We Can Make Mistakes: Fault-tolerant Forward Private Verifiable Dynamic Searchable Symmetric Encryption

1st Dandan Yuan
The University of Auckland
dyua568@aucklanduni.ac.nz

2nd Shujie Cui
Monash University
Shujie.Cui@monash.edu

3rd Giovanni Russello
The University of Auckland
g.russello@auckland.ac.nz

***Abstract*—Verifiable Dynamic Searchable Symmetric Encryption (VDSSE) enables users to securely outsource databases (document sets) to cloud servers and perform searches and updates. The verifiability property prevents users from accepting incorrect search results returned by a malicious server. However, we discover that the community currently only focuses on preventing malicious behavior from the server but ignores *incorrect updates* from the client, which are very likely to happen since there is no record on the client to check. Indeed most existing VDSSE schemes are not sufficient to tolerate incorrect updates from the client. For instance, deleting a nonexistent keyword-identifier pair can break their correctness and soundness.**

In this paper, we demonstrate the vulnerabilities of a type of existing VDSSE schemes that fail them to ensure correctness and soundness properties on incorrect updates. We propose an efficient fault-tolerant solution that can consider any DSSE scheme as a black-box and make them into a fault-tolerant VDSSE in the malicious model. *Forward privacy* is an important property of DSSE that prevents the server from linking an update operation to previous search queries. Our approach can also make any forward secure DSSE scheme into a fault-tolerant VDSSE without breaking the forward security guarantee.

In this work, we take FAST [1] (TDSC 2020), a forward secure DSSE, as an example, implement a prototype of our solution, and evaluate its performance. Even when compared with the previous fastest forward private construction that does not support fault tolerance, the experiments show that our construction saves $9\times$ client storage and has better search and update efficiency.

***Index Terms*—Security and Privacy Protection, Verification, Database Management, Information Search and Retrieval**

1. Introduction

Searchable symmetric encryption (SSE) is a technique that enables a data owner to securely outsource its database to a remote server and search the database efficiently and confidentially. In a common model of SSE, the database is composed of a set of documents, each of which owns a unique identifier and contains a set of keywords. An important research line of SSE focuses on the honest-but-curious adversarial model, where the adversary is assumed to follow the specified protocol honestly, which is not necessarily always true. Once the cloud server is compromised, the attacker could control the protocol arbitrarily, e.g., return incorrect results to the client. Verifiable SSE (VSSE) [2]–[10] is a line of research that aims to ensure the client always receives all the correct documents that match the searched keyword and rejects incorrect ones. These properties are referred to as *correctness* and *soundness* in [6], respectively. These properties are in addition to data confidentiality and are relevant when considering the server as (under the control of) a malicious actor that could mount active attacks.

In VSSE schemes, an additional data structure, called *proof*, is required to verify the correctness of search results. For dynamic SSE (DSSE), the client can insert/delete a keyword into/from a document's keywords set. For Verifiable DSSE (VDSSE), updating the database requires updating the proofs as well. For instance, when adding a new keyword into a document, the proof of the keyword should be changed to cover the identifier of the document. Updating the proofs can be easily achieved. However, we observe that the community currently only focuses on preventing malicious behaviours from the server with the proof but ignores whether the proof still works when the misbehaviour comes from a careless client.

In particular, we discover that incremental-hash-based VDSSE schemes, such as [6]–[9], [11], [12], fail to tolerate *incorrect updates* from the client, such as adding the same keyword-identifier pair multiple times or deleting nonexistent data. For instance, when the client adds the same keyword-identifier pair multiple times, even when the server returns the correct search result, their proofs can not pass the verification and the client will reject the result.

We define the ability of tolerating incorrect updates as *fault tolerance property* for VDSSE scheme. Ensuring this property is necessary as in the setting of outsourced databases there is no local record on the client to check if the updates are correct. Moreover, the collision could happen among the updates issued from different devices. Since all the queries are encrypted, the collision cannot be detected or prevented on the server side with synchronization or locking mechanisms. A trivial solution is to search the keyword (or the document) to be updated and check if the keyword-identifier pair exists in the returned result before executing the update query, which needs 2 rounds of communication and increases the overhead for update queries significantly. Moreover, this solution could leak which keyword (or document) is to be updated. We need a more secure and efficient way to address the issue.

Informally, we say a DSSE scheme is *forward private*

TABLE 1. COMPARISON WITH EXISTING VDSSE SCHEMES

Scheme	Computation		Communication		Client Storage	Forward Private	Fault Tolerant
	Search	Update	Search	Update			
Kurosawa and Ohtaki [4]	$O(\|\mathrm{W}\|\|\mathrm{D}\|)$	$O(\|\mathrm{W}\|)$	$O(m)$	$O(1)$	$O(1)$	✗	-
Kurosawa *et al.* [5]	$O(m)$	$O(\|\mathrm{W}\|)$	$O(m)$	$O(\|\mathrm{W}\|)$	$O(1)$	✗	-
Kamara and Papamanthou [21]	$O(m \log \|\mathrm{D}\|)$	$O(\log(\|\mathrm{D}\|))$	$O(m\cdot \log \|\mathrm{D}\|)$	$O(\log(\|\mathrm{D}\|))$	$O(1)$	✗	-
Bost *et al.*' generic solution [6]	$O(u+ \log \|\mathrm{W}\|)$	$O(\log \|\mathrm{W}\|)$	$O(m+ \log \|\mathrm{W}\|)$	$O(\log \|\mathrm{W}\|)$	$O(1)$	✗	✗
Bost *et al.*' verifiable SPS [6] SPS [18]	$O(\min\{^{u+\log^2 N}_{m\log^3 N}\})$	$O(\log^2 N)$	$O(m+ \log N)$	$O(\log N)$	$O(\lambda \log N)$	✓	✓
Bost [11]	$O(u)$	$O(1)$	$O(m)$	$O(1)$	$O(\|\mathrm{W}\|(\log \|\mathrm{D}\| +\lambda))$	✓	✗
Zhu *et al.* [12]	$O(m + \log \|\mathrm{W}\|)$	$O(\log \|\mathrm{W}\|$	$O(m+ \log \|\mathrm{W}\|)$	$O(\log \|\mathrm{W}\|)$	$O(1)$	✗	✗
Etemad and Küpçü [22]	$O(m \log \|\mathrm{W}\|)$	$O(\log \|\mathrm{W}\| + \log \|\|\mathrm{D}\|\|)$	$O(m\cdot \log \|\mathrm{W}\|)$	$O(\log \|\mathrm{W}\| + \log \|\|\mathrm{D}\|\|)$	$O(1)$	✗	✓
Ge *et al.* [8]	$O(\|\mathrm{D}\|)$	$O(\|\mathrm{W}\|)$	$O(m)$	$O(\|\mathrm{W}\|)$	$O(1)$	✗	✗
Zhang *et al.* [7]	$O(u)$	$O(1)$	$O(m)$	$O(1)$	$O(\|\mathrm{W}\|(\log \|\mathrm{D}\| +\lambda))$	✓	✗
Ours	$O(u)$	$O(1)$	$O(u)$	$O(1)$	$O(\|\mathrm{W}\|(\log \|\mathrm{D}\|))$	✓	✓

$|\mathrm{W}|$ and $|\mathrm{D}|$ are respectively the numbers of keywords and documents in the database. '$-$' represents that the property of fault tolerance does not need to be considered in the scheme of the row. m is the size of the search result, N denotes the total number of keyword-document pairs that exist in the database, u is the number of updates related to the searched keyword, and λ is the security parameter. 'Computation' refers to the consumed computational complexity, 'Communication' is the consumed communication complexity, and 'Client Storage' specially refers to the permanent client storage.

if the server cannot link the updates to previous queries. For DSSE, ensuring forward privacy is fundamental to hamper the file injection attack [13]. A large number of DSSE schemes with forward privacy have been proposed [1], [11], [14]–[17]. However, only a few forward private DSSE schemes are verifiable in the malicious model. In [6], Bost *et al.*make the forward private DSSE scheme in [18] into a fault-tolerant VDSSE by employing verifiable hash table [19] to generate proofs. However, their scheme is not efficient for large databases. To make the schemes in [11] and [7] verifiable, Bost and Zhang *et al.* adopt incremental hash [20] to map the identifiers matched by each keyword to a hash value. However, to achieve sub-linear search and constant update efficiency, the two schemes store the hash values on the client side, resulting in heavy storage overhead on the client. Moreover, they are not fault-tolerant.

Our Contributions. In this work, we propose a solution to make DSSE and forward secure DSSE schemes fault-tolerant and verifiable in the malicious model by only using two simple cryptographic primitives: pseudorandom function (PRF) and authenticated encryption (AE). A comparison with previous work is shown in Table 1. Among existing VDSSE schemes, only when applying Bost *et al.*' solution [6] to Stefanov *et al.*' scheme [18], it achieves both forward privacy and fault tolerance property, however, with heavy computation and communication overhead for both search and update queries. In contrast, our scheme not only is fault-tolerant and forward private, but also achieves acceptable search and update efficiency.

Our contributions can be summarized as below:

- We are the first to demonstrate that incremental-hash-based VDSSE schemes, including [6]–[9], [11], [12], can no longer ensure correctness and soundness when there are incorrect updates from the client.
- We propose a generic solution that can consider any DSSE scheme as a black-box and make them verifiable and fault-tolerant in the malicious model. Moreover, our solution can also make any forward secure DSSE scheme verifiable and fault-tolerant without breaching its forward privacy guarantee.
- We implemented a prototype of our solution on the top of a forward secure DSSE scheme called FAST [1] and evaluated its performance. Our experimental results show that our solution has practical performance.
- Last but not least, we prove the soundness and confidentiality of our solution.

2. Related Work

SSE and DSSE. The first feasible SSE was proposed by Song *et al.* [23] in 2000. Since then, a great deal of literature has contributed to this direction in terms of security, performance, and functionalities. In the early stages, almost all SSE work, such as [24]–[26], only considers static databases. In 2012, Kamara *et al.* [27] present the first DSSE scheme with sub-linear search efficiency. Two subsequent works in [21], [28] present more secure and efficient DSSE schemes.

Forward Private DSSE. Forward privacy of DSSE has been started to be discussed in the community since 2014, and Stefanov et al. [18] present the first forward secure DSSE scheme using ORAM-like technique. Since the file-injection attack [13] has been proposed, forward privacy is becoming an imperative property for DSSE; otherwise previously queried keywords can be recovered easily when the attacker injects well-designed records into the database. Forward privacy was first formally defined by Bost in [11], and he constructs a practical forward private dynamic DSSE scheme using public-key-based trapdoor permutation. After that, several pure symmetric-key-based DSSE schemes proposed in [1], [14]–[17] also ensure forward privacy and achieve better performance.

VSSE and VDSSE. All the schemes discussed above work in the honest-but-curious adversarial model. VSSE schemes are designed to protect outsourced data from malicious adversaries. Early VSSE schemes [2], [3], [29], [30] also just focus on static databases. The works in [4], [5], [21] extend VSSE to support dynamic databases, but the three schemes do not support updating the index

entries (i.e., keywords set) of a document. Instead, the client has to first delete the document and then add the index entries of the document from scratch.

Bost *et al.* [6] and Zhu *et al.* [12] solve the problem by letting the server store incremental hash values of the identifiers matched by every keyword with a verifiable structure. Ge *et al.* [8] also introduce an incremental-hash-like structure to generate the proof of document identifiers and document content matched by every keyword. Etemad and Küpçü [22] achieve a VDSSE scheme by using multi-level authenticated skip list [31].

Amongst, the work in [6], [8], [12] all utilizes the incremental-hash-based method to generate a proof for a keyword, yet they fail to tolerate incorrect updates. Only the scheme presented in [22] allows the client to check incorrect updates and achieves fault tolerance property.

Forward Private VDSSE. To hamper the file-injection attack in the malicious model, the VDSSE needs to achieve forward security as well. Stefanov *et al.* [18] briefly demonstrate how to extend their forward secure DSSE construction to be secure in the malicious model using MAC [32], but as analyzed by Bost *et al.* [6], their method is not effective enough to guarantee security against a malicious adversary. Bost *et al.* [6] propose to use verifiable hash table and AE, whereas their solution withstands high computational and communication complexity. The forward secure VDSSE schemes in [11] and [7] achieve better performance by letting the client hold the incremental hash values of identifiers matched by every keyword as proofs. In [9], Guo *et al.* leverage blockchain [33] and smart contract [34] to preserve forward-secure updates and verify the updated results, respectively. The construction in [6] is fault-tolerant because every step of the server can be verified by the client. However, the incremental-hash-based methods [7], [9], [11] are not fault-tolerant.

3. Preliminaries

We use $\{0,1\}^l$ to denote the set of all binary strings of length l. 0^l denotes a string of length l where every bit is 0. $\{0,1\}^*$ stands for the set of arbitrary length strings. $||$ denotes the concatenation of two strings. $A \leftarrow B$ represents that the value of B is assigned to A. Finally, $A \xleftarrow{\$} \mathrm{X}$ means that A is sampled uniformly at random from the set X.

3.1. Authenticated Encryption

An AE scheme [35], [36] consists of a Probabilistic Polynomial-Time (PPT) algorithm (AE.Gen) and two deterministic polynomial-time algorithms (AE.Enc, AE.Dec). Among them, AE.Gen takes as input 1^λ and outputs a private key k. AE.Enc takes as input a key k, a nonce non, a message mes, and outputs a ciphertext C. Part of the ciphertext C is a tag. AE.Dec takes a key k, a nonce non, a ciphertext C as input and outputs a message mes or a string "$invalid$".

The correctness of AE requires that AE.$Dec(k, non, C) = mes$ if $C \leftarrow \mathrm{AE}.Enc(k, non, mes)$, for every $k \leftarrow \mathrm{AE}.Gen(1^\lambda)$, every $non \leftarrow \{0,1\}^*$, and every $mes \leftarrow \{0,1\}^*$.

