud storage services," *IEEE Trans. Services Comput.*, vol. 14, no. 6, pp. 1675–1689, Nov. 2021, doi: 10.1109/TSC.2019.2897096.

[16] Verizon. (2021). *2021 Data Breach Investigations Report*. Accessed: Dec. 29, 2021. [Online]. Available: https://www.verizon.com/business/resources/reports/dbir/

[17] S. Renwick and K. Martin, "Practical architectures for deployment of searchable encryption in a cloud environment," *Cryptography*, vol. 1, no. 3, p. 19, Nov. 2017.

[18] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption," in *Proc. Annu. Int. Cryptol. Conf.*, Santa Barbara, CA, USA, Aug. 2016, pp. 563–592.

[19] M. Naor, B. Pinkas, and O. Reingold, "Distributed pseudo-random functions and KDCs," in *Proc. Int. Conf. Theory Appl. Cryptograph. Techn.*, Prague, Czech Republic, May 1999, pp. 327–346.

[20] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan, "Key homomorphic PRFs and their applications," in *Proc. 33rd Annu. Cryptol. Conf.*, Santa Barbara, CA, USA, Aug. 2013, pp. 410–428.

[21] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Proc. 3rd Int. Conf. Appl. Cryptogr. Netw. Secur.*, New York, NY, USA, Jun. 2005, pp. 442–455.

[22] D. Cash *et al.*, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2014, pp. 1–16.

[23] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Proc. 17th Int. Conf. Financial Cryptogr. Data Secur.*, Okinawa, Japan, Apr. 2013, pp. 258–274.

[24] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Scottsdale, AZ, USA, Nov. 2014, pp. 310–320.

[25] A. A. Yavuz and J. Guajardo, "Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware," in *Proc. 22nd Int. Conf. Sel. Areas Cryptogr.*, Sackville, NB, Canada, Aug. 2015, pp. 241–259.

[26] 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 Security*, vol. 16, pp. 1538–1549, 2021.

[27] J. Li *et al.*, "Searchable symmetric encryption with forward search privacy," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 1, pp. 460–474, Jan. 2021.

[28] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical private range search revisited," in *Proc. Int. Conf. Manage. Data*, San Francisco, CA, USA, Jun. 2016, pp. 185–198.

[29] I. Demertzis and C. Papamanthou, "Fast searchable encryption with tunable locality," in *Proc. ACM Int. Conf. Manage. Data*, Chicago, IL, USA, May 2017, pp. 1053–1067.

[30] I. Demertzis, R. Talapatra, and C. Papamanthou, "Efficient searchable encryption through compression," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1729–1741, Jul. 2018.

[31] T. Hoang, A. A. Yavuz, and J. Guajardo, "Practical and secure dynamic searchable encryption via oblivious access on distributed data structure," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Los Angeles, CA, USA, Dec. 2016, pp. 302–313.

[32] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward private searchable symmetric encryption with optimized I/O efficiency," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 5, pp. 912–927, Sep. 2020, doi: 10.1109/TDSC.2018.2822294.

[33] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, "Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset," *Proc. Privacy Enhancing Technol.*, vol. 2019, no. 1, pp. 172–191, Jan. 2019, doi: 10.2478/popets-2019-0010.

[34] V. Vo, S. Lai, X. Yuan, S. Nepal, and J. K. Liu, "Towards efficient and strong backward private searchable encryption with secure enclaves," in *Applied Cryptography and Network Security* (Lecture Notes in Computer Science), vol. 12726, K. Sako and N. O. Tippenhauer, Eds. Kamakura, Japan: Springer, Jun. 2021, pp. 50–75, doi: 10.1007/978-3-030-78372-3_3.

[35] G. Amjad, S. Kamara, and T. Moataz, "Forward and backward private searchable encryption with SGX," in *Proc. 12th Eur. Workshop Syst. Secur. (EuroSec)*, Dresden, Germany, Mar. 2019, pp. 4:1–4:6, doi: 10.1145/3301417.3312496.

[36] Open Society Foundations. *OpenSSL*. Accessed: Jun. 22, 2021. [Online]. Available: https://www.openssl.org/

[37] D. F. Aranha and C. P. L. Gouvêa. (2014). *RELIC is an Efficient Library for Cryptography*. Accessed: Jun. 22, 2021. [Online]. Available: https://github.com/relic-toolkit/relic

[38] Free Software Foundation. *The GNU Multiple Precision Arithmetic Library*. Accessed: Jun. 22, 2021. [Online]. Available: https://gmplib.org/

[39] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, May 2007.

Peng Xu (Member, IEEE) received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2010. He held a post-doctoral position with the Huazhong University of Science and Technology, from 2010 to 2013, and as an Associate Research Fellow at the University of Wollongong, Australia, from 2018 to 2019. He is currently a Full Professor with the Huazhong University of Science and Technology. He has authored 40 research articles and three books, and holds 20 patents. His research interest is in the field of cryptography. He was a PI in ten grants, including three NSF projects.

Willy Susilo (Fellow, IEEE) received the Ph.D. degree in computer science from the University of Wollongong, Australia. He is currently a Professor and the Head of the School of Computing and Information Technology and the Director of the Institute of Cybersecurity and Cryptology (iC2), University of Wollongong. His main research interest includes applied cryptography. He was previously awarded the prestigious ARC Future Fellow from the Australian Research Council (ARC).

Wei Wang (Member, IEEE) received the B.E. and Ph.D. degrees in electronic and communication engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2006 and 2011, respectively. She currently works as a Researcher with the Cyber-Physical-Social Systems Laboatory, Huazhong University of Science and Technology. She has authored 30 papers in international journals and conferences. Her research interests include cloud security, network coding, and multimedia transmission.

Tianyang Chen received the B.E. degree in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2017, where he is currently pursuing the Ph.D. degree in cyberspace security. His research interests include cryptography and the IoT.

Qianhong Wu (Member, IEEE) received the Ph.D. degree in cryptography from Xidian University in 2004. Since then, he has been with Wollongong University, Wollongong, NSW, Australia, as an Associate Research Fellow; with Wuhan University, China, as an Associate Professor; and with Universitat Rovira i Virgili, Spain, as the Research Director. He is currently a Professor with Beihang University, China. His research interests include cryptography, information security and privacy, VANET security, and cloud computing security. He has been a holder/co-holder of China/Australia/Spain funded projects. He has authored more than 150 publications and holds 100 patents. He has served as an associate/guest editor in several international ISI journals and in the program committee of dozens of international conferences.

Kaitai Liang (Member, IEEE) received the Ph.D. degree in computer science from the Department of Computer Science, City University of Hong Kong, Hong Kong, in 2014. He is currently an Assistant Professor with the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, The Netherlands. His current research interests include applied cryptography and information security, in particular, encryption, blockchain, postquantum crypto, privacy enhancing technology, and security in cloud computing.

Hai Jin (Fellow, IEEE) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology in 1994. He received the German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Germany, in 1996. He worked at The University of Hong Kong from 1998 to 2000 and as a Visiting Scholar at the University of Southern California from 1999 to 2000. He received the Excellent Youth Award from the National Science Foundation of China in 2001. He is a Cheung Kung Scholars Chair Professor of computer science and engineering of the Huazhong University of Science and Technology. He has coauthored 22 books and published over 800 research articles. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of the CCF and a member of the ACM.