IACR Communications in Cryptology
ISSN 3006-5496, Vol. 1, No. 4, 30 pages.

https://doi.org/10.62056/ayl5w4fe-3

Fault-tolerant Verifiable Dynamic SSE with Forward and Backward Privacy

Bibhas Chandra Das[1,3], Nilanjan Datta[1], Avishek Majumder[2] and Subhabrata Samajder[1,4]

[1] Institute for Advancing Intelligence, TCG CREST, India
[2] UPES, India
[3] Chennai Mathematical Institute, India
[4] Academy of Scientific and Innovative Research, India

Abstract. Dynamic Searchable Symmetric Encryption (DSSE) allows users to securely outsource their data to cloud servers while enabling efficient searches and updates. The verifiability property of a DSSE construction ensures that users do not accept incorrect search results from a malicious server while the fault-tolerance property guarantees the construction functions correctly even with faulty queries from the client (e.g., adding a keyword to a document multiple times, deleting a keyword from a document that was never added). There have been very few studies on fault-tolerant verifiable DSSE schemes that achieve forward privacy, and none of the existing constructions achieve backward privacy. In this paper, we aim to design an efficient fault-tolerant verifiable DSSE scheme that provides both forward and backward privacy. First, we propose a basic fault-tolerant verifiable DSSE scheme, dubbed FVS1, which achieves forward privacy and stronger backward privacy with the update pattern (BPUP). However, the communication complexity for the search operation of this scheme is $O(u)$, where u is the total number of updates for the search keyword. To address this issue, we propose an efficient variant of the previous DSSE scheme, called FVS2, which achieves the same functionality with an optimized communication complexity of $O(m + u')$ for search queries. Here m is the size of the result set and u' is the number of update operations made on the queried keyword after the previous search made on the keyword. This improvement comes at the cost of some additional information leakage, but it ensures the construction achieves backward privacy with the link pattern (BPLP).

Keywords: Verifiable Dynamic Searchable Symmetric Encryption · Conjunctive Queries · Fault Tolerant · Forward Privacy · Backward Privacy

1 Introduction

In the current digital era, our daily activities generate vast amounts of data. With the availability of high-speed internet and affordable third-party cloud storage, storing data in the cloud has become highly convenient. This facilitates easy access and management of the data. However, much of the data we store is sensitive and confidential, necessitating encryption to protect it from malicious servers and third-party attackers. A significant challenge emerges when we need to query this outsourced data, as conventional encryption methods do not support querying the encrypted information. Generic cryptographic tools like *fully homomorphic encryption* (FHE) and *oblivious RAM* (ORAM) can mitigate this

E-mail: bibhaschandra.das@tcgcrest.org (Bibhas Chandra Das), nilanjan.datta@tcgcrest.org (Nilanjan Datta), avishek.majumder1991@gmail.com (Avishek Majumder), subhabrata.samajder@tcgcrest.org (Subhabrata Samajder)

Received: 2024-07-09 Accepted: 2024-12-03

issue by enabling protocols that outsource encrypted data to the server during search queries. However, these tools are impractical for large databases due to their high cost. Therefore, an alternative solution is necessary to allow clients to perform keyword searches efficiently over an encrypted database while minimizing information leakage. *Searchable Symmetric Encryption* (SSE) [SWP00, CM05, CGKO06, KPR12] is a practical solution to this problem, which allows clients to search for keywords in encrypted data efficiently while reducing information leakage.

1.1 Searchable Symmetric Encryption

In SSE, a database is abstracted as a collection of documents, each containing certain keywords. Each document is associated with a unique identifier, thereby viewing the database as a collection of keyword-identifier pairs. To facilitate encrypted queries, SSE generates an index (key-value storage) where each keyword serves as the key, and the list of document identifiers where the keyword appears serves as value. This indexing method is known as an *"inverted index"* [NC07]. This inverted index is then encrypted and stored by SSE, allowing for secure queries. SSE was first introduced in [SWP00]. The first formal definition of SSE, which is still in use today, was presented in [CGKO06].

Dynamic Searchable Symmetric Encryption (DSSE). Most of the constructions mentioned above were designed explicitly for static databases, where once the database is uploaded to the server, it cannot be modified. However, in a practical scenario, we need SSE schemes to allow the addition or deletion of keywords to or from documents. The first definition and security requirements for a truly *dynamic* SSE (DSSE) scheme, allowing updates to the database, was introduced in [KPR12]. Since then, numerous DSSE schemes have been proposed. However, incorporating the ability to update keywords leaks information from the protocol during the update operation. As shown in [IKK12, CGPR15], this leakage could be exploited by a malicious server to compromise data privacy. Later, in [ZKP16], Zhang et al. showed that if the server can trick the client into injecting files with specific keywords of its choice, it can recover all queries made to the database thus far. This attack is known as *file-injection attack*.

Forward and Backward Privacy. The file-injection attack exploited a critical vulnerability of an SSE scheme not being *"forward private"*. Informally, forward privacy prevents an adversary from linking newly inserted documents to any past queries made to the database. Forward privacy was first introduced in [CGPR15], with the first formal definition and a corresponding scheme proposed in [Bos16]. Additionally, the authors of [CGPR15] discussed another important property known as *"backward privacy"*. Informally, backward privacy prevents an adversarial server from learning about any document identifiers that have been added and subsequently deleted from the database before a search query. The formal definition and a scheme achieving backward privacy was first proposed in [BMO17]. Since then forward and backward privacy has been an essential requirement for an SSE scheme.

Efficiency of DSSE Schemes. The efficiency of a DSSE scheme primarily depends on the communication complexity of the search operation. An SSE is said to achieve optimal communication, if the search result size returned by the server is exactly the size as the set of currently matching documents for the searched keyword. However, just having optimal communication complexity might not be enough for real-life applications. The efficiency also critically relies on the client computation (for search and update operations), and the client storage. Since the server is considered extremely powerful, we do not consider server computation or server storage as the efficiency metric of a DSSE scheme. Ideally, while designing a DSSE scheme, we want optimal communication with minimized client computation and storage. Very few SSE schemes achieve optimal communication

complexity. The first forward private scheme that achieves optimal communication is by [Bos16]. However, they used asymmetric key primitives. The work of [SDY+20] achieved forward privacy and was IO efficient maintaining optimal communication complexity only using symmetric key premitives. Their scheme also supported parallel processing. In [BMO17], Bost et al. proposed three backward private schemes, among which only Janus achieved optimal communication using puncturable encryption but at the cost of $O(n_w.d_w)$ search complexity, where n_w is the number of matching document for keyword w and d_w is the number of delete operation performed on w. Thereby, making the scheme impractical for a large number of deletions. Another scheme that achieves optimal communication using symmetric puncturable encryption is [SYL+18]. However, both these schemes are in a re-insertion restriction setting. That is, once a keyword identifier pair is deleted, the keyword can not be re-inserted into the database with the same identifier. The first forward and backward private scheme achieving optimal communication in a general setting (allowing reinsertion) was proposed in [CPS20]. In [CMS25], the authors introduced a generic framework that transforms any DSSE scheme into a more efficient, equally secure, and succinct version, thereby reducing communication and computation overhead and improving the overall efficiency of the DSSE scheme.

1.2 Verifiable and Fault-tolerant DSSE

In general, when designing SSE schemes, it is typically assumed that the server (considered an adversary) is honest but curious. This means the server follows the protocol correctly and does not tamper with search results but attempts to learn about the user's data from its queries. However, in practice, this assumption may not hold. The server could be fully dishonest, trying not only to learn about the user data but also to deviate from the protocol and provide incorrect results. Such adversaries are referred to as *malicious*. A *verifiable* SSE (VSSE) scheme addresses this issue by preventing a malicious server from deceiving the client. It does so by providing a *proof* for every search result, allowing the user to verify the correctness of the results returned by the server.

The first VSSE scheme was proposed in [KO12]. The security of the scheme was proven in the *universal composability* (UC) model against non-adaptive adversaries. The first VSSE against adaptive adversaries was proposed in [CG12]. However, both these works were done in the static SSE setting. The work of [KO12] was first extended to *verifiable dynamic* SSE (VDSSE) scheme in [KO13]. However, the scheme was not forward private. The first VDSSE scheme achieving forward privacy was proposed in [BFP16]. This construction was based on incremental multi-set hashing [CDvD+03]. Following this, Zhang et al. [ZWW+19] proposed another incremental multi-set hash-based SSE in symmetric key primitive. However, both these constructions require high client-side storage. In [GYZ+21], Zhang et al. proposed a novel data structure called *Accumulative Authentication Tag*, that reduces the storage overhead and also using symmetric key primitive.

All the above-mentioned verifiable DSSE constructions focus solely on preventing malicious behavior from the server. However, a client unaware of the outsourced database can also behave maliciously by sending updates for adding the same keyword-identifier pair multiple times or attempting to delete non-existent data. The incremental hash-based VDSSE schemes [ZLW+18, ZWW+19] failed to provide correct proof in such scenarios. A DSSE scheme that can tolerate such careless mistakes from a user is referred to as *fault-tolerant DSSE* (FDSSE).

There has been limited research focused on designing DSSE schemes that are both *fault-tolerant and verifiable* (FVDSSE). In [SPS14], Stefanov et al. proposed a DSSE scheme using an ORAM-style data structure with frequent rebuild operations. This scheme was forward secure and fault-tolerant, but not verifiable. Also, the search operation in this scheme is inefficient, taking linear time (on the number of documents) in the worst case. In 2016, Bost et al. [BFP16] proposed three forward private, verifiable DSSE schemes

GVS-Hash, GVS-Acc and GVS-Acc-RSA. The first construction is based on Merkle tree-like data structures and Multi-Set Hash, while the latter two are based on cryptographic accumulators. However, none of these constructions are fault-tolerant and they do not scale well with large databases. In addition to these constructions, Bost et al. [BFP16] also proposed verifiable versions of Linear SPS and Sublinear SPS introduced in [SPS14]. We refer to these two constructions as Verifiable Linear SPS and Verifiable Sublinear SPS. Since the base constructions are forward private and fault-tolerant, these verifiable versions also inherit both these properties. However, the communication complexity remains quite high. Recently, Yuan et al. [YCR22] proposed an efficient FVDSSE scheme that employs *authenticated encryption* (AE) for verification, avoiding the expensive use of Merkle trees or accumulators, thereby making the construction very fast in practice. Their proposed generic scheme when instantiated with a forward private scheme [SDY+20] ensures forward privacy. However, this construction fails to conceal both operations and identifiers during the search phase, resulting in insecurity against backward privacy. These concerns motivate us to answer the following question.

Can we construct an efficient fault-tolerant verifiable DSSE scheme achieving both forward and backward privacy?

1.3 Our Contribution

In this paper, we affirmatively address the above question and propose two Verifiable DSSE constructions that achieve fault tolerance, and forward and backward privacy. Our contributions are twofold.

- We first propose a basic fault-tolerant verifiable DSSE scheme, dubbed FVS1, that achieves forward privacy and backward privacy with update pattern (BPUP). The construction uses client computation of $O(u)$ and $O(1)$ for update and search operations, respectively. The communication complexity is $O(1)$ for update operations. However, the communication complexity is $O(u)$ for search queries. The client storage required for the construction is $O(|W| \log |D|)$.
- Next, we propose an efficient variant of the previous DSSE scheme, dubbed FVS2, that achieves optimized communication complexity of $O(m + u')$, where m is the size of the result set and u' is the number of update operations made on the searched keyword after the previous search made on the keyword. For keywords that are updated less frequently, this complexity essentially boils down to $O(m)$. This comes at the cost of some additional information leakage, which ensures that the construction achieves a weaker notion of backward privacy, called backward privacy with link pattern (BPLP). The forward privacy remains as it is. The client computations for search and update operations and the client storage remain the same as those of the basic scheme.

To the best of our knowledge, this is the first work that proposes fault-tolerant, verifiable DSSE constructions achieving both forward and backward privacy.

Road Map: We discuss all the notations, necessary background, and security definitions in Sect. 2. In Sect. 3, we propose our first construction FVS1. We provide a high-level overview and the formal specification followed by the security of the construction. We conclude the section by highlighting the limitations of the construction. In Sect. 4, we propose our second and main construction FVS2 that extends the previous construction to address the limitations of the previous scheme. We highlight how the modifications help us to obtain improved efficiency. We provide all security results of the construction. Next, in Sect. 5, we provide a detailed security analysis of our proposals. Finally, we conclude with some open research directions in Sect. 6.

1.4 A Comparative Study with Popular Verifiable DSSE Schemes

In this section, we provide a comparative study of our proposed schemes with state-of-the-art verifiable DSSE schemes in terms of efficiency (communication complexity, client computation, and client storage), security (forward and backward privacy), and fault tolerance. The comparison is summarized in Table 1. We use the shorthand notations FP, BP, and FT to denote forward privacy, backward privacy, and fault-tolerant. By 'Computation' and 'Communication', we mean the computational and communication complexity, respectively. We use the notation W to denote the number of distinct keywords in the database, $|D|$ to indicate the number of documents in the database, and N to represent the total number of keyword-document pairs in the database. Here ϵ is a fixed constant between 0 to 1. α is the number of times the queried keyword was historically added to the database and λ denotes the security parameter. Note that, every scheme except Kurosawa and Ohtaki [KO13], has server-side storage $O(N)$. For [KO13], the storage space is $O(|W| \cdot |D|)$. We would like to point out that a few SSE schemes achieve lower client-side storage as shown in the table. However, that comes at the cost of forward privacy or additional computation and communication costs during update operations. Exploring a DSSE scheme that achieves forward privacy with a constant update cost while maintaining lower client-side storage compared to our scheme would be an intriguing research direction.

Table 1: Comparison of Our Schemes with Existing Verifiable DSSE schemes.