We say that an AE scheme guarantees privacy if for all PPT adversaries $\mathcal{A}$, there exists a negligible function $negl$ such that

$$\Pr[k \xleftarrow{\$} \mathrm{AE}.Gen(1^\lambda) : \mathcal{A}^{\mathrm{AE}.Enc(k,..)} = 1] - \Pr[\mathcal{A}^{\$(.,.)} = 1] \leq negl(\lambda)$$

where $\mathcal{A}^{AE.Enc(k,..)}$ represents that $\mathcal{A}$ has oracle access to AE.Enc $(k, ., .)$, $\mathcal{A}^{\$(.,.,.)}$ means that $\mathcal{A}$ can query $\$(., .)$ with a nonce non and a plaintext message mes as input that returns a uniformly random string of length $|\mathrm{AE}.Enc(k, non, mes)|$, and $\mathcal{A}$ is not allowed to ask two queries with the same nonce.

An AE scheme satisfies authenticity if for all PPT adversaries $\mathcal{A}$, there exists a negligible function $negl$ such that

$$\Pr[k \xleftarrow{\$} \mathrm{AE}.Gen(1^\lambda) : \mathcal{A}^{\mathrm{AE}.Enc(k,..)} \text{ successfully forges}] \leq negl(\lambda)$$

where we say that $\mathcal{A}^{\mathrm{AE}.Enc(k,..)}$ successfully forges if it is able to forge a pair (non, C) such that AE.$Dec(k, non, C)$ $\neq$ "$invalid$" but the query with non as the input and C as the output has never happened before, and the adversary $\mathcal{A}$ is not allowed to ask two queries with the same nonce.

3.2. Verifiable Dynamic Searchable Symmetric Encryption

We assume that the database is $\mathrm{DB} = \{(id_i, \mathrm{W}_i)\}_{i=1}^{|\mathrm{D}|}$ where $id_i \in \{0,1\}^\lambda$ is a document identifier and $\mathrm{W}_i \subseteq \{0,1\}^*$ is a set of keywords contained in the document id_i. We use $\mathrm{W} = \cup_{i=1}^{|\mathrm{D}|} \mathrm{W}_i$ to denote the collection of all keywords and $\mathrm{DB}(w) = \cup_{i=1}^{|\mathrm{D}|} \{id_i | w \in \mathrm{W}_i\}$ to stand for the set of document identifiers matched by the keyword w. Formally, a VDSSE scheme (**Setup**, **Search**, **Update**) is composed of three protocols executed by the client and the server.

- $(K, s; \mathrm{EDB}) \leftarrow$ **Setup**$(\lambda, \mathrm{DB}; \perp)$: On input the security parameter λ and a database DB, the client outputs a secret key K and a secret state s. The server outputs an encrypted database EDB.
- $(s', \mathrm{DB}(w)$ or "Reject"; $\mathrm{EDB}') \leftarrow$ **Search**$(K, s, w; \mathrm{EDB})$: The input of the client includes the secret key K, the current state s, and a keyword w. The server has EDB as input. Finally, the client outputs a possibly updated secret state s'. If the search result returned by the server is verified to be correct, the client also outputs $\mathrm{DB}(w)$, otherwise it outputs the string "Reject". The server outputs a possibly updated encrypted database EDB'.
- $(s'; \mathrm{EDB}') \leftarrow$ **Update**$(K, s, op, id, w; \mathrm{EDB})$: The client has five parameters as input that includes the secret key K, the secret state s, an operator $op \in \{add, del\}$, a document identifier id,, and a keyword w. The server's input is the encrypted database EDB. Finally, this protocol outputs a secret state s' (to be stored by the client) and an updated encrypted database EDB' (to be stored by the server).

A VDSSE scheme should at least achieve three basic properties: correctness, soundness, and confidentiality.

Before defining each property, we want to define our adversarial model. As in [6], we denote with $\mathcal{P} \leftrightarrow \mathcal{A}$ that the execution of the protocol $\mathcal{P}$ involves interactions with the malicious adversary $\mathcal{A}$ that may deviate from the protocol arbitrarily to break the soundness and confidentiality of $\mathcal{P}$.

3.2.1. Correctness. We say a VDSSE scheme is correct if, when the server executes all the protocols honestly, the search result for each keyword w is $\mathrm{DB}(w)$ and passes the verification on the client. The formal definition can be referred to [6].

3.2.2. Soundness. Soundness guarantees that if the server is malicious, the client will not be tricked by the server into accepting incorrect search results. In other words, the client is able to detect the malicious behavior of the server returning invalid search results.

Definition 3.1 (Soundness of VDSSE). Let Σ = {**Setup**, **Search**, **Update**} denote a VDSSE scheme. We say Σ satisfies $soundness$ if for all PPT adversaries $\mathcal{A}$, there exist a negligible function $negl$ such that:

$$\Pr[\mathrm{VDSSESound}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] \leqslant negl(\lambda)$$

where the game $\mathrm{VDSSESound}^{\Sigma}_{\mathcal{A}}(\lambda)$ is defined as:

$\mathrm{VDSSESound}^{\Sigma}_{\mathcal{A}}(\lambda)$: $\mathcal{A}$ chooses a database DB and is given EDB $\leftarrow$ **Setup**(1^{λ}, DB; $\perp$) $\leftrightarrow$ $\mathcal{A}$. Then it adaptively makes search or update queries. For a search query, $\mathcal{A}$ chooses a keyword w and receives the output of **Search**(K, s, w; EDB)$\leftrightarrow$ $\mathcal{A}$. For an update query, it chooses (op, id, w) and is given the output of **Update**(K, s, op, id, w; EDB)$\leftrightarrow$ $\mathcal{A}$. Note that when running any of the above queries, besides choosing the input and receiving the output, $\mathcal{A}$ could control the operations on the server side arbitrarily, such as replying with forged information. The game outputs 1 if the result of a search query on a keyword $w \in \mathrm{W}$ is neither the set $\mathrm{DB}(w)$ nor the string "Reject."

3.2.3. Confidentiality. Intuitively, confidentiality demands that the server cannot learn any useful information from the outsourced data and queries. According to most of the existing SSE schemes, such as [26] and [27], it can be formally defined through the leakage functions $\mathcal{L} = (\mathcal{L}^{Setup}(\mathrm{DB}), \mathcal{L}^{Search}(\mathrm{DB}, w), \mathcal{L}^{Update}(\mathrm{DB}, op, id, w))$. For the simulation-based definition, we can say a VDSSE scheme is $\mathcal{L}$-secure if anything that can be computed by the scheme, can also be computed by a simulator taking only the corresponding $\mathcal{L}$ as the input.

Definition 3.2 (Adaptive security of VDSSE). Let Σ = {**Setup**, **Search**, **Update**} denote a VDSSE scheme. We say Σ is *$\mathcal{L}$-adaptively-secure* if for any PPT adversary $\mathcal{A}$, there exist a simulator $\mathcal{S}$ and a negligible function $negl$ such that:

$$|\Pr[\mathrm{VDSSEReal}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathrm{VDSSEIdeal}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1]| \leqslant negl(\lambda)$$

where $\mathrm{VDSSEReal}^{\Sigma}_{\mathcal{A}}(\lambda)$ and $\mathrm{VDSSEIdeal}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$ are defined as:

- $\mathrm{VDSSEReal}^{\Sigma}_{\mathcal{A}}(\lambda)$: $\mathcal{A}$ first chooses a database DB, and obtains EDB by calling **Setup**(1^{λ}, DB; $\perp$) $\leftrightarrow$ $\mathcal{A}$. Then it repeatedly performs searches **Search**(K, s, w; EDB)$\leftrightarrow$ $\mathcal{A}$ and updates **Update**(K, s, op, id, w; EDB)$\leftrightarrow$ $\mathcal{A}$ in an adaptive way. In the end, $\mathcal{A}$ outputs a bit b.
- $\mathrm{VDSSEIdeal}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$: $\mathcal{A}$ chooses a database DB, and calls $\mathcal{S}(\mathcal{L}^{Setup}(\mathrm{DB})) \leftrightarrow \mathcal{A}$ to obtain the encrypted database EDB. After that, it adaptively performs search/update queries by calling $\mathcal{S}(\mathcal{L}^{Search}(\mathrm{DB}, w)) \leftrightarrow \mathcal{A}$/ $\mathcal{S}(\mathcal{L}^{Update}(\mathrm{DB}, op, id, w)) \leftrightarrow \mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b.

The typical leakage of VDSSE includes *search pattern*, *access pattern*, and *database size*. Specifically, search pattern refers to the repetition of search queries (which search queries were on the same keywords). Access pattern represents the matching identifiers received in a search query and the identifiers added/deleted in an update query. The number of keyword-identifier pairs existing in the database is denoted as database size. This typical leakage implies forward privacy. In brief, forward privacy ensures that any information about the updated keyword will not be leaked to the adversary during update operations. The first formal definition of forward privacy is presented in [11].

Definition 3.3 (Forward privacy of VDSSE). A $\mathcal{L} - adptively - secure$ VDSSE scheme Σ = {**Setup**, **Search**, **Update**} is *forward private* iff the update leakage function $\mathcal{L}^{Updt}$ can be written as:

$$\mathcal{L}^{Update}(\mathrm{DB}, op, id, w) = \mathcal{L}'(op, id)$$

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

Definition 3.4 (Incorrect updates). We say an update query (op, id, w) to be executed is *incorrect* in the following 2 cases:

1) if $op = del$, $id \notin \mathrm{DB}(w)$.
2) if $op = add$, $id \in \mathrm{DB}(w)$.

4. Incremental-Hash-Based VDSSE Vulnerabilities

In this section, we first introduce how different types of incremental multi-set hash work and then illustrate their vulnerabilities when used in VDSSE schemes. To the best of our knowledge, we are the first to report these vulnerabilities.

4.1. Incremental Multi-set hash

Incremental multi-set hash is introduced by Clarke *et al.* in [20]. A multi-set is a finite unordered group of elements where an element can occur as a member more than once. Multi-set hash $\mathcal{H}$ takes a multi-set as the input and outputs a fixed-length string. When an element, say x, is added (deleted) to (from) the input multi-set X, we say the multi-set hash is *incremental* if we can efficiently compute $\mathcal{H}(X \cup \{x\})$ ($\mathcal{H}(X \setminus \{x\})$) from $\mathcal{H}(\mathrm{X})$ and $\mathcal{H}(\{x\})$ with an arithmetic within a field, rather than computing the hash from scratch. $\mathcal{H}$ could be *MSet-Add-Hash*, *MSet-XOR-Hash*, and *MSet-Mu-Hash* when the arithmetic is addition (subtract), XOR, and multiplication (inverse), respectively. For instance, when $\mathcal{H}$ is MSet-XOR-Hash, $\mathcal{H}(X) = H_K(r) \oplus \oplus_{i=1}^{|X|} H_K(x_i)$, where H_K is a keyed

hash function, r is a random number, and x_i is the i-th element of X. $H_K(r)$ is also required for MSet-Add-Hash. r should be recorded for both cases. For MSet-Mu-Hash, $\mathcal{H}(X) = \prod_{i=1}^{|X|} H(x_i)$, where H is a hash function: $H : X \rightarrow \mathbb{F}_q$.

In incremental-hash-based VDSSE schemes, for each keyword w, $\mathcal{H}(I)$ is used as the proof, where I is the set of identifiers matched by w. During a search on w, with the received result R (and r when MSet-Add-Hash or MSet-XOR-Hash is used), the client computes $\mathcal{H}(R)$ and checks if it equals to $\mathcal{H}(I)$. When an update comes, e.g., (add, w, id), either the client or the server can quickly update the proof by computing $\mathcal{H}(I \cup \{id\})$.

The advantage of using incremental multi-set hash is that the size of the proof does not increase with updates and it can be updated efficiently with simple arithmetic. To update the index entries of a document, almost all the existing VDSSE schemes (excluding the one proposed in [22] and the forward private solution in [6]) employ collision-resistant incremental hash [20]. Precisely, the scheme in [6], [12] uses MSet-Mu-Hash, MSet-XOR-Hash are employed in [7] and [9], and the solution in [8] adopts a technique similar to MSet-Add-Hash. All the three incremental hash algorithms work for Bost's solution in [11]. Unfortunately, they are not sufficient to tolerate incorrect updates from the client.

4.2. Correctness Vulnerabilities

Here we first highlight one type of incorrect updates that can break the correctness guarantee of incremental-hash-based VDSSE schemes: deleting a nonexistent keyword-identifier pair. In this case, the incremental multi-set hash will generate wrong proofs. Specifically, assume the client wrongly sends a delete query (del, id_1) over w, where (w, id_1) does not exist in the database, the proof will be modified into $\mathcal{H}(I) - H_K(id_1)$ mod Z (where Z is a big integer), $\mathcal{H}(I) \oplus H_K(id_1)$, and $\mathcal{H}(I) \cdot H(id_1)^{-1}$, when MSet-Add-Hash, MSet-XOR-Hash, and MSet-Mu-Hash are used, respectively. However, the correct search result is I in this case, and the correct proof should be $\mathcal{H}(I)$.

Another scenario that can break the correctness is to add the same keyword-identifier pairs multiple times. Assume the update sequence associated with w is $((add, id_1), (add, id_1))$. When the MSet-Add-Hash is used, the proof of w will be $\mathcal{H}(I) + H_K(id_1) + H_K(id_1)$ mod Z, i.e., $\mathcal{H}(\{I, id_1, id_1\})$. For result-revealing schemes, the server will return $R = \{I, id_1\}$ during a search on w, yet $\mathcal{H}(R) \neq \mathcal{H}(\{I, id_1\})$. Similarly, MSet-XOR-Hash and MSet-Mu-Hash will update the proof of w to $\mathcal{H}(I)$ [1] and $\mathcal{H}(I) \cdot H(id_1) \cdot H(id_1)$, whereas the correct proof should be $\mathcal{H}(I) \oplus H_K(id_1)$ and $\mathcal{H}(I) \cdot H(id_1)$, respectively.

