Multi-User Dynamic Searchable Symmetric Encryption with Corrupted Participants

Javad Ghareh Chamani, Yun Wang, Dimitrios Papadopoulos, Mingyang Zhang, and Rasool Jalili

Abstract—We study the problem of multi-user dynamic searchable symmetric encryption (DMUSSE) where a data owner stores its encrypted documents on an untrusted remote server and wishes to selectively allow multiple users to access them by issuing keyword search queries. Specifically, we consider the case where some of the users may be corrupted and *colluding with the server* to extract additional information about the dataset (beyond what they have access to). We provide the first formal security definition for the dynamic setting as well as forward and backward privacy definitions. We then propose μSE, the first provably secure DMUSSE scheme and instantiate it in two versions, one based on oblivious data structures and one based on update queues, with different performance trade-offs. Furthermore, we extend μSE to support verifiability of results. To achieve this, users need a secure digest initially computed by the data owner and changed after every update. We efficiently accommodate this, without relying on a trusted third party, by adopting a blockchain-based approach for the digests' dissemination and deploy our schemes over the permissioned Hyperledger Fabric blockchain. We prototype both versions and experimentally evaluate their practical performance, both as stand-alone systems and running on top of Hyperledger Fabric.

Index Terms—Dynamic Multi-User Searchable Symmetric Encryption, Secure Cloud Computing, Data Outsourcing

1 Introduction

Data sharing and outsourcing has emerged as an essential component of modern computing environments such as cloud and collaborative computing. However, data security and privacy issues are amongst the greatest concerns for data owners that wish to participate in such services [23]. Data encryption can solve this problem, however, it limits one's ability to compute on encrypted data, unless one is willing to use techniques such as fully homomorphic encryption [24], that significantly hamper performance.

One fundamental task in this setting is that of *searching on encrypted data*. E.g., consider the scenario where a data owner wants to outsource a database of files and retain the ability to search for files that contain specific keywords afterward. *Searchable symmetric encryption (SSE)*, originally proposed by Song et al. [45], aims to solve this problem in an efficient manner by slightly relaxing the strict privacy requirements. It allows the server to directly search on the encrypted data after receiving a query, at the cost of learning some *leakage* information, such as *search pattern* (which search queries refer to the same keyword—but without explicitly revealing the keyword) and *access pattern* (which files are returned for a query). There has since been a plethora of works in SSE, e.g., improving its efficiency, or handling more expressive queries and dynamic datasets (see Section 2).

However, most existing works consider the single-user setting where only *one data owner* uploads data to the server and is the only one that can subsequently issue queries. Although this can be useful for individual data outsourcing, it cannot readily be applied in other real-world scenarios. Consider the setting where multiple institutions wish to collaborate by sharing their data securely (e.g., hospitals that want to share patients' data). For such scenarios where one or more data owners upload their data and wish to allow other participants to query (a subset of) it, we need a stronger cryptographic primitive (Figure 1). *Multi-user searchable symmetric encryption (MUSSE)*, originally proposed by Curtmola et al. [16], is applicable to such cases.

MUSSE with Corrupted Participants. The majority of MUSSE schemes from the literature consider the case where *all users are honest and mutually trusting* and the only adversary is the server who wishes to learn as much as possible about the encrypted dataset. This simplifies things as the only goal is to mitigate the leaked information to the server. However, this assumption is too strong for many applications where corrupted or compromised users may collaborate with the server in order to extract information about parts of the dataset not shared with them, or learn the keywords searched by other users. As demonstrated by the works of Van Rompay et al. [54] and Grubbs et al. [27], this can lead to catastrophic *cross-user leakage-abuse* attacks.

In order to address this issue, some works introduced multi-user schemes that achieve security against *corrupted participants*, a term that refers to users that are corrupted by or collude with the adversary (who controls the server). Note that this is a strictly stronger threat model compared to "standard" searchable encryption that considers only the server to be controlled by the adversary and it captures a broad number of adversarial settings. E.g., users being hacked by the adversary that has also compromised the server, or misbehaving users that collude with the server

- *J.Ghareh Chamani, Y.Wang, and D.Papadopoulos are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. E-mail:*{jgc@cse, ywangik@cse, dipapado@cse}.ust.hk
- *J.Ghareh Chamani, and R.Jalili are with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. E-mail:*{gharehchamani@ce, jalili@}.sharif.edu
- *M.Zhang is with Poisson Lab, Huawei Technologies, Beijing, China. E-mail:*zhangmingyang4@huawei.com

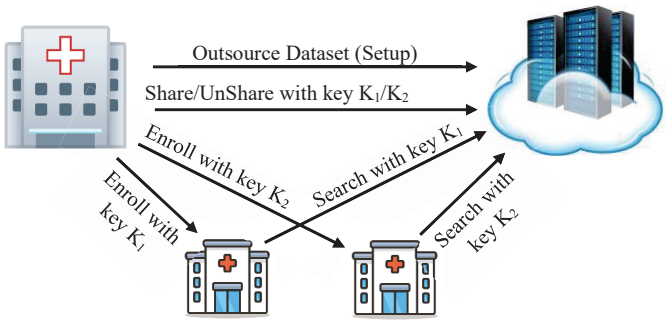


Fig. 1: MUSSE model for patients' datasharing

in order to extract information about the dataset that cannot be inferred by what they have already been granted access to. In the related literature, this "extended" threat model is referred to by different terms, such as client-server collusion (e.g., [27], [42], [43]) or corrupted users [41], whereas other works use the term corrupted, colluding, or adversarially-controlled users interchangeably [28], [42], [54], [55].[1] In terms of security formulation, all these correspond to a definition that allows the adversary to have access to the internal state of corrupted/colluding users (see Section 4).

Popa et al. [42], [43] introduced the notion of "cross-user" leakage, however, their security definition is susceptible to the attack of [27]. Hamlin et al., [28] strengthened this definition to *eliminate* cross-user leakage and proposed a lightweight construction; more recently Su et al. [48] extended this to also achieve verifiability of results. However, both these works only work for *static* datasets, i.e., there is no efficient secure way to handle data insertions or deletions after the initial setup phase. The same limitation holds for the recent work of [61] and other works in the area that moreover are either rather inefficient due to the use of expensive cryptographic primitives [51], [56], [66], or assume a different setting with multiple non-colluding servers [55]. Finally, Patel et al. [41] proposed a different approach by limiting (but not eliminating) cross-user leakage, but their scheme cannot support dataset updates either. Moreover, its cross-user leakage is considerable: If a user has access to the same document as a compromised party, its queried keywords from this document can be leaked to the server.

Dynamic Searchable Encryption. Dynamic searchable encryption schemes can efficiently support updates on the outsourced database (without re-running the "expensive" setup), which is a necessary property for most real-world applications. Besides the efficiency aspect, supporting updates introduces additional security concerns regarding information that can be leaked due to updates. Two security properties have been proposed for dynamic schemes: (i) schemes that do not reveal whether an update operation is related to previously searched keywords are called *forward-private* [13], [46], and schemes that do not reveal during a search that files that contained the searched keyword were deleted since the last search are called *backward-private* [7], [46]. Developing efficient dynamic SSE schemes that achieve these two important properties has attracted significant attention in the literature (e.g., [7], [17], [25], [50]). However, to the best of our knowledge, there is no previous work in *dynamic MUSSE (DMUSSE) with corrupted participants* that considers or formally defines them.

Verifiable MUSSE. One additional property for SSE schemes is *verifiability* [6], [15], i.e., that a misbehaving server cannot return a fake search result (by omitting or erroneously adding documents). For dynamic datasets, this also implies freshness (i.e., the effect of all previous updates is accounted for). While there are many works that study this property, a straight-forward manner that has been heavily adopted is via deploying an authenticated data structure like a Merkle tree [37] on top of the encrypted index stored by the server. Search results can be verified given the Merkle-root *digest* and the corresponding tree paths.

For the single-user case, verifiability can be achieved in this manner, even in the presence of updates, as the owner just locally computes and stores the new digest. However, verifiable DMUSSE poses the additional difficulty that the new digest has to be transmitted to all the users. Doing this via direct communication may not be feasible (e.g., users are not guaranteed to always be online). On the other hand, if the server is used as an intermediary to store the new digest so that users can download it later, it can give outdated digests to the users, e.g., ignoring recent updates. As far as we know, existing works on verifiable MUSSE with corrupted parties do not handle updates—in which case, it is much easier to verify results as the digest is computed only once at the setup phase.

1.1 Our Results

In this paper, we focus on the problem of multi-user dynamic searchable encryption with corrupted participants. As explained above, none of the existing MUSSE schemes in this setting can support dynamic datasets. Hence, we first provide a necessary new security definition for DMUSSE (Section 4), including appropriate leakage functions for forward and backward privacy. Then, we propose μSE[2] a novel scheme that achieves our security definition (Section 5). We instantiate μSE in two versions (O-μSE and Q-μSE) with different efficiency trade-offs. To the best of our knowledge, our constructions are *the first forward and backward-private DMUSSE schemes secure against corrupted participants*. We also extend μSE to achieve verifiability using the standard secure digest approach (e.g., see [6]). To solve the problem of efficient digest dissemination, we use a blockchain protocol. After each update, the owner issues a transaction with the new digest and users can retrieve it from the blockchain and use it for result verification. Since in our target setting all users are known to the data owner, a permissioned blockchain is a natural candidate; in our implementation we use Hyperledger Fabric [21] (Section 6).

Experimental Evaluation. We evaluate the performance of both μSE versions via two sets of experiments (Section 7). First, we measure pure computation time to estimate their computational demands. Then, to benchmark them in a real application scenario, we set up five AWS cloud machines instantiating multiple users and the data server and we mea-

1. A similar notion of user-server collusion has been studied in the context of server-aided multi-party computation, e.g., [3], [30].

2. Pronounced *muse* (/myüz/). In ancient Greece, a Muse was one of the nine inspirational goddesses of arts and science.

sure end-to-end time, including communication overhead, Fabric blockchain overhead, and computation time.

Our results show that both our schemes have low computational overhead. In terms of search time, they take <46ms for retrieving a result of 100 document id's and <111ms for retrieving a large result of 10K id's from a dataset of 1M records. Updates are also quite fast as it takes less than 5ms of computation to add/remove a keyword to/from a file. Since there is no previous DMUSSE secure against corrupted parties for comparison, we compare our schemes' search performance against previous *static* ones. Surprisingly, our results show that our schemes are up to 3 and 5 orders of magnitude faster than the static schemes MKSE [28] and mx-u [41], respectively (while the latter also suffers from cross-user leakage).

Regarding end-to-end times, our experiments show that the blockchain overhead (issuing transactions) is the main bottleneck for searches with small result sizes (e.g., Fabric operations take >85% of the execution time). As the result size increases, the ratio changes (e.g., Fabric takes 31% and 86% of Q-μSE and O-μSE search time for 10K result size in a dataset of 1M records). Overall, Q-μSE is more efficient in our end-to-end experiments. That said, our evaluation shows that each one may be suitable for specific use cases: Q-μSE for active users with frequent searches while O-μSE in settings with infrequent operations and when small permanent client storage is necessary (Section 7.4).

1.2 Overview of Techniques & Challenges

Here, we provide an overview of our techniques and describe the challenges we faced.

Defining security for DMUSSE with corruptions. Our definition extends the one recently proposed by Patel et al. [41] for the static setting. Even though our definition can tolerate cross-user leakage, we stress that our constructions do not have such leakage. Moving from the static to the dynamic setting introduces challenges because the adversary may issue update queries at arbitrary times (intermixed with searches) to infer additional information. Moreover, compared to the single-user definition (e.g., [5]) we need to consider additional sharing or "unsharing" of documents. To capture security in this setting, we adopt a reactive real/ideal game approach (Definition 1). Furthermore, we define forward and backward privacy for the multi-user setting by identifying appropriate leakage functions. Our definitions can be viewed as the analogue of the ones proposed in [7] for the single-user setting.

Building multi-user schemes from single-user ones. Our main idea for building μSE is to start from an efficient single-user scheme and replicate it multiple times, once for each user. In this way, cross-user leakage is avoided as each deployment uses a different key, at the cost of increased storage at the server (which we believe can be a reasonable trade-off, given the low cost of storage). Our selected scheme is MITRA from [25] which is dynamic and forward/backward private. A similar approach is taken in [28] but that scheme can only handle static datasets. Moreover, it requires a separate query for every document in order to search for a keyword, hence, as shown in Section 7 μSE achieves significantly faster search operations.

Handling updates securely. While this replication approach seems to achieve our main security goal, there is one important issue that stops us from using MITRA as is, "off the shelf". The problem is that in single-user schemes during updates the owner/user simply has to modify its local state whereas in DMUSSE communication between the owner and all the affected users is necessary. If we assume such direct communication is always possible, then indeed MITRA can be used for μSE in a black-box manner. However, in practice direct communication may be infeasible (e.g., it is unreasonable to assume users are "always online").

Hence, we need to modify the way MITRA handles updates, storing update information at the server so that users can retrieve it prior to their next query. Done naively this would introduce additional leakage. Somewhat informally, MITRA requires that the user keeps for each keyword a *counter* of how many times it has been inserted/removed, which changes after every update. If the server could deduce which keyword counter is altered by an update this would compromise the forward privacy of our μSE schemes. To address this, we propose two alternative DMUSSE constructions based on MITRA that deviate in how they handle updates: i) O-μSE, and ii) Q-μSE.