Construction	Computation		Communication		Client Storage	FP	FT	BP
	Search	Update	Search	Update				
[KO13]	$O(\|D\|)$	$O(u\|D\|)$	$O(m)$	$O(1)$	$O(1)$	✗	✗	✗
GVS-Hash [BFP16]	$O(m + \log W)$	$O(u \cdot \log W)$	$O(m + \log W)$	$O(\log W)$	$O(1)$	✓	✗	✗
GVS-Acc-Pairing [BFP16]	$O(m)$	$O(uW^{\epsilon})$	$O(m + \log W)$	$O(\log W)$	$O(1)$	✓	✗	✗
GVS-Acc-RSA [BFP16]	$O(m + W^{\epsilon})$	$O(u)$	$O(m + \log W)$	$O(\log W)$	$O(1)$	✓	✗	✗
[ZWW+19]	$O(u)$	$O(1)$	$O(m)$	$O(1)$	$O(\|W\|\lambda)$	✓	✗	✗
Verifiable Linear SPS [BFP16]	$O(\alpha + \log N)$	$O(u \cdot \log^2 N)$	$O(m + \log N)$	$O(\log N)$	$O(\lambda \log N)$	✓	✓	✗
Verifiable Sublinear SPS [BFP16]	$O(m \cdot \log^3 N)$	$O(u \cdot \log^2 N)$	$O(m + \log N)$	$O(\log N)$	$O(\lambda \log N)$	✓	✓	✗
[YCR22]	$O(m + u_w)$	$O(1)$	$O(m + u_w)$	$O(1)$	$O(W \log \|D\|)$	✓	✓	✗
FVS1 (Sec. 3)	$O(u)$	$O(1)$	$O(u)$	$O(1)$	$O(W \log \|D\|)$	✓	✓	BPUP
FVS2 (Sec. 4)	$O(m + u')$	$O(1)$	$O(m + u')$	$O(1)$	$O(W \log \|D\|)$	✓	✓	BPLP

2 Preliminaries

For any natural number $n \in \mathbb{N}$, we write $[n]$ to denote the set $\{1, \ldots, n\}$. For a random variable X, we write $X \xleftarrow{\$} \mathcal{X}$ to denote that the random variable X is sampled uniformly at random from the set $\mathcal{X}$. The output x of a deterministic algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$. For two variables x, y, we write $x \leftarrow y$ to denote the assignment of the value in y to the variable x. We refer to $\lambda \in \mathbb{N}$ as the security parameter and denote by $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ as a generic polynomial function and negligible function of λ, respectively. For a protocol P between $\mathcal{A}$ and $\mathcal{B}$ input (resp. output) is separated by ';' signifies that the first part of the input (resp. output) is for participant $\mathcal{A}$ and second part is for participant $\mathcal{B}$.

Databases: Let $\mathcal{W} = \{w_1, \ldots, w_n\}$ be the set of all distinct keywords, which we call the *dictionary of keywords*. Let $\mathcal{F} = \{f_1, \ldots, f_s\}$ be a collection of files such that each file f_i is associated with an identifier id and it contains keywords from $\mathcal{W}$. Let $\mathcal{ID}$ be the set of all identifiers and $\mathsf{DB} = \{(w, id) : w \in \mathcal{W},\ id \in \mathcal{ID}\}$ be the set of all keyword-identifier pairs such that a given pair $(w, id) \in \mathsf{DB}$ if and only if the keyword w is in a file having the identifier id. For a given keyword $w \in \mathcal{W}$, $\mathsf{DB}(w)$ denotes the set of all file identifiers containing w as keywords. Let, $|\mathcal{W}|$ denote the number of keywords in DB. Similarly, $|\mathsf{DB}(w)|$ denotes the number of files containing the keyword w, $|\mathsf{DB}|$ denotes the number

of distinct keyword-identifier pairs and $|\mathsf{Upd}(w)|$ the number of times update operations have been carried out involving the keyword w.

2.1 Cryptographic Primitives

Pseudo-Random Function (PRF): Let $\mathsf{Func}(\{0,1\}^n)$ be the set of all functions from $\{0,1\}^n$ to $\{0,1\}^n$ and $\mathsf{F} : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a family of keyed functions from $\{0,1\}^n$ to $\{0,1\}^n$. The PRF advantage of F for a distinguisher $\mathcal{A}$ is defined as the difference between the following two probability distributions. The probability that $\mathcal{A}$ interacting with F_K for a randomly sampled secret key K and outputs 1, and the probability that $\mathcal{A}$ interacting with a function R chosen uniformly at random from $\mathsf{Func}(\{0,1\}^n)$, i.e.,

$$\mathbf{Adv}^{\mathrm{PRF}}_{\mathsf{F}}(\mathcal{A}) \triangleq \left| \Pr_K[\mathcal{A}^{\mathsf{F}_K} = 1] - \Pr_R[\mathcal{A}^R = 1] \right|.$$

We say that F is a PRF if the above advantage is negligible for any distinguisher $\mathcal{A}$. Similarly we can define PRP advantage for a function F as $\mathbf{Adv}^{\mathrm{PRP}}_{\mathsf{F}}(\mathcal{A})$ if the adversary can not distinguish it from a random permutation with more than negligible probability.

Hash Function: A hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^n$ is a function that takes a message of arbitrary length and produces a fixed length string. A hash function usually has three properties: (a) pre-image resistance, (b) second pre-image resistance and (c) collision resistance. A hash function H is said to *pre-image* resistant if for a given y, it is "difficult" to find an x such that $\mathsf{H}(x) = y$. A hash function H is said to *second pre-image* resistant if, for a given x and $\mathsf{H}(x)$, it is "difficult" to find an another $x' \neq x$ such that $\mathsf{H}(x') = \mathsf{H}(x)$. Finally, a hash function H is said to *collision* resistant if it is "difficult" to find a pair (x, x') with $x \neq x'$ such that $\mathsf{H}(x) = \mathsf{H}(x')$.

Symmetric-Key Encryption: A symmetric key encryption scheme SE is a tuple of polynomial time algorithms $\mathsf{SE.Gen}$, $\mathsf{SE.Enc}$, and $\mathsf{SE.Dec}$. The key generation algorithm $\mathsf{SE.Gen}$ takes the security parameter λ as input and outputs a secret key $K \in \mathcal{K}$, where $\mathcal{K}$ is the key space. The symmetric encryption algorithm $\mathsf{SE.Enc} : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^*$ takes a secret key $K \in \mathcal{K}$ and a plaintext $M \in \{0,1\}^*$ as input and outputs a ciphertext $C \in \{0,1\}^{|M|}$. Finally, $\mathsf{SE.Dec} : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^*$ is a deterministic algorithm that takes a secret key $K \in \mathcal{K}$ and a ciphertext $C \in \{0,1\}^*$ as input and outputs the decrypted plaintext $M \in \{0,1\}^{|C|}$ iff $\mathsf{SE.Enc}(K, M) = C$. A symmetric encryption scheme SE is said to have privacy, if it has PRF security i.e. no adversary can distinguish SE from a random function R of identical domain and range with non-negligible probability. Formally, we define *advantage* of $\mathcal{A}$ as:

$$\mathbf{Adv}^{\mathrm{Priv}}_{\mathsf{SE}}(\mathcal{A}) \triangleq \left| \Pr_K[\mathcal{A}^{\mathsf{SE.Enc}_K} = 1] - \Pr_R[\mathcal{A}^R = 1] \right|.$$

For SE to achieve privacy, the above advantage should be negligible for any adversary $\mathcal{A}$.

Authenticated Encrypion: An authenticated encryption AE is a tuple of algorithms $(\mathsf{AE.Enc}, \mathsf{AE.VDec})$. The authenticated encryption algorithm $\mathsf{AE.Enc} : \mathcal{K} \times \mathcal{N} \times \{0,1\}^\star \to \{0,1\}^\star$ is a function that takes a key $K \in \mathcal{K}$, a nonce $N \in \mathcal{N}$, and a message $M \in \{0,1\}^\star$ and returns a tagged ciphertext $C \in \{0,1\}^{|M|+\tau}$, where τ is the tag size. The corresponding verified decryption algorithm $\mathsf{AE.VDec} : \mathcal{K} \times \mathcal{N} \times \{0,1\}^\star \to \{0,1\}^\star \cup \{\perp\}$ is a function that takes a key $K \in \mathcal{K}$, a nonce $N \in \mathcal{N}$, and a tagged ciphertext $C \in \{0,1\}^\star$ and returns the corresponding message $M \in \{0,1\}^{|C|-\tau}$, if it is a valid tagged ciphertext. Otherwise, it returns $\perp$. An authenticated encryption algorithm AE demands the following two security requirements:

Privacy. Intuitively, an authenticated encryption scheme AE is said to have privacy,

if it has PRF security i.e. no adversary can distinguish $\mathcal{AE}$ from a random function R of identical domain and range with non-negligible probability. Formally, we define *privacy-advantage* of $\mathcal{A}$ as:

$$\mathbf{Adv}^{\mathrm{Priv}}_{\mathsf{AE}}(\mathcal{A}) \triangleq \big|\Pr_K[\mathcal{A}^{\mathsf{AE.Enc}_K} = 1] - \Pr_R[\mathcal{A}^R = 1]\big|.$$

For AE to achieve privacy, the above advantage should be negligible for any adversary $\mathcal{A}$.

Authenticity. Intuitively, AE is said to have authenticity (also called cipher-text integrity), if no adversary can forge, i.e., construct a fresh, valid (ciphertext, tag) pair with non-negligible probability. More formally, we define *authenticity-advantage* of $\mathcal{A}$ as:

$$\mathbf{Adv}^{\mathrm{Auth}}_{\mathsf{AE}}(\mathcal{A}) \triangleq \Pr_K[(N, C) \leftarrow \mathcal{A}^{\mathsf{AE.Enc}_K},\ \mathsf{AE.VDec}(K, N, C) \neq \perp].$$

We call AE achieves authenticity if the above advantage is negligible for any adversary $\mathcal{A}$.

2.2 Verifiable DSSE Scheme

An VDSSE scheme $\Sigma = (\mathsf{SetUp}, \mathsf{Search}, \mathsf{Update})$ consists of three protocols (possibly probabilistic) SetUp, Search, and Update between the client and the server. We adopted the following definition of FVDSSE from [YCR22].

$(K, \sigma_c; \mathsf{EDB}) \leftarrow \mathsf{SetUp}(1^\lambda, \mathsf{DB}; \perp)$: In the setup protocol, the client takes as input the security parameter 1^λ and the database DB and outputs a key K state σ_c for the client, which is kept secret, and an encrypted database EDB, which is sent to the server.

$((\sigma_c, \mathsf{DB}(w))$ **or** $\mathsf{Reject}; \mathsf{EDB}) \leftarrow \mathsf{Search}(K, \sigma_c, w; \mathsf{EDB})$: In the search protocol, the client takes as input the secret key K, the client's state σ_c, and the keyword w to be searched. The server takes as input the encrypted database EDB. Finally, the client outputs (possibly) an updated state σ_c and the server outputs (possibly) an updated encrypted database EDB. Now, if the search result returned by the server got verified, the client additionally outputs $\mathsf{DB}(w)$, otherwise outputs Reject.

$(\sigma_c; \mathsf{EDB}) \leftarrow \mathsf{Update}(K, \sigma_c, op, (w, id); \mathsf{EDB})$: In the update protocol, the client takes as input the secret key K, the client's state σ_c, an operation $op \in \{\mathsf{add}, \mathsf{del}\}$ and the keyword-identifier pair (w, id) to be updated. The server takes as input the encrypted database EDB. The client outputs an updated state σ_c and the server outputs an updated encrypted database EDB. An update protocol is always considered to be successful.

2.2.1 Correctness of Verifiable DSSE Scheme

Informally, a VDSSE scheme is said to be correct if, when the search result returned by the server for every searched keyword w is $\mathsf{DB}(w)$ and the result passes the verification on the client side except with negligible probability. For a formal definition, readers are requested to visit [BFP16].

2.2.2 Soundness of a Verifiable DSSE Scheme

Soundness guarantees that if the server is malicious, i.e., it returns an incorrect search result, the client can detect the server's malicious behavior. Formally, let us assume $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ denote a verifiable DSSE scheme. Now consider the following $\mathsf{VDSSESound}^{\Sigma}_{\mathcal{A}}(\lambda)$ game:

- $\mathcal{A}$ chooses a database DB and is provided with $\mathsf{EDB} \leftarrow \mathsf{Setup}(1^\lambda)$.

- The adversary adaptively makes search or update queries. For a search query, $\mathcal{A}$ chooses a keyword w and receives the output of $\Sigma.\mathsf{Search}(K, s, w)$. For an update query, it chooses (op, id, w) and is given the output of $\Sigma.\mathsf{Update}(K, s, op, id, w; \mathsf{EDB})$. Note that $\mathcal{A}$ 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 $w \in W$ is neither the set $\mathsf{DB}(w)$ nor the string "Reject".

We say Σ satisfies soundness if for all PPT adversaries $\mathcal{A}$, there exists a negligible function negl such that:

$$\mathbf{Adv}_{\Sigma}^{\mathsf{Sound}}(\mathcal{A}) \triangleq \Pr[\mathsf{VDSSESound}_{\mathcal{A}}^{\Sigma} = 1] \leq \mathsf{negl}(\lambda).$$

2.2.3 Security of a Verifiable DSSE Scheme

The security definition of SSE follows a read-world ideal-world simulation paradigm. The security model of a DSSE is parameterized by a well-defined set of leakage functions $\mathcal{L}$ that models the leakage of the original scheme. So the view of an adversary in the real world can be simulated with all the information given by $\mathcal{L}$. A DSSE scheme $\Sigma = (\mathsf{SetUp}, \mathsf{Search}, \mathsf{Update})$, is said to have the following leakage function given by

$$\mathcal{L} \triangleq \left(\mathcal{L}^{\mathsf{SetUp}}, \mathcal{L}^{\mathsf{Search}}, \mathcal{L}^{\mathsf{Update}}\right),$$

where $\mathcal{L}^{\mathsf{SetUp}}$, $\mathcal{L}^{\mathsf{Search}}$ and $\mathcal{L}^{\mathsf{Update}}$ denotes the information leaked to an adversarial server $\mathcal{A}$ during the setup, search, and update phases, respectively. The following defines the adaptive security of an SSE [KPR12, Bos16, CPS20].