4.3. Soundness Vulnerabilities

Incremental-hash-based VDSSE schemes also fail to ensure their soundness, for instance, when the client wrongly delete a nonexistent keyword-identifier pair (w, id_1) beforehand when he/she indeed wants to add it, *i.e.* conducts $((del, id_1), (add, id_1))$ update sequence for w. In this case, the correct search result of w should be $\{I, id_1\}$, but its proof is still $\mathcal{H}(I)$ for all the three incremental hash structures. Thus, the server can just return I, which is indeed not complete without id_1. There are more scenarios that can break their soundness. Due to the page limit, here we will not give more specific examples.

Ge *et al.* [8] use a method similar to MSet-Add-Hash to generate and update proofs, *i.e.* add or subtract a value derived from the update into or from the original proof. The difference is that in their scheme, the value is derived from both the ciphertext and identifiers of the updated document. Moreover, they use a semantically secure algorithm to encrypt the document. As a result, for each update, the value appended into the proof is unique and cannot be canceled out when there are incorrect updates. Therefore, their solution does not have the soundness vulnerabilities introduced above. However, they fail to ensure the correctness. For instance, when the client adds a keyword-identifier pair multi-times, multiple values of the pair will be appended into the proof, yet only one value of this pair will be included in the proof generated by the client, resulting in failure of verification.

4.4. Discussion

Based on the above discussion, we can find that the vulnerabilities of incremental-hash-based VDSSE are caused mainly due to 3 reasons: ❶ the majority of existing SSE schemes reveal the search result to the server, thus the server can slightly amend the result, such as removing repeated identifiers. For sure, we can ask the server not to amend the result in any way, yet it can just prevent one type of incorrect updates: adding the same keyword-identifier multi-times. As shown in the first scenario given in Section 4.2, the server does nothing over the result; deleting a nonexistent keyword-identifier pair can still break the correctness guarantee in this case. ❷ The client cannot get the update history to check if he or she updates the database properly. ❸ The incremental multi-set hash does not care about the updates sequence and it only records the final proof result. Moreover, it can cancel out conflicting updates. For instance, MSet-Add-Hash will cancel out a pair of add and delete updates on the same keyword-identifier no matter what the update order is, and MSet-XOR-Hash cancels out all the updates on the same keyword-identifier no matter whether the update is add or del and no matter what the order is.

Hiding the search result effectively and efficiently from the server is non-trivial, which is still an open problem in the community. In this work, we still focus on response-revealing schemes and aim to propose a fault-tolerant VDSSE scheme by addressing the last two issues.

5. Overview of Our Approach

In this section, we provide the system model and the threat model we consider, our design goals, and the main idea of our approach.

1. $\mathcal{H}(I) \oplus H_K(id_1) \oplus H_K(id_1) = \mathcal{H}(I)$

5.1. System Model

We consider the typical scenario of data-outsourcing, which encompasses two roles: a client and a server. As defined in Section 3.2, the client and the server set up the system, and interactively perform the search and update protocols over time. Specifically, the client defines the keywords set for each document, generates the index and proofs used to verify search results, and encrypts the documents. The server holds the encrypted index, the proofs, and encrypted documents. For search operation, the client sends a search token of the keyword to the server, and the server returns both the result and associated proofs to the client. The client first verifies the authenticity of the proofs and then verifies if the search result is correct and complete with the proofs. Only when the proofs pass the authenticity and the result passes the verification, the client will accept the search result, otherwise reject it.

5.2. Threat Model

We assume the server is a malicious adversary who is curious about the documents as well as the corresponding index and keywords and tries to collect more information than the permitted leakage. Moreover, the server may also delete or forge data and only return partial or wrong search results. Note that the Denial-of-Service (DoS) and Distributed DoS (DDoS) behavior on the server is out of the scope of this work.

The client is assumed to be trusted. However, considering there is no information stored on the client side, it might carelessly send incorrect updates to the server. For instance, the client might wrongly add a keyword-identifier pair already existing in the database or delete a document that has been deleted.

Note that in this work, we focus on verifying the correctness and completeness of indices. We assume there is a mechanism available to ensure the integrity and freshness of documents, *i.e.* their contents are not tampered with and they are the latest version. For instance, the method proposed in [4], [5] can be leveraged to achieve both integrity and freshness of document content in our scheme.

5.3. Design goal and requirements

In this work, we aim to design a VDSSE scheme that can tolerate incorrect updates and has forward privacy guarantee in the malicious server model if the original DSSE is forward private. To achieve these goals, the scheme should satisfy the following requirements: (**R1**) the client should be able to verify if the search result is correct and complete in the presence of a malicious server; (**R2**) the client should be still able to verify if the search result is correct and complete even when the database has ever been updated incorrectly; and (**R3**) the server should not be able to link the updates over any data structure, including the index or proofs, to previous searches.

5.4. Our approach

To achieve **R1**, the main technique used in previous work [6], [7], [9]–[12] is to append a proof to each keyword and update the proof accordingly whenever it is affected by an update operation. As demonstrated in Section 4, those approaches fail to achieve **R2**, i.e., the proof in their designs cannot ensure the correctness and soundness when the database is updated incorrectly. To meet all of our requirements, **our main idea is to securely record the update history that happened to each keyword on the server side and append a proof to each update operation, rather than to each keyword.** There are 4 key points we highlight here to show how our approach achieves the goal. ❶ We encrypt each proof with AE to ensure all the proofs returned by the server are associated with the searched keyword and are not tampered with. If the server tampers the proofs, then they cannot be decrypted. ❷ Whether an update operation is incorrect is determined by the order of the updates that ever occurred to the keyword. For instance, deleting a keyword-identifier pair must occur after the pair has been added. The client keeps track of the order of updates by storing a counter for each keyword. The counter stores the number of updates that have occurred to the corresponding keyword. The client also stores a sequence number into each update record and proof. ❸ We include the update operation to the proof, e.g., add w into id'. By doing so, the client gets all the updates that ever happened to w and can keep track of the order in which updates occurred based on the associated sequence numbers. In this way, the client is able to identify incorrect operations and figure out the correct search result. ❹ In our scheme, all the updates are actually first recorded on the server side and then committed later during a search query. Moreover, all the update records, including the proof part, are semantically secure, and are stored in a location independent of previous queries. In this way, our scheme achieves the forward privacy property because the server cannot link any newly added records with previous searches, therefore satisfying requirement **R3**. In our approach, during a search query, the client will receive the entire history of the updates related to the searched keyword. The idea of searching for the update history is based on the concept of lazy deletions and keeping duplicates, which is used by many DSSE schemes, such as [1], [11], [14], [16], [17], [28], where the server can obtain the update history associated with the searched keyword. The innovative part achieved by our approach is that we also achieve the verifiability of the update history on top of that.

5.5. Approach Optimization

The basic version of our approach has a drawback in terms of performance: the number of index entries and proofs increases linearly with the updates that happened to the database. As mentioned, we append a proof to each update record, and all the proofs related to the searched keyword should be returned to the client for the verification, which means the overhead on the client also increases linearly with the updates. In the design for achieving forward privacy, the update history stored on the server side also affects the performance of search operation. If the searched keyword has been updated for c times, the server must make c disk accesses during the search. The work proposed in [1], [14] solves the problem by letting the server cache the search result. Inspired by

the above approaches, we have borrowed the idea to cache the search result and renew the proofs, i.e., integrate all the proofs and update records of the searched keyword into a new proof and a new update record. There are two concerns for renewing the proofs: (i) renewing the proofs adds computational overhead on the client; and (ii) after renewing the proof, our scheme should still meet all of our requirements (**R1**, **R2**, and **R3**).

To address the first concern, we set a configurable threshold for renewing the proofs. In this work, we set the threshold to $|\mathrm{DB}(w)|$ as an example, i.e., the client only renews the proofs of w when the number of associated proofs is greater than $|\mathrm{DB}(w)|$. For the second concern, when renewing the proofs, the client performs the following operations: first, it checks the completeness and correctness of the result; then it includes the current correct and complete result into the new proof; this update proof is encrypted with AE and a new key; finally, the client resets the counter of the search keyword to 1.

Setup($\lambda, \perp; \perp$)

Client:

$k_s, k_t, k_p \xleftarrow{\$} \{0,1\}^\lambda$
$K \leftarrow \{k_s, k_t, k_p\}$
$WC \leftarrow$ empty map

return K, WC

Server:

$T \leftarrow$ empty map
return T

Figure 1. **Setup protocol**

Update($K, WC, op, w, id; T$)

Client:

$(c_1, c_2) \leftarrow WC[w]$
if $WC[w]$ does not exist **then**
  $c_1 \leftarrow 0, c_2 \leftarrow 0$
end if
$s_w \leftarrow F_1(k_s, w)$
$c_1 \leftarrow c_1 + 1$
$st \leftarrow F_2(k_t, w||c_1||c_2)$
if $c_1 = 1$ **then**
  $e \leftarrow H_2(s_w||st) \oplus 0^\lambda||op||id$
else
  $oldst \leftarrow F_2(k_t, w||c_1 - 1||c_2)$
  $e \leftarrow H_2(s_w||st) \oplus oldst||op||id$
end if
$k_w \leftarrow F_3(k_p, w||c_2)$
$proof \leftarrow \mathrm{AE}.Enc(k_w, c_1, op||id)$
$WC[w] \leftarrow (c_1, c_2)$
$ut \leftarrow H_1(s_w||st)$
send $(ut, e, proof)$ to the server
return WC

Server:

$T[ut] \leftarrow (e, proof)$
return T

Figure 2. **Update protocol**

Search(K, WC, w; T)

Client:

$s_w \leftarrow F_1(k_s, w)$
$(c_1, c_2) \leftarrow WC[w]$
if $WC[w]$ is empty **then**
  return
end if
$st \leftarrow F_2(k_t, w||c_1||c_2)$
send (s_w, st, c_1, c_2) to the server

Server:

$P \leftarrow$ empty list
$Del, R, Uts \leftarrow$ empty set
$i \leftarrow c_1$
while $i > 1$ **do**
  $ut \leftarrow H_1(s_w||st)$
  $Uts \leftarrow Uts \cup \{ut\}$
  $(e, proof) \leftarrow T[ut]$
  $P[i] \leftarrow proof$
  $st||op||id \leftarrow e \oplus H_2(s_w||st)$
  if $op = add$ and $id \notin Del$ **then**
    $R \leftarrow R \cup \{id\}$
  else if $op = del$ **then**
    $Del \leftarrow Del \cup \{id\}$
  end if
  $i \leftarrow i - 1$
end while
$ut \leftarrow H_1(s_w||st)$
$Uts \leftarrow Uts \cup \{ut\}$
$(e, proof) \leftarrow T[ut]$
$P[i] \leftarrow proof$
if $c_2 > 0$ **then**
  $old\_R \leftarrow \mathrm{Sym}.Dec(st, e)$
  for each id in old_R **do**
    if $id \notin Del$ **then**
      $R \leftarrow R \cup \{id\}$
    end if
  end for
else
  $0^\lambda||op||id \leftarrow e \oplus H_2(s_w||st)$
  if $op = add$ and $id \notin Del$ **then**
    $R \leftarrow R \cup \{id\}$
  end if
end if
if $|P| > |R|$ **then**
  for each ut in Uts **do**
    delete $T[ut]$
  end for
end if
send R and P to the client

Client:

$v \leftarrow \mathrm{Verify}(k_p, WC, w, R, P)$
if $v =$ "Reject" **then**
  return "Reject"
else
  if $|P| \leqslant |R|$ **then**
    return R
  else
    $\mathrm{ReProof}(K, WC, R)$
    return R
  end if
end if

Figure 3. **Search protocol**

Verify(k_p, WC, w, R, P)

$R' \leftarrow$ empty set
$(c_1, c_2) \leftarrow \mathrm{WC}[w]$
if $|P| \neq c_1$ **then**
  return "Reject"
end if
$k_w \leftarrow F_3(k_p, w||c_2)$
$r \leftarrow \mathrm{AE}.Dec(k_w, 1, P[1])$
if $r =$ "$invalid$" **then**
  return "Reject"
else
  if $c_2 > 0$ **then**
    $R' \leftarrow R' \cup r$
  else
    $op||id \leftarrow r$
    if $op = add$ **then**
      $R' \leftarrow R' \cup \{id\}$
    end if
  end if
end if
for $i = 2 \rightarrow c_1$ **do**
  $r \leftarrow \mathrm{AE}.Dec(k_w, i, P[i])$
  if $r =$ "$invalid$" **then**
    return "Reject"
  else
    $op||id \leftarrow r$
    if $op = add$ **then**
      $R' \leftarrow R' \cup \{id\}$
    else
      $R' \leftarrow R' \setminus \{id\}$
    end if
  end if
end for
if $R = R'$ **then**
  return "Accept"
else
  return "Reject"
end if

Figure 4. **Verify algorithm**

6. Solution Details

Our approach can be easily integrated with any forward private DSSE or normal DSSE schemes to make them into fault-tolerant VDSSE schemes. To the best of our knowledge, FAST [1] is the state-of-art forward secure DSSE solution. Here we take the FAST scheme as an example to show how our approach can make it a fault-tolerant VDSSE scheme in the malicious model without breaking its forward privacy guarantee.

To achieve forward privacy, the main idea of FAST and most other existing forward secure DSSE schemes [11], [14], [16], [17], is to store a chain of update records for each keyword. Moreover, when a new update happens to a keyword, the new update records will be inserted in a way that the server cannot link it with previous update records and queries. FAST [1] is a very efficient forward private DSSE scheme as it builds only with symmetric cryptographic primitives. However, FAST does not work in a malicious model.

