Partitioned Searchable Encryption

Jim Barthel[1], Marc Beunardeau[2], Răzvan Roşie[3], and Rajeev Anand Sahu[1](✉)

[1] University of Luxembourg, Esch-sur-Alzette, Luxembourg
{jim.barthel,rajeev.sahu}@uni.lu
[2] Nomadic Labs, Paris, France
marc.beaunardeau@nomadic-labs.com
[3] JAO Luxembourg, Luxembourg, Luxembourg
rosie@jao.eu

Abstract. Symmetric searchable encryption (SSE) allows to outsource encrypted data to an untrusted server and retain searching capabilities. This is done without impacting the privacy of both the data and the search/update queries. In this work we put forth a new flavour of symmetric searchable encryption (SSE):*Partitioned SSE* is meant to capture the cases where the search rights must be partitioned among multiple individuals. We motivate through compelling examples the practical need for such a notion and discuss instantiations based on functional encryption and trapdoor permutations.

– First we leverage the power of *functional encryption* (FE). Our construction follows the general technique of encrypting the set of keywords and the presumably larger datafiles separately, a keyword acting as a "pointer" to datafiles it belongs to. To improve on the constraint factors (large ciphertext, slow encryption/decryption procedures) that are inherent in FE schemes, the keyword check is done with the help of a Bloom filter – one per datafile: the crux idea is to split the filter into buckets, and encrypt each bucket separately under an FE scheme. Functional keys are given for binary *masks* checking if relevant positions are set to 1 inside the underlying bit-vector of the Bloom filter.
– The second construction we present achieves forward security and stems from the scheme by Bost in CCS'16. We show that a simple tweak of the original construction gives rise to a scheme supporting updates in the partitioned setting. Moreover, the constructions take into account the possibility that some specific users are malicious while declaring their search results.

Keywords: SSE · Functional encryption · Partitioned search · Bloom filter

1 Introduction

Searchable encryption [20] is a cryptographic protocol thought to enable its user(s) to perform search queries on encrypted data[1]. In the protocol a set of *keywords* is encrypted and deployed on an untrusted (storage) server. Each keyword originates in some structured *datafile*, which is encrypted separately under a semantic secure symmetric encryption scheme. Ideally, the search operation executed by the client shall work without compromising the privacy of the remaining encrypted data, given that the server has access to the entire history of queries. Speaking about functionality, a client must store some secret information (*key*) that allows to create search tokens corresponding to specific keywords. Tokens are sent to the storage server together with the operation that needs to be executed: *searches* for static schemes, but also *updates* for dynamic schemes. The server uses the tokens to retrieve the index(es) of the encrypted datafile(s) matching the desired keyword(s), but without being able to decrypt the datafile(s) and without learning those keywords. The initial proposals of SSE were designed in a *static* setting where the client cannot perform any *update* on the deployed encrypted data. To address this issue Kamara *et al.* [16] introduced *dynamic SSE* which enables both *searches* and *updates* over the encrypted database. However, update may cause leakage during addition of new (keyword, datafile index) pairs or during the search of a keyword while all the files containing the keyword are deleted. Security against the first case is called *forward privacy* – introduced by Chang and Mitzenmacher [7] and against the second case is referred as *backward privacy* – formalized by Bost et al. [5]. One common issue with the earlier proposals was the search time which was linear in the size of the database. It was until the work of Curtmola *et al.* [9] who put forth the SSE scheme with sublinear search time, in a static setting. The index-based dynamic SSE with sublinear search time was introduced by Kamara *et al.* [15,16]. The current approaches of searchable encryption [4–6] suggest avoiding constructions following from primitives such as fully-homomorphic encryption [10], multi-party computation [23] or oblivious RAM [12]. These are considered non-viable, given their poor practical performance. However, in many settings such techniques can be proven safe as they leak no information on the encrypted data. Consider the case of functional encryption (FE) [3]. A naive but straightforward implementation of searchable encryption consists in issuing functional keys for circuits searching a specific keywords. In the recent years several schemes of SSE have been proposed [17,19] based on different assumptions and targeting advanced properties. Our construction addresses a completely different flavour and unlike the existing schemes has been built using the Bloom filters.

In the existing literature of SSEs, a relevant contribution for our approach is the paper by Goh [11], who proposes a construction – associating an index to each document in a collection – based simply on Bloom filters and pseudorandom functions. Another construction in connection to our results is $\Sigma o\phi o\varsigma$ by Bost [4]

[1] For example a doctor wanting to consult all the medical records of patients having diabetes without having to download the entire database.

which is a scheme supporting sublinear search time and achieving forward privacy with improved security guarantees. In essence, the construction avoids the heavy ORAM model while relying solely on the existence of trapdoor permutations. The later Diana and Janus schemes [5] are improvements on this approach.

1.1 Our Results

We introduce a new flavour of symmetric searchable encryption- *partitioned* searchable encryption. More particularly, we propose two constructions of partitioned SSE using the functionalities of a Bloom filter (BF), one from the functional encryption (FE) and one from trapdoor permutations as used in $\Sigma o\phi o\varsigma$.

Partitioned Symmetric Searchable Encryption (PSSE). Imagine a well known governmental agency intercepts the conversation between the president of its country and a foreign leader. Legally, the transcripts of such recordings must be stored on a secure server and access rights must be given to some investigative authority, after which the role of the security agency ends. Following the law, those recordings may only be accessed by some selected committee of the Senate in its plenitude. That is, no single member of the Senate's committee may access the data independently. To address such a problem, the notion of PSSE may be useful. Such protocol consists of three entities: (1) a trusted authority that encrypts data and subsequently deploys them to a storage server[2]; (2) the server that stores the data; (3) the clients that can gain access to data if and only if *all* agree to do so. We also emphasise that in connection to our partitioned SSE a recent work [1] by Ananth *et al.* is much more relevant as it presents a multi-key FHE with one-round decryption. The process of recovering the plaintext(s), taking place in the final step of their construction, is similar to step (3) above. The *multi-client* in [22], which presents a searchable encryption supporting Boolean queries, refers to a group of clients satisfying certain attributes, and not associated to any partition among them. Each client possessing a *search-authorized* private key issued by the data owner can individually perform *search* where the keyword database is encrypted using a CP-ABE. A related problem would consider malicious users: that is, users that misbehave either when inserting or searching for documents. Relative to our previous example, any senator, independent of his/her political opinion must be able to prove that his/her part of the encrypted database DB and search tokens are correctly generated.

We propose multiple instantiations for such a partitioned searching protocol. The first one exploits the power of functional encryption. A naive approach would encrypt the set of keywords, and then issue search tokens for the search function. However, we introduce a novel and more efficient approach for building searchable encryption from FE. Our key insight is the following: given a document D, we store its set of keywords ($\mathbf{w}$) in a Bloom filter, which is built on top of a bitvector $\vec{b}$. We split $\vec{b}$ into buckets and encrypt each bucket independently under a functional encryption scheme. Then, we issue functional keys for circuits that

[2] The governmental agency in our case.