Definition 1 ($\mathcal{L}$-semantic security)**.** Let, $\Sigma = (\mathsf{SetUp}, \mathsf{Search}, \mathsf{Update})$ be a DSSE scheme and $\mathcal{L} = \left(\mathcal{L}^{\mathsf{SetUp}}, \mathcal{L}^{\mathsf{Update}}, \mathcal{L}^{\mathsf{Search}}\right)$, be stateful functions, we say Σ is *$\mathcal{L}$-semantically secure* if for all probabilistic polynomial time, adaptive adversaries $\mathcal{A}$ there exists a probabilistic polynomial time simulator $\mathcal{S}$ such that

$$\left|\Pr\left[\mathsf{Real}_{\mathcal{A}}^{\Sigma}(\lambda) = 1\right] - \Pr\left[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}^{\Sigma}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda),$$

where the experiment Real and Ideal is defined as follows.

$\mathsf{Real}_{\mathcal{A}}^{\Sigma}(\lambda)$**:** The adversary $\mathcal{A}(1^{\lambda})$ outputs a database DB, and the experiment first computes an encrypted database EDB, where $(K, \sigma_c, \mathsf{EDB}) \leftarrow \Sigma.\mathsf{SetUp}\left(1^{\lambda}, \mathsf{DB}\right)$ and provides EDB to $\mathcal{A}$. Subsequently for every query of $\mathcal{A}$, the experiment returns either $(\mathsf{res}, \sigma_c, \mathsf{EDB}) \leftarrow \Sigma.\mathsf{Search}\left(K, \sigma_c, q; \mathsf{EDB}\right)$ if the query is a search query or, returns $(\sigma_c, \mathsf{EDB}) \leftarrow \Sigma.\mathsf{Update}\left(K, \sigma_c, q;\ \mathsf{EDB}\right)$ it q is an update query. Finally $\mathcal{A}$ outputs a single bit b which the experiment uses as its own output.

$\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}^{\Sigma}(\lambda)$**:** The adversary $\mathcal{A}(1^{\lambda})$ outputs a database DB, and the experiment first returns an encrypted database $\mathsf{EDB} \leftarrow \mathcal{S}(\mathcal{L}^{\mathsf{SetUp}}(\mathsf{DB}))$. Provides EDB to $\mathcal{A}$. Subsequently for every query of $\mathcal{A}$, the experiment returns either a transcript generated by $\mathcal{S}(\mathcal{L}^{\mathsf{Search}}(q))$ if the query is a search query or, $\mathcal{S}(\mathcal{L}^{\mathsf{Update}}(q))$ it q is an update query. Finally $\mathcal{A}$ outputs a single bit b which the experiment uses as its output.

We now define the forward and backward privacy of SSE.

2.3 Forward and Backward Privacy

So far we have informally introduced the notion of forward and backward privacy. Informally, forward privacy prevents linking previously performed search queries to future update queries, and backward privacy prevents revealing document identifiers containing a keyword w which has been removed from the database. The notion of forward and backward privacy was first introduced (informally) in [SPS14] and the first formal definition and a scheme achieving them were provided in [Bos16] and [BMO17], respectively. To formally define the notion of forward and backward privacy we first define a few common leakage functions.

Common Leakage Functions. DSSE is a protocol between the client and the server. The communication between them can be completely captured via a transcript of search and update queries, denoted as $\mathcal{Q}$. The transcript is a list of search and update queries written as (t, w) (for search query) and $(t, op, (w, id))$ (for update query), respectively, where t is the timestamp, $op \in \{\mathsf{add}, \mathsf{del}\}$. Now consider the following leakage functions.

- $\mathsf{sp}(w)$: The search pattern leakage for a keyword w is defined as all the timestamp where the keyword w as been searched for, i.e.,

$$\mathsf{sp}(w) = \{t : (t, w) \in \mathcal{Q}\}.$$

- $\mathsf{Hist}(w)$: The update history leakage of w contains the list of all updates made to w, i.e.,

$$\mathsf{Hist}(w) \;=\; \{(t, op, id) : (t, op, (w, id)) \in \mathcal{Q}\}.$$

- $\mathsf{Updates}(w)$ and $\mathsf{Updates}^{\mathsf{op}}(w)$: The leakage functions $\mathsf{Updates}(w)$ and $\mathsf{Updates}^{\mathsf{op}}(w)$ capture the timestamps and (timestamp, operation) pair of all the update history made for w. Formally,

$$\begin{aligned} \mathsf{Updates}(w) &= \{t : (t, op, (w, id)) \in \mathcal{Q}\}, \quad \text{and} \\ \mathsf{Updates}^{op}(w) &= \{(t, op) : (t, op, (w, id)) \in \mathcal{Q}\}. \end{aligned}$$

- $\mathsf{TimeDB}(w)$: For a keyword w, this leakage function stores the list of (timestamp, identifier) pairs such that the identifier id containing the keyword w was added but never deleted after that. Formally,

$$\mathsf{TimeDB}(w) \;=\; \{(t, id) : (t, add, (w, id)) \in \mathcal{Q} \text{ and } \forall\ t' > t,\ (t', del, (w, id)) \notin \mathcal{Q}\}.$$

- $\mathsf{DelHist}(w)$: The delete history leakage function contains the timestamps of all the deletion operations on w along with the timestamps of the corresponding additions. Formally,

$$\mathsf{DelHist}(w) \;=\; \{(t^+, t^-) : \exists\ id \text{ s.t. } (t^+, add, (w, id)) \in \mathcal{Q} \text{ and } (t^-, del, (w, id)) \in \mathcal{Q}\}.$$

Using these leakage functions, we now define the following notion of forward privacy [Bos16, CPS20]. For any two leakage function $\mathcal{L}_1$ and $\mathcal{L}_2$ we write $\mathcal{L}_1 \preceq \mathcal{L}_2$ to denote $\mathcal{L}_1$ leaks at most as $\mathcal{L}_2$ leaks, and by $\mathcal{L}_1 \prec \mathcal{L}_2$ we mean $\mathcal{L}_1$ leaks strictly less than $\mathcal{L}_2$ leaks.

Definition 2 (Forward Privacy)**.** A DSSE scheme Σ with leakage function $\mathcal{L} = \big(\mathcal{L}^{\mathsf{SetUp}}, \mathcal{L}^{\mathsf{Update}}, \mathcal{L}^{\mathsf{Search}}\big)$ is called *forward private* if the leakage function $\mathcal{L}$ can be written as follows.

$$\mathcal{L}^{\mathsf{SetUp}} \;=\; \emptyset, \quad \mathcal{L}^{\mathsf{Search}} \;\preceq\; \{\mathsf{sp}(w), \mathsf{Hist}(w)\}, \quad \mathcal{L}^{\mathsf{Update}} \;\preceq\; \{op\}.$$

Next, we focus on the backward privacy notions. In [BMO17], Bost et al. defined the following three types of backward privacy in increasing order of leakage and decreasing order of security.

(i) Backward Privacy with Insertion Pattern (BPIP) or Type-I,

(ii) Backward Privacy with Update Pattern (BPUP) or Type-II, and

(iii) Weak Backward Privacy (WBP) or Type-III.

Definition 3 (Backward Privacy)**.** A DSSE scheme Σ with leakage function $\mathcal{L}_{\mathsf{BPIP}} = \left(\mathcal{L}^{\mathsf{SetUp}}_{\mathsf{BPIP}}, \mathcal{L}^{\mathsf{Search}}_{\mathsf{BPIP}}, \mathcal{L}^{\mathsf{Update}}_{\mathsf{BPIP}}\right)$ (or $\mathcal{L}_{\mathsf{BPUP}}$, or $\mathcal{L}_{\mathsf{WBP}}$) is called BPIP (similarly BPUP, or WBP) backward private if the leakage function $\mathcal{L}_{\mathsf{BPIP}}$ (similarly $\mathcal{L}_{\mathsf{BPUP}}$ and $\mathcal{L}_{\mathsf{WBP}}$) can be written as follows.

$$\begin{array}{llll} \mathcal{L}^{\mathsf{SetUp}}_{\mathsf{BPIP}} = \emptyset, & \mathcal{L}^{\mathsf{Search}}_{\mathsf{BPIP}} \preceq \{\mathsf{sp}(w), \mathsf{TimeDB}(w), |\mathsf{Updates}(w)|\}, & \mathcal{L}^{\mathsf{Update}}_{\mathsf{BPIP}} \preceq \{op\}, \\ \mathcal{L}^{\mathsf{SetUp}}_{\mathsf{BPUP}} = \emptyset, & \mathcal{L}^{\mathsf{Search}}_{\mathsf{BPUP}} \preceq \{\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{Updates}(w)\}, & \mathcal{L}^{\mathsf{Update}}_{\mathsf{BPUP}} \preceq \{op, w\}, \\ \mathcal{L}^{\mathsf{SetUp}}_{\mathsf{WBP}} = \emptyset, & \mathcal{L}^{\mathsf{Search}}_{\mathsf{WBP}} \preceq \{\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{Updates}(w), \mathsf{DelHist}(w)\}, & \mathcal{L}^{\mathsf{Update}}_{\mathsf{WBP}} \preceq \{op, w\}. \end{array}$$

Note that the notion of weak backward privacy only applies to a re-insertion restriction setting, which is, once a (keyword, identifier) pair is removed, the same can not be re-inserted into the database. In [CPS20], Chatterjee et al. demonstrated that the leakage function of the above three definitions of backward privacy is not complete. Also the leakage function $\mathcal{L}^{\mathsf{SetUp}}_{\mathsf{WBP}}$ is not in a general non-restricted setting. Finally, Chatterjee et al. in [CPS20], introduced a new definition of backward privacy called Backward Privacy with Link Pattern (BPLP).

To understand this notion, we introduce some additional leakage functions. We use the notation $\mathsf{DB}_x(w)$ to denote the list of identifiers matching a search query on w at timestamp t_x. For two consecutive search queries on w at timestamp t_x and t_{x+1}, we define three link pattern leakage LP1, LP2 and LP3 as follows.

$$\begin{aligned} \mathsf{LP1}(w) &= \{(t, id) : id \in \mathsf{DB}_x(w) \text{ and } (t, op, (w, id)) \in \mathcal{Q};\ t_x < t < t_{x+1}\}, \\ \mathsf{LP2}(w) &= \{(t, id) : id \in \mathsf{DB}_{x+1}(w) \text{ and } (t, op, (w, id)) \in \mathcal{Q};\ t_x < t < t_{x+1}\}, \\ \mathsf{LP3}(w) &= \{(t, t') : (t, op, (w, id)) \in \mathcal{Q} \text{ and } (t', op, (w, id)) \in \mathcal{Q};\ t_x < t < t' < t_{x+1}\}. \end{aligned}$$

Finally, we define, LDB as the list of identifiers matching w for the current search in order of their insertions. With these leakages, they defined backward privacy with link pattern as follows.

Definition 4 (Backward privacy with link pattern (BPLP))**.** A DSSE scheme Σ with leakage $\mathcal{L}_{\mathsf{BPLP}} = \left(\mathcal{L}^{\mathsf{SetUp}}_{\mathsf{BPLP}}, \mathcal{L}^{\mathsf{Search}}_{\mathsf{BPLP}}, \mathcal{L}^{\mathsf{Update}}_{\mathsf{BPLP}}\right)$ is called *BPLP backward private* if the leakage function $\mathcal{L}$ can be written as following.

$$\begin{aligned} \mathcal{L}^{\mathsf{SetUp}}_{\mathsf{BPLP}} &= \emptyset, \quad \mathcal{L}^{\mathsf{Update}}_{\mathsf{BPLP}} = \emptyset, \\ \mathcal{L}^{\mathsf{Search}}_{\mathsf{BPLP}} &\preceq \{\mathsf{sp}(w), \mathsf{Updates}^{op}(w), \mathsf{LP1}(w), \mathsf{LP2}(w), \mathsf{LP3}(w), \mathsf{LDB}(w)\}. \end{aligned}$$

3 FVS1: The Basic Construction

In this section, we propose our first construction, called FVS1. Our construction ensures correctness at the same time is verifiable and fault-tolerant.

High-level Overview: The core idea is to record the update history securely on the server. Additionally, proof is generated for each update operation using an authenticated encryption scheme. During a search query, the server is expected to return the proofs corresponding to all the updates on the searched keyword. The client stores the number of updates corresponding to each keyword in its local memory. The server might attempt to tamper with the result by omitting some proofs, sending random values, or even adding extra proofs. In any such case, there will either be a mismatch between the number of proofs and the number of updates on the keyword, or some of the verified decryption algorithms will fail on the client side. Consequently, our scheme, $\mathsf{FVS1}$, is verifiable against an active malicious adversary. Regarding the fault tolerance aspect, an incorrect update query on keyword w depends on the sequence of updates on w. Both the server and client maintain this order when storing or retrieving the values generated from the updates. An index id is added to the Result ID set corresponding to a keyword w only once. Moreover, the server checks updates in reverse order, ensuring that id is added to the Result ID set before it becomes part of the set Delete ID set. This ensures situations such as deleting a keyword from an identifier before adding it to the identifier are handled properly.