6.1. Our Construction

Our construction uses 3 PRFs (F_1 $F_2, F_3 : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$), two hash functions ($H_1 : \{0,1\}^* \rightarrow \{0,1\}^\zeta, H_2 : \{0,1\}^* \rightarrow \{0,1\}^{2\lambda+1}$), an authenticated

ReProof(K, WC, R)

Client:

1: $(c_1, c_2) \leftarrow WC$
2: $c_1 \leftarrow 1, c_2 \leftarrow c_2 + 1$
3: $st \leftarrow F_2(k_t, w||c_1||c_2)$
4: $e \leftarrow \text{Sym}.Enc(st, R)$
5: $k_w \leftarrow F_3(k_p, w||c_2)$
6: $proof \leftarrow \text{AE}.Enc(k_w, 1, R)$
7: $WC[w] \leftarrow (c_1, c_2)$
8: $ut \leftarrow H_1(s_w||st)$
9: send $(ut, e, proof)$ to the server
10: **return** WC

Server:

11: $T[ut] \leftarrow (e, proof)$
12: **return** T

Figure 5. **ReProof algorithm**

encryption scheme AE (AE.Enc : $\{0,1\}^\lambda \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$; AE.$Dec$: $\{0,1\}^\lambda \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$), and a symmetric encryption scheme (Sym.Enc : $\{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^*$; Sym.$Dec$: $\{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^*$). In the following, we show how the **Setup**, **Update**, and **Search** protocols are constructed in our scheme. The details of the three protocols are shown in Fig. 1, Fig. 2, and Fig. 3, respectively. Particularly, the pseudocode highlight in blue is the parts we add or modify to support fault-tolerant verification for FAST in the malicious model.

- $(K, WC; T) \leftarrow$ **Setup**$(\lambda, \perp; \perp)$: This phase aims to initialize the data structures that should be stored on the client and the server. With input λ, the client randomly selects three keys k_s, k_t, and k_p from $\{0,1\}^\lambda$ in a uniform way, and creates an empty map WC, *i.e.* the secret state s. The secret keys $K = \{k_s, k_t, k_p\}$ will be used to encrypt different data structures. WC is used to map each keyword w to two counters (c_1, c_2), where c_1 represents the number of proofs of w that are stored on the server side, and c_2 records the number of times the proofs of w have been renewed.
  This phase also outputs the storage on the server side, *i.e.* EDB, which is an initialized empty hash table T. T is used to store the encrypted update history and proofs.
- $(WC'; T') \leftarrow$ **Update**$(K, WC, op, id, w; T)$: When the client needs to add/delete a document or add/delete a keyword to/from a document's keyword set, it needs to run the **Update** protocol. As mentioned, any update operation is first uploaded into the server as a new record and is indeed committed later during search operations.
  Basically, when the client needs to add or delete a keyword-identifier pair (w, id), it takes the tuple (op, w, id) as input, where $op = add$ or del, and sends the output: the update token ut, the encrypted update record e, and the proof $proof$, to the server. The server stores $(e, proof)$ in $T[ut]$.
  Compared with the FAST scheme, the main difference is that we add a new data structure $proof$. As shown in line 15 in Fig. 2, $proof$ contains the counter c_1, the operation type op, and the associated document identifier id, and it is encrypted with AE and the key k_w derived from w and c_2. By storing c_1 and c_2 on the client and including them into $proof$, the client can check if the returned proofs are complete and correct; and by encrypting $proof$ with AE the client can check if the server generates the proof properly.
  Another difference with FAST is that our approach generates the search token st from w, c_1 and c_2 (line 7 in Fig 2), whereas FAST derives st from its previous version stored on the client. This modification just aims to reduce the client storage overhead.
- $(WC, \text{DB}(w)$ or "Reject"; $T') \leftarrow$ **Search**$(K, WC, w; T)$: When searching for a keyword w, the client generates a label s_w for w with F_1 and k_s, generates the search token st with F_2 and k_t, and sends them to the server. As done in **Update** protocol, the generation of st also involves the latest values of c_1 and c_2. Moreover, both c_1 and c_2 are sent to the server for searching.
  With the received query (s_w, st, c_1, c_2), the server searches over T and returns the result R and proof list P to the client. This process can be virtually divided into 2 parts: getting R and P (line 11–40) and updating the database when renewing proofs is necessary (line 41–45).
  For getting R and P, the server generates c_1 update tokens with H_1 and fetches their $(e, proof)$ pairs from T. With s_w and st, the server also can recover the plaintext of e, which is $st||op||id$, and obtain the update history of the searched keyword. Based on op of each update record, the server constructs the search result R, *i.e.* adds id into R (Del) if the associated op is add (del). However, no matter what the operation type is, the server needs to send the proof of each matching record to the client, which is a new process added into FAST. Due to the proof renew operation, the way to process the first matching record is also different from FAST (line 28–40). Recall that c_2 represents the times the proofs have been renewed. If $c_2 > 0$, the first matching record must contain a set of matching identifiers that have been verified, *i.e.* old_R, and will be added into R (line 28–34); otherwise, the first record should be a normal update record and is processed in the same way as others (line 35–40).
  The database is updated over time, the size of P increases with update operations, which increases the overhead on the client and the server. To improve the performance of our scheme, the client renews the proofs when $|P| > |R|$ by integrating the matching update records that have been verified into one. To save storage on the server side, in the second part of the search protocol the server removes those update records as they will never be used.
  After getting R and P from the server, the client verifies P and R with **Verify**, and renews the proofs with **ReProof** if necessary.

Proof and Result Verification. In the malicious model, the client could verify the proofs and results in case the server returns wrong data. The details of the verification process are given in Fig. 4. The proofs are used to verify if the returned result is correct and complete. However, the malicious server might tamper with the proofs in order to bypass the verification. Thus, the client first needs to verify the integrity of P. In our approach, P is verified in 2 aspects: the client firstly verifies if P contains c_1 proofs and secondly verifies if each proof can

be decrypted with AE. From the pseudocode in line 7 and line 21, we can see that the proof $P[i]$ is decrypted with both i and k_w. Thus, there are 2 requirements to decrypt each proof successfully: the proof never be tampered with and the proof is in the right order. If P contains c_1 proofs and each proof can be decrypted successfully with AE, then the client can collect the right identifiers and check if R is correct and complete.

Renew Proofs. To improve the performance of the scheme, when $|P| > |R|$, the client integrates the matching records that have passed the verification into one and inserts it into T. The details of renewing the proofs are given in **ReProof** in Fig. 5. As the number of proofs associated with w will be reduced to 1, the client first resets $c_1 = 1$ and increases c_2 by 1. Second, the client generates the new update record e, the new proof $proof$, and the new update token ut. The difference with a normal update record is that $op||id$ is replaced with R in renewed e and $proof$. Moreover, the key k_w used to generate the new proof is also refreshed with the latest counter c_2. This operation can prevent the server from returning expired proofs, such as those that should be removed after proof renewing.

6.2. Generalization of Our Approach

Our solution is generic, which can treat any (forward secure) DSSE scheme as a black-box and make them fault-tolerant and verifiable in the malicious model. From the above description and the details shown in Fig. 2 and 3, we see that the dependency between our construction with FAST is minor. We mainly insert additional operations and data into FAST for achieving fault-tolerance and verifiability, rather than modifying operations of the original scheme. Specifically, the **Verify** and **ReProof** algorithms and the verification process on the client (line 47-57 in Fig. 3) are totally new processes, and they can be added into any DSSE scheme. The only combination between our approach and FAST is in **Update** and **Search** protocols. The pseudocode in blue presents all new things added into FAST, which are also independent of the original FAST. In terms of the data storage, we just need to add WC to the client. On the server side, our approach requires a hash table T to store the update history. If the original scheme has such structure, like FAST, we just need to induce a $proof$ to each record. On the contrary, if the original scheme does not store update history, we just need to add T into it to store the proofs. For FAST, the only thing we modify is the generation of search token st. As mentioned, this modification only aims to save the storage on the client. Moreover, this minor change can also be easily conducted in any other DSSE schemes. In Appendix C, we analyze the security of our generic solution.

7. Security Analysis

In this section, we provide the analysis of correctness, soundness, and security achieved by our solution.

7.1. Correctness

Theorem 1. If there is no collision in the hash table T, and the authenticated encryption scheme AE and the symmetric encryption scheme Sym satisfy correctness, our scheme achieves correctness even when there are incorrect updates.

The VDSSE scheme is correct if for each keyword w the search result returned by the server is $\mathrm{DB}(w)$ when the server honestly follows the designed search and update protocols, and the client accepts the search result. In our scheme, as shown in Fig. 2, for each update, the update record and the corresponding proof are uploaded to the server. Both the update record e and the proof contain the values op and id related to the update operation. When the proofs related to a keyword w have not been renewed yet, and there is no collision in the hash table T, all the updates related to w are stored in T properly. For a search on w, by following the **Search** protocol, the server obtains all the update records related to w in the correct order. By decrypting all the obtained update records, the server gets the full update history for keyword w. In this way, the server is able to detect any incorrect updates and recover $\mathrm{DB}(w)$. The server also obtains all proofs associated with the searched keyword w and returns them to the client. Recall that each proof also contains op and id. Thus, when the client decrypts the proofs it is able to recover the full update and correct update history of w as long as the scheme AE is correct. From this, the client computes $\mathrm{DB}(w)$. Since each $(e, proof)$ pair related to w contains the same op and id, the search result recovered from the proofs must be equal to the one received from the honest server.

The correctness is also ensured when the proofs are renewed. In this case, the server obtains the search result from the packaged search result and the update records inserted after renewing the proofs. The matching identifiers included in update records must be correct as they are processed in the same way as before. The packaged search result included in the renewed index and proof must be correct as well because the index and proof are renewed after the verification. As long as the scheme Sym is correct, the server can obtain the packaged result. The client can also get the packaged result from the renewed proof as long as AE is correct and the returned proofs match the latest k_w.

7.2. Soundness

Theorem 2. If AE satisfies authenticity defined in Section 3.1 and our scheme Σ satisfies correctness, then our scheme Σ achieves soundness even when there are incorrect updates.

The VDSSE scheme is sound when the client can detect malicious behavior from the server. Our approach achieves soundness by appending a proof to each update record, ensuring the integrity of each proof with AE, numbering each proof, and recording the total number of proofs on the client. When the server forges a proof, returns out-of-order or incomplete proofs, due to the authenticity of AE, the client can detect these malicious behaviours and reject the results. In particular, for the i-th update (op, w, id) related to w, the proof is generated by computing $\mathrm{AE}.Enc(k_w, i, op||id)$. Recall that only when both the key k_w and the nonce i are correct can AE decrypt successfully. Therefore, only when the server returns all

the correct proofs in the correct order, the client can decrypt them with AE. In any other case, such as a forged proof, a proof associated with another keyword, a tampered proof, and a proof in the wrong order, it will all be rejected because AE cannot decrypt correctly. If the returned proofs are verified to be correct and complete, from Section 7.1, we know that the client can obtain $\mathrm{DB}(w)$ after decrypting all the proofs such that it can detect whether the server returns an invalid search result by comparing if the search result is equal to $\mathrm{DB}(w)$.

We provide the formal soundness proof in **Appendix A**.

7.3. Security proof

We use $\mathcal{Q}$ to represent the list of the performed queries, each of which is in the form of (t, w) (search) or (t, op, id, w) (update) where t is the timestamp and define two leakage functions $\mathrm{sp}(w)$, $\mathrm{UpdHis}(w)$. The search pattern on w is denoted as $\mathrm{sp}(w)$ and the update history related to w are denoted as $\mathrm{UpdHis}(w)$. Formally,

$$\mathrm{sp}(w) = \{t | (t, w) \in \mathcal{Q}\}$$
$$\mathrm{UpdHis}(w) = \{(t, op, id) | (t, op, id, w) \in \mathcal{Q}\}$$

Theorem 3. If F_1, F_2, F_3 are PRFs, AE satisfies the privacy and authenticity defined in Section 3.1, and H_1, H_2 are two hash functions modeled as two random oracles outputting ζ and λ bits respectively, then our construction Σ is $\mathcal{L}_\Sigma$-adaptively-secure where

(A) $\mathcal{L}_\Sigma^{Setup}(\lambda) = \perp$
(B) $\mathcal{L}_\Sigma^{Search}(\mathrm{DB}, w) = (\mathrm{sp}(w), \mathrm{UpdHis}(w))$
(C) $\mathcal{L}_\Sigma^{Update}(\mathrm{DB}, op, id, w) = \perp$

The proof for Theorem 3 is presented in **Appendix B**.

8. Performance Analysis

In this section, we analyze the performance overhead of our scheme in terms of computation, communication, and storage. Table 1 shows the comparison between our solution and other VDSSE schemes that support single-keyword search queries.

8.1. Computation and Communication Overhead

Update. For an update query, as shown in Fig. 2, our scheme only requires the client to generate an encrypted update record and a proof with $O(1)$ computational cost, and upload them to the server at the cost of $O(1)$ communication overhead.

Search. For a search query, the main overhead occurs due to the search process on the server side and the verification process on the client side. Assume there are u updates for the searched keyword w. The computational complexity on the server, on the client, and the communication overhead between them are all linear with the number of $(e, proof)$ pairs associated with w.

In our scheme, each $(e, proof)$ pair is stored in a hash table and indexed with an update token. The server has to return to the client all the related $(e, proof)$ pairs and the search result $\mathrm{DB}(w)$. When the proofs related to w have not been renewed, the main operation on the server side is to generate u update tokens and decrypt u encrypted update records. The computation overhead on the server is thus $O(u)$. The client is required to verify the proofs. Unlike incremental-hash-based VDSSE schemes, our scheme has u proofs associated with w and it requires the client to verify each proof separately. This takes $O(u)$ computational cost. The server sends the search result and u proofs to the client, thus the communication overhead between the server and client is $O(u)$.

When $u > |\mathrm{DB}(w)|$, the $(e, proof)$ pairs of w stored in the hash table will be significantly reduced by renewing proofs. However, in this case, decrypting the packaged e and $proof$ is more expensive than decrypting a normal one as the length of them is much longer which is linear to $|old\_R|$. In the best case, there is no new update after renewing the proofs, $old\_R = \mathrm{DB}(w)$, and only the packaged $(e, proof)$ pair matches w. In this case, the main overhead on the server and the client will be decrypting the packaged e and $proof$, respectively. In this case, the computation overhead on both the server and the client is $O(m)$, where $m = |\mathrm{DB}(w)|$. Likewise, the server just needs to send the search result and one packaged proof to the client, thus the communication overhead is also $O(m)$ in this case.