O-μSE: Storing counters obliviously at the server. In O-μSE, we instantiate the array of keyword counters of MITRA via a separate *oblivious map (OMAP)* [60] for each user, stored at the server. This is an oblivious key-value dictionary, that entirely hides the access patterns from the server. During updates, the data owner updates the necessary OMAPs on the server (incrementing counters). During a search, the user first accesses the correct keyword counter from its OMAP and then proceeds to run the search. In this manner, the clients need tiny local storage as the counters are stored obliviously at the server. For our scheme, we use the OMAP of [60] which is based on the classic PathORAM of [47]. One issue that we face when deploying it is that it is an OMAP "with stash", a small state that needs to be maintained locally and accessed during every OMAP access. In our setting, each OMAP is accessed by both the data owner and its respective user, hence this stash needs to be transferred. To avoid direct communication we store this stash at the server, downloading it before every operation (by the respective party) and uploading it again afterwards. This has no effect in the asymptotics of O-μSE and minimal effect in practice (the fixed-upper-bound stash is just a few KB in our evaluation).

Q-μSE: Queue-based μSE. The downside of O-μSE is that operations are somewhat slowed down due to the OMAP access. With Q-μSE we take a different approach, requiring users to store keyword counters locally. The server stores an encrypted *update queue* for each user. During updates, the data owner appends to the users' queues the keyword and the incremented counter. Then, during a search the user first "flushes" its queue, increments its local counters, and then runs the query. Compared to O-μSE, this construction can have faster search and updates (especially in high latency networks as the oblivious map of O-μSE requires multiple roundtrips) as long as a user interacts frequently and its queue does not grow too long, at the cost of increased local storage. Both constructions are proven secure and for-

ward/backward private as per our definitions. We discuss the performance trade-offs they achieve in Section 7.4.

Verifiability & Deployment on Hyperledger Fabric. Making our schemes verifiable is rather straight-forward by creating a Merkle-ized version of the involved data structures. For O-μSE this entails the OMAP (which is already in a "nice" tree structure as it relies on PathORAM). For Q-μSE, we build a Merkle tree for a user's queue with the updates sequentially stored at the leafs. In this way, no matter how many updates the user is "missing" the proofs consist only of two paths. In both cases, a Merkle tree is deployed over the MITRA dictionary that stores the actual entries.

However, the challenge of how to securely propagate the Merkle root after each update remains. We solve this problem using a permissioned blockchain. After every update, the new digest is posted to the Hyperledger Fabric [21] blockchain via a transaction by the data owner. Prior to a search, users can securely retrieve the latest digest via the blockchain. In this way, result verifiability is guaranteed from the blockchain protocol's consensus security. Note that the blockchain is only used for digest retrieval. The encrypted dataset is still stored off-chain at the untrusted server, as storing it on-chain would be inefficient.

2 Related Work

Dynamic searchable encryption has been extensively studied, both in the single-user (e.g., [5], [17], [25], [31], [46]) and the multi-user (e.g., [62], [63], [67]) setting. However, none of the existing multi-user schemes are secure against corrupted users colluding with the server, neither has this property been defined in the dynamic setting. Multi-user searchable symmetric encryption was defined by Curtmola et al. [16] and numerous subsequent works have revisited and improved it in several aspects (e.g., [1], [32], [33], [35], [49], [53], [56], [59]). However, only a handful of them remain secure in the presence of corrupted and colluding users [28], [41], [48], [55], [56], [66] and all of them work only for static datasets. Furthermore, with the exception of [28] none of them explicitly considers the sharing of different subsets of the dataset with different users.

Backward privacy was formally defined in [7] and subsequent works proposed improved constructions (e.g., [17], [25], [50]) in the single-user setting.

Verifiability for SSE has been the focus of many works (e.g., [6], [12], [48], [65]). We adopt the "black-box" approach of building a "Merkle-ized" version of the encrypted dataset (e.g., see [6]). The interplay between oblivious primitives and SSE has also been studied extensively [7], [17], [18], [25], [40]. Another line of work combines keyword search with blockchain protocols to guarantee fairness of payments [9], [29], [65]. Finally, other works have studied SSE for more expressive queries [11], [19] and public-key searchable encryption [4]; both are beyond the scope of this paper.

Relation to Multi-key Searchable Encryption. The notion of multi-key searchable encryption was first introduced by Popa et al. [42], [43] to refer to multi-user searchable encryption systems where data is encrypted with different keys per user. It explicitly requires that a user's query can be submitted as a constant-size token even when multiple different datasets (owned by different parties) are shared with her. Hence, while in principle a multi-key searchable encryption can be built by running multiple instances of searchable encryption, this would increase the search token size to linear in the number of datasets shared with the user so it does not satisfy this efficiency requirement. The first multi-key searchable encryption construction [43] was based on elliptic curves with bilinear pairings, however, that construction (and inherently that definition) was shown to be susceptible to leakage-abuse attacks [27].

Since then there have been a few follow-up works that proposed alternative formulations, the most recent of which was by Hamlin et al. [28] (also adopted in [48]). Comparing that definition with ours (Definition 1) in terms of security, we note that the one from [28] is weaker as it is indistinguishability-based and selective-only (i.e., the adversary must declare all its queries before receiving any other information), whereas ours is simulation-based and adaptively secure. However, their model integrates multiple data-owners that may possibly also be corrupted (collaborating with the server to learn the contents of users' queries). On the contrary, we assume that the data owner is honest.

The same paper introduces two schemes, an efficient one based on PRFs and another based on indistinguishability obfuscation that is mostly of theoretical interest at this point. Interestingly, similar to our μSE, their first scheme is also based on replication as it maintains a separate encoding of each document for each user. The difference is that searching for a keyword requires a separate lookup for each document, whereas with μSE this takes only one lookup per previous insertion/deletion for this keyword. This can make a big difference in search efficiency as shown in our experimental evaluation (Section 7). Apart from this, we believe that, when restricted to the static setting, both schemes can be shown to satisfy the same security properties with some modifications, mainly to replace PRFs with a hash function modeled as a random oracle for both of them. To make our search token constant-size and achieve security against server-owner collusion, we can moreover use the "classic" token trick of [10] in the random oracle model (already mentioned for MITRA in [25]) and run all sharing phases locally at the user instead of the owner, as in [28].

TEE-assisted Searchable Encryption. A relatively recent line of work tries to build searchable encryption by relying on *trusted execution environments*, such as Intel SGX (e.g., [2], [14], [22], [38], [57], [58]—a comparative survey can be found in [64]). From a security perspective, using such TEE one needs to consider possible sources of information leakage, such as memory access patterns and other side-channels [8], [34]. In particular, one aspect of leakage in TEE-assisted searchable encryption is leakage due to enclave memory accesses. At a high level, to avoid meaningful leakage all the executed algorithms need to be oblivious which may introduce additional overhead. Indeed, some existing works (e.g., [14], [38], [58]) either omit this from their leakage profile or assume it can be solved by orthogonal techniques for eliminating such leakage. That said, the existence of a trusted area inside the untrusted server allows for increased efficiency (e.g., by running client-side routines inside the TEE one can avoid costly round-trips). In Section 7.4, we discuss the potential of combining μSE with TEE solutions.

3 Cryptographic Preliminaries

We denote by $\lambda \in \mathbb{N}$ a security parameter and by $v(\lambda)$ a negligible function in λ. PPT stands for probabilistic polynomial-time. For a set D, we denote its size by $|D|$. Our protocols involve three types of parties: a data owner, a server, and several users. The more complex setting with multiple data owners can be covered by running a separate instance of our protocol for each of them. Assume a file set D with identifiers $id_1, \ldots, id_{|D|}$, each of which contains textual keywords from a given alphabet Λ. Let the dataset DB consist of (keyword, file identifier) pairs, such that $(w, id) \in DB$ if and only if the file id contains keyword w. For each w, let $DB(w)$ denote the set of files that contain keyword w. Let W denote the set of keywords that contains all the keywords from DB, $|W|$ denote the number of distinct keywords and N denote the total number of (keyword, file identifier) pairs. For each user u, $Access(u)$ denotes all files that user u has access to and $AccList(id)$ the set of users with access to the file with identifier id. Therefore, $id \in Access(u)$ if and only if $u \in AccList(id)$. For a keyword w and user u, $DB_u(w)$ consists of the identifiers of all files that contain w and to which u has access to. For a set of users U, $Access(U)$ is the set of file identifiers to which at least one $u \in U$ has access. Finally, for file identifier id, $Kw(id)$ shows the keywords in that file at the setup time. $Kw^t(id)$ shows those keywords at timestamp t but, when it is clear from the context we drop the superscript t.

Pseudorandom Functions. Let $Gen(1^\lambda) \in \{0,1\}^\lambda$ be a key generation function, and $F : \{0,1\}^\lambda \times \{0,1\}^\ell \to \{0,1\}^{\ell'}$ be a pseudorandom function (PRF) family. F is secure if for all PPT adversaries Adv, $|Pr[K \leftarrow Gen(1^\lambda); \mathsf{Adv}^{F(K,\cdot)}(1^\lambda) = 1] - Pr[\mathsf{Adv}^{R(\cdot)}(1^\lambda) = 1]| \leq v(\lambda)$, where $R : \{0,1\}^\ell \to \{0,1\}^{\ell'}$ is a random function.

Oblivious Data Structures. Oblivious data structures are privacy-preserving versions of regular data structures. Formally, all equal-length sequences of data accesses (reads/writes) become indistinguishable from each other. Oblivious RAM (ORAM) [26] provides oblivious array accesses. One of the most popular and efficient ORAMs is PathORAM [47]. Wang et al. [60] proposed more general oblivious data structures such as oblivious maps (OMAP). An OMAP is a key/value dictionary that provides three procedures: (i) SETUP for initialization, (ii) INSERT to add a key/value pair, and (iii) FIND to retrieve a value for a given key. In the following, we use dictionary notation to denote Find/Insert from/to an OMAP (e.g., $Dict[k]$ refers to the value corresponding to key k in OMAP $Dict$). For our O-μSE scheme, we use the OMAP from [60] as a black box; for completeness we include the OMAP algorithms in Appendix A.

4 DMUSSE and Verifiability Definitions

In this section, we propose our dynamic multi-user SSE definition which follows that of [41] with some modifications for generality and to support dataset modification (e.g., adding an $Update$ algorithm and converting the $Search$ algorithm to an interactive protocol). Then, we define forward and backward privacy in the dynamic multi-user setting. Finally, we provide the appropriate definition of verifiability.

4.1 DMUSSE Definition

A *dynamic multi-user searchable symmetric encryption scheme (DMUSSE)* consists of five algorithms: Setup, Enroll, Share, Update, and Search that are executed between a data owner, a server, and several users:

- Setup($1^\lambda, N, |W|, |U|, D$): on input security parameter λ, total size of dataset N, size of distinct keyword set $|W|$, total number of users $|U|$, and a set of initial files D consisting of tuples $\{id, Kw(id)\}_{id \in \mathrm{D}}\}$ of a file identifier id and the set of keywords $Kw(id)$ in file id, it outputs $(K, xSet, \{K_{id}\}_{id \in D}, uSet, aux^D)$ where K is the master secret key of the owner, $xSet$ is an encrypted version of D, $\{K_{id}\}$ is a separate secret key for each file in D, $uSet$ is an initially empty set for users' authorization tokens, and aux^D is a set of auxiliary data of D for the owner. Both $xSet$ and $uSet$ are sent to the server while K, $\{K_{id}\}_{id \in D}$, and aux^D are kept private.
- Enroll($1^\lambda, u$): on input user id u and security parameter λ, it outputs user key K_u. It adds the new user u to the system. The data owner stores K_u privately and sends it to the user u. When the new user is enrolled, it does not have any access rights yet.
- Share($K, u, id, Kw^t(id), mod$): on input data owner master key K, user u, file identifier id, set of keywords $Kw^t(id)$ in file id, and mode mod (which can be $share$ or $add\&share$), either shares an existing file with user u or first adds a new file and then shares it with user u. It outputs $xSet'$ which is the encrypted version of file id for user u, and aux_u^D which is auxiliary data for user u and contains $\{K_{id}\}_{id \in Access(u)}$. It may also output a (new or existing) key K_{id} for file id. Then, $xSet'$ is sent to the server who includes it to $xSet$ and updates $uSet$ to note that u has access to id. K_{id} and aux_u^D are stored at the owner and privately shared with user u.
- Update($K, id, WList, op, \{aux_u^D\}_{u \in U}, \{K_u\}_{u \in U}$) is a protocol for inserting/removing a list of keywords to/from an existing file. On input master secret key K, file identifier id, list of keywords $WList$, operation op which can be add or del, users' auxiliary information $\{aux_u^D\}_{u \in U}$, and the users' secret key, it outputs $aux_u'^D$ which is the updated version of the auxiliary information for each user. Furthermore, it outputs $xSet'$, which is sent to the server to update $xSet$.
- Search(w, K_u, aux_u^D; $xSet, uSet$): this is a (possibly interactive) protocol run between the server and a user u that wishes to retrieve $DB_u(w)$. The user's inputs are search keyword w, secret key K_u, and auxiliary data aux_u^D. The server's inputs are encrypted dataset $xSet$ and authorization list $uSet$. After executing the interactive protocol, besides $DB_u(w)$ the user may get an updated version of aux_u^D. The server gets a possibly updated version of $xSet$.

We tried to make this definition general for reasons of backwards-compatibility. For instance, our constructions *do not use* file keys K_{id}, however, previous static schemes [41] need such keys. Finally, we use the standard simplifying convention from the literature that if the user wishes to retrieve the actual files for w after a search, this can be done by an additional roundtrip, not included in the above.

An additional functionality which can be helpful in DMUSSE is UnShare. It revokes access of a user to a file and

prevents the user from searching over that file afterward. We assume that files that have been unshared with a user u cannot be shared with u again.