3.1 Specification

Here, we describe the setup, search (also verify), and update functionality of $\mathsf{FVS1}$. The construction uses two data structures TSet and RSet. The data structure TSet stores necessary information about each update on w and RSet stores the result sets corresponding to a keyword w. The construction also keeps two counters ver_w and upd_w corresponding to each keyword w that represents the total number of search operations performed on the current searched keyword w before that instance and the number of update operations performed on w after the previous search query (on w), respectively. The client stores these two counters in a $\sigma_c[w]$ state array. We have used five PRFs F_S, F_T, F_V, F_R, and F_P with domain $\{0,1\}^\lambda \times \{0,1\}^*$ and range $\{0,1\}^*$, two hash functions H_1, H_2 with domain $\{0,1\}^*$ and range $\{0,1\}^{2\lambda}$. Also, we have used an authenticated encryption scheme AE with $\mathsf{AE.Enc} : \{0,1\}^\lambda \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ and $\mathsf{AE.Dec} : \{0,1\}^\lambda \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ and finally a symmetric encryption scheme SE with $\mathsf{SE.Enc} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$ and $\mathsf{SE.Dec} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$.

Setup: In the setup phase, all the keys are chosen uniformly at random from $\{0,1\}^\lambda$, TSet, RSet data structures are initialized as empty and the encrypted database EDB is defined as $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{RSet})$. The state array σ_c indexed by the keyword $w \in \mathcal{W}$ is also initialized to $\perp$.

Algorithm 1 : $\mathsf{FVS1.SetUp}(1^\lambda, \perp)$

1: $K_S, K_T, K_V, K_R, K_P \xleftarrow{\$} \{0,1\}^\lambda$
2: $\sigma_c, \mathsf{TSet}, \mathsf{RSet} \leftarrow []$
3: $\mathsf{K} \leftarrow (K_S, K_T, K_V, K_R, K_P)$
4: $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{RSet})$
5: Store (K, σ_c) at Client
6: Send EDB to Server

Update: For any update query $(op, (w, id))$, the client generates a search token st uniformly at random and generates a *value* corresponding to the update by masking the PRF evaluation of the current update query with the op, w, and id involved in the update. In addition, it generates a *link* that will interconnect the current update with the previous update on this same keyword following the idea of [Bos16, SDY+20]. This is done in such a manner that once the server is provided with the search token and the desired hash key, it will be able to traverse back on all the updates on w but will not be able to guess any

future update. This, by definition, guarantees the forward privacy. Along with the link and the value, a proof is generated to correspond to the update operation. The proof is a ciphertext of a secure authenticated encryption scheme that uses a (keyword, version) dependent key, $op\|id$ as input and upd_w as the nonce. The $value$, $proof$, and $link$ are then sent to the server, which stores these in the TSet array. An algorithmic description of the update operation of our proposed construction $\mathsf{FVS1}$ is given in Algorithm 2.

Algorithm 2 : $\mathsf{FVS1.Update}(\mathsf{K}, \sigma_c, op, w, id; \mathsf{EDB})$

function Client
- $\mathsf{K} = (K_S, K_T, K_V, K_R, K_P)$
- $(\mathsf{ver}_w, \mathsf{upd}_w) \leftarrow \sigma_c[w]$
- **if** $(\mathsf{ver}_w, \mathsf{upd}_w) = \perp$ **then**
  - $\mathsf{ver}_w \leftarrow 0, \mathsf{upd}_w \leftarrow 0$
- $\mathsf{upd}_w \leftarrow \mathsf{upd}_w + 1$
- $s_w \leftarrow F_S(K_S, w)$
- $st \leftarrow F_T(K_T, w\|\mathsf{upd}_w\|\mathsf{ver}_w)$
- $value \leftarrow F_V(K_V, st) \oplus op\|id$
- **if** $\mathsf{upd}_w = 1$ **then**
  - $pst \leftarrow \{0,1\}^\lambda$
- **else**
  - $pst \leftarrow F_T(K_T, w\|\mathsf{upd}_w - 1\|\mathsf{ver}_w)$
- $addr \leftarrow H_1(s_w, st)$
- $link \leftarrow H_2(s_w, st) \oplus pst$
- $p_w \leftarrow F_P(K_P, w\|\mathsf{ver}_w)$
- $proof \leftarrow \mathsf{AE.Enc}(p_w, \mathsf{upd}_w, op\|id)$
- $\sigma_c[w] \leftarrow (\mathsf{ver}_w, \mathsf{upd}_w)$
- Send $(addr, value, link, proof)$ to Server

function Server
- Parse EDB as $(\mathsf{TSet}, \mathsf{RSet})$
- $\mathsf{TSet}[addr] \leftarrow (value, link, proof)$

Search: Our search protocol is a two-round protocol. In the first phase of the search operation with the keyword w, the client computes the current search token st from the state array $\sigma_c[w]$ and sends it to the server. We follow the approach of [SDY+20, CPS20] and store the search results in a separate data structure RSet. So, if the search is not the first search with w, an extra $addr$ value is sent to the server along with st. This is to enable the server to learn about the encrypted identifiers from the previous search. With the st value the server gets back the corresponding $value$, $proof$ and $link$ from TSet. The $value$ and $proof$ are then stored sequentially in Val and Proof array respectively whereas, the $link$ is used to get back the previous search token pst. This process continues until the server reaches the first update. Also, if there is a result stored from the previous search query in RSet, the corresponding $value$ and $proof$ are retrieved and are then put in $\mathsf{Val}[0]$ and $\mathsf{Proof}[0]$ respectively. The Val and Proof arrays are then sent to the client. This concludes the first round of the search operation. An algorithmic description of this round is given in Algorithm 3. In the second round of the search query, the client first unmasks the information about the op and id values for each entry of Val array starting from the end of the array. This is to take care of any faulty operation during updates. According to the op the client decides whether to put the corresponding id in the the result set Rid or not. For the encrypted identifiers in $\mathsf{Val}[0]$, first they are decrypted, and the corresponding ids are again added to the result set if they are not deleted between the two latest searches. Now the client runs the verification algorithm on Rid and Proof. By security of AE, the client becomes confident about the server's behavior once the $\mathsf{AE.VDec}$ on any $\mathsf{Proof}[i]$ with nonce i returns a valid message of the form $op\|id$. This way, another Rid_P is prepared from the Proof array. Only if Rid matches with Rid_P the client then outputs Rid as the search result. It also encrypts Rid along with a new proof corresponding to Rid and sends

it to the server. The server stores the encrypted result set and the corresponding proof in RSet. An algorithmic description of this round is given in Algorithm 4.

Algorithm 3 : FVS1.Search $(\mathsf{K}, \sigma_c, w; \mathsf{EDB})$ - Round 1

function Client
- $\mathsf{K} = (K_S, K_T, K_V, K_R, K_P)$
- $(\mathsf{ver}_w, \mathsf{upd}_w) \leftarrow \sigma_c[w]$
- $s_w, st \leftarrow \perp$
- $addr_w \leftarrow F_R(K_R, w)$
- **if** $(\mathsf{ver}_w, \mathsf{upd}_w) = \perp$ **then**
  - **return** $\emptyset$
- **if** $\mathsf{upd}_w \neq 0$ **then**
  - $s_w \leftarrow F_S(K_S, w)$
  - $st \leftarrow F_T(K_T, w \| \mathsf{upd}_w \| \mathsf{ver}_w)$
- Send $(addr_w, st, s_w, \sigma_c[w])$ to Server.

function Server
- $\mathsf{Proof}, \mathsf{Val} \leftarrow$ empty list
- $\mathsf{RAddr}, \mathsf{Pval} \leftarrow \emptyset$
- $(\mathsf{ver}_w, \mathsf{upd}_w) \leftarrow \sigma_c[w]$
- **for** $i = \mathsf{upd}_w$ *down to* 1 **do**
  - $addr \leftarrow H_1(s_w, st)$
  - $\mathsf{RAddr} \leftarrow \mathsf{RAddr} \cup \{addr\}$
  - $(value, link, proof) \leftarrow \mathsf{TSet}[addr]$
  - $\mathsf{Proof}[i] \leftarrow proof$
  - $\mathsf{Val}[i] \leftarrow value$
  - $st \leftarrow link \oplus H_2(s_w, st)$
- **if** $\mathsf{ver}_w > 0$ **then**
  - $(eRid, proof) \leftarrow \mathsf{RSet}[addr_w]$
  - $\mathsf{Proof}[0] \leftarrow proof$
  - $\mathsf{Val}[0] \leftarrow eRid$
- **for** $addr \in \mathsf{RAddr}$ **do**
  - Delete $\mathsf{TSet}[addr]$
- Send $(\mathsf{Val}, \mathsf{Proof})$ to Client.

3.2 Security Results

In this section, we describe the leakage profile of FVS1 and then state the privacy and soundness security results of FVS1. The proof of the Theorems is deferred to Sect. 5.

3.2.1 Privacy of FVS1

We first define the leakage profile of FVS1, denoted as $\mathcal{L}_{\mathsf{FVS1}}$ as given below.

$$\mathcal{L}_{\mathsf{FVS1}} = \left(\mathcal{L}^{\mathsf{SetUp}}_{\mathsf{FVS1}}, \mathcal{L}^{\mathsf{Update}}_{\mathsf{FVS1}}, \mathcal{L}^{\mathsf{Search}}_{\mathsf{FVS1}}\right), \text{ where}$$

$$\mathcal{L}^{\mathsf{SetUp}}_{\mathsf{FVS1}} = \emptyset, \quad \mathcal{L}^{\mathsf{Update}}_{\mathsf{FVS1}} = \emptyset, \quad \mathcal{L}^{\mathsf{Search}}_{\mathsf{FVS1}} \preceq \{\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{Updates}(w)\}.$$

We want to highlight that as per the forward and backward privacy definitions stated in Sect. 2.3, the above leakage profile ensures that our construction FVS1 achieves forward privacy and backward privacy with update pattern (BPUP).

For the above-mentioned leakage profile, in the following, we state the security theorem for our FVS1 construction as:

Theorem 1. *Let F_S, F_T, F_V, F_R, and F_P be independent and secure PRFs, H_1, and H_2 be hash functions modeled as random oracles, SE be a secure symmetric encryption scheme and AE be a secure authenticated encryption scheme. Then, our proposed construction FVS1 is L_{FVS1}-adaptively secure, achieves forward and backward privacy with update pattern (BPUP). Formally, we have:*

$$\begin{aligned}
&|\Pr[\mathit{FVS1}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \\
&\leq \mathbf{Adv}^{PRF}_{F_S}(\mathcal{A}_1) + \mathbf{Adv}^{PRF}_{F_T}(\mathcal{A}_2) + \mathbf{Adv}^{PRF}_{F_V}(\mathcal{A}_3) + \mathbf{Adv}^{PRF}_{F_R}((\mathcal{A}_4) + \mathbf{Adv}^{PRF}_{F_P}(\mathcal{A}_5) + \\
&\quad \mathbf{Adv}^{Priv}_{\mathit{SE}}(\mathcal{B}_1) + \mathbf{Adv}^{\mathrm{Priv}}_{\mathit{AE}}(\mathcal{B}_2) + \mathbf{Adv}^{\mathrm{Auth}}_{\mathit{AE}}(\mathcal{B}_3) + \frac{2q^2}{2^{2\lambda}} + \frac{q^2}{2^s}.
\end{aligned}$$

Algorithm 4 : FVS1.Search(K, σ_c, w; EDB) - Round 2

function Client
- $\mathsf{K} = (K_S, K_T, K_V, K_R, K_P)$
- $\mathsf{Rid}, \mathsf{Del}, \mathsf{PrvRes} \leftarrow \emptyset$
- $(\mathsf{ver}_w, \mathsf{upd}_w) \leftarrow \sigma_c[w]$
- **for** $i = \mathsf{upd}_w$ *down to* 1 **do**
  - $op \| id \leftarrow F_V(K_V, st) \oplus \mathsf{Val}[i]$
  - **if** $op = add$ *and* $id \notin \mathsf{Del}$ **then**
    - $\mathsf{Rid} \leftarrow \mathsf{Rid} \cup \{id\}$
  - **if** $op = del$ **then**
    - $\mathsf{Del} \leftarrow \mathsf{Del} \cup \{id\}$
- **if** $\mathsf{ver}_w \neq 0$ **then**
  - $R_K \leftarrow F_T(K_T, w \| 0 \| \mathsf{ver}_w + 1)$
  - $\mathsf{PrvRes} \leftarrow \mathsf{SE.Dec}(R_K, \mathsf{Val}[0])$
  - **for** *each* $id \in \mathsf{PrvRes}$ **do**
    - **if** $id \notin \mathsf{Del}$ **then**
      - $\mathsf{Rid} \leftarrow \mathsf{Rid} \cup \{id\}$
- $v \leftarrow \mathsf{Verify}(\mathsf{Rid}, \mathsf{Proof}, \mathsf{K}, \sigma_c, w)$
- **if** $v = 0$ **then**
  - **return** Reject
- **else**
  - $\mathsf{ver}_w \leftarrow \mathsf{ver}_w + 1$, *and* $\mathsf{upd}_w \leftarrow 0$
  - $R_K \leftarrow F_T(K_T, w \| \mathsf{upd}_w \| \mathsf{ver}_w)$
  - $eRid \leftarrow \mathsf{SE.Enc}(R_K, \mathsf{Rid})$
  - $p_w \leftarrow F_P(K_P, w \| \mathsf{ver}_w)$
  - $proof \leftarrow \mathsf{AE.Enc}(p_w, \mathsf{upd}_w, \mathsf{Rid})$
  - $\sigma_c(w) \leftarrow (\mathsf{ver}_w, \mathsf{upd}_w)$
  - Send $eRid$, $proof$ to Server.
  - **return** Rid

function Server
- $\mathsf{RSet}[addr_w] \leftarrow (eRid, proof)$

function Verify
- $\mathsf{K} = (K_S, K_T, K_V, K_R, K_P)$
- $\mathsf{Rid}_P \leftarrow \emptyset$
- $(\mathsf{ver}_w, \mathsf{upd}_w) \leftarrow \sigma_c[w]$
- **if** $\mathsf{ver}_w = 0$ **then**
  - **if** $|\mathsf{Proof}| \neq \mathsf{upd}_w$ **then**
    - **return** 0
- **else**
  - **if** $|\mathsf{Proof}| \neq \mathsf{upd}_w + 1$ **then**
    - **return** 0
- **if** $\mathsf{ver}_w > 1$ **then**
  - $p_w \leftarrow F_3(K_P, w \| \mathsf{ver}_w - 1)$
  - $r \leftarrow \mathsf{AE.VD}(p_w, 0, \mathsf{Proof}[0])$
  - **if** $r = \perp$ **then**
    - **return** 0
  - **else**
    - $\mathsf{Rid}_P \leftarrow \mathsf{Rid}_P \cup \{r\}$
- $p_w \leftarrow F_3(K_P, w \| \mathsf{ver}_w)$
- **for** $i = 1$ *to* upd_w **do**
  - $r \leftarrow \mathsf{AE.VD}(p_w, i, \mathsf{Proof}[i])$
  - **if** $r = \perp$ **then**
    - **return** 0
  - **else**
    - $op \| id \leftarrow r$
    - **if** $op = add$ **then**
      - $\mathsf{Rid}_P \leftarrow \mathsf{Rid}_P \cup \{id\}$
    - **else**
      - $\mathsf{Rid}_P \leftarrow \mathsf{Rid}_P \setminus \{id\}$
- **if** $\mathsf{Rid} = \mathsf{Rid}_P$ **then**
  - **return** 1
- **else**
  - **return** 0

Remark 1. We have modeled cryptographic primitives in both constructions using PRFs and random oracles. While PRFs (or PRPs) are a standard assumption in symmetric key modes, the random oracle assumption is a stronger one. However, note that almost all the state-of-the-art schemes (including all the DSSE schemes mentioned in Table 1) use this assumption to argue for security. One possible solution to mitigate this is to replace the hash (which is assumed to be a random oracle) with a PRF in the same manner as shown in [SPS14]. However, in that case, the client needs to provide more tokens during a search operation that requires additional client computation and communication costs up to $O(u')$.