8.2. Storage Overhead

Our scheme requires the client to permanently store two counters for every keyword, resulting in the permanent client storage of $O(|\mathrm{W}| \log |\mathrm{D}|)$. In addition, the client also needs to store search results and proofs temporarily. Proofs are additional compared with DSSE schemes, which is $O(u\lambda)$ in the worst case. Precisely, during a search query, the client needs to keep m document identifies and u proofs. This can be reduced to $O(m\lambda)$ by letting the client process the proofs in a stream fashion while renewing or decrypting proofs.

Assuming that the total number of document/keyword pairs that are historically added and deleted from the database is N^+, the server at most needs $O(N^+)$ storage space, which happens when there are no renewed proofs.

TABLE 2. STORAGE OVERHEAD

Scheme		Number of keywords			
		5×10^4	2×10^5	8×10^5	32×10^5
Zhang *et al.*'	Client	4.0MB	16MB	61MB	224MB
	Server	124MB	321MB	1.1GB	4.1GB
Ours	Client	536KB	1.7MB	6.1MB	21MB
	Server	179MB	488MB	1.7GB	6.2GB

9. Performance Evaluation

We implemented a prototype of our scheme and evaluated its performance with diverse datasets.

Experiment setting. Our prototype is implemented in C++ with Crypto++ library [37]. In our implementation, F_1, F_2, F_3, and symmetric encryption are implemented with AES-CTR-128, H_1 and H_2 are implemented with SHA-256. AE is implemented with the

project of OCB3 from [38]. RocksDB [39] is used to store the data on the client and the server. gRPC [40] is used for implementing the communication between the client and the server.

We ran our experiments on two desktops in two isolated LANs that are connected with VPN [41]. The bandwidth of the VPN during the test ranges between about 200KB/s and 900KB/s. Note that the performance of our scheme will be much better when the server and the client are located in the same LANs, as the bandwidth of the network will be much larger and more stable. The desktop that plays as the server is deployed with 16 cores (Intel Core i9-9900 CPU 3.10 GHz), 31 GB memory, and 483 GB SSD disk space, and the client desktop has 4 cores (Intel Core i5-6600 CPU 3.30GHz), 7.7 GB memory, and 984 GB HDD disk space. Both of them run on Ubuntu 18.04. We also ran our experiments in a local setting where both the client and the server are deployed on the former desktop. For each test case in the following, we test 10 times and take the average.

Datasets. We used five diverse datasets for the test. The first one is a real-world dataset from Wikimedia [42]. It consists of 470,483 keywords and 8,388,608 keyword-identifier pairs. We created another four datasets with different numbers of keywords.The keywords and keyword-identifier pairs contained by each of them are 5×10^4 and 1,019,750, 2×10^5 and 4,079,000, 8×10^5 and 16,316,000, and 32×10^5 and 65,264,000, respectively.

Comparison. In order to better show the performance of our scheme, we also compared the performance of our scheme with the scheme proposed by Zhang *et al.* [7]. To the best of our knowledge, Zhang *et al.* present the fastest forward secure VDSSE despite its failure to support fault tolerance. For the comparison, we downloaded their source code [43] and evaluated their performance with the same datasets on the same testbed. Since the code of OCB3 is optimized with AES-NI, for fairness, we also optimized the implementation of MSet-XOR-Hash in Zhang *et al.*' work with AES-NI.

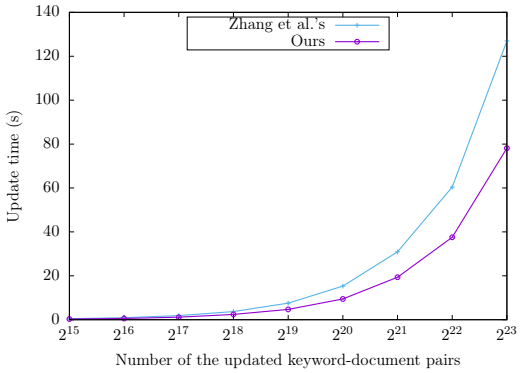
Bost *et al.* [6] do not implement the prototype of their solution and evaluate the performance. Therefore, we cannot provide the specific performance comparison with the only exiting VDSSE scheme that achieves fault tolerance and forward privacy.

9.1. Storage Overhead

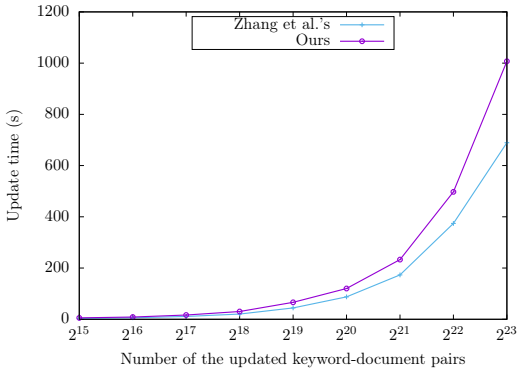
We measured the storage overhead on the client and the server in our scheme and Zhang *et al.*'s scheme, and the results are shown in Table 2. It is measured after creating the encrypted database of the four synthetic datasets. From Table 2, we can see that our scheme saves nearly $9\times$ storage on the client compared with Zhang *et al.*'s scheme and approximately adds 0.5 times more storage than Zhang *et al.*'s scheme on the server side due to the proofs.

9.2. Update Efficiency

We evaluated the performance of the update protocol by measuring the time of inserting different numbers of (add, w, id) records into an empty database in batch, and the results are shown in Fig 6. From Fig. 6(a), we can see that Zhang *et al.*'s scheme takes about 0.5 times longer than ours to complete batch updates in the local setting. The main reason is that Zhang *et al.*'s scheme needs to perform 3 hashes to update the proof while our scheme only requires 1 AE encryption, which is more efficient. However, when considering network latency, our update speed is about 0.4 times slower than theirs. This is because our update protocol requires more communication overheads to convey proofs. The required communication overheads in batch updates are displayed in Table 3, from which we can see our batch update protocol needs to transmit approximately 0.7 times more data than Zhang *et al.*'s scheme.



(a) Without network latency



(b) With network latency

Figure 6. Performance of the update protocol

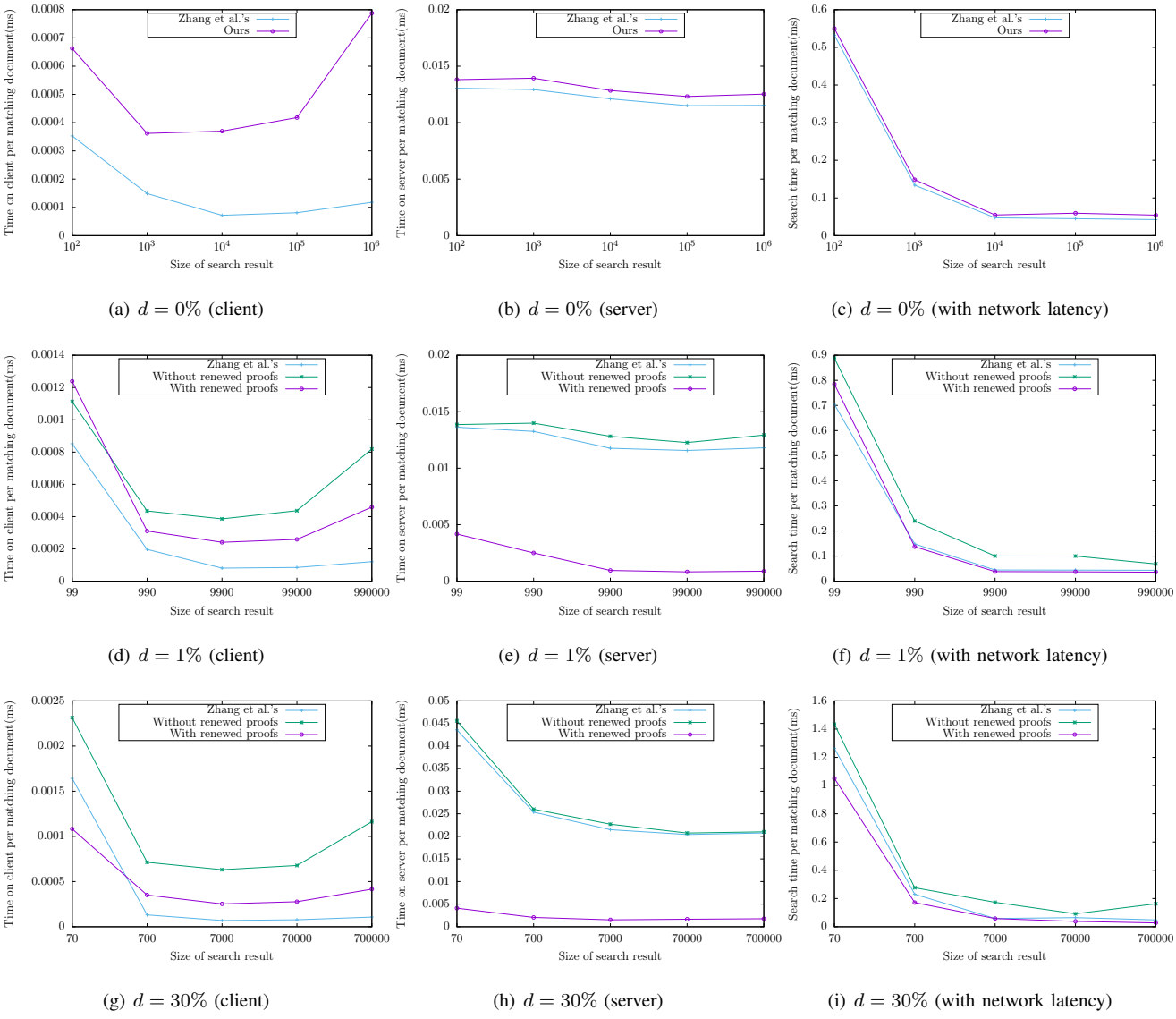
9.3. Search Efficiency

The search efficiency of our scheme is linear to the matching $(e, proof)$ entries. Renewing proofs periodically is our method to improve search efficiency. In this part, we measured the time for search queries when the proofs are not renewed and when the proofs are renewed to show the performance of our solution. Moreover, we also evaluated the overhead of **ReProof**.

To test search performance comprehensively, we first tested the time consumed by the client and the server respectively in the local setting. The time on the client is mainly spent on computing the search token and verifying search results, and the time on the server is for searching the encrypted database. Then we measured the search efficiency in our network setting to test the impact of network latency.

TABLE 3. COMMUNICATION OVERHEADS IN BATCH UPDATES

Scheme	Number of keyword-document pairs								
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
Zhang *et al.*'	1.8MB	3.6MB	7.2MB	15MB	29MB	57MB	115MB	228MB	457MB
Ours	3.1MB	6.2MB	13MB	25MB	49MB	98MB	196MB	392MB	783MB



(a) $d = 0\%$ (client) (b) $d = 0\%$ (server) (c) $d = 0\%$ (with network latency)

(d) $d = 1\%$ (client) (e) $d = 1\%$ (server) (f) $d = 1\%$ (with network latency)

(g) $d = 30\%$ (client) (h) $d = 30\%$ (server) (i) $d = 30\%$ (with network latency)

Figure 7. **Performance of the search protocol**

TABLE 4. COMMUNICATION OVERHEADS IN SEARCHES ($d = 0\%$)

Scheme	Size of search result				
	10^2	10^3	10^4	10^5	10^6
Zhang *et al.*'	4KB	8KB	80KB	784KB	7.7MB
Ours	8KB	48KB	480KB	4.7MB	47MB

TABLE 5. COMMUNICATION OVERHEADS IN SEARCHES ($d = 1\%$), WHERE "WITHOUT" REFERS TO "WITHOUT RENEWED PROOFS" AND "WITH" REFERS TO "WITH RENEWED PROOFS"

Scheme	Size of search result				
	99	990	9900	99000	990000
Zhang *et al.*'	4KB	8KB	80KB	776KB	7.6MB
Ours (without)	8KB	48KB	480KB	4.7MB	47MB
Ours (with)	16KB	28KB	276KB	2.7MB	27MB

Search without renewed proofs. Fig. 7(a), Fig. 7(b), and Fig. 7(c) show the performance of our search protocol when there are no deletion operations related to the searched keywords. There is no renewed proof in this case as the client only renews the proofs when $|P| > |R|$. The tested queries match 10^2, 10^3, 10^4, 10^5, and 10^6 documents, respectively. From Fig.7(a), we can see that the average time consumed by the client on

TABLE 6. COMMUNICATION OVERHEADS IN SEARCHES ($d = 30\%$), WHERE "WITHOUT" REFERS TO "WITHOUT RENEWED PROOFS" AND "WITH" REFERS TO "WITH RENEWED PROOFS"

Scheme	Size of search result				
	70	700	7000	70000	700000
Zhang *et al.*'	4KB	8KB	56KB	548KB	5.4MB
Ours (without)	8KB	60KB	576KB	5.6MB	56MB
Ours (with)	4KB	20KB	196KB	1.9MB	19MB

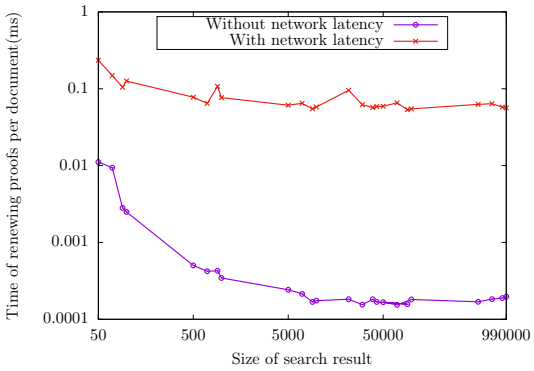


Figure 8. Performance of renewing proofs

a single matching document drops first and then rises as the size of the search results increases. The reason for the decrease is that the start-up time is distributed to more searched entries. For example, the client computes a search token for a search query regardless of the size of the search result, so when the size of the search result increases, the time allocated to each matching document for computing the search token will be less. However, when the size of the search result is large, frequent construction and destruction of data structures cause a negative impact on the performance. Fig.7(a) also shows that Zhang *et al.*'s scheme is more efficient than ours in terms of the computation on the client. That is because, for the verification, the AE decryption algorithm used in our protocol is more expensive than their MSet-XOR-hash technique. Besides, when the client verifies the search result in our protocol, after decrypting all the proofs, it also needs to traverse all the proofs to restore the matching documents, which is not required by Zhang *et al.*'s scheme.