- UnShare($K, u, id, Kw^t(id), aux_u^D$): on input master key K, user u, file identifier id, set of keywords $Kw^t(id)$ in file id, and user's auxiliary data, it revokes the user access to id. It outputs $xSet'$ which is the updated version of encrypted files that u has access to, $uSet'$ which is the updated set of authorization tokens, and $aux_u'^D$ which is the updated auxiliary data for u. $xSet'$ and $uSet'$ are sent to the server to be included in $xSet$ and $uSet$, and $aux_u'^D$ is stored at the owner and is privately shared with u.

4.2 Security vs. Corrupted Participants

Here we provide our DMUSSE security definition for the setting where some users may be corrupted or colluding with the server. As mentioned above, all previous multi-user searchable encryption definitions only consider such a threat model in the static setting, so our definition for the dynamic case can be viewed as a generalization.

We adopt the standard real/ideal game approach [16], [46] parametrized by leakage functions $\mathcal{L} = \{\mathcal{L}^{Stp}, \mathcal{L}^{Enrl}, \mathcal{L}^{Shr}, \mathcal{L}^{Updt}, \mathcal{L}^{Srch}, \mathcal{L}^{UnShr}\}$ where $\mathcal{L}^{Stp}$ corresponds to the leakage during $Setup$, and likewise for the rest of the leakages and functions. Our definition considers an interactive game between the adversary Adv and a challenger. At a high level, to capture the concept of corrupted users, we allow Adv to explicitly select the sets of users U and corrupted users $\mathcal{C}$ ($\mathcal{C} \subseteq U$) to be enrolled in the system. For those users declared corrupted, Adv receives all their information and internal state.

In the real game, Adv interacts with the algorithms of the scheme, while in the ideal one it interacts with a simulator that is only given the leakage function for each operation Adv requests and must simulate the behaviour of the algorithms. A secure DMUSSE scheme with leakage $\mathcal{L}$ should reveal nothing about DB other than this leakage, i.e., the adversary should be unable to distinguish between the real and the ideal execution with non-negligible advantage. The two games are presented in Figures 2, 3 and the formal security definition is as follows:

Definition 1. *A DMUSSE scheme Σ is adaptively-secure in the presence of corrupted participants with respect to leakage function $\mathcal{L}$, iff for any PPT adversary Adv issuing a polynomial number of queries q, there exists a stateful PPT simulator Sim =* (SimSetup, SimEnroll, SimShare, SimUpdate, SimSearch, SimUnShare) *such that* $|\Pr[\mathsf{Real}_{Adv}^{U,\mathcal{C},\Sigma}(\lambda, q) = 1] - \Pr[\mathsf{Ideal}_{Adv,Sim,\mathcal{L}}^{U,\mathcal{C},\Sigma}(\lambda, q) = 1]| \leq v(\lambda)$.

We believe our definition is a natural extension of the one from [41] to the dynamic setting. The main difference is the interactive nature of the game but this deviation is necessary in order to capture adaptive security in the dynamic setting, where the adversary may issue updates at arbitrary times. We stress that the definition of [28] achieves a stronger security property against a colluding server and data owners that wish to infer users' queries. However, we have two observations regarding the definition of [28]. First, the original file encryption phase is run by each user separately (i.e., the owner essentially "delegates" file ownership to the user) in order to guarantee security against server-owner collusion. This seems to be problematic in the dynamic setting, as one would have to delegate update responsibility to all entailed users (who have to remain always online) as well. Second, the ability for the owner (and only the owner) to periodically learn which keywords were searched may be an attractive feature in certain data sharing scenarios (e.g., the owner may want to check hospitals' queries in our patients' data sharing scenario).

4.3 Forward and Backward Privacy

Forward and backward privacy aim to control what information is leaked in relation to modifications to the dataset. Here, we provide the first formal definition of these properties for the multi-user case in the presence of corrupted users. Informally, a scheme is *forward-private* if a new modification cannot be connected to a previous operation when it occurs [7]. E.g., it should be impossible to tell whether an insertion is for a new keyword or one that was previously searched. To capture forward privacy, we first define some auxiliary functions, for timestamp t.

$$KwLeakage(u, id, t) = \begin{cases} Kw^t(id) & \text{if } u \text{ is corrupted} \\ |Kw^t(id)| & \text{if } u \text{ is honest} \end{cases}$$

$$WLeakage(u, WList) = \begin{cases} WList & \text{if } u \text{ is corrupted} \\ |WList| & \text{if } u \text{ is honest} \end{cases}$$

We will use these functions to demonstrate the leakage in modification operations for different user types (honest and corrupted). Now, we can define forward privacy as follows.

Definition 2. *A dynamic multi-user searchable symmetric encryption scheme with a coalition $\mathcal{C}$ of corrupted users is forward-private iff the* Share, UnShare, *and* Update *leakage functions $(\mathcal{L}^{Shr}, \mathcal{L}^{UnShr}, \mathcal{L}^{Updt})$ can be written as:*

$$\mathcal{L}^{Shr}(K, u, id, Kw^t(id), mod, t) = \mathcal{L}'(id, u, KwLeakage(u, id, t), mod)$$
$$\mathcal{L}^{UnShr}(K, u, id, Kw^t(id), aux_u^D, t) = \mathcal{L}'(id, u)$$
$$\mathcal{L}^{Updt}(K, id, WList, op, \{aux_u^D\}_{u\in U}, \{K_u\}_{u\in U}, t) = \mathcal{L}'(id, WLeakage(u, WList), op, AccList(id))$$

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

Compared to the single-user forward-privacy definition [7], [46], ours must account for leakage not only from file updates but also from file sharing/unsharing. Therefore, sharing a file with a corrupted user should not leak any information about previous operations (including search, update, and share) which were executed for honest users. In contrast to this, with the mx-u scheme of Patel et al. [41], if a file is shared between corrupted and honest users at time t, the server can learn all previous searched keywords on that file even before time t.

When a file is shared with a corrupted user the adversary trivially learns all keywords in it. If the user is honest, the only leakage is the number of keywords. Furthermore, the size and content of the file depend on the timestamp of the query execution (due to updates). Thus, we defined $KwLeakage(u, id, t)$ to receive as input both the user type and the timestamp. Similar to share, update should either reveal the updated keywords or their numbers according to

$b \leftarrow \mathsf{Real}^{U,\mathcal{C},\Sigma}_{\mathrm{Adv}}(\lambda, q)$:

$(D, U, \mathcal{C}) \leftarrow \mathsf{Adv}(1^\lambda)$;
$(\sigma_0 = \{K, aux^D, \{K_{id}\}_{id \in D}\}, EDB_0 = \{xSet, uSet\}) \leftarrow$Setup$(1^\lambda, N, |W|, |U|, D)$;
$(\sigma_0 = \sigma_0 \cup \{K_u\}_{u \in U}) \leftarrow$Enroll$(1^\lambda, U)$; ▷ Run Enroll for all users $u \in U$
for $i = 1$ to q **do**
  $q_i \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, aux^D_{\mathcal{C}}, \tau_1, \ldots, \tau_{i-1})$; ▷ τ_k is messages from user/owner to server in each query protocol
  if $q_i.type$ is Share **then**
    $(\sigma_i = \{aux^D_{q_i.u}\}, EDB_i) \leftarrow$Share$(K, q_i.u, q_i.id, Kw(q_i.id), q_i.mod, \sigma_{i-1}; EDB_{i-1})$;
  else if $q_i.type$ is Update **then**
    $(\sigma_i = \{aux^D_u\}_{u \in U}, EDB_i) \leftarrow$Update$(K, q_i.id, q_i.WList, q_i.op, \{aux^D_u, K_u\}_{u \in U}, \sigma_{i-1}; EDB_{i-1})$;
  else if $q_i.type$ is Search **then**
    $(\sigma_i = \{aux^D_{q_i.u}\}, DB_{q_i.u}(q_i.w); EDB_i) \leftarrow$Search$(q_i.w, K_{q_i.u}, aux^D_{q_i.u}, \sigma_{i-1}; EDB_{i-1})$;
  else if $q_i.type$ is UnShare **then**
    $(\sigma_i = \{aux^D_{q_i.u}\}, EDB_i) \leftarrow$UnShare$(K, q_i.u, q_i.id, Kw(q_i.id), aux^D_{q_i.u}, \sigma_{i-1}; EDB_{i-1})$;
return $b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, aux^D_{\mathcal{C}}, \tau_1, \tau_2, \ldots, \tau_q)$

Fig. 2: Real experiment for the DMUSSE scheme with user set U and coalition $\mathcal{C}$ of corrupted users.

$b \leftarrow \mathsf{Ideal}^{U,\mathcal{C},\Sigma}_{\mathrm{Adv},Sim,\mathcal{L}}(\lambda, q)$:

$(D, U, \mathcal{C}) \leftarrow \mathsf{Adv}(1^\lambda)$;
$(st_{\mathcal{S}}, EDB_0) \leftarrow$SimSetup$(1^\lambda, N, |W|, |U|, |D|)$
$(st_{\mathcal{S}}) \leftarrow st_{\mathcal{S}} \cup$ SimEnroll$(1^\lambda, U)$;
for $i = 1$ to q **do**
  $q_i \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, aux^D_{\mathcal{C}}, \tau_1, \ldots, \tau_{i-1})$;
  if $q_i.type$ is Share **then**
    $(st_{\mathcal{S}}; \tau_i, EDB_k) \leftarrow$SimShare$(st_{\mathcal{S}}, \mathcal{L}^{Shr}(q_i, q_i.u); EDB_{i-1})$
  else if $q_i.type$ is Update **then**
    $(st_{\mathcal{S}}; \tau_i, EDB_k) \leftarrow$SimUpdate$(st_{\mathcal{S}}, \mathcal{L}^{Updt}(q_i, q_i.u); EDB_{i-1})$
  else if $q_i.type$ is Search **then**
    $(st_{\mathcal{S}}; \tau_i, EDB_k) \leftarrow$SimSearch$(st_{\mathcal{S}}, \mathcal{L}^{Srch}(q_i, q_i.u); EDB_{i-1})$
  else if $q_i.type$ is UnShare **then**
    $(st_{\mathcal{S}}; \tau_i, EDB_k) \leftarrow$SimUnShare$(st_{\mathcal{S}}, \mathcal{L}^{UnShr}(q_i, q_i.u); EDB_{i-1})$
return $b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, aux^D_{\mathcal{C}}, \tau_1, \tau_2, \ldots, \tau_q)$;

Fig. 3: Ideal experiment for the DMUSSE scheme with user set U and coalition $\mathcal{C}$ of corrupted users.

the user type ($WLeakage(u, WList)$). Although the explicit leakage of the share operation in some cases can be smaller (e.g., when a file is shared with an honest user after being added to the dataset the number of keywords in the file is not newly leaked information as it can be computed based on the initial size and the query history), for simplicity we assume all share operations leak the same information.

Backward privacy limits the information that the server can learn during a search for keyword w for which some entries have been deleted since the last search. Bost et al. [7] gave the formal definition for the single-user case. Here, we propose our backward privacy definition for DMUSSE schemes. Again, we must first define some helper functions.

Let Q be a list with one entry for each executed query in a sequence of operations. For searches, the entry in Q is $(t, w, u, Search)$ where t is the timestamp, w is the searched keyword, and u is the user who runs the search query. For updates, the entry is $(t, id, WList, op, U, Update)$ where id is the identifier of modified file, $WList$ is the set of keywords for update operation, $op = add/del$, and U is the set of users who are affected by the operation ($AccList(id)$). For shares, the entry is $(t, id, Kw^t(id), mod, u, Share)$ where $mod = share/add\&share$. Finally, for unsharing, the entry is $(t, id, u, UnShare)$. Now we can define $\boldsymbol{Time}(w, u) = \{(t, id) |\ ((t, id, Kw^t(id), mod, u, Share) \in Q : w \in Kw^t(id) \lor (t, id, WList, add, U, Update) \in Q : w \in WList \land u \in U)) \land \forall t'(((t', id, WList, del, U, Update) \in Q : w \in WList \land u \in U) \lor ((t', id, u, UnShare) \in Q : w \in Kw^{t'}(id))) \rightarrow (t' < t)\}$. To put it simply, this is the function that returns all (timestamp, file-identifier) tuples of keyword w for user u that have been added by either a Share or an Update operation to DB and have not been deleted or unshared afterwards. Next, function $\boldsymbol{Update}(w, u) = \{t | ((t, id, Kw^t(id), mod, u, Share) \in Q : w \in Kw^t(id) \lor (t, id, WList, op, U, Update) \in Q : w \in WList, u \in U)\}$ returns the timestamps of sharing and of each addition/deletion operation related to keyword w for user u. Finally, we define:

$$SrchLeakage(w, u) = \begin{cases} w & \text{if } u \text{ is corrupted} \\ \perp & \text{if } u \text{ is honest} \end{cases}$$

Using these functions, we now define backward privacy.

Definition 3. *A dynamic multi-user searchable symmetric encryption scheme with a coalition $\mathcal{C}$ of corrupted users is backward-private iff the* Search *leakage function can be written as:*

$$\mathcal{L}^{Srch}(w, K_u, aux^D_u; xSet, uSet) = \mathcal{L}'(u, \boldsymbol{Time}(w, u), \boldsymbol{Update}(w, u), SrchLeakage(w, u))$$

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

Our definition reveals two important pieces of information for w during a search: (i) the files currently containing w that u has access to, and (ii) the timestamps of all previous updates for w affecting u. The first is unavoidable if the user wishes to subsequently retrieve the actual files (except if the files are stored in oblivious storage)—none of our constructions explicitly leaks this information during file id retrieval. The second captures backward privacy by hiding the id of files previously containing w and only revealing when

previous related updates took place, along the lines of the definition of [7]. Indeed, if we assume no corrupted users, our definition would be a natural extension of the Type-II backward privacy of [7] to the multi-user setting. Finally, $SrchLeakage(w, u)$ leaks the searched keyword only if the query is executed by a corrupted user.