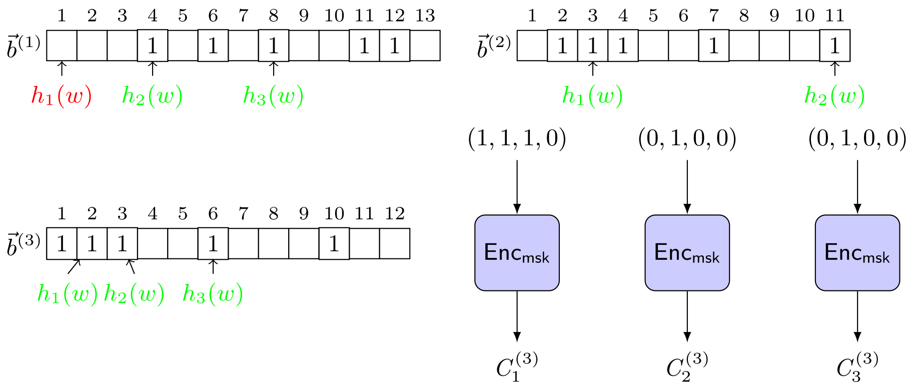

Fig. 1. Each Bloom Filter $\mathsf{BF}^{(i)}$ is associated to a datafile $\mathsf{D}^{(i)}$ (not represented) and consists of a bitvector $\vec{b}^{(i)}$ of size B_i. As seen at the lower right, $\vec{b}^{(3)}$ is split in 3 buckets of size 4, which are encrypted independently under a functional encryption scheme. Thus, the message space supported by the FE scheme consists of 4 bits. Functional keys are issued to check if the j-th bit is set to 1, in order to simulate the membership testing in the Bloom filter specification, as seen in the upper part of the picture.

check if the desired set of bits corresponding to a search query is set to 1. This is illustrated in Fig. 1. The second class of proposed constructions stems from the $\Sigma o\phi o\varsigma$ scheme introduced by Bost in [4]. $\Sigma o\phi o\varsigma$ uses classical primitives, such as simple trapdoor permutations. Whenever a new keyword is inserted, the index of the keyword is *masked* with a pseudorandom value the client is aware of. We achieve a PSSE scheme by distributing such a masked value amongst the participants into the protocol. Concretely, we employ the usage of a Bloom filter to store the index. Then, we split the Bloom filter into buckets, and each party will independently mask a bucket of the Bloom filter. Pictorially, this would correspond to a parallel execution of Bost's protocol. Finally, during the combine (Comb) step, the results of the searches are gathered and the question of a keyword belonging to a document can be settled.

Organization of the Paper. Section 2 introduces the common notations and the definitions we work with in the following parts. Section 3 defines partitioned SSE. In Sect. 4 we introduce the main results of this work: by devising simple PSSE protocols starting from FE and trapdoor permutations, static or supporting updates. Section 5 concludes the contributions.

2 Preliminaries

Mathematical and Algorithmic Conventions. In this work, $\lambda \in \mathbb{N}^*$ stands for the security parameter. We assume λ is implicitly given to all algorithms in the unary representation 1^λ. We consider an algorithm to be equivalent to a Turing machine, and unless stated, we assume that algorithms are randomized. PPT stands for "probabilistic polynomial-time" in the security parameter (rather than the total length of its inputs). Given a randomized algorithm $\mathcal{A}$ we denote the action of running $\mathcal{A}$ on input(s) $(1^\lambda, x_1, \dots)$ with coins r (sampled uniformly at random) and assigning the output(s) to $(y_1, \dots)$ by the expression $(y_1, \dots) \leftarrow_{\$} \mathcal{A}(1^\lambda, x_1, \dots; r)$. We write $\mathcal{A}^{\mathcal{O}}$ for the case that $\mathcal{A}$ is given oracle access to some procedure $\mathcal{O}$. We denote the cardinality of a finite set S by $|S|$ and the action of sampling a uniformly at random element x from X by $x \leftarrow_{\$} X$. $[k]$ stands for the set $\{1, \dots, k\}$. A real-valued function is called negligible if it belong to $O(\lambda^{-\omega(1)})$. We denote the set of all negligible functions by NEGL. Throughout the paper $\perp$ stands for a special error symbol. We use $||$ to denote concatenation.

2.1 Searchable Encryption

To structure the discussion, we assume a classical *client-server* model, with a client wishing to deploy its data on some untrusted third party; at the same time, the client wants to retain its ability of searching over the encrypted, deployed data. In the first part—the *Setup* phase—the client proceeds as follows with its datafiles $\{\mathsf{D}^{(1)}, \dots, \mathsf{D}^{(d)}\}$: (1) extracts all the keywords for each $\mathsf{D}^{(i)}$ (let this set of keywords be written in a structure denoted DB); (2) encrypts each $\mathsf{D}^{(i)}$ to $\mathsf{ED}^{(i)}$, using a semantic-secure symmetric encryption scheme; (3) encrypts the keywords under a scheme that supports searches (let this resulting database of encrypted keywords be denoted as EDB); (4)uploads EDB and the encrypted datafiles on some untrusted storage cloud server.

In the *Search* phase, whenever looking for a datafile corresponding to a specific keyword $\mathbf{w}$ the client should have the ability to identify the datafile(s) $\mathsf{D}^{(i)}$ containing $\mathbf{w}$. Then, it will retrieve the encrypted file $\mathsf{ED}^{(i)}$ corresponding to $\mathsf{D}^{(i)}$ and decrypt it. Our protocols do *not* explicitly mention the last phase consisting of simply downloading and decrypting the datafiles identified as containing the keywords. A rigorous formulation capturing the aforementioned intuition is given below.

Definition 1 (Multi-Keyword SSE). *Let* $\mathsf{D}_i \subseteq \{0,1\}^*$ *denote a datafile, for any* $i \in [d]$. *Let* $\mathsf{DB} := \{(i, \mathbf{w}^{(i)})\}_{i \in [d]}$ *denote the set of pairs containing a datafile index* i *and a set of keywords* $\mathbf{w}^{(i)}$. *A static searchable encryption scheme* $\Sigma = (\Sigma.\mathsf{Setup}, \Sigma.\mathsf{Search})$ *consists of a* PPT *algorithm* $\Sigma.\mathsf{Setup}$ *and a protocol* $\Sigma.\mathsf{Search}$ *between a client and server, such that:*

- $(\mathsf{EDB}, K, \sigma) \leftarrow_{\$} \mathsf{Setup}(\mathsf{DB})$*: takes as input the* DB*, encrypts it to obtain* EDB*, and deploys the resulting ciphertext to a server. It returns the key* K *and a state* σ *to the client.*
- $i \leftarrow \mathsf{Search}(\mathsf{EDB}, \sigma, K, \mathbf{w}, \mathsf{I})$*: is a protocol between the client and the server. The client inputs its key* K*, its state* σ*, a search query* $\mathbf{w}$ *which can consist of a single or multiple keywords and an index set* I*, consisting of file indices that should be searched through. In case* I *is absent, we consider it as* $\mathsf{I} = \{1\}$ *meaning the system stores only a single datafile. The server's input is* EDB*.*

It then returns the index(es) in I *that correspond to datafile(s) containing the queried keyword(s).*

In addition, we call a symmetric searchable encryption scheme dynamic if there exists a third algorithm:

– $\mathsf{EDB}' \leftarrow \mathsf{Update}(\mathsf{EDB}, \sigma, K, \mathbf{w}, \mathsf{I}, \mathsf{op})$*: the client encrypts a keyword* $\mathbf{w}$ *and sends an update query for a specific index set* I *of datafiles. In case* I *is absent, we consider it as* $\mathsf{I} = \{1\}$ *meaning the system stores only a single datafile. The operation* op *can either be a delete or insert request.* Update *then returns the updated encrypted database.*

We require an SSE *scheme to satisfy correctness, meaning that the search protocol must return correct results for every query, except with negligible probability.*

Security of SSE. Security of an SSE scheme corresponds to the amount of information a server can gather about the database (file) and the keywords queried. More concretely, it is parametrized by the stateful leakage functions incorporating the leakage of the Setup, Search, Update algorithms. We denote this by a leakage function $\mathcal{L} = \left(\mathcal{L}^{\mathsf{Setup}}, \mathcal{L}^{\mathsf{Search}}, \mathcal{L}^{\mathsf{Update}}\right)$[3]. Security requires that the adversary should not learn more than the outputs of the corresponding leakage function $\mathcal{L}$ after triggering the Setup, Search or Update operations. Of particular interest is the notion of *forward privacy* [21], which ensures that an update query does not leak information on the updated keyword, the server being unable to tell if a particular document leaks the updated keywords.