Fig. 7(b) shows the time consumed by the server to obtain matching documents and the related proofs. In this case, our search protocol is slightly less efficient than Zhang *et al.*'s scheme. This is because the server in our protocol also needs to find the related proofs. Instead, in Zhang *et al.*'s scheme, the proofs are only stored on the client side. By comparing Figure. 7(a) and Fig.7(b), we can discover that the overhead of the server accessing the database is the main bottleneck of the search efficiency in the local setting. This point is also emphasized in several forward secure SSE works [1], [11], [14].

Fig. 7(c) shows the total search overhead when introducing network latency. In our settings, all the communication is encrypted and needs to be relayed by the VPN server; the network latency is a dominant performance factor in our experiments. In addition, our protocol also requires more communication overheads than Zhang *et al.*'s scheme due to the need to transmit the proofs from the server to the client. The comparison of communication overheads in the search queries is shown in Table 4.

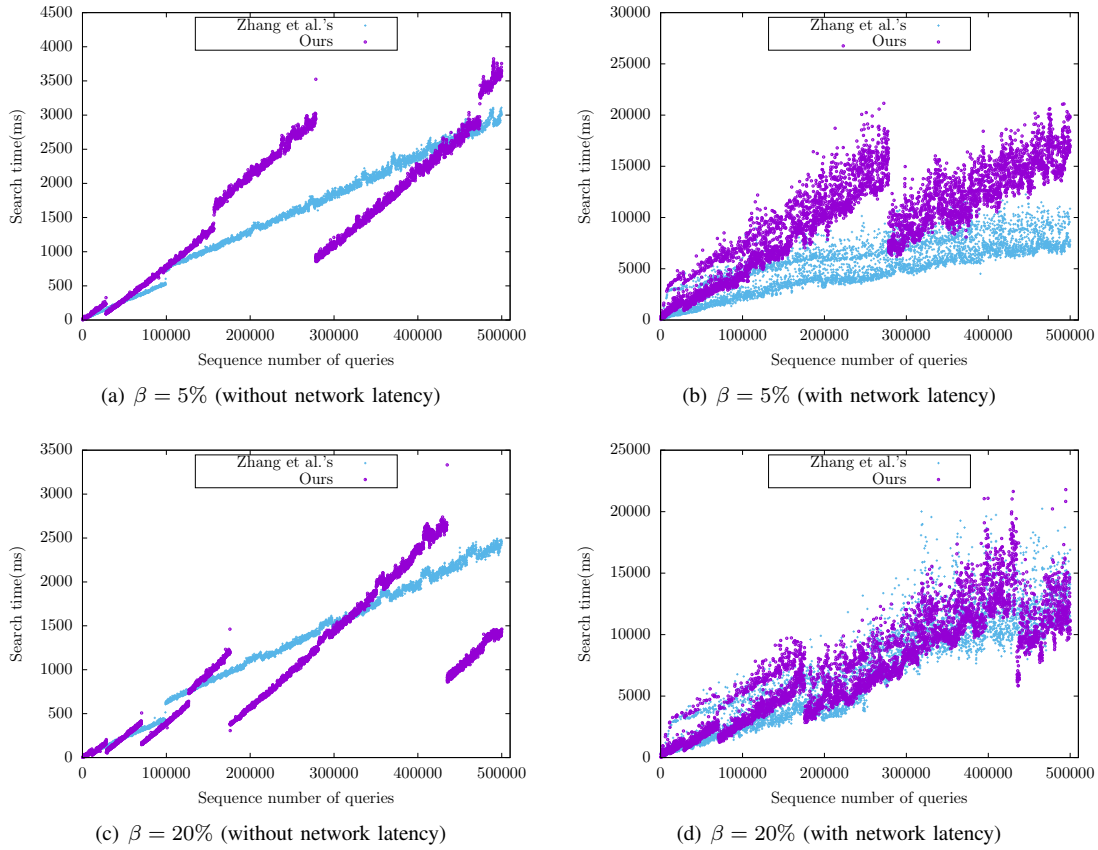
Search with renewed proofs. We also tested the search performance of our scheme when there are two different rates of delete operations. With delete queries, the **ReProof** process will be triggered. The results are shown in Fig. 7(d) – Fig. 7(i). In these figures, d represents the percentage of documents that are marked with del among the matching documents. For example, when $d = 30\%$, the numbers of matching documents after the deletion operations for the five queries are 70, 700, 7000, 70000, and 700000, respectively. In this case, the proofs will be renewed as $|P| > |R|$. To show how the performance is improved with proof renewing, we also tested the search time without renewed proofs as a baseline.

In the case of $d = 1\%$, from Fig. 7(d), we can see that with renewed proofs, we reduced the time spent by the client per matching document by approximately 30% - 40%, compared to the case without renewed proofs, when the size of the search result ranges from 990 - 990000. When the size of the search result is 99, which means that only one matching document was deleted, the operation of renewing proofs slightly reduces verification efficiency. Fig. 7(e) shows the time spent on the server side per matching document. With renewed proofs, the number of times the server accesses the database is reduced to 1, making our scheme very efficient. In general, with renewed proofs, the efficiency on the client and server of our solution is 7-9 times faster than Zhang *et al.*'s solution. However, as shown in Fig.7(f), this advantage has been greatly weakened when including network latency. Our search performance with renewed proofs is only slightly better than Zhang *et al.*'s scheme. The reason is that, as shown in Table 5, even if the proofs were renewed, our communication overheads are still greater than Zhang *et al.*'s scheme, which caused an increase in network delay in our solution.

When $d = 30\%$, the impact of renewing proofs on the search performance of our solution is more obvious. We can see from Fig. 7(g) that for our solution, the search efficiency was improved by 50% - 65% after renewing proofs. Similar to Fig. 7(e), Fig. 7(h) also shows that the search time spent by the server is reduced significantly. The total time spent on the client and server is 7-12 times less than Zhang *et al.*'s scheme needs. When counting network latency, as shown in Fig. 7(i), our solution has more obvious advantages than when $d = 1\%$ and is 0.2-0.7 times faster than Zhang *et al.*'s scheme. The communication overheads consumed by our solution and Zhang *et al.*'s scheme can be seen in Table 6.

9.4. Renewing proofs

Fig. 8 displays the average time per document when renewing the proof without and with network latency, respectively. To do this experiment, we first deleted part $(1\%, 10\%, 30\%, 50\%)$ of the documents matched by the keywords searched in the above paragraph, then measured the time to renew the proofs in the next search, and divided the obtained time by the size of the search result. From Fig. 8, we can see that the overheads of renewing proofs are very small without network latency. If network latency



(a) $\beta = 5\%$ (without network latency)

(b) $\beta = 5\%$ (with network latency)

(c) $\beta = 20\%$ (without network latency)

(d) $\beta = 20\%$ (with network latency)

Figure 9. **Performance of the search protocol by trace simulation**

is considered, the consumed time mainly depends on the network bandwidth between the client and the server.

9.5. Trace simulation

To better measure the search performance, we tested the search time for Zhang *et al.*'s scheme and our solution in the dynamic setting through producing traces of search and update queries. Specifically, we chose a keyword and produced two traces with different deletion frequencies for the keyword. Each trace contains 500000 queries on the chosen keyword, where the frequency of search, deletion, and addition queries is 1%, β (β is 5% or 20%), $99\% - \beta$, respectively. In the experiment, for each trace, we ran every query in the trace and recorded the time consumed by each search query in the local and network setting, respectively. Fig.9(a) and 9(b) show the results for the trace with 5% deletion frequency. The results for the other trace are presented in Fig.9(c) and 9(d).

Through observing the four figures, we first can confirm again that the procedure of renewing proofs helps improve the search performance of our solution a lot. Second, the trend also demonstrates that our scheme exhibits more competitive efficiency as the number of queries increases. Third, from Fig.9(b) and 9(d), we can see that network instability has a considerable impact on search time. Besides, the overall search performance gap between our scheme and Zhang *et al.*'s solution is relatively small. When $\beta = 5\%$, the average search time consumed by Zhang *et al.*'s scheme is 20% and 34% less than our scheme without and with network latency, respectively. In the case that $\beta = 20\%$, our average search efficiency is 13% faster than Zhang *et al.*'s scheme without network latency, and is almost equal to their scheme when network latency is considered.

10. Conclusion and Future Work

In this paper, we demonstrate that most of the VDSSE schemes cannot guarantee the correctness and soundness when the client updates the database improperly. We propose an efficient VDSSE scheme that can tolerate incorrect updates with forward privacy guarantee. However, our solution currently only supports single-keyword search in the single-user scenario. In our future work, we will consider complex queries and extend our scheme to the multi-user scenario.

11. Acknowledgement

Yuan and Russello would like to acknowledge the MBIE-funded programme STRATUS (UOWX1503) for its support and inspiration for this research.

References

[1] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward private searchable symmetric encryption with optimized i/o efficiency," IEEE Transactions on Dependable and Secure Computing, vol. 17, no. 5, pp. 912–927, 2020.

[2] K. Kurosawa and Y. Ohtaki, "Uc-secure searchable symmetric encryption," in 16th International Conference on Financial Cryptography and Data Security, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, pp. 285–298.

[3] Q. Chai and G. Gong, "Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers," in IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, June 10-15, 2012, pp. 917–922.

[4] K. Kurosawa and Y. Ohtaki, "How to update documents verifiably in searchable symmetric encryption," in 12th International Conference on Cryptology and Network Security, CANS 2013, Paraty, Brazil, November 20-22. 2013, pp. 309–328.

[5] K. Kurosawa, K. Sasaki, K. Ohta, and K. Yoneyama, "Uc-secure dynamic searchable symmetric encryption scheme," in 11th International Workshop on Security, IWSEC 2016, Tokyo, Japan, September 12-14, 2016, pp. 73–90.

[6] R. Bost, P.-A. Fouque, and D. Pointcheval, "Verifiable dynamic symmetric searchable encryption: Optimality and forward security," https://eprint.iacr.org/2016/062, 2016.

[7] Z. Zhang, J. Wang, Y. Wang, Y. Su, and X. Chen, "Towards efficient verifiable forward secure searchable symmetric encryption," in 24th European Symposium on Research in Computer Security, ESORICS 2019, Luxembourg, September 23-27, 2019, pp. 304–321.

[8] X. Ge, J. Yu, H. Zhang, C. Hu, Z. Li, Z. Qin, and R. Hao, "Towards achieving keyword search over dynamic encrypted cloud data with symmetric-key based verification," IEEE Transactions on Dependable and Secure Computing, vol. 18, no. 1, pp. 490–504, 2021.

[9] Y. Guo, C. Zhang, and X. Jia, "Verifiable and forward-secure encrypted search using blockchain techniques," in IEEE International Conference on Communications, ICC 2020, Dublin, Ireland, June 7-11, 2020, pp. 1–7.

[10] M. Miao, Y. Wang, J. Wang, and X. Huang, "Verifiable database supporting keyword searches with forward security," ELSEVIER Computer Standards & Interfaces, vol. 77, p. 103491, 2021.

[11] R. Bost, "$\Sigma o \varphi o \varsigma$–forward secure searchable encryption," in 23rd ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, Vienna, Austria, October 24-28, 2016, pp. 1143–1154.

[12] J. Zhu, Q. Li, C. Wang, X. Yuan, Q. Wang, and K. Ren, "Enabling generic, verifiable, and secure data search in cloud services," IEEE Transactions on Parallel and Distributed Systems, vol. 29, no. 8, pp. 1721–1735, 2018.

[13] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in 25th USENIX Security Symposium, USENIX Security 2016, Austin, TX, USA, August 10-12, 2016, pp. 707–720.

[14] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in 24th ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, pp. 1465–1482.

[15] R. W. Lai and S. S. Chow, "Forward-secure searchable encryption on labeled bipartite graphs," in 15th International Conference on Applied Cryptography and Network Security, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, pp. 478–497.

[16] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in 24th ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, pp. 1449–1463.

[17] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans, "Efficient dynamic searchable encryption with forward privacy," in 18th Privacy Enhancing Technologies Symposium, PETS 2018, Barcelona, Spain, July 24–27, 2018, pp. 5–20.

[18] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014.

[19] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Cryptographic accumulators for authenticated hash tables," https://eprint.iacr.org/2009/625, 2009.

[20] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in 9th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT 2003, Taipei, Taiwan, November 30 - December 4, 2003, pp. 188–207.

[21] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in 17th International Conference on Financial Cryptography and Data Security, FC 2013, Okinawa, Japan, April 1-5, 2013, pp. 258–274.

[22] M. Etemad and A. Küpçü, "Verifiable dynamic searchable encryption," Turkish Journal of Electrical Engineering & Computer Sciences, vol. 27, no. 4, pp. 2606–2623, 2019.

[23] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in 21st IEEE Symposium on Security and Privacy, S&P 2000, Berkeley, CA, USA, May 14-17, 2000, pp. 44–55.

[24] E. J. Goh, "Secure indexes," http://eprint.iacr.org/2003/216, 2003.

[25] Y. C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in 13th International Conference on Applied Cryptography and Network Security, ACNS 2005, New York, NY, USA, June 7-10, 2005, pp. 442–455.

[26] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, Virginia, USA, October 30-November 3, 2006, pp. 79–88.

[27] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in 19th ACM conference on Computer and communications security, CCS 2012, Raleigh, North Carolina, USA, October 16-18, 2012, pp. 965–976.

[28] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014, pp. 23–26.

[29] J. Wang, X. Chen, X. Huang, I. You, and Y. Xiang, "Verifiable auditing for outsourced database in cloud computing," IEEE Transactions on Computers, vol. 64, no. 11, pp. 3293–3303, 2015.