4.4 Verifiability

In what follows, we give the definition of *verifiable* DMUSSE (VDMUSSE). The basic idea of VDMUSSE is to modify DMUSSE algorithms and use authenticated data structures (e.g., [36], [52]) so that the user can check whether the server is sending back the correct result of search queries ($DB_u(w)$) without dropping/modifying parts of it. To implement such an idea, Setup creates a verification token Δ at the beginning. When a user is enrolled in the system, Δ is shared with it. Subsequent operations (including Share, Update, UnShare, and maybe Search) may update Δ and all users are notified. During a search for w, the server sends a verification proof Π with the result. Using Π and Δ, the user verifies the correctness of the result. A VDMUSSE contains an additional algorithm Verify:

- Verify$(K_u, aux_u^D, \Delta, \Pi, w, IdSet)$: on input secret key K_u, auxiliary data aux_u^D, verification token Δ, proof Π, keyword w, and a set of file identifiers which is the response of the search query, it outputs $true$ if the verification test passes, otherwise outputs $false$.

We define the verifiability of VDMUSSE by computing the advantage of the adversary in producing incorrect results that still pass verification via the following game between adversary and challenger. The adversary, playing the role of the server, chooses the dataset and the corrupted users and it receives from the challenger the result of running Setup and the corrupted parties' keys. The adversary then issues polynomially many queries of Share/Update/UnShare/Search to the challenger and receives corresponding responses. The challenger executes these queries while recording the current state of the dataset and all scheme variables. Finally, the adversary provides a result set $IdSet$, user u, and proof Π' and the challenger runs Verify on them using the latest locally stored digest Δ. The adversary wins if $IdSet$ is not the correct result ($\neq DB_u(w)$, where DB_u is the latest dataset version for u, according to the challenger's state), u is not corrupted, and Verify accepts. The following definition captures the above (similar to the one from [6] for the single-user case):

Definition 4. *A dynamic multi-user searchable symmetric encryption scheme is verifiable if for all λ, any PPT adversary Adv who plays the above game can win with probability at most $v(\lambda)$.*

5 μSE Construction

In this section, we propose our dynamic multi-user forward and backward private SSE constructions. We explain how to extend them to achieve verifiability in Section 6.

One of the main challenges in proposing DMUSSE is to provide an efficient mechanism for distributing updates among the affected users. We propose two constructions that address this issue in different ways: (i) O-μSE which is based on an oblivious map, and (ii) Q-μSE which uses update queues. At a high level, both our schemes adopt the approach of MITRA from [25] for building the encrypted dataset index. That is, they use a key-value map to store the encrypted values of the form (file identifier, operation) for the keywords in each file. The location of these key-values in the map are computed via a PRF, using a separate incremental per-keyword counter. These counters are stored at the data owner and are incremented after each update. A copy of these counters may also be stored at each user, depending on the files it has access to. To search for a specific keyword w, the user first looks up the corresponding counter (cnt) and then asks the server to return all encrypted values of the map that correspond to keys $(w, 1), \ldots, (w, \mathsf{cnt})$. Finally, it decrypts the returned ciphertexts and extracts the file identifiers for w, excluding deleted files. Note that since we do not store the whole address space for all keywords and file identifiers (we only keep the existing mappings by computing PRFs), collisions may happen and two (file identifier, operation) pairs may be mapped to the same location. The probability of such a collision can be made negligibly small in the parameter λ by setting a large enough PRF output (see [20] for more details). In what follows, we explain our two schemes and analyze their efficiency and security. Since they are similar in most of their parts, we present their pseudocodes together.

5.1 OMAP-based μSE

O-μSE stores each user's keyword counters in a separate OMAP on the server that maps keywords to counters. We note that each OMAP is instantiated with a different key which is only shared with the data owner and that user. To execute an update or share, the data owner interacts with the corresponding users' OMAPs on the server to modify keyword counters as necessary. When a user wants to execute a search, it must first fetch the last value of the keyword counter from its OMAP and then proceed to request the map keys as explained above. Below, we explain the O-μSE procedures in more detail.

Setup. The setup algorithm is presented in Algorithm 1. In the beginning, on input security parameter λ, total dataset size N, size of keyword set $|W|$, total number of users $|U|$, and an empty initial data set, the data owner generates a secret key K (we omit initial dataset encryption from setup to simplify the presentation). Then, it initiates two empty maps **DictW** and **UserKeys**. **DictW** is used to store encrypted entries and **UserKeys** is used to store users' secret keys. After that, the data owner creates a map of lists **AccessList** to store mappings of file identifiers to users who have access to them. **UserKeys** and **AccessList** are stored locally in σ (which plays the role of aux^D in DMUSSE definition and is given to all other procedures), whereas **DictW** and a copy of **AccessList** are sent to the server (**DictW** corresponds to $xSet$ and **AccessList** corresponds to $uSet$ according to our DMUSSE definition). Note that **AccessList** is used by the server for access control during file retrieval—it does not appear in the file identifier extraction process below. Note that, although we could store the encrypted values of each user in a separate map, we store all encrypted values in one map (**DictW**) for simplicity of design and explanation.

Algorithm 1 μSE Setup$(1^\lambda, N, |W|, |U|, \perp)$

Owner:
1: $K \leftarrow Gen(1^\lambda)$
2: **DictW**,**UsersKeys**$\leftarrow$ empty map
3: **FileCnts** $\leftarrow$ empty map of maps ▷ For Queue-based
4: **Queues** $\leftarrow$ empty map of queues ▷ For Queue-based
5: **AccessList** $\leftarrow$ empty map of lists
6: $\sigma \leftarrow$ (**FileCnts**, **AccessList**, **UsersKeys**)
7: $EDB \leftarrow$ (**DictW**, **Queues**, **AccessList**)
8: Send EDB to the server
User (u):
9: **if** $u.type$ is OMAP-based **then OMAP.key**$\leftarrow\perp$
10: **else if** $u.type$ is Queue-based **then FileCnt**$\leftarrow$ empty

Enroll. The data owner registers user u for future files sharing. First, it generates secret key $K_u \leftarrow Gen(1^\lambda)$ for user u and stores it in **UserKeys**. Then, it initializes an OMAP for user u with capacity $|W|$ and sends it to the server. Finally, it sends K_u and **OMAP**$_u$**.key** to the user.

Share. In the share algorithm (Algorithm 2), the owner first adds user u to the list of users who have access to the given file (**AccessList**[id]). Then, it loads the user key K_u from **UserKeys**[u] (line 3). For each keyword w in file id, it checks whether the keyword counter **OMAP**$_u$[w] is initialized or not. In the latter case, the value will be initialized to 0. Next, the data owner increments the keyword counter and stores its current value in cnt (line 8). After that, it computes the encrypted values and their locations on the server. For the location, the PRF G with key K_u and (keyword, keyword counter) tuple as the input of the function is computed (line 13). The output of the PRF function is XORed with the file identifier concatenated with "add" operator to construct the encrypted value. In the last step, all pairs of $(addr, val)$ and (u, id) are sent to the server who adds u to **AccessList**[id] and stores $(addr, val)$ in **DictW** (lines 17-18). Clearly, different modes ($share$ and $add\&share$) do not change the algorithm significantly since a separate copy of the target file id is created for each user u. Indeed, all Share operations are implicitly executed in $add\&share$ mode.

Update. Algorithm 3 describes the update process. First, the data owner gets all the users who have access to the target file id from **AccessList**[id]. For each user u in **AccessList**[id], it extracts the corresponding key K_u and for each keyword w in set $WList$ checks whether the user's keyword counter has been initialized or not (**OMAP**$_u$[w]). If not, the counter is set to 0. Then, it increments the counter and runs the PRF G with key K_u to compute the encrypted value val and its location $addr$ on the server for each keyword (lines 13-15). Finally, the $(addr, val)$ pairs for all keywords and users are sent to the server to be stored in **DictW** (lines 17-18).

Search. The search process is shown in Algorithm 4. When user u wants to search for files containing keyword w, it first retrieves the keyword counter **OMAP**$_u$[w] from the server, which corresponds to the total number of updates related to w (line 2). Then, it generates a list of locations for these entries in **DictW** by evaluating the PRF G on input $G_{K_u}(w, i||0)$ for $i = 1, \ldots,$ **OMAP**$_u$[w] (line 9). Note that these are the locations computed during earlier shares and updates for w and user u. The list is sent to the server who retrieves the encrypted values from **DictW** and sends them back to the user (lines 11-13). The user decrypts them by computing the PRF values $G_{K_u}(w, i||1)$ for $i = 1, \ldots,$ cnt and XORing the i-th of them with the i-th ciphertext. Finally, it extracts all $(id||op)$ pairs and removes deleted id's.

Algorithm 2 μSE Share$(K, u, id, Kw^t(id), mod, \sigma)\}$

Owner:
1: **AccessList**[id] $\leftarrow$ **AccessList**[id] $\cup\{u\}$
2: KeyValues $\leftarrow\perp$; CntDiffs $\leftarrow\perp$
3: **if** $u.type$ is OMAP-based **then** $K_u \leftarrow$**UsersKeys**[u]
4: **else** $(K_u, K_u^{'}) \leftarrow$**UsersKeys**[$u$]
5: **for** w in $Kw^t(id)$ **do**
6: **if** $u.type$ is OMAP-based **then**
7: **if OMAP**$_u$[w] = $\perp$ **then OMAP**$_u$[w]=0
8: **OMAP**$_u$[w]++ ; cnt $\leftarrow$ **OMAP**$_u$[w]
9: **else if** $u.type$ is Queue-based **then**
10: **if FileCnts**[u][w] = $\perp$ **then FileCnts**[u][w] $= 0$
11: **FileCnts**[u][w]++ ; cnt $\leftarrow$ **FileCnts**[u][w]
12: CntDiffs $\leftarrow$ CntDiffs $\cup\ (u, Enc_{K_u^{'}}(w, \text{cnt}))$
13: $addr = G_{K_u}(w, \text{cnt}||0)$
14: $val = (id||add) \oplus G_{K_u}(w, \text{cnt}||1)$
15: KeyValues $\leftarrow$ KeyValues $\cup\ (addr, val)$
16: Send KeyValues, CntDiffs, and (u, id) to the server
Server:
17: **AccessList**[id] $\leftarrow$ **AccessList**[id] $\cup\{u\}$
18: Insert all $(addr, val)$ in KeyValues into **DictW**
19: Insert all (u, diff) in CntDiffs into **Queues**[u]

Algorithm 3 μSE Update$(K, id, WList, op, \sigma)$

Owner:
1: userList $\leftarrow$ **AccessList**[id]; KeyValues $\leftarrow\perp$; CntDiffs $\leftarrow\perp$
2: **for** u in userList **do**
3: **if** $u.type$ is OMAP-based **then** $K_u \leftarrow$**UsersKeys**[u]
4: **else** $(K_u, K_u^{'}) \leftarrow$**UsersKeys**[$u$]
5: **for** w in $WList$ **do**
6: **if** $u.type$ is OMAP-based **then**
7: **if OMAP**$_u$[w] = $\perp$ **then OMAP**$_u$[w]=0
8: **OMAP**$_u$[w]++ ; cnt $\leftarrow$ **OMAP**$_u$[w]
9: **else if** $u.type$ is Queue-based **then**
10: **if FileCnts**[u][w] = $\perp$ **then FileCnts**[u][w] $= 0$
11: **FileCnts**[u][w]++ ; cnt $\leftarrow$ **FileCnts**[u][w]
12: CntDiffs $\leftarrow$ CntDiffs $\cup\ (u, Enc_{K_u^{'}}(w, \text{cnt}))$
13: $addr = G_{K_u}(w, \text{cnt}||0)$
14: $val = (id||op) \oplus G_{K_u}(w, \text{cnt}||1)$
15: KeyValues $\leftarrow$ KeyValues $\cup\ (addr, val)$
16: Send KeyValues and CntDiffs to the server
Server:
17: Insert all $(addr, val)$ in KeyValues into **DictW**
18: Insert all (u, diff) in CntDiffs into **Queues**[u]

5.2 QUEUE-BASED μSE

Q-μSE uses a different approach from O-μSE for maintaining keyword counters. It stores keyword counters locally at each user and whenever the data owner performs an update or share it stores the new values of the counters in a separate queue for each user on the server in encrypted form, using symmetric encryption. The secret key for each queue is

Algorithm 4 μSE Search$(w, K_u, K_u^{'}, \sigma; EDB)$

User:
1: TList = { }; $R_w = \{\}$
2: **if** $u.type$ is OMAP-based **then** cnt $\leftarrow$ **OMAP**$_u$[w]
3: **else if** $u.type$ is Queue-based **then**
4: diffs $\leftarrow$ fetch **Queues**$[u]$ from the server
5: **for** diff in diffs **do**
6: $(w', \text{cnt}) \leftarrow Dec_{K_u'}(\text{diff})$
7: **FileCnt**$[w'] =$ cnt
8: cnt $\leftarrow$ **FileCnt**$[w]$
9: **for** $i = 1$ to cnt **do** TList = TList $\cup \{G_{K_u}(w, i||0)\}$
10: Send TList to the server
Server:
11: $F_w = \{\}$
12: **for** $i = 1$ to TList.size **do** $F_w = F_w \cup$ **DictW**[TList$[i]$]
13: Send F_w to the user
User:
14: **for** $i = 1$ to $F_w.size$ **do**
15: $(id||op) = F_w[i] \oplus G_{K_u}(w, i||1)$
16: $R_w = R_w \cup (id||op)$
17: Remove deleted id's from R_w and return R_w

shared only between data owner and target user. When the user wants to execute a search query, it first "flushes" its queue and fetches all counter modifications that happened since the last search from the server. Then, it updates the local keyword counters and proceeds with the search. We stress that all other parts of the scheme remain the same as O-μSE. In what follows, we explain the Q-μSE procedures omitting parts that are similar to O-μSE:

Setup. The setup algorithm is the same as O-μSE (Algorithm 1), except that the owner initiates two empty maps: **FileCnts** stores each user's keyword counters and **Queues** stores the updates of the counters for each user separately on the server. On the other hand, each user initiates an empty map **FileCnt** to store its own counters locally.