3.2.2 Soundness of FVS1

In the following, we state the soundness theorem for FVS1 as follows:

Theorem 2. *If AE satisfies authenticity as defined in section 2.1, then our proposed scheme FVS1 achieves soundness even in the presence of faulty update queries. Formally we have*

$$\Pr[\mathit{VDSSESound}_{\mathcal{A}}^{\mathit{FVS1}}(\lambda)] \leq \mathbf{Adv}_{\mathit{AE}}^{\mathit{Auth}}(\mathcal{A}_1) + \mathbf{Adv}_{\mathit{FVS1}}^{\mathit{Correct}}(\mathcal{A}_2).$$

3.3 Limitations of FVS1

Our proposed construction is highly efficient from the leakage point of view - it reveals very little information to the adversarial server during update and search queries. Specifically, the update leakage is nonexistent, and the only leakages related to the search queries are the timestamps of all update and search queries on the search keyword, $\mathsf{Updates}(w)$ and $\mathsf{sp}(w)$. However, the communication complexity of the search algorithm is proportional to the number of update operations on w because the server sends all masked (op, id) pairs related to keyword w to the client for further processing. Since the client must process each of these ciphertexts to obtain the actual result set, the computation complexity on the client side is similarly high. This limits the practicality of our FVS1 scheme and motivates us to a modified construction that achieves better communication complexity.

4 FVS2: An Efficient Fault-Tolerant Verifiable DSSE

The objective of this work is to design a DSSE scheme that achieves optimal communication complexity while ensuring forward and backward privacy. In this section, we present an updated version of the previous scheme, dubbed FVS2, which meets these goals. We begin with a high-level overview of the modifications and then provide a formal description of all the algorithms. Following this, we outline the security results, with detailed proofs deferred to Sect. 5.

4.1 Specification

Our goal is to achieve optimal communication and client-side computation during search operations. We need the server to provide information corresponding precisely to the set of documents currently matching the searched keyword w. To achieve this, we have assigned a tag to each updated identifier with the following property: between two consecutive search queries on w, regardless of the underlying update op, the same tag will be generated for all updates involving w, and id. Additionally, during a search query, the server is delegated to learn about the actual update op. This enables the server to compute the set of tags belonging to the result set independently. Consequently, both the communication cost and client-side computation cost are reduced to optimal levels while maintaining backward privacy. The algorithmic descriptions of the setup and update procedures for FVS2 are presented in Algorithms 5 and 6, respectively. The search algorithm is divided into two parts - Round 1 and Round 2 - with the corresponding algorithms provided in Algorithm 7 and Algorithm 8, respectively.

Algorithm 5 : FVS2.Setup$(1^\lambda, \perp)$

1: $K_S, K_T, K_R, K_P, K_G \xleftarrow{\$} \{0,1\}^\lambda$
2: $\sigma_c, \mathsf{TSet}, \mathsf{RSet} \leftarrow \emptyset$
3: $\mathsf{K} \leftarrow (K_S, K_T, K_R, K_P, K_G)$
4: $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{RSet})$
5: Store (K, σ_c) at Client
6: Send EDB to Server

4.2 Security Results

In this subsection, we describe the leakage profile of FVS2 and then state the privacy and soundness security results of FVS2. The proof of the Theorems is deferred to Sect. 5.

Algorithm 6 : FVS2.Update $(\mathsf{K}, \sigma_c, op, w, id; \mathsf{EDB})$

function Client
- $\mathsf{K} = (K_S, K_T, K_G, K_R, K_P)$
- $(\mathsf{ver}_w, \mathsf{upd}_w) \leftarrow \sigma_c[w]$
- **if** $(\mathsf{ver}_w, \mathsf{upd}_w) = \perp$ **then**
  - $\mathsf{ver}_w \leftarrow 0, \mathsf{upd}_w \leftarrow 0$
- $\mathsf{upd}_w \leftarrow \mathsf{upd}_w + 1$
- $s_w \leftarrow F_S(K_S, w)$
- $st \leftarrow F_T(K_T, w \| \mathsf{upd}_w \| \mathsf{ver}_w)$
- $addr \leftarrow H_1(s_w, st)$
- $tag \leftarrow G(K_G, w \| id \| \mathsf{ver}_w)$
- **if** $\mathsf{upd}_w = 1$ **then**
  - $value \leftarrow H_2(s_w, st) \oplus 0^\lambda \| op \| tag$
- **else**
  - $pst \leftarrow F_T(K_T, w \| \mathsf{upd}_w - 1 \| \mathsf{ver}_w)$
  - $value \leftarrow H_2(s_w, st) \oplus pst \| op \| tag$
- $p_w \leftarrow F_P(K_P, w \| \mathsf{ver}_w)$
- $proof \leftarrow \mathsf{AE.Enc}(p_w, \mathsf{upd}_w, op \| id)$
- $\sigma_c \leftarrow (\mathsf{ver}_w, \mathsf{upd}_w)$
- Send $(addr, value, proof_u)$ to Server

function Server
- Parse EDB as $(\mathsf{TSet}, \mathsf{RSet})$
- $\mathsf{TSet}[addr] \leftarrow (value, proof)$

Algorithm 7 : FVS2.Search$(\mathsf{K}, \sigma_c, w;$ EDB) - Round I

function Client
- $\mathsf{K} = (K_S, K_T, K_G, K_R, K_P)$
- $(\mathsf{ver}_w, \mathsf{upd}_w) \leftarrow \sigma_c[w]$
- **if** $(\mathsf{ver}_w, \mathsf{upd}_w) = \perp$ **then**
  - **return** ϕ
- $addr_w \leftarrow F_R(K_R, w)$
- $s_w \leftarrow F_S(K_S, w)$
- $st \leftarrow F_T(K_T, w \| \mathsf{upd}_w \| \mathsf{ver}_w)$
- Send $(addr_w, st, s_w, \sigma_c[w])$ to Server.

function Server
- Proof $\leftarrow$ empty list
- RAddr, TagSet, DelTag, PrvTag $\leftarrow \emptyset$
- **for** $i = \mathsf{upd}_w$ *down to* 1 **do**
  - $addr \leftarrow H_1(s_w, st)$
  - RAddr $\leftarrow$ RAddr $\cup \{addr\}$
  - $(value, proof) \leftarrow \mathsf{TSet}[addr]$
  - Proof$[i] \leftarrow proof$
  - $st \| op \| tag \leftarrow value \oplus H_2(s_w, st)$
  - **if** $op = add$ *and* $tag \notin$ DelTag **then**
    - TagSet $\leftarrow$ TagSet $\cup \{tag\}$
  - **if** $op = del$ **then**
    - DelTag $\leftarrow$ DelTag $\cup \{tag\}$
- **if** $\mathsf{ver}_w > 0$ **then**
  - $(\mathsf{PrvTag}, proof) \leftarrow \mathsf{RSet}[addr_w]$
  - Proof$[0] \leftarrow proof$
  - **for** *each* $tag \in$ PrvTag **do**
    - **if** $tag \notin$ DelTag **then**
      - TagSet $\leftarrow$ TagSet $\cup \{tag\}$
- **for** *each* $addr \in$ RAddr **do**
  - Delete $\mathsf{TSet}[addr]$
- Send (TagSet, Proof) to Client.

4.2.1 Privacy of FVS2

The leakage profile of our FVS2 construction is defined as a stateful function $\mathcal{L}_{\mathsf{FVS2}}$, defined as:

$$\mathcal{L}_{\mathsf{FVS2}} = \left(\mathcal{L}^{\mathsf{SetUp}}_{\mathsf{FVS2}}, \mathcal{L}^{\mathsf{Update}}_{\mathsf{FVS2}}, \mathcal{L}^{\mathsf{Search}}_{\mathsf{FVS2}}\right),$$

Algorithm 8 : FVS2.Search(K, σ_c, w; EDB) - Round II

function Client
- $\mathsf{K} = (K_S, K_T, K_G, K_R, K_P)$
- $\mathsf{Rid}, \mathsf{PrvTag} \leftarrow \emptyset$
- **for** *each* $tag \in \mathsf{TagSet}$ **do**
  - $w \| id \| tag \leftarrow G^{-1}(K_G, tag)$
  - $\mathsf{Rid} \leftarrow \mathsf{Rid} \cup \{id\}$
- $v \leftarrow \mathsf{Verify}(\mathsf{Rid}, \mathsf{P}, \mathsf{K}, st, w)$
- **if** $v = 0$ **then**
  - **return** Reject
- **else**
  - $(\mathsf{ver}_w, \mathsf{upd}_w) \leftarrow \sigma_c[w]$
  - $\mathsf{ver}_w \leftarrow \mathsf{ver}_w + 1$
  - $\mathsf{upd}_w \leftarrow 0$
  - **for** *each* $tag \in \mathsf{TagSet}$ **do**
    - $tag \leftarrow G(K_G, w \| id \| \mathsf{ver}_w)$
    - $\mathsf{PrvTag} \leftarrow \mathsf{PrvTag} \cup \{tag\}$
  - $p_w \leftarrow F_P(K_P, w \| \mathsf{ver}_w)$
  - $proof \leftarrow \mathsf{AE.Enc}(p_w, \mathsf{upd}_w, \mathsf{Rid})$
  - Send $\mathsf{PrvTag}, proof$ to Server.
  - **return** Rid

function Server
- $\mathsf{RSet}[\mathsf{addrw}] \leftarrow (\mathsf{PrvTag}, proof)$

function Verify
- $\mathsf{K} = (K_S, K_T, K_G, K_R, K_P)$
- $\mathsf{Rid}_P \leftarrow \emptyset$
- $(\mathsf{ver}_w, \mathsf{upd}_w) \leftarrow \sigma_c[w]$
- **if** $\mathsf{ver}_w = 0$ **then**
  - **if** $|\mathsf{Proof}| \neq \mathsf{upd}_w$ **then**
    - **return** 0
- **else**
  - **if** $|\mathsf{Proof}| \neq \mathsf{upd}_w + 1$ **then**
    - **return** 0
- **if** $\mathsf{ver}_w > 1$ **then**
  - $p_w \leftarrow F_3(K_P, w \| \mathsf{ver}_w - 1)$
  - $r \leftarrow \mathsf{AE.VDec}(p_w, 0, \mathsf{Proof}[0])$
  - **if** $r = \perp$ **then**
    - **return** 0
  - **else**
    - $\mathsf{Rid}_P \leftarrow \mathsf{Rid}_P \cup r$
- $p_w \leftarrow F_3(K_P, w \| \mathsf{ver}_w)$
- **for** $i = 1$ *to* upd_w **do**
  - $r \leftarrow \mathsf{AE.VDec}(p_w, i, \mathsf{Proof}[i])$
  - **if** $r = \perp$ **then**
    - **return** 0
  - **else**
    - $op \| id \leftarrow r$
    - **if** $op = add$ **then**
      - $\mathsf{Rid}_P \leftarrow \mathsf{Rid}_P \cup \{id\}$
    - **else**
      - $\mathsf{Rid}_P \leftarrow \mathsf{Rid}_P \setminus \{id\}$
- **if** $\mathsf{Rid} = \mathsf{Rid}_P$ **then**
  - **return** 1
- **else**
  - **return** 0

where

$$\begin{aligned} \mathcal{L}^{\mathsf{SetUp}}_{\mathsf{FVS2}}() &= \mathcal{L}^{\mathsf{Update}}_{\mathsf{FVS2}}(op, (w, id)) = \emptyset, \\ \mathcal{L}^{\mathsf{Search}}_{\mathsf{FVS2}}(w) &\preceq \{\mathsf{sp}(w),\ \mathsf{Updates}^{op}(w),\ \mathsf{LP1}(w),\ \mathsf{LP2}(w),\ \mathsf{LP3}(w),\ \mathsf{LDB}(w)\}. \end{aligned}$$

We justify the leakage profile of FVS2 in Sect. A. Note that as per the definitions of forward and backward privacy stated in Sect. 2.3, the above leakage profile ensures that our construction FVS2 achieves forward privacy and backward privacy with link pattern (BPLP).