Forward Privacy: informally, it states that newly updated documents do not leak information about newly added files that match the query. Alternatively, the Update queries do not leak the keyword/file being updated.

Definition 2 (Forward Privacy for SSE). *We say an* $\mathcal{L}$*-adaptive-secure multi-keyword* SSE *is forward private if the update leakage function is defined as* $\mathcal{L}^{\mathsf{Update}}(\mathsf{op}, \mathsf{in}) := \mathcal{L}'^{\mathsf{Update}}(\mathsf{op}, (f', \mathbf{w}'))$*, for operation* op *with input* in *where* $\mathcal{L}'$ *is stateless and the set* $(f', \mathbf{w}')$ *denotes all updated documents for which the keyword* $\mathbf{w}'$ *is modified in file* f'*.*

Security requires that the adversary does not learn more than the outputs of the corresponding leakage function $\mathcal{L}$ after triggering the Setup, Search or Update operations. The corresponding security game is described in Fig. 2.

Definition 3 (Adaptive Security for SSE). *We say a multi-keyword* SSE *scheme achieves* $\mathcal{L}$*-adaptive-security if the advantage of any* PPT *adversary* $\mathcal{A}$ *in winning the* FS − SSE *experiment defined in Fig. 2 is negligible. i.e.:* $|\Pr[\mathsf{FS} - \mathsf{SSE}^{\mathcal{A}}_{\mathsf{SSE}}(\lambda) = 1] - \frac{1}{2}| \in \textsc{Negl}(\lambda)$.

Definition 4 (Functional Encryption - Public Key Setting). *A functional encryption scheme* FE *in the public-key setting consists of a quadruple of* PPT *algorithms* (FE.Setup, FE.KDer, FE.Enc, FE.Dec) *such that:*

[3] For a static scheme $\mathcal{L}^{\mathsf{Update}} := \emptyset$.

```
$\underline{\mathsf{FS-SSE}^{\mathcal{A}}_{\mathsf{SSE}}(\lambda)}$:
$b \leftarrow_{\$} \{0,1\}$
$\mathrm{DB} \leftarrow_{\$} \mathcal{A}(1^\lambda)$
if $b = 0$:
    $(\mathrm{EDB}, \sigma) \leftarrow_{\$} \mathsf{SSE.Setup}(1^\lambda)$
else:
    $(\mathrm{EDB}, \sigma) \leftarrow_{\$} \mathcal{S}(\mathcal{L}_{\mathsf{Setup}})$
for $q \leftarrow 1, Q$:
  if $b = 0$:
    $(\mathsf{op}, \mathbf{w}) \leftarrow_{\$} \mathcal{A}(\mathrm{EDB}, \sigma)$
    if $\mathsf{op} =$"Update":
      $R_q \leftarrow_{\$}$ Update($\sigma$,EDB, $\mathbf{w}$)
    if $\mathsf{op} =$"Search":
      $R_q \leftarrow_{\$}$ Search($\sigma$,EDB, $\mathbf{w}$)
  if $b = 1$:
    $(\mathsf{op}, \mathbf{w}) \leftarrow_{\$} \mathcal{A}(\mathrm{EDB}, \sigma)$
    if $\mathsf{op} =$"Update":
      $R_q \leftarrow_{\$} \mathcal{S}(\sigma, \mathcal{L}_{\mathsf{Update}}(\mathbf{w}))$
    if $\mathsf{op} =$"Search":
      $R_q \leftarrow_{\$} \mathcal{S}(\sigma, \mathcal{L}_{\mathsf{Search}}(\mathbf{w}))$
$b' \leftarrow_{\$} \mathcal{A}(\{R_q\}_{q \in [Q]})$
return $b = b'$
```

```
$\underline{\mathsf{PSSE}^{\mathcal{A}}_{\mathsf{PSSE}}(\lambda, N)}$:
$b \leftarrow_{\$} \{0,1\}$
$\mathsf{DB} \leftarrow_{\$} \mathcal{A}(1^\lambda)$
if $b = 0$:
  $(\mathsf{pp}, \mathsf{DB}_1, \ldots, \mathsf{DB}_N) \leftarrow_{\$} \mathsf{PSSE.Setup}(1^\lambda, N, \mathsf{DB})$
  for $j \leftarrow 1, N$:
    $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow_{\$} \mathsf{SSE.ClientSetup}(\mathsf{DB}_j, j)$
else:
  $(\mathsf{pp}, \mathsf{DB}_1, \ldots, \mathsf{DB}_N) \leftarrow_{\$} \mathcal{S}(\mathcal{L}_{\mathsf{Setup}})$
  for $j \leftarrow 1, N$:
    $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow_{\$} \mathcal{S}(\mathcal{L}_{\mathsf{ClientSetup}})$
for $q \leftarrow 1, Q$:
  $(\mathsf{op}, \mathbf{w}) \leftarrow_{\$} \mathcal{A}(\mathsf{pp}, \mathsf{pk}_j, \mathsf{EDB}_j), \forall j \in [N]$
  if $b = 0$:
    if $\mathsf{op} =$"Update":
      $R_{q,j} \leftarrow_{\$}$ Update$(\sigma, \mathsf{sk}_j, j, \mathsf{EDB}_j, \mathbf{w}), \forall j \in [N]$
    if $\mathsf{op} =$"Search":
      $R_{q,j} \leftarrow_{\$}$ Search$(\sigma, \mathsf{sk}_j, j, \mathsf{EDB}_j, \mathbf{w}), \forall j \in [N]$
  if $b = 1$:
    if $\mathsf{op} =$"Update":
      $R_{q,j} \leftarrow_{\$} \mathcal{S}(\sigma, j, \mathcal{L}_{\mathsf{Update}}(\mathbf{w})), \forall j \in [N]$
    if $\mathsf{op} =$"Search":
      $R_{q,j} \leftarrow_{\$} \mathcal{S}(\sigma, j, \mathcal{L}_{\mathsf{Search}}(\mathbf{w})), \forall j \in [N]$
$b' \leftarrow_{\$} \mathcal{A}(\{R_{q,j}\}_{q,j})$
return $b = b'$
```

Fig. 2. The $\mathsf{FS-SSE}$-security defined for a symmetric searchable encryption scheme (left). Simulation-security for a PSSE scheme.