Enroll. The only difference with O-μSE is that the owner creates an empty map **FileCnts**$[u]$ for the user's keyword counters which it keeps locally, instead of **OMAP**$_u$. Furthermore, it generates a key $K_u^{'}$ for a semantically-secure symmetric encryption scheme using $KeyGen(1^\lambda)$. This will be used to encrypt encrypt counter updates later.

Share. For each distinct keyword w in file id, the data owner checks whether the corresponding counter **FileCnts**$[u][w]$ has been initialized or not. In the latter case, it will be initialized to 0 (line 10). Then, it increments the counter and encrypts its new value (lines 11-12). Finally, $(addr, val)$, (u, id) pairs, and the encrypted counter information will be sent to the server to be stored in the user's queue **Queues**$[u]$ (line 19). Similar to O-μSE, $share$ and $add\&share$ modes are treated in the same manner.

Update. As with Share, the only difference between Q-μSE and O-μSE is that keyword counters are updated locally and stored in the corresponding users' queues on the server.

Search. In the search process (Algorithm 4), the only difference is that instead of retrieving the keyword counter from the oblivious map, the user first fetches the contents of its queue, decrypts them, and updates its local counter array **FileCnt**$[w]$ (lines 4-8). Then, the user uses the updated value of the counter and continues with the search as in O-μSE.

5.3 Revoking Document Access

In Section 4, we included in the DMUSEE algorithms an UnShare operation whose goal is to revoke a user's access to a document. Intuitively, its purpose in a dynamic scheme is to limit the user's knowledge of the document to whatever has been queried so far. A first observation regarding unsharing in our security model where corrupt users may collaborate with the server is that this "strong" goal is *unobtainable*: The server can always store the old version of the encrypted dataset (prior to the unshare) and keep sharing it with the user. Hence, a more realistic goal in this setting is to ensure that the user *cannot learn any information about future versions of the unshared document*.

In both versions of μSE, unsharing id is implemented by notifying the server to remove the user from the document's authorization list. This incurs negligible overhead as it does not entail any cryptographic operations and it satisfies our forward privacy definition from Section 4. Then, the owner does not include the user whose access has been revoked from future iterations of Update regarding id, which guarantees the goal outlined above. Observe that the user may still get information about versions of id prior to unsharing. If one wants to avoid this, one alternative that maintains forward privacy would be to implement unsharing by re-enrolling the user with a new key and re-sharing all the documents it has access to (excluding id). This clearly incurs a very large overhead. Moreover, we argue that it does not satisfy our goal in a meaningful way: due to the counter-based approach of μSE the user may still infer information about the state of id prior to unsharing via future queries: If a counter for searched keyword w remains the same right before and after unsharing, then $w \notin id$, else $w \in id$. Hence, we chose to implement the first method for unsharing due to its much better performance.

5.4 Security Analysis and Asymptotic Efficiency

Here, we analyze the security of our schemes and their asymptotic performance.

Security Analysis. Both our schemes are secure with respect to Definition 1 with leakage profiles that achieve forward and backward privacy (Definitions 2,3). In both of them the (add, val) pairs during updates and shares are computed with a PRF using a fresh counter, thus they are indistinguishable from random. For O-μSE, the deployed oblivious map entirely hides from the server which are the affected keywords during all accesses to it; the server only learns who are the affected users. For Q-μSE, due to the semantic security of the encryption scheme used to encrypt the queues again the only leaked information is which users' queues are accessed. Hence, during searches, the only thing that is revealed to the server, beyond the id of the querying user u, is the number of previous updates for w and when they took place (since the server can always remember when it last accessed the corresponding dictionary locations). Finally, since separate OMAPs/queues with different keys are used for each user and the PRF key used for each user is also different, we manage to eliminate all cross-user leakage.

We are now ready to state the following theorem for the security of our schemes (full proofs in Appendix B).

Theorem 1. *Assuming G is a secure PRF, queues are encrypted with a semantically secure encryption scheme, and OMAP is a secure oblivious map,* Q-μSE *and* O-μSE *are secure (Definition 1) and forward/backward private (Definitions 2,3) DMUSSE schemes with the following leakage:*
$\mathcal{L}^{Shr} = \{id, u, KwLeakage(u, id, t), mod\}$
$\mathcal{L}^{UnShr} = \{id, u\}$
$\mathcal{L}^{Updt} = \{id, WLeakage(u, WList), op, AccList(id)\}$
$\mathcal{L}^{Srch} = \{u, \textbf{\textit{Time}}(w, u), \textbf{\textit{Update}}(w, u), SrchLeakage(w, u)\}$

Asymptotic Efficiency. Setting up an initially empty dataset takes $O(1)$ operations. Enroll takes $O(1)$ with Q-μSE and $O(|W|)$ with O-μSE as the latter sets up the user's OMAP. Share for a single keyword w requires computing add, val ($O(1)$ operations). It also entails updating the counter in OMAP for O-μSE ($O(\log^2 |W|)$ operations, assuming constant block size), or updating **Queues** for Q-μSE ($O(1)$ operations). Clearly, this grows linearly with the number of keywords in $Kw^t(id)$. This also characterizes communication size. Update entails the same overheads (per keyword and per affected user). Finally, UnShare has $O(1)$ computation and communication overhead.

Regarding searches, let a_w denote the counter for the total number of previous updates for keyword w and user u. For O-μSE, u has to execute an OMAP look-up to extract this counter ($O(\log^2 |W|)$ computation time and communication size, as above). For Q-μSE, $|\mathsf{diffs}|$ denotes the size of the queue received from the server. This can vary between 0 and N (size of dataset). In both cases, after computing a_w the user executes $O(a_w)$ operations to retrieve the encrypted file identifiers. Indeed, the computation and communication complexity of search for O-μSE and Q-μSE would be $O(a_w + \log^2 |W|)$ and $O(a_w + |\mathsf{diff}|)$, respectively. Comparing the two, for settings with few updates or small intervals between searches, Q-μSE can be better while with frequent updates or "sparse" searches O-μSE may be better.

The server storage in both schemes is $O(N)$ *per user*, assuming N (keyword-file-operation) entries in the dataset. Of course, this corresponds to the worst case where the entire dataset is shared with each user. On the other hand, the OMAP stash size for O-μSE is $O(\log |W| \cdot \omega(1))$ but this downloaded by the users for each operation, without asymptotically increasing computation or communication, so the permanent user storage is $O(1)$ O-μSE. For Q-μSE it is $O(|W|)$ as users need to store their keyword counters. Finally, with Q-μSE Share, Update, and Search, require $O(1)$ roundtrips, whereas with O-μSE they require $O(log|W|)$ roundtrips, due to OMAP look-ups. Depending on the network latency in a given application, this may significantly affect the performance.

6 VERIFIABLE μSE

Our above schemes are not secure against a server that provides a false result, e.g., by dropping or modifying (parts of) it. In this section, we explain how to extend our μSE schemes to VDMUSSE to achieve result verifiability.

As explained previously, a general blueprint for verifiability (e.g., see [6]) is to protect the integrity of the encrypted data via a Merkle tree. Our schemes' setup algorithm builds an encrypted index (**DictW**) that can be treated as an array. It is then straight-forward to compute a Merkle tree over it and publish its root as the verification digest. Users issuing searches get the corresponding tree proofs from the server and verify each **DictW** access with respect to this digest. Hence, they can ensure that no element of the result was omitted, changed, or erroneously added.

In addition to the encrypted data itself, the integrity of the keyword counters needs to be protected. Otherwise, a misbehaving server can provide an outdated OMAP or queue version during the search operation which leads to an outdated or incorrect search result. To prevent this, a similar Merkle tree technique can be used. We note that the situation is somewhat more complicated for O-μSE than Q-μSE. With the latter, queue entries are updated only by the owner during Update/Share operations—consequently, so does the digest. With the former, OMAP blocks are shuffled whenever keyword counters are accessed, including during Search queries issued by users. Hence, even search operations change the digest. Clearly, the overall additional overhead of verifiability in terms of computation complexity and communication size is logarithmic.

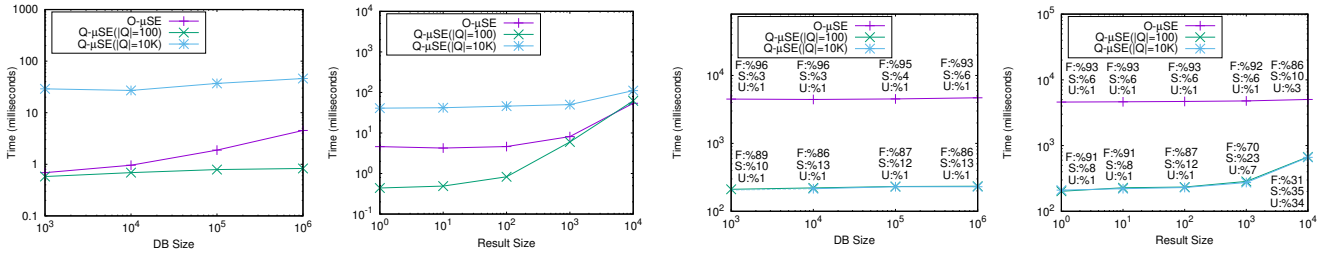
We refer to the verifiable versions of our schemes that are achieved by using the above technique as VQ-μSE and VO-μSE. We state the following theorem for their verifiability.

Theorem 2. *Assuming H is a collision-resistant hash function used to build the Merkle trees described above,* VO-μSE *and* VQ-μSE *are verifiable according to Definition 4.*

Proof Sketch. Recall that during the verifiability game, the challenger stores the dataset DB initially chosen by the adversary. It then makes a list $\mathbf{I}$ and on each Update/Share operation it stores entry $(addr, val, u, w, id, op, type, \Delta)$ in the $i-$th cell of $\mathbf{I}$, for all $u \in AccList(id)$ and all $w \in WList$ in Update and for all $w \in Kw^t(id)$ in Share where type is $Share$ or $Update$, respectively, and op is add or del. Eventually, the adversary produces its challenge $IdSet, u, w, \Pi'$. The challenger runs Verify for it, using the latest $\Delta \in I$. Let us examine the case $|IdSet| \neq |DB_u(w)|$, where $DB_u(w)$ is the correct search result for w and user u with respect to the latest dataset version. Hence, the keyword w counter computed locally by the challenger is different but $Verify$ accepts for the correct Δ. This means that Π' includes a collision in H, for both schemes. The only remaining case we need to argue is when $|IdSet| = |DB_u(w)|$ but $IdSet \neq DB(u)$. Again, as part of verification, the challenger recursively hashes $IdSet$ with Π', computes the Merkle root and compares it with Δ. Clearly, if $IdSet \neq DB(u)$ and the comparison succeeds, Π' contains a collision for H along the path in the tree. We conclude that the advantage of the adversary is negligible. □

6.1 Deploying Verifiable μSE on Hyperledger Fabric

Next, we explain how verifiable μSE can be implemented in practice. Recall that the main issue has to do with the dissemination of digests among users. Direct communication between the data owner and users may be infeasible, whereas relying on the server (or a third party) for storing



(a) (b) (c) (d)

Fig. 4: Search computation vs.: (a) variable $|DB|$ for result size 100 on stand-alone machine, (b) variable result size for $|DB| = 1$M on stand-alone machine, (c) variable $|DB|$ for result size 100 in end-to-end setting, (d) variable result size for $|DB| = 1$M in end-to-end setting.

the digests is insecure in cases of compromise or misbehavior. Our approach is to use a blockchain protocol for storing and managing the digests without relying on a trusted third party. In particular, we deployed our μSE schemes atop Hyperledger Fabric [21], a permissioned blockchain protocol designed for a closed set of participants who, while they may not fully trust each other (they may be competitors in the same industry), can still cooperate under a governance model that is built off of what trust does exist between them, such as a legal agreement. In the simplest implementation, each participant runs a blockchain node (except for the server that does not need access to the blockchain). For completeness, we provide the necessary code for setting up μSE on Fabric in Appendix C.

Modifications to μSE. To implement verifiable μSE on Hyperledger Fabric, its procedures should be modified as follows. At the setup phase, in addition to uploading EDB to the server, each data owner issues a Fabric transaction to publish its digest (i.e., the root of its dataset's Merkle tree), computed as explained above. Whenever a new user is enrolled for this data owner, the latter provides read permission of its digest to the user. The data owner also executes a blockchain transaction to store the digest for the user's counters (i.e., the root of the "Merkle-ized" OMAP tree for VO-μSE, or the root of the queues Merkle tree for VQ-μSE). Finally, it gives permissions for this digest to the user. For VQ-μSE this is a read permission, whereas for VO-μSE this is a read&write permission; recall that the OMAP state changes after every access, including reads, hence the user must be able to compute the new digest after searches.