For the above-mentioned leakage profile, in the following, we state the main security theorem for our FVS2 construction as:

Theorem 3. *Let F_S, F_T, F_R and F_P be independent and secure PRFs, G be a secure PRP, H_1, H_2 be hash functions modelled as random oracles, and AE be a secure authenticated encryption scheme. Then, our proposed construction FVS2 is $\mathcal{L}_{FVS2}$-adaptively secure, achieves forward privacy and backward privacy with link pattern (BPLP). Formally, we*

have:

$$\begin{aligned}&|\Pr[\mathit{FVS2}_{\mathcal{A}}(\lambda)=1]-\Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)=1]|\\ &\quad\leq\ \mathbf{Adv}^{PRF}_{F_S}(\mathcal{A}_1)+\mathbf{Adv}^{PRF}_{F_T}(\mathcal{A}_2)+\mathbf{Adv}^{PRF}_{F_R}(\mathcal{A}_3)+\mathbf{Adv}^{PRF}_{F_P}(\mathcal{A}_4)+\mathbf{Adv}^{PRP}_{G}(\mathcal{A}_5)+\\ &\qquad\mathbf{Adv}^{\mathrm{Priv}}_{\mathit{AE}}(\mathcal{B}_1)+\mathbf{Adv}^{\mathrm{Auth}}_{\mathit{AE}}(\mathcal{B}_2)+\frac{2q^2}{2^{2\lambda}}.\end{aligned}$$

4.2.2 Soundness of FVS2

Theorem 4. *If AE satisfies authenticity as defined in section 2.1, then FVS2 achieves soundness even in the presence of faulty update queries. Formally, $\mathcal{A}_2$, such that*

$$\Pr[\mathit{VDSSESound}^{\mathit{FVS2}}_{\mathcal{A}}(\lambda)]\ \leq\ \mathbf{Adv}^{\mathrm{Auth}}_{\mathit{AE}}(\mathcal{A}_1)+\mathbf{Adv}^{\mathrm{Correct}}_{\mathit{FVS2}}(\mathcal{A}_2).$$

5 Security Analysis

In this section, we prove all the theorems.

5.1 Proof of Theorem 1 and Theorem 3

In this section, we first rigorously prove Theorem 3 and then provide a sketch of Theorem 1 that follows from the previous theorem. To prove Theorem 3, we follow the standard approach, as used in [CJJ+13, CGKO06], and define a sequence of games between the challenger $\mathcal{C}$ and the adversary $\mathcal{A}$. The first game (i.e., Game G0) computes a distribution identical to the real experiment $\mathsf{FVS2}_{\mathcal{A}}(\lambda)$ and the final game (i.e., Game G5) considers a simulator $\mathcal{S}$ that perfectly simulates a distribution identical to the ideal experiment $\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$, given the leakage function of FVS2. As per the basic requirement for security, we have shown that the design of the games is done in such a manner that the views of the adversary $\mathcal{A}$ in each pair of consecutive games are computationally indistinguishable.
Game G0: This game is identical to the real experiment $\mathsf{FVS2}_{\mathcal{A}}(\lambda)$. The challenger generates the transcript corresponding to each update and search query using Algorithms 6, 7, 8 for FVS2.
Game G1: This game is identical to Game G0 except that response to every call to the PRFs F_S, F_T, F_R and F_P are generated using tables $\mathsf{Tab}_S, \mathsf{Tab}_T, \mathsf{Tab}_R$ and Tab_P, respectively. The table access is done following the lazy sampling [Zha19] technique: when an entry in a table is first accessed it is chosen at random and the chosen value is used for further access. Since all of F_S, F_T, F_R, and F_P are secure PRFs, it is easy to see that:

$$\big|\Pr[\mathsf{G1}=1]-\Pr[\mathsf{G0}=1]\big|\ \leq\ \mathbf{Adv}^{\mathrm{PRF}}_{F_S}(\mathcal{A}_1)+\mathbf{Adv}^{\mathrm{PRF}}_{F_T}(\mathcal{A}_2)+\mathbf{Adv}^{\mathrm{PRF}}_{F_R}(\mathcal{A}_3)+\mathbf{Adv}^{\mathrm{PRF}}_{F_P}(\mathcal{A}_4). \tag{1}$$

Game G2: This game is identical to Game G1 except that the response to every call to the PRP G is prepared using table Tab_G. Similarly as above the table access is done following the lazy smapling [Zha19] technique. Compared to the PRFs the only difference for PRP G is that the responses are chosen without replacement in this case. Since G is a secure PRP, it is easy to see that:

$$\big|\Pr[\mathsf{G2}=1]-\Pr[\mathsf{G1}=1]\big|\ \leq\ \mathbf{Adv}^{\mathrm{PRP}}_{G}(\mathcal{A}_5). \tag{2}$$

Game G3: In this game, we replace the hash functions H_1 and H_2 with random oracles while generating the transcript for different queries. As example consider H_1: During update query $addr$ is chosen randomly from $\{0,1\}^\lambda$ and then the value $addr$ is stored in table Tab_{addr} with corresponding $w\|\mathsf{upd}_w\|\mathsf{ver}_w$. Now during the search query when H_1 is accessed for some fixed $key\|arg$, the client randomly programs the oracle H_1 using table

Algorithm 9 : $H_1(key\|arg)$

1: $v \leftarrow \mathsf{Tab}_{H_1}(key\|arg)$
2: **if** $v = \perp$ **then**
3: $\quad v \xleftarrow{\$} \{0,1\}^\lambda$
4: $\quad \mathsf{Tab}_{H_1}(key\|arg) \leftarrow v$
5: **return** v

Tab_{H_1} as described in algorithm 9. The Tab_{H_1} stores the value $\mathsf{Tab}_{addr}[w\|\mathsf{upd}_w\|\mathsf{ver}_w]$ corresponding to the key $sw\|st$ where $st \leftarrow \sigma_c[w\|\mathsf{upd}_w\|\mathsf{ver}_w]$.

Note that inconsistency can occur during this client calls to H_1, to be specific for some $w\|\mathsf{upd}_w\|\mathsf{ver}_w$ it may happen that $H_1(sw\|\sigma_c[w\|\mathsf{upd}_w\|\mathsf{ver}_w] \neq \mathsf{Tab}_{addr}[w\|\mathsf{upd}_w\|\mathsf{ver}_w]$. This happens when the client randomly selects some value x for $\sigma_c[w\|\mathsf{upd}_w\|\mathsf{ver}_w]$ but $sw\|x$ is already used as input of H_1 by the adversary. But since in Game G3 all these values are generated randomly, the probability that such an inconsistency occurs is upper bounded by $p/2^{2\lambda}$, when the adversary makes p many queries to H_1. Assuming the total number of triples $w\|\mathsf{upd}_w\|\mathsf{ver}_w$ to be $p*$, for H_1 and H_2 together we have,

$$\big|\Pr[\mathsf{G3}=1] - \Pr[\mathsf{G2}=1]\big| \;\leq\; \frac{2pp^*}{2^{2\lambda}} \;\leq\; \frac{2q^2}{2^{2\lambda}}. \tag{3}$$

Game G4: Moving forward, we now change the way the response to the encryption and decryption query to the authenticated encryption AE scheme is generated. When there is a query to AE.Enc with key k, message m and nonce n, a response is chosen randomly from $\{0,1\}^s$, where s indicates the size of the actual ciphertext. This value is then stored in Tab_{proof} corresponding to (k,n,m). The decryption queries are then replied to following algorithm 10. Since the authenticated encryption scheme AE is secure, it is easy to see

Algorithm 10 : AE.VDec$(k,n,proof)$

1: **if** $\mathsf{Tab}_{proof} = \perp$ **then**
2: $\quad r \xleftarrow{\$} \perp$
3: **else**
4: $\quad (K,N,M) \leftarrow \mathsf{Tab}_{proof}[proof]$
5: $\quad$ **if** $k \neq K$ or $n \neq N$ **then**
6: $\quad\quad r \leftarrow \perp$
7: $\quad$ **else**
8: $\quad\quad r \leftarrow M$
9: **return** r

that:

$$\big|\Pr[\mathsf{G4}=1] - \Pr[\mathsf{G3}=1]\big| \;\leq\; \mathbf{Adv}^{\mathrm{Priv}}_{\mathsf{AE}}(\mathcal{B}_1) + \mathbf{Adv}^{\mathrm{Auth}}_{\mathsf{AE}}(\mathcal{B}_2). \tag{4}$$

Game G5: Observe that each operation during the update query can uniquely be characterized by its corresponding global timestamp t. Hence in this game, we modify the computations during the update queries as: every table access with $w\|\mathsf{upd}_w\|\mathsf{ver}_w$ is replaced by table access with corresponding time stamp t. Hence,

$$\big|\Pr[\mathsf{G5}=1] - \Pr[\mathsf{G4}=1]\big| \;=\; 0. \tag{5}$$

Now we will construct a simulator $\mathcal{S}_{\mathsf{FVS2}}$ that, given only the access of the leakage function $\mathcal{L}_{\mathsf{FVS2}}$ but not the actual queries, can generate transcripts for the update and search queries. Remember that the transcript needs to follow the same distribution defined in Game G5. The simulator's behavior for the setup and update operations are described formally in the Supporting Material C. Finally the theorem follows as we combine Eqn. (1), (2), (3), (4), (5) together.

The proof of Theorem 1 follows similarly and we provide the sketch in the Supplementary Material B.

5.2 Proof of Theorem 2 and 4

In this section, we first rigorously prove Theorem 2. The proof of Theorem 4 is identical to the proof of Theorem 2, and hence, we skip it. Before delving into the formal proof we recall that Theorem 2 says that if the underlying AE has authenticity security and the construction FVS1 is correct, then the client should be able to identify any malicious behavior of the server. We follow the standard verification proof technique as used in [Bos16, YCR22] and reduce the soundness advantage to the authenticity of AE and the correctness of FVS1 through a sequence of games S0 to S3. The difference between these games lies strictly in the corresponding verify algorithm and everything else remains the same.

Game S0: In this game, the verify algorithm is the same as the function Verify in algorithm 4. Hence, by definition

$$\Pr[\mathsf{VDSSESound}_{\mathcal{A}}^{\mathsf{FVS1}}] \;=\; \Pr[\mathsf{S0}(1^{\lambda}) = 1] \tag{6}$$

Game S1: The first change that we have done in this game is that the function Verify here besides returning 1 for the correct and complete result, also returns 1, if the adversary produces any forged proof $FP[i],\ i \in \{0, 1, \cdots, c\}(\{1, \cdots, c\}\ for\ \mathsf{ver}_w > 0)$, that passes the verification step of AE.VDec. The modified Verify function is depicted in the left part of Algorithm 11. For $FP[i]$, to be a forged proof, the corresponding ciphertext must not be produced by encryption of AE and with key p_w and nonce i. Remember that S0 outputs 1, if all the c ($c+1\ for\ \mathsf{ver}_w > 0$) proofs pass the verification algorithm of the AE.VDec. Thus, we have

$$\Pr[\mathsf{S0}(1^{\lambda}) = 1] \;\leq\; \Pr[\mathsf{S1}(1^{\lambda}) = 1]. \tag{7}$$

Game S2: In this game we further change the conditions of returning 1 of the function Verify as depicted in the right part of Algorithm 11. Observe that if the adversarial server can forge a proof $FP[i]$ which passes the verification but is not a ciphertext produced by the encryption of AE and with key p_w and nonce i, S1 outputs 1 but S0 will output 0. Consequently,

$$\Pr[\mathsf{S1}(1^{\lambda}) = 1] - \Pr[\mathsf{S2}(1^{\lambda}) = 1] \;\leq\; \mathbf{Adv}_{\mathsf{AE}}^{\mathsf{Auth}}(\mathcal{A}_1). \tag{8}$$

Game S3: Finally, in this game, we assume that there is no collision in the hash table TSet, i.e. given the latest σ_c the adversary can unambiguously find the recently added entry related to searched keyword w. So, if some adversary can distinguish between game S2 and game S3, it can break the correctness of our FVS1 scheme:

$$\Pr[\mathsf{S2}(1^{\lambda}) = 1] - \Pr[\mathsf{S3}(1^{\lambda}) = 1] \;\leq\; \mathbf{Adv}_{\mathsf{FVS1}}^{\mathsf{Correct}}(\mathcal{A}_2). \tag{9}$$

Now let us try to compute $\Pr[\mathsf{S3}(1^{\lambda}) = 1]$. Note that, if an adversary behaves maliciously, then either of the following three cases occur:

- *Case 1: The adversary sends an incomplete proof.* Here, as described in function Verify in game S3, the client uses the current update count upd_w to check whether the number of proofs in Proof list is correct or not. Since the update counter is stored on the client side, such an incomplete array of proofs will always be rejected.
- *Case 2: Some of the proofs are forged or positioned incorrectly.* Again, by definition of the game S3, such an array of proofs will always be rejected.