- $(\mathsf{msk}, \mathsf{mpk}) \leftarrow_{\$} \mathsf{FE.Setup}(1^\lambda)$: *given the unary representation of the security parameter λ, the setup procedure outputs a pair of master secret/public keys.*
- $\mathsf{sk}_f \leftarrow_{\$} \mathsf{FE.KDer}(\mathsf{msk}, f)$*: given the master secret key* msk *and a function f, the (potentially randomized) key-derivation procedure generates a corresponding functional key* sk_f*.*
- $C \leftarrow_{\$} \mathsf{FE.Enc}(\mathsf{mpk}, M)$*: the randomized encryption procedure encrypts, using the master public key* mpk*, the plaintext M into a ciphertext C.*
- $\mathsf{FE.Dec}(C, \mathsf{sk}_f)$*: decrypts the ciphertext C using the functional key* sk_f *in order to either learn a valid message $f(M)$ or, in case the decryption procedure fails, a special error symbol $\perp$.*

We say a scheme FE *achieves correctness if $\forall f \in \mathcal{F}_\lambda$, for any $M \in \mathcal{M}$, the following quantity is negligibly close to* 1*:*

$$\Pr\left[y = f(M) \;\middle|\; \begin{array}{l} (\mathsf{msk}, \mathsf{mpk}) \leftarrow_{\$} \mathsf{FE.Setup}(1^\lambda) \wedge \mathsf{sk}_f \leftarrow_{\$} \mathsf{FE.KDer}(\mathsf{msk}, f) \wedge \\ C \leftarrow_{\$} \mathsf{FE.Enc}(\mathsf{mpk}, M) \wedge y \leftarrow \mathsf{FE.Dec}(C, \mathsf{sk}_f) \end{array} \right]$$

2.2 Bloom Filters

Bloom filters (BF), introduced in [2] are *probabilistic* abstract data structures allowing for constant time *searches*, *insertions* and *deletions*. Thus, they improve over both running time and memory space over the existing approaches using *hash-tables* or different flavours of *tree-based* structures. The core idea behind Bloom filters is to store a representation of a keyword $\mathbf{w}$ instead of storing $\mathbf{w}$ itself. To do so, one can imagine an underlying data structure consisting of a bitvector $\vec{b}$ of B bits, that is populated by hashing the inserted strings $\mathbf{w} \in \mathbf{W}$ as depicted in Fig. 3:

– given $\mathbf{w}$, compute $i \leftarrow \mathsf{Hash}(\mathbf{w})$, where $\mathsf{Hash} : \{0,1\}^* \rightarrow \{0,\ldots,B-1\}$ denotes a hash function.
– for each hash function outputting an index i, set $\vec{b}_i \leftarrow 1$.

False positives are possible, since b_i can be set to 1 by multiple strings. Still, by controlling the number of hash functions to be used, one can bound the probability of false positives when inserting n elements through γ hash functions:

$$\Pr\left[\exists \mathbf{w} \notin \mathbf{W} \wedge |\mathbf{W}| \geq 1 \wedge \mathsf{BF.Search}(\vec{b}, \mathbf{w}) = 1\right] \approx \left(1 - \left(1 - \frac{1}{B}\right)^{\gamma \cdot n}\right)^{\gamma}.$$

Therefore, the optimal number of hash functions is simply: $\gamma \approx \ln(2) \cdot \frac{B}{n}$. Moreover, a technique to reduce the false positive rate in a Bloom filter is discussed in [8] specially with the view of its use in searchable encryption.

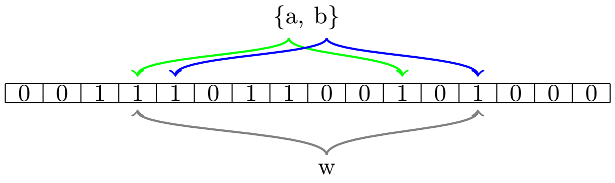

Fig. 3. A depiction of a Bloom filter BF storing elements a, b. Deciding if $\mathbf{w}$ belongs to BF implies a check over the corresponding positions. In this example $\mathbf{w}$ is in the set $\{a, b\}$.

Data Representation. Let $\mathsf{D}^{(1)}, \ldots, \mathsf{D}^{(d)}$ be d datafiles, represented in binary. For each $\mathsf{D}^{(i)}$ we instantiate the vector $\mathbf{w}^{(i)} = (\mathbf{w}_1^{(i)}, \ldots \mathbf{w}_{n_i}^{(i)})$, as the vector of keywords, where $\mathbf{w}_t^{(i)}$ denotes the t^{th} keyword belonging to the i^{th} document $\mathsf{D}^{(i)}$. For each $\mathsf{D}^{(i)}$, we instantiate a Bloom filter $\mathsf{BF}^{(i)}$, whose bit-vector $\vec{b}^{(i)}$ can be split into $l^{(i)}$ buckets of equal size $\frac{B_i}{l^{(i)}}$, where B_i denotes the length of $\vec{b}^{(i)}$.

3 Partitioned Symmetric Searchable Encryption

Partitioned symmetric searchable encryption (PSSE) extends the standard definition of SSE in a natural way: the Search algorithm must be post-processed jointly by a group of N users instead of a single one, in order to identify the document(s) with the corresponding keyword(s) by confirming whether a (set of) keyword(s) belongs to some document or not, with sufficient probability[4]. The protocol works by pre-sharing public parameters between the users and then combining the outcome of their results in a similar way to a distributed PRF [18]. As motivated in Sect. 1, certain scenarios benefit from such a setting.

In its simplest setting, a partitioned protocol requires all users to be honest while declaring their outcome, which jointly validates if a keyword belongs or not to some document. We emphasize that a cheater in the group that deliberately changes the result of his/her finding is tantamount to changing the truthfulness of the global outcome. We proceed with a definition and a security notion for the honest model. Our definition encompasses both static and dynamic SSE schemes.

Definition 5 (Partitioned SSE - Honest Setting). *Let N stand for the number of users. Let $\mathsf{D}_i \subseteq \{0,1\}^*$ denote a datafile, for any $i \in [d]$. Let $\mathsf{DB} = \big\{(i, \mathbf{w}^{(i)})\big\}_{i\in[d]}$ denote the set of pairs containing a datafile index i and a set of keywords $\mathbf{w}^{(i)}$. An N-party PSSE consists of a tuple of algorithms $\big(\mathsf{PSSE.Setup}, \mathsf{PSSE.ClientSetup}, \mathsf{PSSE.Search}, \mathsf{PSSE.Comb}\big)$ such that:*

- $(\mathsf{pp}, \mathsf{DB}_1, \ldots, \mathsf{DB}_N) \leftarrow_{\$} \mathsf{PSSE.Setup}(1^\lambda, \mathsf{DB}, N)$*: is a* PPT *algorithm that takes as input a database of keywords* DB *and a number N of users; it extracts the keywords based on which it generates N individual databases denoted* DB_j *and sends then database* DB_j *to user j; further auxiliary information may be computed and added to* pp *(taken as input by all other algorithms).*
- $(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow_{\$} \mathsf{PSSE.ClientSetup}(\mathsf{DB}_j, j)$*: party j samples a private/public key pair* $(\mathsf{sk}_j, \mathsf{pk}_j)$*. When omitted, the public key is set to $\emptyset$. At this stage, party j encrypts* DB_j *under* pk_j*, obtains* EDB_j *and sends it to the server.*
- $b_j \leftarrow \mathsf{PSSE.Search}(\mathsf{EDB}, \mathsf{sk}_j, \mathbf{w}, \mathsf{I})$*: is a protocol between client j with input its secret key* sk_j*, and the server with input* $\mathsf{EDB} := \cup_{\ell=1}^{N} \mathsf{EDB}_\ell$*. Using* sk_j*, party j can query for a keyword $\mathbf{w}$ in the datafiles with index in* I*. A search query can support one or multiple (conjunctions of) keywords. The* Server *returns to client j a bit b_j indicating if the partial search found $\mathbf{w}$ in any datafile with index in* I *or not.*
- $b \leftarrow \mathsf{Comb}(b_1, \ldots, b_N)$*: after running the* Search *procedure, the parties combine their individual outcomes locally without interaction with the server and generate the final outcome of the search query.*

In addition, we say a partitionable symmetric searchable encryption scheme is dynamic if there exists a fifth algorithm:

[4] Some of the constructions we propose admit false positives, and therefore we require that correctness holds with a good enough probability, rather than having overwhelming/perfect correctness.