During each Update/Share, the data owner also issues a new blockchain transaction with the updated digest; this serves to ensure the integrity and freshness of subsequent search results. When executing a Search operation, the user first retrieves the digest from Fabric. Then, it proceeds to run the remainder of the query with the server, verifying the result against the retrieved digest (note that the server does not need to have access to the blockchain at any point; it receives data directly from the data owners). Finally, for VO-μSE, the user needs to compute the new counters digest and issue a Fabric transaction to update it on the blockchain.

7 Experimental Evaluation

For our experimental evaluation we implement the verifiable versions of O-μSE and Q-μSE in C++ with OpenSSL [44], using AES-256 as PRF. We consider two settings: (i) a single machine with four-core Intel i5-7600 3.5GHz processor, running Ubuntu 16.04 LTS, with 40GB RAM, 500GB HDD, and AES-NI for measuring computation times, and (ii) five t2.xlarge AWS machines with four-core Intel Xeon E5-2676 v3 2.4GHz processor, running Ubuntu 16.04, with 16GB RAM, 100GB SSD hard disk, and AES-NI, for measuring end-to-end times. The average RTT among the AWS machines is 20ms (various EU locations). Since there is no other DMUSSE scheme in the corrupted user setting, we compare ours with state-of-art static MUSSE constructions. We implement the MKSE scheme of [28] to compare its search performance with ours. We also report comparison with mx-u [41] which, however, is considerably slower as it relies on public-key operations—it also suffers from cross-user leakage, unlike our schemes and MKSE.

We mainly measure execution time and communication size for search and update operations (with verification). We consider synthetic datasets of size $|DB| = 10^3$–10^6, setting $|W|$ and number of files to one-hundredth of $|DB|$ (unless otherwise specified). We vary the search result size from 1–10^4. Since the performance of Q-μSE depends on the size of the update queue, we execute experiments separately with a *small* and *large* update queue size (100 and 10K). Before searches, we delete 10% of the corresponding entries at random to account for the impact of deletions (except when the result size is 1). We report the average of 10 executions.

7.1 Stand-Alone Machine Measurements

First, we examine the pure computational time of O-μSE and Q-μSE on a single machine, thus ignoring overheads due to blockchain transactions and transmission time.

Search Performance. Figure 4(a) shows the search time of O-μSE and Q-μSE (for update queue size 100 and 10K) and variable database size (for $|DB| = 10^3$, update queue 10K would correspond to an extreme case where the entire dataset is deleted and re-written multiple times; we only include this point for completeness). As is evident, the search computation time for Q-μSE increases very slightly with the database size; this increase is due to the change in the Merkle tree height for verification. On the other hand, the O-μSE time increases more steeply since the cost of each OMAP access increases with $|DB|$. Figure 4(b) shows the search computation time for variable result size. Observe that it increases more sharply with the result size than with $|DB|$, as per our analysis in Section 5.4. As the search result size increases, the file identifier extraction (i.e., PRF

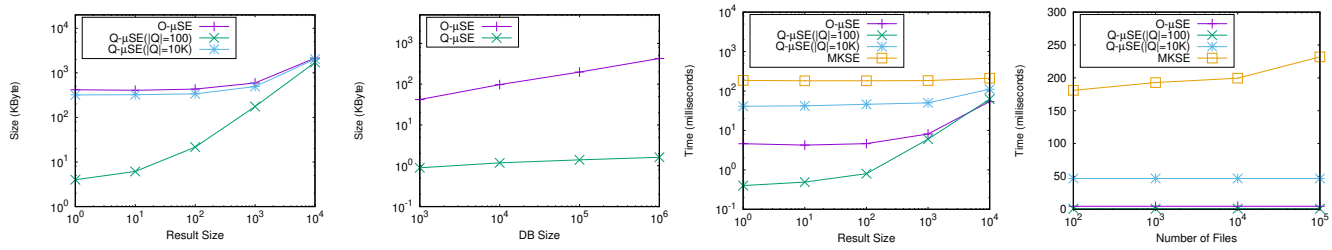


Fig. 5: Communication size for: (a) search vs. variable result size for $|DB| = 1$M, (b) update vs. variable $|DB|$. Search time vs: (c) variable result size for $|DB| = 1$M, (d) variable number of files for $|DB| = 1$M and $|\mathbf{Result}| = 100$.

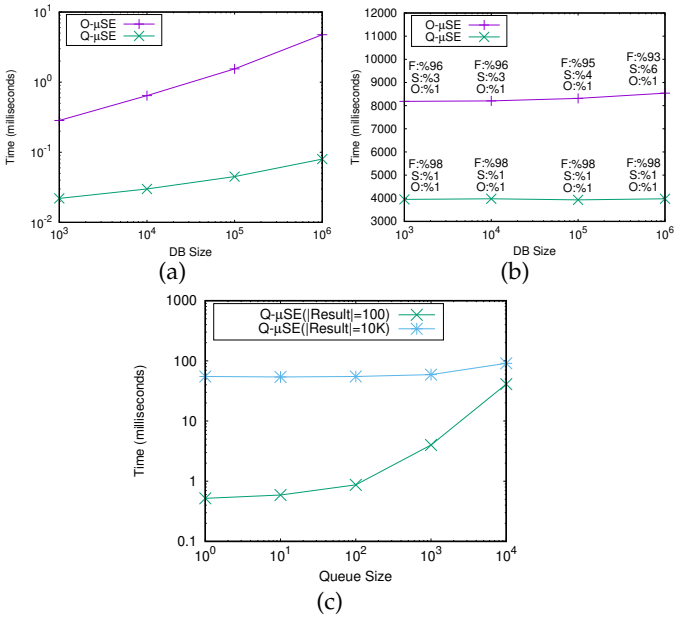


Fig. 6: Update computation vs. variable $|DB|$ in (a) standalone machine (b) end-to-end setting. (c) Search vs. variable queue size for $|\mathbf{Result}| = 100$ and $|\mathbf{Result}| = 10$K.

evaluations) time dominates all other overheads and all three methods' search times become approximately equal.

Overall, both schemes have excellent performance and their execution time is in the order of milliseconds (even for a large result size of 10K it is <111ms). Further comparing the two schemes, we see that in general O-μSE lies between Q-μSE with large queue and Q-μSE with small queue. Throughout all executions, Q-μSE with small queue is 0.9-10$\times$ faster than O-μSE and 1.7-93$\times$ faster than Q-μSE with large queue. One can argue that O-μSE would be more appropriate for scenarios where users execute search queries rarely or the update frequency is high (hence the update queue size grows fast). On the other hand, Q-μSE may be more useful for settings where users run search queries frequently or the number of updates is low.

Figure 5(a) shows the total communication size for searches as the result size changes. The communication size of Q-μSE with small queue size is roughly linear. For O-μSE, when the result size is small, communication is dominated by OMAP accesses. Even for searches with small results it is not below a threshold (406KB). As the result size increases, the search result itself exceeds the OMAP overhead and becomes the main factor for communication size. Likewise, Q-μSE with large queue needs to fetch many counter updates which enforces a big overhead even for small result sizes. Concretely, O-μSE and Q-μSE with large queue have very similar communication size (the ratio of the former over the latter is 1-1.3$\times$). Throughout all our experiments, Q-μSE with small queue size needs 1-103$\times$ less communication than O-μSE (e.g., Q-μSE with queue size 100, 10K, and O-μSE transmit 6KB, 323KB, and 406KB in total for a database of 1M records and result size 10).

Effect of Queue Size. Clearly, for Q-μSE the queue size plays a crucial role in the scheme's performance. To measure this, we measure the search computation time of Q-μSE for variable queue sizes (Figure 6(c)). We fix $|DB|$ to 1M and vary the queue size between 1 and 10K for two result sizes (100 and 10K). As expected, the search computation time increases proportionally to the queue size. This implies that the overhead of updating the counters for a user with infrequent search operation can become high with Q-μSE. Furthermore, the impact of queue size on search performance with small results is more evident than on searches with truly large results (e.g. for result 10K, the impact is almost imperceptible in the figure).

Update Performance. Figure 6 (a) shows the update computation time vs. variable database size for a single keyword insertion/deletion. As expected, Q-μSE is faster (12-59$\times$) than O-μSE as it only needs to store one key-value pair and an encrypted counter on the server while O-μSE requires accessing the OMAP. That said, both schemes have very good update performance (<5ms). Regarding update communication size (Figure 5 (b)), for both schemes it increases almost linearly with $|DB|$. However, it increases more steeply for O-μSE due to the increase of the OMAP tree height (e.g., the communication size of Q-μSE for a database with size 1M is 1KB while O-μSE transfers 426KB).

Storage. Recall that in O-μSE we store the OMAP stash at the server which minimizes the client storage, making it $O(1)$ for O-μSE and $O(|W|)$ for Q-μSE (which requires maintaining the update counters). Concretely, for a database of size 10^6 and 10K keywords, the permanent user storage is approximately 64B for O-μSE and and 360KB for Q-μSE.

7.2 End-to-End Execution Time with Fabric

In this section, we evaluate the end-to-end performance of our schemes. We run O-μSE and Q-μSE over five AWS machines (in Sweden, Germany, Ireland, Italy, and France) connected over WAN with 20ms average latency. We configure a Hyperledger Fabric blockchain over four of these machines and use the last one as the data server. Each of

the blockchain machines hosts a peer node of Fabric. One of them runs as the data owner (also operating a Fabric Orderer and Certificate Authority) while the rest run users.

Figure 4(c) shows the search time breakdown for variable $|DB|$ and result size 100 (**F** denotes the time related to Fabric operations, **S** the server time, and **U** the user computation time). O-μSE is considerably slower than Q-μSE due to the extra Fabric transaction for updating the digest. Furthermore, Q-μSE performs similarly for both queue sizes (100 or 10K). Despite the difference in the number of counter updates, Fabric transactions and roundtrips (which is the bottleneck of search time) are the same. In general, Q-μSE is 19.4-21.2$\times$ faster than O-μSE depending on the database size (e.g., the search time of Q-μSE and O-μSE is approximately 234ms and 4.6s for $|DB| = 10^6$).

We also evaluated the search performance for database size 1M and variable result size. As Figure 4(d) shows, for small result sizes O-μSE is considerably slower than Q-μSE. However, as the result size increases, the user computation time (in order to extract the ids) becomes the dominating factor (the user and server computation time vary from 1% and 8% of the computation to 33% and 35% while the Fabric cost decreases from 91% to 31%). Since this is the same for all schemes, the performance gap between them closes.

Finally, Figure 6(b) shows the update time breakdown for variable database sizes. The performance gap between O-μSE and Q-μSE is mainly the result of the extra Fabric transaction of O-μSE for updating the Merkle root of the OMAP (the Fabric cost is >93% of the total update time). According to our experiments, Q-μSE is 2-2.1$\times$ faster than O-μSE for update operations (e.g., Q-μSE and O-μSE update time are roughly 4s and 8.5s respectively).

7.3 Comparison with Previous Static MUSSE Schemes

Here, we compare the performance of μSE with MSKE [28] and mx-u [41]. Recall that both these schemes are static and do not support updates after the initial setup, hence we can only compare search times.

Figure 5(c) compares the search computation times of μSE with MKSE, for variable result size and $|DB| = 1M$. Clearly, MKSE is slower than both our schemes. This is expected as its performance is proportional to the total number of files in the dataset while the search time of μSE only depends on the actual result size. Obviously, as the result size increases, the performance of all schemes becomes similar. Throughout these experiments, Q-μSE and O-μSE are 3-417$\times$ and 3-39$\times$ faster than MKSE. The effect of the number of files on the performance of the schemes is depicted in Figure 5(d) which shows the search time for variable number of files, and fixed $|DB| = 1M$ and result size 100. As the figure shows, MKSE search time increases with the number of files and is considerably slower than μSE while our schemes' performance is virtually unaffected by the number of files in the dataset (recall that the performance of μSE is linear in the result size which is fixed for this experiment). In general, Q-μSE and O-μSE are 4-279$\times$ and 39-50$\times$ faster than MKSE (e.g., Q-μSE and O-μSE take less than 46ms while MKSE takes 180-230ms).

Turning our attention to the mx-u scheme, the comparison is more straight-forward, as it requires a number of group exponentiations that is linear in the number of files in the dataset which makes it significantly slower than our schemes. According to the reported numbers (Fig 1.c of [41]), its search time in a database of 10^4 files is 10.5s while both our schemes take less than 46ms to search in a similar dataset and on the same machine (228-12635$\times$ faster). Moreover, recall that mx-u has cross-user leakage which makes it less secure than μSE. That said, it has the advantage that its encrypted dataset size (stored at the server) does not grow with the number of users, as is the case for μSE and MKSE.

7.4 Discussion

Performance Trade-offs Between our Schemes. Our experimental evaluation allows us to draw some conclusions about the trade-offs offered by our two schemes and identify in which cases one is preferable to the other. First, updates are always much more efficient with Q-μSE hence in applications where update performance is important this should be the adopted solution. On the other hand, with Q-μSE the search performance depends on the number of updates executed since the previous search for this user; in theory this can be huge, as large as $|DB|$ itself. O-μSE achieves a better worst-case bound as retrieving the word counter always takes only one OMAP operation. However, due to the additional roundtrips for the OMAP, in practice Q-μSE would only be worse in terms of search for extreme cases where a user has been "offline" for very long periods.

The main drawback of Q-μSE is that it requires dedicated permanent storage at the client side for the keyword-counter pairs. Assuming 10^6 keywords, each of size 16Bytes plus one update counter, this would result in at least 20MB of local storage. Although this is small enough for most applications, it could be problematic for applications that support multi-device access for the same user (see discussion in [17]). On the other hand, with O-μSE each user only stores its secret key (as the OMAP stash is stored on the server), which makes it better for applications where "portability" of access is required.