Algorithm 11 : $\mathsf{Verify}(\mathsf{K}, \sigma_c, w; \mathsf{EDB})$ Algorithm for Game S1 (left) and S2 (right)

```
function Client
    K = (K_S, K_T, K_V, K_R, K_P)
    Rid_P ← ∅
    (ver_w, upd_w) ← σ_c[w]
    if ver_w = 0 then
        if |Proof| ≠ upd_w then
            return 0
    else
        if |Proof| ≠ upd_w + 1 then
            return 0
    if ver_w > 1 then
        p_w ← F_3(K_P, w||ver_w − 1)
        r ← AE.VDec(p_w, 0, Proof[0])
        if r ≠ ⊥ and FP[0] is forged then
            return 1
        if r = ⊥ then
            return 0
        else
            Rid_P ← Rid_P ∪ r
    p_w ← F_3(K_P, w||ver_w)
    for i = 1 to upd_w do
        r ← AE.VDec(p_w, i, P[i])
        if r ≠ ⊥ and FP[i] is not forged then
            return 1
        if r = ⊥ then
            return 0
        else
            op||id ← r
            if op = add then
                Rid_P ← Rid_P ∪ {id}
            else
                Rid_P ← Rid_P \ {id}
    if Rid = Rid_P then
        return 1
    else
        return 0
```

```
function Client
    K = (K_S, K_T, K_V, K_R, K_P)
    Rid_P ← ∅
    (ver_w, upd_w) ← σ_c[w]
    if ver_w = 0 then
        if |Proof| ≠ upd_w then
            return 0
    else
        if |Proof| ≠ upd_w + 1 then
            return 0
    if ver_w > 1 then
        p_w ← F_3(K_P, w||ver_w − 1)
        if FP[0] is not forged then
            return 0
        else
            Rid_P ← Rid_P ∪ r
    p_w ← F_3(K_P, w||ver_w)
    for i = 1 to upd_w do
        if FP[i] is not forged then
            return 0
        else
            op||id ← r
            if op = add then
                Rid_P ← Rid_P ∪ {id}
            else
                Rid_P ← Rid_P \ {id}
    if Rid = Rid_P then
        return 1
    else
        return 0
```

- *Case 3: The adversary sends an incomplete or incorrect Val.* Now, from the above two cases, it is guaranteed that the Proof array is correct and complete. So, the client can easily compute the set Rid_P. However, such a tampered Val array will produce a different Rid, and hence, the output of the function Verify will be rejected. Consequently, here also, the output of S3 will not be 1.

Hence, we have $\Pr[\mathsf{S3}(1^\lambda) = 1] = 0$. Now, combining this with Eqn. (6), (7), (8), (9) with the lemma to obtain the result.

6 Conclusion

This paper proposes two designs achieving forward and backward private verifiable DSSE schemes. While the first construction achieves stronger BPUP backward privacy but high communication complexity, the latter achieves optimal communication at the cost of having slightly weaker BPLP backward privacy. Extending the work to design forward and backward secure verifiable conjunctive DSSE scheme is an interesting open problem.

References

[BFP16] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. *IACR Cryptol. ePrint Arch.*, page 62, 2016. URL: http://eprint.iacr.org/2016/062.

[BMO17] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1465–1482. ACM, 2017. doi:10.1145/3133956.3133980.

[Bos16] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1143–1154. ACM, 2016. doi:10.1145/2976749.2978303.

[CDvD+03] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In Chi-Sung Laih, editor, *Advances in Cryptology - ASIACRYPT 2003, 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30 - December 4, 2003, Proceedings*, volume 2894 of *Lecture Notes in Computer Science*, pages 188–207. Springer, 2003. doi:10.1007/978-3-540-40061-5_12.

[CG12] Qi Chai and Guang Gong. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In *Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, June 10-15, 2012*, pages 917–922. IEEE, 2012. doi:10.1109/ICC.2012.6364125.

[CGKO06] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 79–88. ACM, 2006. doi:10.1145/1180405.1180417.

[CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 668–679. ACM, 2015. doi:10.1145/2810103.2813700.

[CJJ+13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373. Springer, 2013. doi:10.1007/978-3-642-40041-4_20.

[CM05] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455, 2005. doi:10.1007/11496137_30.

[CMS25] Debrup Chakraborty, Avishek Majumder, and Subhabrata Samajder. Making searchable symmetric encryption schemes smaller and faster. *Int. J. Inf. Sec.*, 24(1):10, 2025. URL: https://doi.org/10.1007/s10207-024-00915-y, doi:10.1007/S10207-024-00915-Y.

[CPS20] Sanjit Chatterjee, Shravan Kumar Parshuram Puria, and Akash Shah. Efficient backward private searchable encryption. *J. Comput. Secur.*, 28(2):229–267, 2020. doi:10.3233/JCS-191322.

[GYZ+21] Xinrui Ge, Jia Yu, Hanlin Zhang, Chengyu Hu, Zengpeng Li, Zhan Qin, and Rong Hao. Towards achieving keyword search over dynamic encrypted cloud data with symmetric-key based verification. *IEEE Trans. Dependable Secur. Comput.*, 18(1):490–504, 2021. doi:10.1109/TDSC.2019.2896258.

[HWKS98] Chris Hall, David A. Wagner, John Kelsey, and Bruce Schneier. Building prfs from prps. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 370–389. Springer, 1998. URL: https://doi.org/10.1007/BFb0055742, doi:10.1007/BFB0055742.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. URL: https://www.ndss-symposium.org/ndss2012/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation.

[KO12] Kaoru Kurosawa and Yasuhiro Ohtaki. Uc-secure searchable symmetric encryption. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, February 27 - March 2, 2012, Revised Selected Papers*, volume 7397 of *Lecture Notes in Computer Science*, pages 285–298. Springer, 2012. doi:10.1007/978-3-642-32946-3_21.

[KO13] Kaoru Kurosawa and Yasuhiro Ohtaki. How to update documents verifiably in searchable symmetric encryption. In Michel Abdalla, Cristina Nita-Rotaru, and Ricardo Dahab, editors, *Cryptology and Network Security - 12th International Conference, CANS 2013, Paraty, Brazil, November 20-22. 2013. Proceedings*, volume 8257 of *Lecture Notes in Computer Science*, pages 309–328. Springer, 2013. doi:10.1007/978-3-319-02937-5_17.

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 965–976. ACM, 2012. doi:10.1145/2382196.2382298.

[NC07] Alexandros Ntoulas and Junghoo Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In Wessel Kraaij, Arjen P. de Vries, Charles L. A. Clarke, Norbert Fuhr, and Noriko Kando, editors, *SIGIR 2007: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Amsterdam, The Netherlands, July 23-27, 2007*, pages 191–198. ACM, 2007. doi:10.1145/1277741.1277776.

[SDY+20] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. Forward private searchable symmetric encryption with optimized I/O efficiency. *IEEE Trans. Dependable Secur. Comput.*, 17(5):912–927, 2020. doi:10.1109/TDSC.2018.2822294.

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine 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*. The Internet Society, 2014. URL: https://www.ndss-symposium.org/ndss2014/practical-dynamic-searchable-encryption-small-leakage.

[SWP00] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 44–55. IEEE Computer Society, 2000. doi:10.1109/SECPRI.2000.848445.

[SYL+18] Shifeng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 763–780. ACM, 2018. doi:10.1145/3243734.3243782.

[YCR22] Dandan Yuan, Shujie Cui, and Giovanni Russello. We can make mistakes: Fault-tolerant forward private verifiable dynamic searchable symmetric encryption. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*, pages 587–605. IEEE, 2022. URL: https://doi.org/10.1109/EuroSP53844.2022.00043, doi:10.1109/EUROSP53844.2022.00043.

[Zha19] Mark Zhandry. How to record quantum queries, and applications to quantum indifferentiability. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 239–268. Springer, 2019. doi:10.1007/978-3-030-26951-7_9.

[ZKP16] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 707–720. USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang.

[ZLW+18] Jie Zhu, Qi Li, Cong Wang, Xingliang Yuan, Qian Wang, and Kui Ren. Enabling generic, verifiable, and secure data search in cloud services. *IEEE*

Trans. Parallel Distributed Syst., 29(8):1721–1735, 2018. doi:10.1109/TPDS.2018.2808283.

[ZWW+19] Zhongjun Zhang, Jianfeng Wang, Yunling Wang, Yaping Su, and Xiaofeng Chen. Towards efficient verifiable forward secure searchable symmetric encryption. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, volume 11736 of *Lecture Notes in Computer Science*, pages 304–321. Springer, 2019. doi:10.1007/978-3-030-29962-0_15.

A Understanding the Leakage Profile for FVS2

During the execution of our FVS2 scheme, let $\mathcal{Q}$ be the list of all queries issued by the client with $|\mathcal{Q}| = q$. From Algorithms 5 it is clear that the setup phase leaks no information to the server, as we outsource an empty database EDB which does not contain any information about the original database. Hence we conclude,

$$\mathcal{L}^{\mathsf{SetUp}}_{\mathsf{FVS2}}() \;=\; \emptyset.$$

Now with every search and update operation, the server starts learning information about the encrypted database. Some of these information (leakages) are trivial to an SSE scheme (like update token leaks the number of (w, id) being updated) and hence are not considered as leakage, while the others leakages are specific to construction. In the following, we will elaborate on the non-trivial leakages of our construction. We'll show that the leakage of our construction is subsumed by the leakage function for forward privacy and backward privacy with link pattern (BPLP) defined in Section 2.3. Hence, if there exists a PPT simulator that can simulate an indistinguishable transcript with the leakages of our construction then by definition of forward and backward privacy construction FVS2 is also forward and backward private with link pattern.

Leakage on Updates: During the update phase (see Algorithm 6) for every update the client sends the server three values $(addr, value, proof_u)$, where,

1. $addr \leftarrow H_1(s_w, st)$,
2. $value \leftarrow H_2(s_w, st) \oplus 0^\lambda \| op \| tag$
3. $proof \leftarrow \mathsf{AE.Enc}(p_w, \mathsf{upd}_w, op \| id)$.

Also, $s_w \leftarrow F_S(K_S, w \| \mathsf{ver}_w)$, $st \leftarrow F_T(K_T, w \| \mathsf{upd}_w \| \mathsf{ver}_w)$. $tag \leftarrow G(K_G, w \| id \| \mathsf{ver}_w)$, and $p_w \leftarrow F_P(K_P, w \| \mathsf{ver}_w)$ For every update operation on a keyword-identifier pair (w, id), the pair $(\mathsf{upd}_w, \mathsf{ver}_w)$ is unique. Thus, as H_1 and H_2 are secure hash functions, the $addr$ and $value$ generated for every update are unique and unlikable, hence, they do not leak anything about the input of the update function. Now the $proof$ that is being sent, as the pair $(\mathsf{upd}_w, \mathsf{ver}_w)$ are always unique for a (w, id) pair, either the p_w or upd_w will be unique for any two update. Thus leaks nothing. So we conclude,

$$\mathcal{L}^{\mathsf{Update}}_{\mathsf{FVS2}}(op, (w, id)) \;=\; \emptyset.$$

Leakage on Search: Now we argue that, during a search query for keyword w the leakage to the server is,

$$\mathcal{L}^{\mathsf{Search}}_{\mathsf{FVS2}}(w) \;\preceq\; \{\mathsf{sp}(w),\ \mathsf{Updates}^{op}(w),\ \mathsf{LP1}(w),\ \mathsf{LP2}(w),\ \mathsf{LP3}(w),\ \mathsf{LDB}(w)\}\,.$$

For every search query, the client always sends $s_w \leftarrow F_S(K_S, w)$ to the server. Thus whenever the same keyword is searched, the value s_w will be the same, and the server learns $\mathsf{sp}(w)$. Now during a search query, the server checks all the updates on w and figures out whether an encrypted identifier (specifically the tag) belongs to the search result or not. In this process, the server learns $\mathsf{LDB}(w)$. At the same time, it learns about the timestamps of the update operations on w along with the specific update op (add or delete). This is captured by the notion $\mathsf{Updates}^{\mathsf{op}}$ from [CPS20], described as follows:

$$\mathsf{Updates}^{op}(w) \;=\; \{(t, op) : (t, op, (w, id)) \in \mathcal{Q}\}.$$

Now as tag, (which is $tag \leftarrow G(K_G, w\|id\|\mathsf{ver}_w)$) is the same for a (w, id) pair for a fixed ver_w (that is in between two searches). Thus, the server can able to link which are the updates that correspond to the same (w, id) in between two searches. The server (adversary) learns,

$$\mathsf{LP3}(w) \;=\; \{(t, t') : (t, op, (w, id)) \in \mathcal{Q} \text{ and } (t', op, (w, id)) \in \mathcal{Q};\; t_x < t < t' < t_{x+1}\}\,.$$

Now let us consider two consecutive searches on w at t_x and t_{x+1}, and the corresponding search result DB_x and DB_{x+1} respectively. If and $id \in \mathsf{DB}_x(w)$ and any further operation on that id with the same keyword happened then the server always learns that as the tags generated are the same. A similar argument holds for $id \in \mathsf{DB}_{x+1}(w)$. Hence the following two leakages.

$$\begin{aligned}
\mathsf{LP1}(w) &= \{(t, id) : id \in \mathsf{DB}_x(w) \text{ and } (t, op, (w, id)) \in \mathcal{Q};\; t_x < t < t_{x+1}\}\,,\\
\mathsf{LP2}(w) &= \{(t, id) : id \in \mathsf{DB}_{x+1}(w) \text{ and } (t, op, (w, id)) \in \mathcal{Q};\; t_x < t < t_{x+1}\}\,.
\end{aligned}$$

Taken everything together, we summarize the leakage profile of $\mathsf{FVS2}$ as follows:

$$\mathcal{L}_{\mathsf{FVS2}} \;=\; \left(\mathcal{L}^{\mathsf{SetUp}}_{\mathsf{FVS2}}, \mathcal{L}^{\mathsf{Update}}_{\mathsf{FVS2}}, \mathcal{L}^{\mathsf{Search}}_{\mathsf{FVS2}}\right),$$

where

$$\begin{aligned}
\mathcal{L}^{\mathsf{SetUp}}_{\mathsf{FVS2}}() &= \mathcal{L}^{\mathsf{Update}}_{\mathsf{FVS2}}(op, (w, id)) \;=\; \emptyset,\\
\mathcal{L}^{\mathsf{Search}}_{\mathsf{FVS2}}(w) &\preceq \{\mathsf{sp}(w),\; \mathsf{Updates}^{\mathsf{op}}(w),\; \mathsf{LP1}(w),\; \mathsf{LP2}(w),\; \mathsf{LP3}(w),\; \mathsf{LDB}(w)\}\,.
\end{aligned}$$

So following definition 2.3 we can argue that $\mathsf{FVS2}$ achieves forward privacy and BPLP backward-privacy notion for single keyword DSSE.