– $(\mathsf{EDB}'_j, \sigma') \leftarrow \mathsf{Update}(\mathsf{EDB}_j, \sigma, \mathsf{sk}_j, \mathsf{I}, \mathbf{w}, \mathsf{op})$*: client j encrypts a keyword $\mathbf{w}$ and sends an update query for a specific datafile index set* I*. The operation* op *can be either a delete or insert.*

We require any PSSE *scheme to satisfy correctness, in the sense that for any* $\mathbf{w} \in \{\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(d)}\}$*, the following quantity is negligibly close to 1:*

$$\Pr\left[b \leftarrow \mathsf{Comb}(\{b_j\}) \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow_{\$} \mathsf{PSSE.Setup}(1^\lambda, \mathsf{DB}, N) \wedge \\ \{(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow_{\$} \mathsf{PSSE.ClientSetup}(\mathsf{DB}_j, j)\}_{j \in [N]} \wedge \\ \{b_j \leftarrow \mathsf{PSSE.Search}(\mathsf{EDB}, \mathsf{sk}_j, \mathbf{w}, \mathsf{I})\}_{j \in [N]} \end{array} \right]$$

A PSSE *scheme is adaptive secure if the advantage of any* PPT *adversary in winning the game in Fig. 2 is negligibly close to 1/2.*

We emphasise that a secret sharing scheme would be an option for combining the individual shares of the parties but it does not allow *searching* over the encrypted database for which a SSE is required. PSSE naturally combines both functionalities.

3.1 Dealing with Malicious Users

Real scenarios are more complex to describe, and often contain entities that are able to actively cheat, in the sense that they may want to change the outcome of a search result. To deal with malicious users we modify Definition 5 by giving the server the possibility to access some verification methods. For example, in our PSSE using FE, the server checks the ciphertexts, and the keys. Let this methods be denoted by $\mathsf{VerCT}, \mathsf{VerKey}$, and we assume they are globally accessible. In such a setting, we enforce ClientSetup to return a public key pk_j for each user. Formally, *correctness* can then be described by requiring the following quantity to be negligibly close to 1:

$$\Pr\left[1 \leftarrow \mathsf{Comb}(\{b_j\}) \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow_{\$} \mathsf{PSSE.Setup}(1^\lambda, \mathsf{DB}, N) \wedge \\ \{(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow_{\$} \mathsf{PSSE.ClientSetup}(\mathsf{pp}, j)\}_{j \in [N]} \wedge \\ \{b_j \leftarrow \mathsf{PSSE.Search}(\mathsf{pp}, \mathsf{EDB}, K, j, \mathbf{w})\}_{j \in [N]} \wedge \\ \mathsf{VerCT}(\mathsf{pk}_j, \mathsf{EDB}_j) = 1 \wedge \mathsf{VerKey}(\mathsf{pk}_j, K) = 1 \end{array} \right]$$

4 PSSE Instantiations from FE and Trapdoor Permutation Using BF

In this section we present our PSSE constructions from FE and trapdoor permutation using the functionality of Bloom filters.

4.1 A PSSE Scheme from FE

This part introduces a PSSE protocol based on FE and Bloom filters. The scheme discussed stems from the one by Goh [11]. It differs significantly from the new generation of SSE schemes in the sense that a data structure indexed by datafiles is used, as opposed to recent works that use structures indexed by keywords [4,5]. This is somehow natural, in the sense that Bloom filters are meant to store massive datafiles associated to particular documents. It works in two phases: an *offline* setup phase is responsible for generating the required parameters for each Bloom filter and for inserting all document/keyword sets; and an *online* search protocol partitioned between N clients allowing them to recover the indices of the searched elements and to get, through a combine step (Comb), the final result.

The $\mathsf{SSE.Setup}$ algorithm instantiates a symmetric encryption scheme SE with key K_{SE} to be used to encrypt the (structured) datafiles. A sufficiently large set of hash functions are also sampled at this stage in order to instantiate the Bloom filters to be used. The Setup, given the database of keywords $\mathsf{DB} = \{(i, \mathbf{w}^{(i)})\}_{i\in[d]}$ and the number of users N, proceeds as follows: (1) a set of Bloom filters $\mathsf{BF}^{(i)}$, each consisting in a bit-vector $\vec{b}^{(i)}$, is instantiated. For simplicity, we assume the length of each Bloom filter to be B; (2) each keyword $\mathbf{w}_j^{(i)} \in \mathsf{DB}$ is inserted into $\mathsf{BF}^{(i)}$. Next, the clients proceed as follows: (3) for each Bloom filter $\mathsf{BF}^{(i)}$, the underlying bit-vector $\vec{b}^{(i)}$ is split into l buckets. For simplicity we assume[5] that $l = N$; (4) each bucket j in $\mathsf{BF}^{(i)}$ is encrypted independently using the mpk_j of some party j. Again, for simplicity we assume a canonical association: bucket j corresponds to party j. During the $\mathsf{SSE.ClientSetup}$ phase, each user independently samples a public-key functional encryption scheme supporting inner products: $(\mathsf{msk}, \mathsf{mpk}) \leftarrow_{\$} \mathsf{FE.Setup}$. We note that for the purpose of searching, a linear functional encryption scheme suffices. However, more convoluted constructions [13] may support re-encryption queries and thus they may allow insertions. Each user stores its own secret msk_j and publishes its mpk_j. As the scheme is static, each party encrypts its associated chunk of the Bloom filters and sends the resulting ciphertext to the server, which stores them.

On the *client side*, the Search protocol is given a set of query keywords $\mathbf{W}$. For every queried keyword $\mathbf{w} \in \mathbf{W}$, the client j proceeds as follows: first, it determines the places in some bitvector $\mathbf{b}$ that must be set to 1 by the hash functions. Then, each client looks into its allocated chunk: say user j is allocated chunk j. Finally, the client j derives a functional key for the circuit C_f that checks if all required corresponding bits to $\mathbf{W}$ are set to 1 in chunk j. This is done by the functionality $\mathsf{CircuitCheckBitsAreOne}$: for each bucket in some Bloom filter, if some bits are set to 1, then a single circuit-value checker is built. The output of this circuit consists of a single bit. The server is expected to run $\mathsf{FE.Dec}$ under these circuits (i.e. apply the functional key) for each bucket: $\mathsf{FE.Dec}(\mathsf{sk}_{f_j}, C_j^{(i)}) = 1$.