Deploying μSE with TEE. One possible way to improve our O-μSE scheme's efficiency is by relying on a TEE at the server. The advantage would be that client-side overheads would be delegated to this TEE, in particular the OMAP client-side routines that O-μSE uses for storing keyword counters, including storing the OMAP stash inside the enclave. This approach would eliminate the need for multiple communication roundtrips that, as shown in our experimental evaluation, is currently the main bottleneck.

However, recall that to avoid additional leakage due to enclave memory accesses, we must ensure that all algorithms executed in TEE are oblivious. To eliminate this in TEE-O-μSE we would have to replace the OMAP of [60] with a *doubly-oblivious map* [39]. In practice this would incur additional overhead but we still believe the performance benefits make it an interesting goal for future work.

8 Conclusion

In this work, we studied the problem of dynamic multi-user searchable encryption (DMUSSE) that is secure against

corrupted users that may collaborate with the server. We proposed a formal security definition, as well as appropriate forward and backward privacy notions for this setting. We developed the first provably secure and forward-backward-private DMUSSE schemes and described how they can be modified to achieve result verifiability. We prototyped both our constructions and experimentally evaluated their performance both on a stand-alone machine and when implemented over the Hyperledger Fabric permissioned blockchain in order to facilitate efficient digest dissemination. Our experiments show that our schemes have very low practical overheads, outperforming previous ones that do not support updates and/or suffer from cross-user leakage.

Our work leaves many open problems, such as investigating whether we can remove the oblivious data structure from our O-μSE construction to reduce the communication overhead, and to leverage TEE to improve our schemes' efficiency. Furthermore, it would be interesting to examine whether it is possible to propose a generic secure compiler for efficiently converting single-user DSSE schemes to (forward-and-backward private) DMUSSE.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by Hong Kong RGC grant ECS-26208318 and a Huawei HIRP award.

REFERENCES

[1] James Alderman, Keith M Martin, and Sarah Louise Renwick. Multi-level access in searchable symmetric encryption. In *International conference on Financial Cryptography and Data Security*, 2017.

[2] Ghous Amjad, Seny Kamara, and Tarik Moataz. Forward and backward private searchable encryption with sgx. In *Proceedings of the 12th European Workshop on Systems Security*, pages 1–6, 2019.

[3] Foteini Baldimtsi, Dimitrios Papadopoulos, Stavros Papadopoulos, Alessandra Scafuro, and Nikos Triandopoulos. Server-aided secure computation with off-line parties. In *ESORICS 2017, Proceedings, Part I*, volume 10492, pages 103–123, 2017.

[4] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *EUROCRYPT*, 2004.

[5] Raphael Bost. $\Sigma o \varphi o \varsigma$: Forward secure searchable encryption. In *ACM CCS*, pages 1143–1154, 2016.

[6] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. *IACR Cryptol. ePrint Arch.*, 2016:62, 2016.

[7] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM CCS 2017*, pages 1465–1482.

[8] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security 2018,*. USENIX Association.

[9] Chengjun Cai, Jian Weng, Xingliang Yuan, and Cong Wang. Enabling reliable keyword search in encrypted decentralized storage with fairness. *IEEE TDSC*, 18(1):131–144, 2021.

[10] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, M Rosu, and Michael Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*, 2014.

[11] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO 2013*, pages 353–373. 2013.

[12] 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.

[13] 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.

[14] Yaxing Chen, Qinghua Zheng, Zheng Yan, and Dan Liu. Qshield: Protecting outsourced cloud data queries with multi-user access control based on SGX. *IEEE Trans. Parallel Distributed Syst.*, 32(2):485–499, 2021.

[15] Rong Cheng, Jingbo Yan, Chaowen Guan, Fangguo Zhang, and Kui Ren. Verifiable searchable symmetric encryption from indistinguishability obfuscation. In *ASIA CCS*, 2015.

[16] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 79–88. ACM, 2006.

[17] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[18] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *CRYPTO 2018*, pages 371–406, 2018.

[19] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security 2020*, pages 2433–2450, 2020.

[20] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *PoPETs*, 2018(1):5–20, 2018.

[21] Hyperledger Fabric. https://hyperledger-fabric.readthedocs.io/en/release-1.4/whatis.html.

[22] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and secure index with SGX. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 386–408. Springer, 2017.

[23] Gartner Risk Management. Top 10 emerging risks of q2 2018. https://www.gartner.com/ngw/globalassets/en/risk-audit/documents/top-ten-emerging-risks-Q2-2018.pdf, 2018.

[24] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[25] Javad Ghareh Chamani, Dimitrios Papadopoulos, C. Papamanthou, and R. Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, 2018.

[26] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[27] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS' 16*, pages 1353–1364, 2016.

[28] Ariel Hamlin, Abhi Shelat, Mor Weiss, and Daniel Wichs. Multi-key searchable encryption, revisited. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018*, pages 95–124. Springer, 2018.

[29] Shengshan Hu, Chengjun Cai, Qian Wang, Cong Wang, Xiangyang Luo, and Kui Ren. Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization. In *IEEE Conference on Computer Communications, INFOCOM 2018*, 2018.

[30] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011.

[31] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *ACM CCS 2012*.

[32] Masahiro Kamimura, Naoto Yanai, Shingo Okamura, and Jason Paul Cruz. Key-aggregate searchable encryption, revisited: Formal foundations for cloud applications, and their implementation. *IEEE Access*, 8:24153–24169, 2020.

[33] Aggelos Kiayias, Ozgur Oksuz, Alexander Russell, Qiang Tang, and Bing Wang. Efficient encrypted keyword search for multi-user data sharing. In *ESORICS*, 2016.

[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security 2018*, pages 973–990, 2018.

[35] Zheli Liu, Zhi Wang, Xiaochun Cheng, Chunfu Jia, and Ke Yuan. Multi-user searchable encryption with coarser-grained access control in hybrid cloud. In *IEEE EIDWT*, 2013.

[36] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[37] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[38] Antonis Michalas, Alexandros Bakas, Hai-Van Dang, and Alexandr Zalitko. MicroSCOPE: Enabling Access Control in Searchable Encryption with the Use of Attribute-Based Encryption and SGX. In *NordSec 2019, Proceedings*, pages 254–270, 2019.

[39] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *IEEE Symposium on Security and Privacy (SP)*, pages 279–296.

[40] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. Dynamic searchable encryption via blind storage. In *2014 IEEE Symposium on Security and Privacy, SP 2014*, pages 639–654, 2014.

[41] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Symmetric searchable encryption with sharing and unsharing. In *ESORICS 2018*, pages 207–227. Springer, 2018.

[42] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using Mylar. In *USENIX NSDI 14*.

[43] Raluca Ada Popa and Nickolai Zeldovich. Multi-key searchable encryption. Cryptology ePrint Archive, Report 2013/508, 2013.

[44] The OpenSSL Project. https://www.openssl.org/, 2003.

[45] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, 2000.

[46] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, 2014.

[47] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS' 13*.

[48] Yaping Su, Jianfeng Wang, Yunling Wang, and Meixia Miao. Efficient verifiable multi-key searchable encryption in cloud computing. *IEEE Access*, 7:141352–141362, 2019.

[49] Shi-Feng Sun, Joseph K Liu, Amin Sakzad, Ron Steinfeld, and Tsz Hon Yuen. An efficient non-interactive multi-client searchable encryption with support for boolean queries. In *ESORICS' 16*.

[50] Shi-Feng 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 *CCS*, 2018.

[51] Shi-Feng Sun, Cong Zuo, Joseph K Liu, Amin Sakzad, Ron Steinfeld, Tsz Hon Yuen, Xingliang Yuan, and Dawu Gu. Non-interactive multi-client searchable encryption: Realization and implementation. *IEEE TDSC*, 2020.

[52] Roberto Tamassia. Authenticated data structures. In *European symposium on algorithms*, pages 2–5. Springer, 2003.

[53] Cédric Van Rompay, Refik Molva, and Melek Önen. Multi-user searchable encryption in the cloud. In *Information Security*, pages 299–316, 2015.

[54] Cédric Van Rompay, Refik Molva, and Melek Önen. A leakage-abuse attack against multi-user searchable encryption. *Proceedings on Privacy Enhancing Technologies*, 2017(3):168 – 178, 2017.

[55] Cédric Van Rompay, Refik Molva, and Melek Önen. Fast two-server multi-user searchable encryption with strict access pattern leakage. In *ICICS*, 2018.

[56] Cédric Van Rompay, Refik Molva, and Melek Önen. Secure and scalable multi-user searchable encryption. In *SCC@AsiaCCS*, 2018.

[57] Viet Vo, Shangqi Lai, Xingliang Yuan, Surya Nepal, and Joseph K Liu. Towards efficient and strong backward private searchable encryption with secure enclaves. In *ACNS 2021*, pages 50–75, 2021.

[58] Viet Vo, Shangqi Lai, Xingliang Yuan, Shi-Feng Sun, Surya Nepal, and Joseph K Liu. Accelerating forward and backward private searchable encryption using trusted execution. In *ACNS 2020*, pages 83–103, 2020.

[59] Guofeng Wang, Chuanyi Liu, Yingfei Dong, Peiyi Han, Hezhong Pan, and Binxing Fang. Idcrypt: A multi-user searchable symmetric encryption scheme for cloud applications. *IEEE Access*, 6:2908–2921, 2018.

[60] Xiao S. Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *CCS' 14*.

[61] Yun Wang and Dimitrios Papadopoulos. Multi-User Collusion-Resistant Searchable Encryption with Optimal Search Time. In *AsiaCCS 2021*, pages 252–264, 2021.

[62] Lei Xu, Chungen Xu, Joseph K Liu, Cong Zuo, and Peng Zhang. A multi-client dynamic searchable symmetric encryption system with physical deletion. In *ICICS*, 2017.

[63] Yanjiang Yang, Haibing Lu, and Jian Weng. Multi-user private keyword search for cloud computing. In *IEEE CLOUDCOM*, 2011.

[64] Hyundo Yoon and Junbeom Hur. A comparative analysis of searchable encryption schemes using sgx. In *ICTC 2020*, pages 526–528. IEEE, 2020.

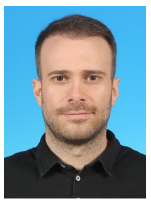
[65] Yinghui Zhang, Robert H Deng, Jiangang Shu, Kan Yang, and Dong Zheng. TKSE: Trustworthy keyword search over encrypted data with two-side verifiability via blockchain. *IEEE Access*, 6:31077–31087, 2018.

[66] Rang Zhou, Xiaosong Zhang, Xiaojiang Du, Xiaofen Wang, Guowu Yang, and Mohsen Guizani. File-centric multi-key aggregate keyword searchable encryption for industrial internet of things. *IEEE Transactions on Industrial Informatics*, 14(8):3648–3658, 2018.

[67] Rang Zhou, Xiaosong Zhang, Xiaofen Wang, Guowu Yang, and Wanpeng Li. Keyword searchable encryption with fine-grained forward secrecy for internet of thing data. In *ICA3PP*, pages 288–302, 2018.

Javad Ghareh Chamani received his M.Sc. degree in software engineering from Sharif University of Technology, Iran, in 2014. He is currently pursuing a Dual Ph.D. degree at the Hong Kong University of Science and Technology and Sharif University of Technology. His research interests include searchable encryption, database security, data outsourcing, and secure cloud-computing.

Yun Wang received a MPhil degree in computer science and engineering from the Hong Kong University of Science and Technology in 2020. She is currently a PhD student at the Hong Kong University of Science and Technology and her research interests include cloud computing, cloud security, and database outsourcing.



Dimitrios Papadopoulos is an assistant professor at the Computer Science and Engineering Department of the Hong Kong University of Science and Technology. He received his Ph.D. in computer science from Boston University in 2016 and has published multiple papers in international conferences and journals. His research is focused on the development of cryptographic protocols for verifiable computation, zero-knowledge proofs, searchable encryption, oblivious computation, and other applications.

Mingyang Zhang graduated from Tsinghua University, China in 2018 with a master's degree, and is working at Huawei's Poisson Lab as a blockchain software engineer.

Rasool Jalili received his Ph.D. in computer science from University of Sydney, Australia, in 1995. He then joined the Department of Computer Engineering, Sharif University of Technology in 1995. He has published more than 140 papers in international journals and conference proceedings. He is now an associate professor,doing research in the areas of computer dependability and security, access control, distributed systems, and database systems in his Data and Network Security Laboratory.

APPENDIX

APPENDIX A
OMAP

The OMAP of [60] is based on storing an AVL-tree inside PathORAM [47]. Each AVL-tree node stores a key, a value, and some information about corresponding PathORAM nodes. Each OMAP find/insert operation consists of a logarithmic number of PathORAM accesses to traverse the height of the AVL-tree including dummy accesses, if necessary, to reach maximum height. In what follows, we explain this OMAP API briefly. We refer interested readers to [60] for more detailed descriptions.

- $(T, root) \leftarrow$ SETUP$(1^\lambda, N)$: Given security parameter λ, and an upper bound N on the number of elements, the client executes $T \leftarrow$ ORAM.INITIALIZE$(1^\lambda, N)$ to initialize an ORAM (PathORAM). Then, it creates the root of the AVL tree by allocating an empty node $rootID$, assigns a random position $rootPos$, and stores it in T. Finally, the client uploads T to the server and maintains the ORAM state and $root$ information locally.
- $(root', data) \leftarrow$ FIND$(key, root)$: Given the search key key and the root $root$, it returns the corresponding value $data$ using an interactive protocol between the client and server. In the first step, the client fetches the tree root and compares the given key with the retrieved one. Based on the comparison result, it retrieves the left or the right child using the PathORAM position stored in the root node. It repeats this process until the node with key key is found (if it exists) and pads the number of accesses with dummies to hide the depth at which the block was found. Next, all the fetched nodes are re-mapped to new random positions in the PathORAM, re-encrypted, and sent to the server with PathORAM eviction. The client also updates its local state and root information.
- $root' \leftarrow$ INSERT$(key, val, root)$: Given a key-value pair (key, val) and $root$, it inserts this entry to the map, in a similar manner as FIND except that it creates a new node for the entry. After insertion, it may have to execute a BALANCE sub-protocol to re-balance the AVL tree.