[30] W. Ogata and K. Kurosawa, "Efficient no-dictionary verifiable searchable symmetric encryption," in 21st International Conference on Financial Cryptography and Data Security, FC 2017, Sliema, Malta, April 3-7, 2017, pp. 498–516.

[31] M. T. Goodrich, R. Tamassia, and N. Triandopoulos, "Efficient authenticated data structures for graph connectivity and geometric search problems," Algorithmica, vol. 60, no. 3, pp. 505–552, 2011.

[32] O. Goldreich, Foundations of cryptography, 2004.

[33] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum project yellow paper, vol. 151, no. 2014, pp. 1–32, 2014.

[34] N. Szabo, "Smart contracts," 1994.

[35] P. Rogaway, M. Bellare, J. Black, and T. Krovetz, "Ocb: A block-cipher mode of operation for efficient authenticated encryption," ACM Transactions on Information and System Security, vol. 6, no. 3, p. 365–403, 2003.

[36] T. Krovetz and P. Rogaway, "The software performance of authenticated-encryption modes," in 18th International Workshop on Fast Software Encryption, FSE 2011, Lyngby, Denmark, February 13-16, 2011, pp. 306–327.

[37] W. Dai, "Crypto++ library 8.2," https://www.cryptopp.com.

[38] T. Krovetz and P. Rogaway, "OCB," https://www.cs.ucdavis.edu/~rogaway/ocb.

[39] Facebook, "Rocksdb 6.6.4," https://github.com/facebook/rocksdb/tree/v6.6.4.

[40] Google, "grpc," https://github.com/grpc/grpc.

[41] Fortinet, "Forticlient vpn," https://www.fortinet.com/.

[42] "Wikimedia dump service," https://dumps.wikimedia.org/enwiki/.

[43] Z. Zhang, "VFSSSE," https://github.com/zhangzhongjun/VFSSSE.

Verify(k_p, WC, w, R, P^*)

1: $R' \leftarrow$ empty set
2: $(c_1, c_2) \leftarrow \mathrm{WC}[w]$
3: **if** $|P^*| \neq c_1$ **then**
4: **return** "Reject"
5: **end if**
6: $k_w \leftarrow F_3(k_p, w||c_2)$
7: $r \leftarrow \mathrm{AE}.Dec(k_w, 1, P^*[1])$
8: **if** $r \neq$ "$invalid$" and $P^*[1]$ is not a ciphertext produced by $\mathrm{AE}.Enc$ with the key k_w and the nonce 1 **then**
9: **return** "Accept"
10: **end if**
11: **if** $r =$ "$invalid$" **then**
12: **return** "Reject"
13: **else**
14: **if** $c_2 > 0$ **then**
15: $R' \leftarrow R' \cup r$
16: **else**
17: $op||id \leftarrow r$
18: $R' \leftarrow R' \cup \{id\}$
19: **end if**
20: **end if**
21: **for** $i = 2 \rightarrow c_1$ **do**
22: $r \leftarrow \mathrm{AE}.Dec(k_w, i, P^*[i])$
23: **if** $r \neq$ "$invalid$" and $P^*[i]$ is not a ciphertext produced by $\mathrm{AE}.Enc$ with the key k_w and the nonce i **then**
24: **return** "Accept"
25: **end if**
26: **if** $r =$ "$invalid$" **then**
27: **return** "Reject"
28: **else**
29: $op||id \leftarrow r$
30: **if** $op = add$ **then**
31: $R' \leftarrow R' \cup \{id\}$
32: **else**
33: $R' \leftarrow R' \setminus \{id\}$
34: **end if**
35: **end if**
36: **end for**
37: **if** $R = R'$ **then**
38: **return** "Accept"
39: **else**
40: **return** "Reject"
41: **end if**

Figure 10. **Game** G_0^*

Verify(k_p, WC, w, R, P^*)

1: $R' \leftarrow$ empty set
2: $(c_1, c_2) \leftarrow \mathrm{WC}[w]$
3: **if** $|P| \neq c_1$ **then**
4: **return** "Reject"
5: **end if**
6: $k_w \leftarrow F_3(k_p, w||c_2)$
7: **if** $P^*[1]$ is not a ciphertext produced by $\mathrm{AE}.Enc$ with the key k_w, the nonce 1, and the plaintext r **then**
8: **return** "Reject"
9: **else**
10: **if** $c_2 > 0$ **then**
11: $R' \leftarrow R' \cup r$
12: **else**
13: $op||id \leftarrow r$
14: $R' \leftarrow R' \cup \{id\}$
15: **end if**
16: **end if**
17: **for** $i = 2 \rightarrow c_1$ **do**
18: **if** $P^*[i]$ is not a ciphertext produced by $\mathrm{AE}.Enc$ with the key k_w, the nonce i, and the plaintext r **then**
19: **return** "Reject"
20: **else**
21: $op||id \leftarrow r$
22: **if** $op = add$ **then**
23: $R' \leftarrow R' \cup \{id\}$
24: **else**
25: $R' \leftarrow R' \setminus \{id\}$
26: **end if**
27: **end if**
28: **end for**
29: **if** $R = R'$ **then**
30: **return** "Accept"
31: **else**
32: **return** "Reject"
33: **end if**

Figure 11. **Game** G_1^*

Appendix A. Soundness Proof

Considering the client can recover the correct and complete search results from proofs, to deceive the client into accepting wrong or incomplete search results, the adversary must be able to forge proofs that can bypass the authenticity of AE. Thus, as done by Bost *et al.* in [6], we reduce the soundness of our scheme to the authenticity of AE and the correctness of our scheme through a hybrid of games.

Proof: **Game** G_0^* : The difference between G_0^* and $\mathrm{VDSSESound}_{\mathcal{A}}^{\Sigma}(\lambda)$ locates at **Verify** algorithm. The **Verify** algorithm in G_0^* is shown in Fig. 10. Compared with the original algorithm, the newly added part is highlighted in red colour. Generally speaking, G_0^* outputs 1 as long as one forged proof passes the authenticity of AE. Here the forged proof $P^*[i]$ must not be a ciphertext produced by $AE.Enc$ with k_w and i, where $1 \leq i \leq c_1$. However, $\mathrm{VDSSESound}_{\mathcal{A}}^{\Sigma}(\lambda)$ outputs 1 only when all the proofs pass the authenticity of AE. Forging one proof successfully is much easier than forging c_1 successful proofs. Thus, the adversary $\mathcal{A}$ has a greater probability of success in G_0^* than in $\mathrm{VDSSESound}_{\mathcal{A}}^{\Sigma}(\lambda)$.

Formally,

$$\Pr[\mathrm{VDSSESound}_{\mathcal{A}}^{\Sigma}(\lambda) = 1] \leq \Pr[G_0^* = 1]$$

Game G_1^*: The **Verify** algorithm in G_1^* is shown in Fig. 11. G_1^* requires that each proof in P^* must be a ciphertext generated from $AE.Enc$ with a nonce and k_w.

To distinguish G_0^* and G_1^*, the adversary should be able to forge a proof $P^*[i]$ which is not a ciphertext of $AE.Enc(k_w, i,)$ but pass the verification of $AE.Dec(k_w, i, P^*[i])$ $(1 \leq i \leq c_1)$, as G_0 outputs 1 but G_1 outputs 0 in this case. Thus, we can reduce the difficulty of distinguishing G_0^* and G_1^* to the authenticity of AE and construct a reduction $\mathcal{C}$ such that

$$\Pr[G_0^* = 1] - \Pr[G_1^* = 1] \leq \mathrm{Adv}_{\mathcal{C}}^{Auth,AE}$$

Game G_2^*: In G_2^*, we assume that the adversary can successfully find the recently added c_1 entries related to the searched keyword in the hash table T with the latest search token st related to the searched keyword. This implies that there is no collision in the hash table T. The distinguishability between G_2^* and G_1^* can be reduced to the ability to break the correctness of our scheme. Formally, there exists a reduction D such that

$$\Pr[G_1^* = 1] - \Pr[G_2^* = 1] \leq \mathrm{Adv}_{\mathcal{D}}^{Correct,\Sigma}$$

From the three schemes, we can conclude that

$$\Pr[\mathrm{VDSSESound}_{\mathcal{A}}^{\Sigma}(\lambda) = 1] \leq \Pr[G_2^* = 1] + \mathrm{Adv}_{\mathcal{C}}^{Auth,AE} + \mathrm{Adv}_{\mathcal{D}}^{Correct,\Sigma}$$

The soundess of G_2^* Here we prove the soundness of game G_2^*. Formally, we will prove that

$$\Pr[G_2^* = 1] = 0$$

In G_2^*, during a search query, the adversary has three possible malicious behaviours, which are shown below:

1) The adversary returns incomplete proofs.
2) The adversary returns enough proofs, but gives a forged proof or puts a proof in a wrong position in P.
3) The adversary returns incorrect or incomplete search results.

First, because the client uses c_1 to record the number of proofs matched by each keyword, it can determine whether the number $(|P|)$ of proofs returned by the server is equal to c_1 corresponding to the searched keyword. If the returned proofs are incomplete $(|P| < c_1)$, the client will return "Reject."

Second, if the case 2) happens during a search on a keyword w, there must exist a proof $proof$ (suppose it is

at the k-th position in P) that is not a ciphertext produced by AE.Enc with the key k_w and the nonce k. In this case, following game G_1^*, G_2^* returns "Reject."

Third, suppose that neither of the first two situations happened, the client will obtain correct and complete proofs associated with the searched keyword w. According to the correctness of our scheme, the client can decrypt each proof and finally obtain $\mathrm{DB}(w)$. Consequently, if the case 3) happens, the client will detect that the search result is not equal to $\mathrm{DB}(w)$ and return "Reject."

From the above analysis, we can conclude that G_2^* either outputs $\mathrm{DB}(w)$ or "Reject." Formally,

$$\Pr[G_2^* = 1] = 0$$

In summary, we can get that

$$\Pr[\mathrm{VDSSES}_{\mathrm{ound}}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] \leq \mathrm{Adv}_{\mathcal{C}}^{Auth,AE} + \mathrm{Adv}_{\mathcal{D}}^{Correct,\Sigma}$$

where $\mathcal{C}$ and $\mathcal{D}$ are the adversaries for the authenticity of AE and the correctness of our scheme Σ, respectively. □

Algorithm 1 S.Setup($\perp$)

Client:
1: $t \leftarrow 0$

Appendix B. Security Proof

Here we define a new leakage function $\mathrm{ReProof}(w) \subset \mathrm{sp}(w)$ to denote a series of timestamps of renewing the proof for w. Formally,

$$\mathrm{ReProof}(w) = \{t | (t, w) \in \mathcal{Q} \text{ and during the search query, the proof for } w \text{ was renewed}\}$$

Additionally, we assume the number of bits of a proof generated in an update query is l and the number of bits of the renewed proof produced in a search is l_R. For a set S, we use $\mathrm{Min}(S)$ to represent the minimal value in S.

Proof: ***Game*** G_0: The game G_0 is just the VDSSE game of the real world.

$$\Pr[\mathrm{VDSSER}_{\mathrm{EAL}}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] = \Pr[G_0 = 1]$$

Game G_1: In G_1, we create a mapping table for F_1, F_2, and F_3, respectively. The difference with G_0 is that we convert the calls to F_1, F_2, and F_3 into choosing outputs uniformly at random from their ranges $\{0,1\}^{\lambda}$ and storing the input-output pairs in corresponding mapping tables if the calls are on new inputs, and otherwise taking the values mapped by the inputs in the corresponding tables. Below we use SW and ST to represent the mapping tables corresponding to F_1 and F_2. By doing so, we can reduce the difficulty of distinguishing G_0 and G_1 to that of the security of the PRFs. More precisely, a reduction $\mathcal{B}_1$ that can make at most B calls to the PRFs can be constructed such that

$$\Pr[G_0 = 1] - \Pr[G_1 = 1] \leq 3 \cdot \mathrm{Adv}_{\mathcal{B}_1}^{prf}$$

Game G_2: In G_2, calls to H_1 are replaced by picking random values, and the random oracle H_1 is programmed during the search protocol. Specifically, the codes that the

Algorithm 2 S.Search($\mathrm{sp}(w), \mathrm{UpdHis}(w)$))