On the *server side*, the Search protocol will simply evaluate the circuit computing f on the encrypted buckets corresponding to each datafile $\mathsf{D}^{(i)}$ and returns $\mathsf{D}^{(i)}$ if all decryptions succeed in returning 1 for the buckets corresponding to

[5] One can also consider l as a multiple of the number of participants N.

$\mathsf{BF}^{(i)}$. As discussed in Sect. 2.2, false positives are possible hence the identification of datafile(s) corresponding to specific keyword(s) is considered with a reasonable probability. Also, if even a single bit in any chunk is not 1 at the desired place then the queried keyword does not correspond to the datafile searched, hence false negative is not possible. Intuitively, the semantic security of FE should guarantee that nothing is leaked on the message apart from $f(M)$.

Our PSSE Construction from FE

Definition 6 (Basic Construction). *Assuming the existence of sub-exponentially semantic-secure* FE *scheme in the public-key setting, the construction in Fig. 4 is a* PSSE *scheme for multiple keywords.*

```
$\mathsf{PSSE.Setup}(1^\lambda, \mathsf{DB}, N)$:
for $i \leftarrow 1, d$:
    $\mathsf{BF}^{(i)} \leftarrow\$ \mathsf{BF.Setup}(1^\lambda)$
    for $j \in 1, N$:
        $\mathsf{BF}^{(i)}.\mathrm{Insert}(\mathbf{w}_j^{(i)})$
    $L^{(i)} \leftarrow \mathsf{BF}^{(i)}.Split(\vec{b}^{(i)}, N)$
    assert($L^{(i)}$ is a list)
    foreach $x_j^{(i)} \in L^{(i)}$:    // $x_j^{(i)}$ is a bucket
        SendClient($j, x_j^{(i)}$)

$\mathsf{PSSE.ClientSetup}(j, \mathsf{DB}_j = \{x_j^{(1)}, \ldots, x_j^{(d)}\})$:
$(\mathsf{msk}_j, \mathsf{mpk}_j) \leftarrow\$ \mathsf{FE.Setup}(1^\lambda)$
for $i \leftarrow 1, d$:
    $C_j^{(i)} \leftarrow\$ \mathsf{FE.Enc}(\mathsf{mpk}_j, x_j^{(i)})$
SendServer($C_j^{(1)}, \ldots, C_j^{(d)}$)
Store($\mathsf{msk}_j, \mathsf{mpk}_j$)

$\mathsf{PSSE.Search}()$:
ClientSide($j, \mathbf{W}$):
$\mathsf{BF}'.\mathsf{Setup}(1^\lambda)$
foreach $\mathbf{w} \in \mathbf{W}$:
    $\mathsf{BF}'.\mathrm{Insert}(\mathbf{w})$
$x_j \leftarrow \mathsf{BF}'.\mathsf{getChunk}(j)$
$f_j \leftarrow \mathsf{CircuitCheckBitsAreOne}(x_j)$
$\mathsf{sk}_{f_j} \leftarrow\$ \mathsf{FE.KDer}(\mathsf{msk}_j, f_j)$
SendServer($\mathsf{sk}_{f_j}$)

ServerSide($j, \mathsf{mpk}_j, \mathsf{sk}_{f_j}, I$):
$b_j \leftarrow 0$
foreach $i \in I$:
    Parse $(C_1^{(i)}, \ldots, C_N^{(i)})$
    $b_j \leftarrow (\mathsf{FE.Dec}(\mathsf{sk}_{f_j}, C_j^{(i)}) = 1) \vee b_j$
SendClient($b_j$)

$\mathsf{PSSE.Comb}(b_1, \ldots, b_N)$:
return $b_1 \wedge \ldots \wedge b_N$
```

Fig. 4. A PSSE scheme using a functional encryption schemes as an underlying primitive.

Correctness. Assuming a query set $\mathbf{W}$ is to be checked, for each index i (corresponding to the datafile $\mathsf{D}^{(i)}$), the client computes the positions of 1s as pointed by some pseudorandom function[6]. Each position will be part of some bucket. The client builds the appropriate circuits to check if the required bits in bucket x are set to 1.

The server uses the functional decryption procedure[7] for each bucket and then for each document. If the FE decryption returns 1 for all positions pointed by the hash functions, then $\mathsf{D}^{(i)}$ contains the searched key with high probability.

[6] In some sense we want to preserve the idea behind the Bloom filter construction, and work with hash functions having pseudorandom outputs.

[7] Note however that this step is highly parallelizable.

Lemma 1. *If the static* PSSE *in Fig. 4 is built on a semantic secure functional encryption scheme* FE *supporting a bounded number of functional keys, then it is* $\mathsf{PSSE}^{\mathcal{A}}_{\mathsf{PSSE}}$*-secure (Fig. 2).*

Proof (Lemma 1). We construct the simulator $\mathcal{S}_{\mathsf{PSSE}}$ described in Definition 5 using the simulator of the underlying FE scheme. Namely, during the $\mathsf{PSSE.Setup}$ procedure, the pp are computed, consisting of the hash functions used to instantiate the Bloom filters, which are then handed to $\mathcal{S}_{\mathsf{PSSE}}$, which outputs them. For the $\mathsf{ClientSetup}$ case, N $\mathcal{S}_{\mathsf{FE}}$ simulators are instantiated, and $\mathcal{S}_{\mathsf{PSSE}}$ simply returns as EDB the ciphertexts it receives from the N simulators. During the Search procedure, the leakage function obtained from the functional keys (and the ciphertexts) that are exchanged between the clients and the server, consists only in the $\mathsf{FE.Dec}(\mathsf{sk}_f, C)$, which is also leaked by the real experiment. Thus, the two settings are indistinguishable. □

We observe that an inner-product FE scheme is sufficient for our purpose. For instance, if a bucket in some bitvector contains m positions set to 1, we issue a functional key for exactly the same bucket and check if its output is m (assuming m is small).

4.2 PSSE from Trapdoor Permutation: PSSE from $\Sigma o\phi o\varsigma$

This section proposes a forward secure SSE scheme supporting a partitioned search amongst N honest users.

$\Sigma o\phi o\varsigma$**.** The starting point of our proposal is $\Sigma o\phi o\varsigma$ [4]. The construction is keyword-indexed and easy to follow. It uses a master secret key, denoted as K_S to derive a keyword key $K_{\mathbf{w}}$ for each keyword $\mathbf{w}$ of interest using a PRF:

$$K_{\mathbf{w}} \leftarrow \mathsf{PRF}(K_S, \mathbf{w}),$$

where $K_{\mathbf{w}}$ is used in conjunction with some randomly sampled search token ST_0 from the space of admissible tokens $\mathcal{M}$ in order to attain the first insertion of $\mathbf{w}$ in the encrypted database corresponding to some document with index in $\vec{\mathsf{I}}$. Then, additional queries will generate new search tokens ST_c. Concretely, the server maintains a table $\mathbf{T}$ with lines corresponding to users and rows corresponding to hash value $\mathsf{Hash}_1(K_{\mathbf{w}}||ST_c)$ containing the hidden value $(\mathsf{Hash}_2(K_{\mathbf{w}}||ST_c) \oplus \vec{\mathsf{I}})$, while the client maintains a table $\mathbf{T}^{(j)}$ saving the relation between search tokens ST_c and $\mathbf{w}$. To insert $\mathbf{w}$ in any new document with index i_c, a new search token ST_c is derived from ST_0. This happens by the means of a trapdoor permutation Π_c:

$$ST_c \leftarrow \Pi_c^{-1}(ST_0)$$

obtained through the repeated application of a pseudorandom permutation PRP.