B Proof of Theorem 1 - A Brief Sketch

The proof of this result follows the same standard approach of defining a sequence of games between the challenger $\mathcal{C}$ and the adversary $\mathcal{A}$. The first game (i.e., Game $\mathsf{B0}$) computes a distribution identical to the real experiment $\mathsf{FVS1}_{\mathcal{A}}(\lambda)$ and the final game (i.e., Game $\mathsf{B5}$) considers a simulator $\mathcal{S}$ that perfectly simulates a distribution identical to the ideal experiment $\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$, given the leakage function of $\mathsf{FVS1}$.

Game $\mathsf{B0}$: This game is identical to the real experiment $\mathsf{FVS1}_{\mathsf{FVS1}}(\lambda)$. The challenger generates the transcript corresponding to each update and search query using Algorithms 2, 3, 4 for $\mathsf{FVS1}$.
Game $\mathsf{B1}$: This game is identical to Game $\mathsf{B0}$ except that response to every call to the PRFs F_S, F_T, F_V, F_R and F_P are generated using tables $\mathsf{Tab}_S, \mathsf{Tab}_T, \mathsf{Tab}_V, \mathsf{Tab}_R$ and Tab_P, respectively. The table access is done following the lazy sampling [Zha19] technique: when

an entry in a table is first accessed it is chosen at random and the chosen value is used for further access. Since all of F_S, F_T, F_V, F_R, and F_P are secure PRFs, it is easy to see that:

$$\begin{aligned}\big|\Pr[\mathsf{B1}=1]-\Pr[\mathsf{B0}=1]\big| \;\leq\; & \mathbf{Adv}^{\mathrm{PRF}}_{F_S}(\mathcal{A}_1)+\mathbf{Adv}^{\mathrm{PRF}}_{F_T}(\mathcal{A}_2)+\mathbf{Adv}^{\mathrm{PRF}}_{F_V}(\mathcal{A}_3)+\\ & \mathbf{Adv}^{\mathrm{PRF}}_{F_R}(\mathcal{A}_4)+\mathbf{Adv}^{\mathrm{PRF}}_{F_P}(\mathcal{A}_5). \end{aligned} \tag{10}$$

Game B2: In this game, we program the hash functions H_1 and H_2 with random oracles while generating the transcript for different queries. During update query $addr$ is chosen randomly from $\{0,1\}^\lambda$ and then the value $addr$ is stored in table Tab_{addr} with corresponding $w\|\mathsf{upd}_w\|\mathsf{ver}_w$. Now during the search query when H_1 is accessed for some fixed $key\|arg$, the client randomly programs the oracle H_1 using table Tab_{H_1} as described in algorithm 9. The Tab_{H_1} stores the value $\mathsf{Tab}_{addr}[w\|\mathsf{upd}_w\|\mathsf{ver}_w]$ corresponding to the key $sw\|st$ where $st \leftarrow \sigma_c[w\|\mathsf{upd}_w\|\mathsf{ver}_w]$.

As in the proof of theorem 3, inconsistency can occur during this client calls to H_1. To be specific for some $w\|\mathsf{upd}_w\|\mathsf{ver}_w$ it may happen that $H_1(sw\|\sigma_c[w\|\mathsf{upd}_w\|\mathsf{ver}_w] \neq \mathsf{Tab}_{addr}[w\|\mathsf{upd}_w\|\mathsf{ver}_w]$. Similarly as before, for H_1 and H_2 together we have,

$$\big|\Pr[\mathsf{B2}=1]-\Pr[\mathsf{B1}=1]\big| \;\leq\; \frac{2pp^*}{2^{2\lambda}} \;\leq\; \frac{2q^2}{2^{2\lambda}}. \tag{11}$$

Game B3: We now change the way the $eRid$s are generated by the symmetric encryption scheme SE in Algorithm 4. When there is a query to SE.Enc with key k, message m, a response is chosen randomly from $\{0,1\}^s$, where s indicates the size of the actual ciphertext. Since the symmetric encryption scheme SE is modeled as a secure PRP, following PRP/PRF switching lemma [HWKS98] we have:

$$\big|\Pr[\mathsf{B3}=1]-\Pr[\mathsf{B2}=1]\big| \;\leq\; \mathbf{Adv}^{\mathrm{Priv}}_{\mathsf{SE}}(\mathcal{B}_1)+\frac{q^2}{2^s}. \tag{12}$$

Game B4: Moving forward, we now change the way the response to the encryption and decryption query to the authenticated encryption AE scheme is generated. When there is a query to AE.Enc with key k, message m and nonce n, a response is chosen randomly from $\{0,1\}^s$, where s indicates the size of the actual ciphertext. This value is then stored in Tab_{proof} corresponding to (k,n,m). The decryption queries are then replied to by algorithm 10. Since the authenticated encryption scheme AE is secure, it is easy to see that:

$$\big|\Pr[\mathsf{B4}=1]-\Pr[\mathsf{B3}=1]\big| \;\leq\; \mathbf{Adv}^{\mathrm{Priv}}_{\mathsf{AE}}(\mathcal{B}_2)+\mathbf{Adv}^{\mathrm{Auth}}_{\mathsf{AE}}(\mathcal{B}_3). \tag{13}$$

Game B5: Observe that each operation during the update query can uniquely be characterized by its corresponding global timestamp t. Hence in this game, we modify the computations during the update queries as: every table access with $w\|\mathsf{upd}_w\|\mathsf{ver}_w$ is replaced by table access with corresponding time stamp t. Hence,

$$\big|\Pr[\mathsf{B5}=1]-\Pr[\mathsf{B4}=1]\big| \;=\; 0. \tag{14}$$

Now we claim that it is possible to construct a simulator S_{FVS1} that generates the transcript for each update and search query. Note that the transcripts will have to follow the same distribution as in Game B5. Also, S_{FVS1} does not have access to the actual queries but it has access to the leakage function $\mathcal{L}_{\mathsf{FVS1}}$. The description of the simulator S_{FVS1} is similar to that of S_{FVS2}, given in Supplementary Material C, and hence skipped. Finally the theorem follows as we combine Eqn. (10), (11), (12), (13), (14) together.

C Description of the Simulator $\mathcal{S}_{\mathsf{FVS2}}$

The Setup Algorithm of $\mathcal{S}_{\mathsf{FVS2}}$: The simulator simply initializes the timestamp to 0. The algorithm is given in Algorithm 12.

Algorithm 12 : $\mathcal{S}$.Setup($\perp$)

1: $t \leftarrow 0$

The Update Algorithm of S_{FVS2}: Following Game $\mathsf{G5}$, the simulator randomly chooses $addr$, $value$, and $proof$ and stores them in the corresponding tables for further use. In each table the values are stored corresponding to the update time stamp t. The algorithm description of the update operation is given in Algorithm 13.

Algorithm 13 : $\mathcal{S}$.Update ($\perp$)

1: **function Client**
2: $\quad \mathsf{Tab}_A[t] \xleftarrow{\$} \{0,1\}^{\lambda}$
3: $\quad \mathsf{Tab}_V[t] \xleftarrow{\$} \{0,1\}^{5\lambda}$
4: $\quad proof \xleftarrow{\$} \{0,1\}^{|\mathsf{AE.Enc}(*,*,*)|}$
5: $\quad \mathsf{Tab}_P[proof] \leftarrow t$
6: $\quad t \leftarrow t+1$

The Search Algorithm of S_{FVS2}: During the search queries S_{FVS2} first sends latest st, s_w and $(\mathsf{upd}_w, \mathsf{ver}_w)$ to the server. This is done by accessing the corresponding tables $\mathsf{Tab}_T, \mathsf{Tab}_S$. $(\mathsf{upd}_w, \mathsf{ver}_w)$ are retrieved from the search leakage of $\mathsf{FVS2}$. Also accessing Tab_R sends $addr_w$ to the server. The detailed description of the two search phases is depicted in Algorithm 14 and 15.

Algorithm 14 : $\mathcal{S}$.Search($\mathsf{sp}(w), \mathsf{Updates}^{\mathsf{op}}(w), \mathsf{LP1}(w), \mathsf{LP2}(w), \mathsf{LP3}(w), \mathsf{LDB}(w)$) Round I

function Client
  $\{t_1, t_2, \cdots, t_s\} \leftarrow \mathsf{sp}(w)$
  $v \leftarrow s - 1$
  if $v = 0$ **then**
    Denote searched keyword as $\overline{w}$
    $s_{\overline{w}} \xleftarrow{\$} \{0,1\}^{\lambda}$
    $\mathsf{Tab}_S[t_1] \leftarrow (s_w, \overline{w})$
  else
    $(s_w, \overline{w}) \leftarrow \mathsf{Tab}_S[t_1]$
  if $\mathsf{Updates}^{\mathsf{op}}[t_s, \overline{w}] = \perp$ **then**
    return ϕ
  if $v = 0$ **then**
    $addr_w \xleftarrow{\$} \{0,1\}^{\lambda}$
    $\mathsf{Tab}_R(\overline{w}) \leftarrow addr_w$
  else
    $addr_w \leftarrow \mathsf{Tab}_R(\overline{w})$
  $\{(t_{u_1}, op_{u_1}), (t_{u_2}, op_{u_2}), \cdots (t_{u_c}, op_{u_c})\} \leftarrow \mathsf{Updates}^{\mathsf{op}}(t_s, \overline{w}) \setminus \mathsf{Updates}^{\mathsf{op}}(t_{s-1}, \overline{w})$
  for $i = 1$ *to* c **do**
    $st[i] \xleftarrow{\$} \{0,1\}^{\lambda}$
    $\mathsf{Tab}_T[\overline{w} \| i \| v] \leftarrow st[i]$
    Program H_1 such that $H_1(s_w, st[i]) \leftarrow \mathsf{Tab}_A[t_{u_i}]$
    if $(id, t_{u_i}) \in \mathsf{LP1}$ *for some* id **then**
      $tag \leftarrow \mathsf{Tab}_G[v \| id]$
    else if $(id, t_{u_i}) \in \mathsf{LP2}$ *for some* id **then**
      if $(t, t_{u_i}) \in \mathsf{LP3}$ *for some* t **then**
        $tag \leftarrow \mathsf{Tab}_G[v \| id]$
      else
        $tag \xleftarrow{\$} \{0,1\}^{\lambda}$
        $\mathsf{Tab}_G[v \| id] \leftarrow tag$
    else
      if $((t_{u_j}, t_{u_i}) \in \mathsf{LP3}$ *for some* t_{u_j} **then**
        $tag \leftarrow \mathsf{Tab}_G[v, id_{u_j}]$
      else
        $id_{u_i} \xleftarrow{\$} \{0,1\}^{\lambda}$
        $tag \xleftarrow{\$} \{0,1\}^{\lambda}$
        $\mathsf{Tab}_G[v, id_{u_i}] \leftarrow tag$
    Program H_2 such that $H_2(s_w, st[i]) \leftarrow \mathsf{Tab}_V[t_{u_i}] \oplus st[i-1] \| op_{u_i} \| tag$
  Send $(addr_w, st_{u_c}, s_w, (v, c))$ to Server.

Algorithm 15 : $\mathcal{S}$.Search($\mathsf{sp}(w), \mathsf{Updates}^{op}(w), \mathsf{LP1}(w), \mathsf{LP2}(w), \mathsf{LP3}(w), \mathsf{LDB}(w)$) Round II

function Client
  $\mathsf{Rid}, \mathsf{PrvTag} \leftarrow \emptyset$
  $\{t_1, t_2, \cdots, t_s\} \leftarrow \mathsf{sp}(w)$
  $v \leftarrow s - 1$
  $(s_w, \overline{w}) \leftarrow \mathsf{Tab}_S[t_1]$
  $\{(t_{u_1}, op_{u_1}), (t_{u_2}, op_{u_2}), \cdots (t_{u_c}, op_{u_c})\} \leftarrow \mathsf{Updates}^{\mathsf{op}}(t_s, \overline{w}) \setminus \mathsf{Updates}^{\mathsf{op}}(t_{s-1}, \overline{w})$
  Obtain $\mathsf{DB}(\overline{w})$ from $\mathsf{LDB}(\overline{w})$
  if $v = 0$ **then**
    if $|\mathsf{Proof}| \neq c$ **then**
      return Reject
    else
      for $i = 1$ *to* c **do**
        if $\mathsf{Tab}_P[\mathsf{Proof}[i]] \neq t_{u_i}$ **then**
          return Reject
  else
    if $|\mathsf{Proof}| \neq c + 1$ **then**
      return Reject
    else
      for $i = 1$ *to* c **do**
        if $\mathsf{Tab}_P[\mathsf{Proof}[i]] \neq t_{u_i}$ **then**
          return Reject
      if $\mathsf{Tab}_P[\mathsf{Proof}[0]] \neq t_{s-1}$ **then**
        return Reject
  for *each* $tag \in \mathsf{TagSet}$ **do**
    if $\exists$ id such that $\mathsf{Tab}_G[v \| id] = tag$ **then**
      $\mathsf{Rid} \leftarrow \mathsf{Rid} \cup \{id\}$
    else
      return Reject
  if $\mathsf{Rid} \neq \mathsf{DB}(\overline{w})$ **then**
    return Reject
  else
    $v \leftarrow v + 1$
    for *each* $id \in \mathsf{Rid}$ **do**
      $tag \xleftarrow{\$} \{0,1\}^{\lambda}$
      $\mathsf{Tab}_G[v \| id] \leftarrow tag$
      $\mathsf{PrvTag} \leftarrow \mathsf{PrvTag} \cup \{tag\}$
    $proof \leftarrow \{0,1\}^{|\mathsf{AE.Enc}(*,*,*)|}$
    $\mathsf{Tab}_P[proof] \leftarrow t_s$
    Send $\mathsf{PrvTag}, proof$ to Server.
    return Rid