Client:
$\overline{w} \leftarrow \mathrm{Min}(\mathrm{sp}(w))$
$s_w \leftarrow \mathrm{SW}[\overline{w}]$
if $\mathrm{UpdHis}(w) = \phi$ **then**
  return
else
  $\{(t_1, op_1, id_1), \cdots, (t_u, op_u, id_u)\} \leftarrow \mathrm{UpdHis}(w)$
  $\mathrm{Updates}(w) \leftarrow \{t_1, \cdots, t_u\}$
end if
Obtain $\mathrm{ReProof}(w)$ from $\mathrm{sp}(w)$ and $\mathrm{UpdHis}(w)$
if $\mathrm{ReProof}(w) = \perp$ **then**
  $c_1 \leftarrow u$
  $c_2 \leftarrow 0$
  $c_3 \leftarrow 2$
  $st \leftarrow \mathrm{ST}[\overline{w}||1||0]$
  Program H_1, s.t. $H_1(s_w||st) \leftarrow \mathrm{UT}[t_1]$
  Program H_2, s.t. $H_2(s_w||st) \leftarrow \mathrm{E}[t_1] \oplus 0^{\lambda}||op_1||id_1$
else
  $\{t'_1, \cdots, t'_{c_2}\} \leftarrow \mathrm{ReProof}(w)$
  $c_3 \leftarrow \mathrm{Min}(\{c | t_c \in \mathrm{Updates}(w) \text{ and } t_c > t'_{c_2}\})$
  if $c_3 \neq \perp$ **then**
    $c_1 \leftarrow u - c_3 + 2$
  else
    $c_1 \leftarrow 1$
  end if
  $st \leftarrow \mathrm{ST}[\overline{w}||1||c_2]$
  Program H_1, s.t. $H_1(s_w||st) \leftarrow \mathrm{UT}[t'_{c_2}]$
end if
for $i = 2$ to c_1 **do**
  $oldst \leftarrow \mathrm{ST}[\overline{w}||i-1||c_2]$
  $st \leftarrow \mathrm{ST}[\overline{w}||i||c_2]$
  $j \leftarrow c_3 - i + 2$
  Program H_1, s.t. $H_1(s_w||st) \leftarrow \mathrm{UT}[t_j]$
  Program H_2, s.t. $H_2(s_w||st) \leftarrow \mathrm{E}[t_j] \oplus oldst||op_j||id_j$
end for
send (s_w, st, c_1, c_2) to the server
Server:
send (R, P) to the client
Client:
if $|P| \neq c_1$ **then**
  return "Reject"
end if
if $c_2 > 0$ and $\mathrm{ADEC}[P[1]] \neq t' c_2$
or $c_2 = 0$ and $\mathrm{ADEC}[P[1]] \neq t_1$ **then**
  return "Reject"
end if
for $i = 2$ to c_1 **do**
  if $\mathrm{ADEC}[P[i]] \neq t_{c_3+i-2}$ **then**
    return "Reject"
  end if
end for
Obtain $\mathrm{DB}(w)$ from $\mathrm{UpdHis}(w)$
if $R \neq \mathrm{DB}(w)$ **then**
  return "Reject"
end if
if $|P| > |R|$ **then**
  $\mathrm{UT}[t] \xleftarrow{\$} \{0,1\}^{\zeta}$
  $proof \xleftarrow{\$} \{0,1\}^{l_R}$
  $\mathrm{ADEC}[proof] \leftarrow t$
  $st \leftarrow \mathrm{ST}[\overline{w}||1||c_2 + 1]$
  $\mathrm{E}[t] \leftarrow \mathrm{Sym}.Enc(st, R)$
  send $(\mathrm{UT}[t], \mathrm{E}[t], \mathrm{PROOF}[t])$ to the server
  $t \leftarrow t + 1$
end if

Algorithm 3 S.Update($\perp$)

Client:
$\mathrm{UT}[t] \xleftarrow{\$} \{0,1\}^{\zeta}$
$\mathrm{E}[t] \xleftarrow{\$} \{0,1\}^{2\lambda+1}$
$proof \xleftarrow{\$} \{0,1\}^{l}$
$\mathrm{ADEC}[proof] \leftarrow t$
send $(\mathrm{UT}[t], \mathrm{E}[t], proof)$ to the server
$t \leftarrow t + 1$

client calls H_1 (line 17 in Fig.2 & line 8 in Fig.5) are replaced by:

$$ut \xleftarrow{\$} \{0,1\}^{\zeta}$$

$$\mathrm{UT}[w||c_1||c_2] \leftarrow ut$$

In the search protocol, the client uses a table h_1 to program the random oracle H_1. The relevant pseudocodes are shown below where 1) should be added to the place ahead of line 7 in Fig.3 and 2) displays the random oracle H_1.

1)

for $i = 1$ to c_1 **do**
  for $j = 0$ to c_2 **do**
    $st \leftarrow \mathrm{ST}[w||c_1||c_2]$
    $h_1(s_w||st) \leftarrow$
      $\mathrm{UT}[w||c_1||c_2]$
  end for
end for

2) $H_1(k||x)$

$v \leftarrow h_1(k||x)$
if $v = \perp$ **then**
  $v \xleftarrow{\$} \{0,1\}^{\xi}$
  $h_1(k||x) \leftarrow v$
end if
return v

However, there exists an event ***bad*** that causes inconsistency when the client calls H_1. Specifically, for a triple (w, c_1, c_2), the event ***bad*** of $H_1(s_w||\mathrm{ST}[w\ ||c_1||c_2]) \neq \mathrm{UT}[w||c_1||c_2]$ happens in overwhelming probability if, during an update query or when renewing the proof, the client selects a value x as $\mathrm{ST}[w||c_1||c_2]$ where $s_w||x$ was used by the adversary as the input to H_1. Fortunately, s_w and x are both generated randomly, so for the adversary that makes η queries to H_1, the probability that ***bad*** on (w, c_1, c_2) occurs is $q/2^{2\lambda}$. Now assuming that there are a total of N^* distinct triples, we can derive the probability of distinguishing G_1 from G_2.

$$\Pr[G_1 = 1] - \Pr[G_2 = 1] = \frac{N^* \cdot \eta}{2^{2\lambda}}$$

Game G_3: In G_3, we make the same changes for H_2 as for H_1 in G_2. Hence:

$$\Pr[G_2 = 1] - \Pr[G_3 = 1] = \frac{N^* \cdot \eta}{2^{2\lambda}}$$

Game G_4: In G_4, we create a mapping table ADEC, and replace all calls to $\mathrm{AE}.Enc(k, non, mes)$ (line 15 in Fig.2, line 6 in Fig.5) by the following pseudocode.

$$proof \xleftarrow{\$} \{0,1\}^{|\mathrm{AE}.Enc(k,non,mes)|}$$

$$\mathrm{ADEC}[proof] \leftarrow (k, non, mes)$$

Then we replace all the calls to $\mathrm{AE}.Dec(k, non, proof)$ (line 7 and line 21 in Fig. 4) by the following pseudocode.

if $\mathrm{ADEC}[proof]$ does not exist **then**
  $r \leftarrow$ "$invalid$"
else
  $(k', non', mes') \leftarrow \mathrm{ADEC}[proof]$
  if $k \neq k'$ or $non \neq non'$ **then**
    $r \leftarrow$ "$invalid$"
  else
    $r \leftarrow mes'$
  end if
end if

In this way, we can build a reduction $\mathcal{B}_2$ to reduce the advantage of distinguishing G_4 from G_3 to the privacy and the authenticity of AE:

$$\Pr[G_3 = 1] - \Pr[G_4 = 1] \leq \mathrm{Adv}^{Priv,Auth,AE}_{\mathcal{B}_2}$$

The Simulator S: The simulator S is built only with the leakage function $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{Setup}, \mathcal{L}_\Sigma^{Search}, \mathcal{L}_\Sigma^{Update})$ as the input, which is shown in Algorithm 1, Algorithm 2, and Algorithm 3. Recall that the leakage functions of our scheme are:

(A) $\mathcal{L}_\Sigma^{Setup}(\lambda) = \perp$
(B) $\mathcal{L}_\Sigma^{Search}(\mathrm{DB}, w) = (\mathrm{sp}(w), \mathrm{UpdHis}(w))$
(C) $\mathcal{L}_\Sigma^{Update}(\mathrm{DB}, op, id, w) = \perp$

Conclusion: In summary, we can get that:

$$|\Pr[\mathrm{VDSSEReal}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathrm{VDSSEIdeal}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1]| \leqslant 3 \cdot \mathrm{Adv}^{prf}_{\mathcal{B}_1} + 2 \cdot \frac{N^* \cdot q}{2^{2\lambda}} + \mathrm{Adv}^{priv,auth,AE}_{\mathcal{B}_2}$$

where $\mathcal{B}_1$ and $\mathcal{B}_2$ are the adversaries for PRFs and AE respectively.

□

Appendix C. Security proof of the generic solution

For the multi-round VDSSE, as described in [6], the adversary has the chance to lure the client into sending messages that reveal private information about queries and the database by returning forged results to the client in previous rounds. Owing to the forward private DSSE instantiation being used as a black-box in our generic solution, we do not know its implementation details and thus cannot analyze its security against a malicious adversary if it needs multi-rounds. Here we assume the DSSE instantiation are implemented in a single-round way. This assumption is rather reasonable considering that almost all forward private DSSE schemes [1], [11], [14]–[17] are implemented with only one round of interaction. For ease of explanation, we use Π to denote the DSSE instantiation and $G\Sigma$ to denote our generic VDSSE solution. We assume the leakages of Π during setup, search, update operations are formally captured by $\mathcal{L}_\Pi^{Setup}$, $\mathcal{L}_\Pi^{Search}$, and $\mathcal{L}_\Pi^{Update}$, respectively. The simulators of Π and our generic solution are denotes as S_Π and $S_{G\Sigma}$, respectively. In addition, we define $\mathrm{DB}_t(w)$ to record the identifiers matched by the keyword w at the timestamp t, $\mathrm{spR}(w)$ to describe the timestamps and the matching identifiers at the time of every search query on the keyword w, and $\mathrm{Updates}(w)$ to capture the timestamps of all the update operations related to the keyword w. Formally, the latter two leakage functions are shown below.

$$\mathrm{spR}(w) = \{(t, \mathrm{DB}_t(w))|(t, w) \in \mathcal{Q}\}$$
$$\mathrm{Updates}(w) = \{t|\exists id : (t, add/del, id, w) \in \mathcal{Q}$$

Theorem 4. If F_1, F_2, F_3 are PRFs, AE satisfies privacy and authenticity defined in Section 3.1, H_1, H_2 are two hash functions modeled as two random oracles outputting ζ and λ bits respectively, and the DSSE construction Π is $\mathcal{L}_\Pi$-adaptively-secure, then our generic solution $G\Sigma$ is $\mathcal{L}_{G\Sigma}$-adaptively-secure where

Algorithm 4 $S_{G\Sigma}.\text{Search}(\mathcal{L}_{\Pi}^{Srch}, \text{spR}(w), \text{Updates}(w)))$

Client:
Run $S_{\Pi}.\text{Search}(\mathcal{L}_{\Pi}^{Srch})$ to obtain the search result R.
Obtain $sp(w)$ from $\text{spR}(w)$ by removing $\text{DB}_t(w)$ from $\text{spR}(w)$
$\overline{w} \leftarrow \text{Min}(\text{sp}(w))$
$s_w \leftarrow \text{SW}[\overline{w}]$
$\{t_1, \cdots, t_u)\} \leftarrow \text{Updates}(w)$
Obtain $\text{ReProof}(w)$ from $\text{spR}(w)$ and $\text{Updates}(w)$
if $\text{ReProof}(w) = \perp$ **then**
  $c_1 \leftarrow u$
  $c_2 \leftarrow 0$
  $c_3 \leftarrow 2$
  $st \leftarrow \text{ST}[\overline{w}||1||0]$
  Program H_1, s.t. $H_1(s_w||st) \leftarrow \text{UT}[t_1]$
  Program H_2, s.t. $H_2(s_w||st) \leftarrow \text{E}[t_1] \oplus 0^{\lambda}$
else
  $\{t'_1, \cdots, t'_{c_2}\} \leftarrow \text{ReProof}(w)$
  $c_3 \leftarrow \text{Min}(\{c | t_c \in \text{Updates}(w) \text{ and } t_c > t'_{c_2}\})$
  if $c_3 \neq \perp$ **then**
    $c_1 \leftarrow u - c_3 + 2$
  else
    $c_1 \leftarrow 1$
  end if
  $st \leftarrow \text{ST}[\overline{w}||1||c_2]$
  Program H_1, s.t. $H_1(s_w||st) \leftarrow \text{UT}[t'_{c_2}]$
end if
for $i = 2$ to c_1 **do**
  $oldst \leftarrow \text{ST}[\overline{w}||i-1||c_2]$
  $st \leftarrow \text{ST}[\overline{w}||i||c_2]$
  $j \leftarrow c_3 - i + 2$
  Program H_1, s.t. $H_1(s_w||st) \leftarrow \text{UT}[t_j]$
  Program H_2, s.t. $H_2(s_w||st) \leftarrow \text{E}[t_j] \oplus oldst$
end for
send (s_w, st, c_1, c_2) to the server
Server:
send (R, P) to the client
Client:
if $|P| \neq c_1$ **then**
  return "Reject"
end if
if $c_2 > 0$ and $\text{ADEC}[P[1]] \neq t'c_2$
or $c_2 = 0$ and $\text{ADEC}[P[1]] \neq t_1$ **then**
  return "Reject"
end if
for $i = 2$ to c_1 **do**
  if $\text{ADEC}[P[i]] \neq t_{c_3+i-2}$ **then**
    return "Reject"
  end if
end for
Obtain $\text{DB}(w)$ from $\text{spR}(w)$
if $R \neq \text{DB}(w)$ **then**
  return "Reject"
end if
if $|P| > |R|$ **then**
  $\text{UT}[t] \xleftarrow{\$} \{0,1\}^{\zeta}$
  $proof \xleftarrow{\$} \{0,1\}^{l_R}$
  $\text{ADEC}[proof] \leftarrow t$
  $st \leftarrow \text{ST}[\overline{w}||1||c_2 + 1]$
  send $(\text{UT}[t], \text{PROOF}[t])$ to the server
  $t \leftarrow t + 1$
end if

(A) $\mathcal{L}_{G\Sigma}^{Setup}(\lambda) = \mathcal{L}_{\Pi}^{Setup}$
(B) $\mathcal{L}_{G\Sigma}^{Search}(\text{DB}, w) = (\mathcal{L}_{\Pi}^{Search}, \text{spR}(w), \text{Updates}(w))$
(C) $\mathcal{L}_{G\Sigma}^{Update}(\text{DB}, op, id, w) = \mathcal{L}_{\Pi}^{Update}$

Proof: The way to prove the security of the generic solution overlaps mostly with that to prove the specific solution shown in Appendix B. The games G_1, G_2, G_3, and G_4 are constructed in the same way as in Appendix B. To construct the simulator $S_{G\Sigma}$, we can obtain $S_{G\Sigma}.\text{Setup}(\mathcal{L}_{G\Sigma}^{Setup}(\lambda))$ and $S_{G\Sigma}.\text{Update}(\mathcal{L}_{G\Sigma}^{Update}(op, w, id))$ by simply combining the execution of $S_{\Pi}(\mathcal{L}_{\Pi}^{Setup})$ and $S_{\Pi}(\mathcal{L}_{\Pi}^{Setup})$ and that of $S_{\Sigma}.\text{Setup}(\lambda)$ and $S_{\Sigma}.\text{Update}(\lambda)$, respectively. Besides, $S_{G\Sigma}.\text{Search}(\mathcal{L}_{\Pi}^{Search}, \text{spR}(w), \text{Updates}(w))$ can be constructed as shown in Algorithm 4. □