On the server side, things are remarkably simple. Once the client sends $K_{\mathbf{w}}$ and some search token ST_c, the server recomputes the hash values $\mathsf{Hash}_1(K_{\mathbf{w}}||ST_c)$, $\mathsf{Hash}_2(K_{\mathbf{w}}||ST_c)$ and obtains the index. If more than one index has been inserted per keyword $\mathbf{w}$, then all search tokens can be recovered, simply by taking:

$$ST_0 \leftarrow \Pi_c(ST_c)$$

and the previous step is applied. Intuitively, $\Sigma o\phi o\varsigma$ enjoys forward security for the following reason: whenever a new (keyword, index) pair is inserted via the hash function, the server is not able to derive the search token that is used. Thus it can't say which keyword has been inserted, or to which document it belongs to.

A Forward Secure Partitioned SSE Scheme. We aim at having a scheme handling update queries while reaching forward security. The gist of the modifications we make on $\Sigma o\phi o\varsigma$ is to replace the index that identifies the document, with a more convenient data structure – a Bloom filter. Then, we allocate to each party a chunk over the bitvector associated to BF. That is, whenever a keyword is to be inserted at some location, a new Bloom filter is instantiated. Party j will compute independently its search tokens and will XOR the j^{th} chunk of BF. The server will store all these chunks in a 2-dimensional array with rows corresponding to the N possible parties and columns to hashes. Whenever queried by a client for some keyword $\mathbf{w}$, it will identify and return the value of the BF chunk. In the Comb protocol, the parties would be able to recreate the entire BF and check if the words are included or not. For technical reasons, we employ the usage of several hash functions (impersonating random oracles) in order to obtain a joint computation corresponding to the value during the insert and the search phases. An algorithmic description of our scheme can be found in Fig. 5.

```
$\mathsf{PSSE.Setup}(1^\lambda, \mathsf{DB}, N)$:
  $\mathsf{Hash}_1.\mathsf{Setup}(1^\lambda)$
  $\mathsf{Hash}_2.\mathsf{Setup}(1^\lambda)$
  $\mathbf{T} \leftarrow \emptyset$
  for $j \in [N]$
    $\mathsf{Hash}^j.\mathsf{Setup}(1^\lambda)$
  ServerStore($\mathbf{T}$)
  return

$\mathsf{PSSE.ClientSetup}(j, \emptyset)$:
  $\mathbf{T}^{(j)} \leftarrow \emptyset$
  $K^{(j)} \leftarrow\$ \mathsf{PRF.Setup}(1^\lambda)$
  $(\mathsf{sk}^{(j)}, \mathsf{pk}^{(j)})$
      $\leftarrow\$ \mathsf{PRP.Setup}(1^\lambda)$
  ClientStore($j, K^{(j)}, \mathsf{sk}^{(j)},$
      $\mathsf{pk}^{(j)}, \mathbf{T}^{(j)}$)
  return

$\mathsf{PSSE.Update}(\mathbf{w}, j, \mathsf{op})$:
  ClientSide($j, K^{(j)}, \mathsf{sk}^{(j)}, \mathsf{pk}^{(j)}, \mathbf{w}$):
  BF.InitAndInsert($\mathbf{w}||\vec{\mathbf{I}}$)
  $K_\mathbf{w} \leftarrow \mathsf{PRF.Eval}(K^{(j)}, \mathbf{w})$
  $(\mathsf{ST}_c, c) \leftarrow \mathbf{T}^{(j)}[\mathbf{w}]$
  if $(\mathsf{ST}_c, c) = \perp$:
    $\mathsf{ST}_0 \leftarrow\$ \mathcal{M}$
    $\mathbf{T}^{(j)}[\mathbf{w}] \leftarrow (\mathsf{ST}_0, 0)$
  else:
    $\mathsf{ST}_{c+1} \leftarrow \mathsf{PRP}^{-1}(\mathsf{ST}_c)$
    $\mathbf{w}^{(j)}[\mathbf{w}] \leftarrow (\mathsf{ST}_{c+1}, c+1)$
  $h_1 \leftarrow \mathsf{Hash}_1(K_\mathbf{w}||\mathsf{ST}_{c+1})$
  $h_2 \leftarrow \mathsf{Hash}_2(K_\mathbf{w}||\mathsf{ST}_{c+1})$
  SendServer($j$, ($h_1$,
      $h_2 \oplus \mathsf{BF.getChunk}[j]$))

  ServerSide($j, v$):
  $\mathbf{T}[j][h_1] \leftarrow (h_2 \oplus \mathsf{BF.getChunk}[j])$

$\mathsf{PSSE.Search}()$:
  ClientSide($j, K^{(j)}, \mathsf{sk}^{(j)}, \mathsf{pk}^{(j)}, \mathbf{w}$):
  $K_\mathbf{w} \leftarrow \mathsf{PRF.Eval}(K^{(j)}, \mathbf{w})$
  $(\mathsf{ST}_c, c) \leftarrow \mathbf{T}^{(j)}[\mathbf{w}]$
  if $(\mathsf{ST}_c, c) = \perp$:
    return 0
  else:
    SendServer($j, K_\mathbf{w}, \mathsf{ST}_c, c$)

  ServerSide($j, K_\mathbf{w}, \mathsf{ST}_c, c$):
  while $c \geq 0$:
    $h_1 \leftarrow \mathsf{Hash}_1(K_\mathbf{w}||\mathsf{ST}_c)$
    $h_2 \leftarrow \mathsf{Hash}_2(K_\mathbf{w}||\mathsf{ST}_c)$
    SendClient($j, h_2 \oplus \mathbf{T}[j][h_1]$)
    $\mathsf{ST}_{c-1} \leftarrow \mathsf{PRP}(\mathsf{ST}_c); c \leftarrow c - 1$

$\mathsf{PSSE.Comb}(v^{(1)}, \ldots, v^{(N)})$:
  BF.Init($v^{(1)}, \ldots, v^{(N)}$)
  return BF.Search($\mathbf{w}$,
      $\mathsf{Hash}^1, \ldots, \mathsf{Hash}^n) \overset{?}{=} 1$
```

Fig. 5. A construction of forward-private PSSE based on trapdoor permutations.

Forward Security of PSSE. We show the partitioned $\Sigma o\phi o\varsigma$ achieves forward security. To this end, we mainly adapt the proof given by Bost in his paper to our partitioned version in Fig. 5.

Lemma 2 (Forward Security). *Let $\mathcal{S}_S$ denote the simulator used in the forward-security of [4]. The partitioned* SSE *scheme in Fig. 5 achieves partitioned forward security against any* PPT *adversary $\mathcal{A}$, as defined in Definition 2 under the advantage* $\mathbf{Adv}^{FW}_{\Sigma,\mathcal{R}}(\lambda) \leq N \cdot \mathbf{Adv}^{FW}_{\Sigma o\phi o\varsigma,\mathcal{A}}(\lambda)$.

Proof (Proof Sketch). Observe that scheme Σ in Fig. 5 can be viewed as N parallel executions of $\Sigma o\phi o\varsigma$. A key difference consists in the indices that are passed by each of the users whenever a keyword is inserted. Naturally, we would like to re-use the already proven forward security in order to attain the same property for our partitioned scheme. In doing so, we proceed using a hybrid argument. A new hybrid game corresponds to simulating the output of the i^{th} client using the simulator put forward by Bost in [4].

Game$_0$: corresponds to the real experiment in the security game $\mathsf{PSSE}^{\mathcal{A}}_{\mathsf{PSSE}}$.

Game$_i$: we use the simulator in [4] to generate the transcript. The distance to the previous game is bounded by the forward security of the scheme in [4].

Game$_N$: is identical to the simulated experiment in Defintion 6. □

```
PSSE.Setup(1^λ, DB, N):
  Hash_1.Setup(1^λ)
  Hash_2.Setup(1^λ)
  T ← ∅
  for j ∈ [N]
    Hash^j.Setup(1^λ)
  ServerStore(T)

PSSE.ClientSetup(j, ∅):
  T^(j) ← ∅
  (K^(j), vk^(j)) ←$ VRF.Setup(1^λ)
  (sk^(j), pk^(j)) ←$ PRP.Setup(1^λ)
  ClientStore(j, K^(j), sk^(j), pk^(j), T^(j))

PSSE.Update(w, j, op):
  ClientSide(j, K^(j), sk^(j), pk^(j), w):
  BF.Insert(w||I⃗)
  (K_w, π_w) ← VRF.Eval(K^(j), vk^(j), w)
  (ST_c, c) ← T^(j)[w]
  if (ST_c, c) = ⊥:
    ST_0 ← M
  else:
    ST_{c+1} ← PRP^{-1}(ST_c)
  T^(j)[w] ← (ST_{c+1}, c + 1)
  h_1 ← Hash_1(K_w||ST_{c+1})
  h_2 ← Hash_2(K_w||ST_{c+1})
  SendServer(h_1, h_2 ⊕ BF.getChunk[j])

  ServerSide(j, v):
  T[j][h_1] ← (h_2 ⊕ BF.getChunk[j])

PSSE.Search():
  ClientSide(j, K^(j), sk^(j), pk^(j), w):
  (K_w, π_w) ← VRF.Eval(K^(j), pk^(j), w)
  (ST_c, c) ← T^(j)[w]
  if (ST_c, c) = ⊥:
    return 0
  else:
    SendServer(j, K_w, π_w, ST_c, c)

  ServerSide(j, K_w, π_w, ST_c, c):
  while c > 0:
    if VRF.Ver(vk^(j), K_w, π_w, w) ≠ 1:
      abort
    h_1 ← Hash_1(K_w||ST_c)
    h_2 ← Hash_2(K_w, ||ST_c)
    SendClient(j, h_2 ⊕ T[j][h_1])
    c ← c − 1

PSSE.Comb(v^(1), ..., v^(N)):
  BF.Init(v^(1), ..., v^(N))
  return BF.Search(w, Hash^1, ..., Hash^N) =? 1
```

Fig. 6. A candidate construction of forward-private PSSE based on trapdoor permutations and verifiable random functions (VRFs) for handling malicious users.

4.3 Dealing with Malicious Users

Far more challenging is the scenario that deals with malicious users. A generalization of the preceding scheme may also deal with this additional problem by replacing pseudorandom functions in Fig. 5 by verifiable random functions (VRF). A VRF consists of a setup phase – generating the public parameters, an *evaluation* algorithm – returning an evaluation and a proof for a given input, and a verification algorithm – validating the outcome of the evaluation process. We refer to [14] for the formal definition of VRFs. The verifiability property allows the server to check if a keyword key $K_{\mathbf{w}}$ has been correctly generated by the client and to abort the search otherwise. We present a candidate construction of forward-private PSSE based on trapdoor permutations in Fig. 6.

5 Conclusion

In this work, we proposed a new variant of searchable encryption schemes (SSE). We call it partitioned SSE (PSSE) where datafiles can only be found whenever all search parties give a collective search request. This combines the best parts of searchable encryption and secret sharing. We accompany this new variant with two pragmatic schemes, one based on functional encryption and one on $\Sigma o\phi o\varsigma$. Additionally, we showed how the latter scheme can be used in the presence of malicious users.

Acknowledgements. Jim Barthel was supported in part by the Luxembourg National Research Fund through grant PRIDE15/10621687/SPsquared. The last two authors were supported by ERC Advanced grant CLOUDMAP 787390.

References

1. Ananth, P., Jain, A., Jin, Z., Malavolta, G.: Multi-key fully-homomorphic encryption in the plain model. In: Pass, R., Pietrzak, K. (eds.) TCC 2020. LNCS, vol. 12550, pp. 28–57. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64375-1_2
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
3. Boneh, D., Sahai, A., Waters, B.: Functional encryption: definitions and challenges. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 253–273. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19571-6_16
4. Bost, R.: $\Sigma o\phi o\varsigma$: forward secure searchable encryption. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S., (eds.) ACM CCS 16, pp. 1143–1154. ACM Press, October 2016
5. Bost, R., Minaud, B., Ohrimenko,O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D., (eds.) ACM CCS 17, pp. 1465–1482. ACM Press, October/November 2017
6. Cash, D., et al.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: NDSS 2014. The Internet Society, February 2014

7. Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005). https://doi.org/10.1007/11496137_30
8. Chum, C.S., Zhang, X.: A new bloom filter structure for searchable encryption schemes. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 143–145 (2017)
9. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Juels, A., Wright, R.N., De Capitani di Vimercati, S. (eds.) ACM CCS 06, pp. 79–88. ACM Press, October/November 2006
10. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) 41st ACM STOC, pp. 169–178. ACM Press, May/June 2009
11. Goh, E.-J.: Secure indexes. Cryptology ePrint Archive, Report 2003/216 (2003). http://eprint.iacr.org/2003/216
12. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM (JACM) **43**(3), 431–473 (1996)
13. Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) 45th ACM STOC, pp. 555–564. ACM Press, June 2013
14. Hofheinz, D., Jager, T.: Verifiable random functions from standard assumptions. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9562, pp. 336–362. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49096-9_14
15. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_22
16. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 12, pp. 965–976. ACM Press, October 2012
17. Mishra, P., Poddar, R., Chen, J., Chiesa, A., Popa, R.A.: Oblix: an efficient oblivious search index. In: 2018 IEEE Symposium on Security and Privacy, pp. 279–296. IEEE Computer Society Press, May 2018
18. Naor, M., Pinkas, B., Reingold, O.: Distributed pseudo-random functions and KDCs. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 327–346. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48910-X_23
19. Patranabis, S., Mukhopadhyay, D.: Forward and backward private conjunctive searchable symmetric encryption. In: NDSS Symposium 2021 (2021)
20. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, pp. 44–55. IEEE Computer Society Press, May 2000
21. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS 2014. The Internet Society, February 2014
22. Sun, S.-F., Liu, J.K., Sakzad, A., Steinfeld, R., Yuen, T.H.: An efficient non-interactive multi-client searchable encryption with support for Boolean queries. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9878, pp. 154–172. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45744-4_8
23. Yao, A.C.-C.: Protocols for secure computations (extended abstract). In: 23rd FOCS, pp. 160–164. IEEE Computer Society Press, November 1982