APPENDIX B
PROOF OF THEOREM 1

B.1 O-μSE Proof

We prove the security of O-μSE using the following games. It is important to note that all the following descriptions for the simulator are related to honest users. For the corrupted users, the simulator executes real operations because the adversary has access to all inputs and states of the corrupted users (all this information is given to the simulator as input). Therefore, any execution other than the actual execution can be distinguished by the adversary.

Game-0 This is the $\mathsf{Real}^{U,\mathcal{C},\Sigma}$ game as defined in Figure 2.

Game-1 This game is the same as **Game**-0, except that the values $G_K(w, \mathsf{cnt}||b)$ for $b = 0, 1$ which are computed during Update and Share operations are replaced by values sampled uniformly at random from the range of G ($\{0,1\}^{\lambda'}$). Since the output of G is used for $addr$ construction in the search operation, the simulator makes a list **I**.

Algorithm 5 μSE Simulator

Setup$(1^\lambda, N, |W|, |U|, |D|)$ timestamp = 0
1: **DictW**$\leftarrow$ empty maps with size
2: **UsersKeys**$\leftarrow$ empty map with
3: **Queues** $\leftarrow$ empty map of $|U|$ queues
4: **AccessList** $\leftarrow$ empty map of lists
5: **I** $\leftarrow$ empty list for bookkeeping
6: $EDB \leftarrow$ (**DictW**, **Queues**, **AccessList**)
7: Return EDB

Enroll$(1^\lambda, u \in U)$
1: **if** u is corrupted **then**
2: $K_u \leftarrow Gen(1^\lambda)$
3: Set **FileCnts**[u] to 0 for all keywords
4: **if** u is OMAP-Based **then**
5: **UsersKeys**[u]= K_u; Run OMAP.SETUP$(1^\lambda, |W|)$
6: Return K_u, the OMAP and its key
7: **else**
8: $K_u^{'} \leftarrow KeyGen(1^\lambda)$; **UsersKeys**[$u$]= $(K_u, K_u^{'})$
9: Return $(K_u, K_u^{'})$
10: **else if** u is OMAP-Based **then**
11: Return simulated OMAP.SETUP$(1^\lambda, |W|)$
12: **else** $K_u^{'} \leftarrow KeyGen(1^\lambda)$; **UsersKeys**[$u$]= $K_u^{'}$

Share$(id, u, KwLeakage(u, id, t), mod)$
1: **AccessList**[id] $\leftarrow$ **AccessList**[id] $\cup\{u\}$
2: KeyValues $\leftarrow\perp$; CntDiffs $\leftarrow\perp$
3: Load K_u,$K_u^{'}$ from **UsersKeys**[u] based on u type
4: **if** u is corrupted **then**
5: **for** w in $Kw^t(id)$ **do**
6: **if** $u.type$ is OMAP-based **then**
7: **if OMAP**$_u[w] = \perp$ **then OMAP**$_u[w]$=0
8: **OMAP**$_u[w]$++ ; cnt $\leftarrow$ **OMAP**$_u[w]$
9: **else if** $u.type$ is Queue-based **then**
10: **if FileCnts**$[u][w] = \perp$ **then FileCnts**$[u][w] = 0$
11: **FileCnts**$[u][w]$++ ; cnt $\leftarrow$ **FileCnts**$[u][w]$
12: CntDiffs $\leftarrow$ CntDiffs $\cup\ (u, Enc_{K_u^{'}}(w, \mathsf{cnt}))$
13: $addr = G_{K_u}(w, \mathsf{cnt}||0)$
14: $val = (id||add) \oplus G_{K_u}(w, \mathsf{cnt}||1)$
15: KeyValues $\leftarrow$ KeyValues $\cup\ (addr, val)$
16: **else**
17: **for** 1 to $|Kw^t(id)|$ **do**
18: **if** $u.type$ is OMAP-based **then**
19: execute OMAP simulator
20: **else if** $u.type$ is Queue-based **then**
21: CntDiffs $\leftarrow$ CntDiffs $\cup\ (u, Enc_{K_u^{'}}(0))$
22: $addr \xleftarrow{\$} \{0,1\}^{\lambda'}$; $val \xleftarrow{\$} \{0,1\}^{\lambda'}$
23: **I**[timestamp, u]=$(addr, val)$; timestamp++
24: KeyValues $\leftarrow$ KeyValues $\cup\ (addr, val)$
25: Return KeyValues, CntDiffs, and (u, id)

Let us assume that the timestamp of the update/share is i. The simulator stores entry $(addr, val, u, w, id, op, type)$ in the $i-$th cell (timestamp) of **I** for all $u \in AccList(id)$ and all $w \in WList$ in Update and for all $w \in Kw^t(id)$ in Share, where type is $share$ or $update$ and op is add or del. During Unshare, the simulator only removes the user

Algorithm 6 μSE Simulator

UnShare(id, u)
1: **AccessList**$[id] \leftarrow$ **AccessList**$[id] \setminus \{u\}$

Update$(id, WLeakage(u, WList), op, AccList(id))$
1: userList $\leftarrow$ **AccessList**$[id]$
2: KeyValues $\leftarrow\perp$; CntDiffs $\leftarrow\perp$
3: **for** u in userList **do**
4: Load K_u, $K_u^{'}$ from **UsersKeys**[u] based on u type
5: **if** u is corrupted **then**
6: **for** w in $WList$ **do**
7: **if** $u.type$ is OMAP-based **then**
8: **if OMAP**$_u[w] = \perp$ **then OMAP**$_u[w]$=0
9: **OMAP**$_u[w]$++ ; cnt $\leftarrow$ **OMAP**$_u[w]$
10: **else if** $u.type$ is Queue-based **then**
11: **if FileCnts**$[u][w] = \perp$ **then**
12: **FileCnts**$[u][w] = 0$
13: **FileCnts**$[u][w]$++ ; cnt $\leftarrow$ **FileCnts**$[u][w]$
14: CntDiffs $\leftarrow$ CntDiffs $\cup (u, Enc_{K_u'}(w, \text{cnt}))$
15: $addr = G_{K_u}(w, \text{cnt}||0)$
16: $val = (id||op) \oplus G_{K_u}(w, \text{cnt}||1)$
17: KeyValues $\leftarrow$ KeyValues $\cup (addr, val)$
18: **else**
19: **for** 1 to $|WList|$ **do**
20: **if** $u.type$ is OMAP-based **then**
21: Execute OMAP simulator
22: **else if** $u.type$ is Queue-based **then**
23: CntDiffs $\leftarrow$ CntDiffs $\cup (u, Enc_{K_u'}(0))$
24: $addr \xleftarrow{\$} \{0,1\}^{\lambda'}$; $val \xleftarrow{\$} \{0,1\}^{\lambda'}$
25: KeyValues $\leftarrow$ KeyValues $\cup (addr, val)$
26: **I**[timestamp, u]=$(addr, val)$; timestamp++
27: Return KeyValues and CntDiffs.

Search$(u, \textbf{Time}(w, u), \textbf{Update}(w, u), SrchLeakage(w, u))$
1: TList = { }
2: **if** u is corrupted and $u.type$ is OMAP-based **then**
3: cnt $\leftarrow$ **OMAP**$_u[w]$
4: **else if** u is honest and $u.type$ is OMAP-based **then**
5: Execute OMAP simulator
6: **else if** $u.type$ is Queue-based **then**
7: fetch **Queues**$[u]$ from the server
8: **if** u is corrupted **then**
9: **for** $i = 1$ to **FileCnt**$[u][w]$ **do**
10: TList = TList $\cup \{G_{K_u}(w, i||0)\}$
11: **else**
12: TList$\leftarrow$Extract $addr$s from list **I** using **Update**(w, u)
13: Return TList and the result R using **Time**(u, w)

from the access list of the target file. In a search operation for keyword w, it performs a scan over **I** to identify the entries that match w (instead of creating **TList** by PRF evaluation) and sends their corresponding $addr$s to the server. Furthermore, it constructs set R which contains files currently containing the target keyword for the target user and sends R to the server too. Since in **Game**-0 the PRF is never computed on the same input twice during updates and shares, **Game**-1 is indistinguishable from **Game**-0 from the security of the PRF.

Game-2 This game is the same as **Game**-1, except that all OMAP operations (SETUP, INSERT, and FIND) are replaced with corresponding OMAP simulated operations for each user, computed by the OMAP simulators that exist from the OMAP security. The simulator also expands list **I** of **Game**-1 to store all the information about insertions and modifications of files (such as update time and file creation time) during Update and Share. During search operations, first the simulator executes the OMAP simulator find. Furthermore, the counter of the target keyword in the search operation (which was previously retrieved from OMAP) is derived from **I**. Specifically, it scans **I** to extract $addr$s related to the target user and target keyword and sends the corresponding entries to the server. Finally, it scans **I** and extracts search result R (files contain search keyword for target user). Since OMAP is secure, **Game**-2 is indistinguishable from **Game**-1.

Game-3 This game is the same as **Game**-2, except that we remove w and id from table **I**. The latter was used in Update/Share operations to construct val by computing $id||op \oplus r$, where r was chosen uniformly at random. Since the output of such XOR operations is also distributed uniformly at random, we can replace val by a value sampled uniformly at random. Therefore, during the update/share, there would not be any need for id anymore. During the search, we assume that the simulator gets some external information including ***Time***(w, u) and ***Update***(w, u). Based on these, it uses ***Update***(w, u) to find all $addr$s and uses ***Time***(w, u) to find R. Observe that ***Time***(w, u), ***Update***(w, u), and the modified version of **I** (which does not have w and id) provide exactly the same information as **Game**-2. Therefore, **Game**-3 is indistinguishable from **Game**-2.

Game-4 This is $\mathsf{Ideal}^{U,\mathcal{C},\Sigma}$ as defined in Figure 3 where the Sim runs the code described in the previous game.

The pseudocode of the μSE simulator for both honest and corrupted users is presented in Algorithms 5 and 6.

B.2 Q-μSE Proof

The proof of Q-μSE is the same as O-μSE, except that in **Game**-2, instead of simulating OMAP we replace all encryptions of counter updates with encryptions of 0's. The games for the Q-μSE security proof are as follows:

Game-0 This is the $\mathsf{Real}^{\mathcal{UC},\Sigma}$ game as defined in Figure 2.

Game-1 The same as **Game**-1 in Appendix B.1.

Game-2 This game is the same as **Game**-1, except that during Update/Share it sends encryptions of 0s for all users in $AccList(id)$ to the server. During Search operations, first it retrieves these "encrypted" queues. Then, it scans **I** to extract $addr$s of updates that have been executed for the target user and target keyword to send to the server. Finally, it scans **I** to extract the set R which contains files currently containing the target keyword for the user. **Game**-2 is indistinguishable from **Game**-1 from the semantic security of the encryption scheme.

Game-3 The same changes to **Game**-2 as those described in **Game**-3 in Appendix B.1.

Game-4 Same as **Game**-4 in Appendix B.1.

APPENDIX C
HYPERLEDGER FABRIC PSEUDOCODE

To implement verifiable μSE in Hyperledger Fabric, we need to define three components in the Fabric architecture. (i) *Model* defines the entities and their relation in the blockchain. (ii) *Permission rules* describe the resources each participant has access to. (iii) *Transaction logic* explains the operations that need to be executed for each blockchain transaction. The pseudocode of these components for verifiable μSE are provided in Algorithms 7, 8 and 9.

Algorithm 7 Hyperledger Fabric Model

```
asset MerkleRoot identified by rootId {
o String rootId
o String rootHash
→ Owner owner
}
participant Owner identified by ownerId {
o String ownerId
o String secret
}
participant Client identified by clientId {
o String clientId
}
transaction ChangeRoot {
→ MerkleRoot root
o String newHash
o String secret
}
event RootChangeNotification {
→ MerkleRoot root
}
```

Algorithm 8 μSE Transaction Logic

```
function changeRoot(rootChange) {
if (rootChange.secret == rootChange.root.owner.secret){
rootChange.root.rootHash = rootChange.newHash;
                                        ▷ Update root hash
return getAssetRegistry("MerkleRoot")
    .then(function(assetRegistry){
      {return assetRegistry.update(rootChange.root);})
    .then(function(){
      {var event = getFactory().newEvent(
        "RootChangeNotification");        ▷ Create event
      event.root = rootChange.root;
      emit(event);});                     ▷ Fire off the event
}}
```

Algorithm 9 Hyperledger Fabric Permissions

```
rule EveryoneAccessSystemRessources {
description: "Grant r/c access to system resources"
participant: "**"
operation: READ, CREATE
resource: "org.hyperledger.composer.system.**"
action: ALLOW
}
rule OwnerAccessChangeroot {
description: "Grant owner access to create ChangeRoot"
participant: "Owner"
operation: CREATE
resource: "ChangeRoot"
action: ALLOW
}
rule OwnerAccessItsObject {
description: "Grant owner u/c access to its own record"
participant(t): "Owner"
operation: UPDATE, CREATE
resource(v): "Owner"
condition: (v.getIdentifier() == t.getIdentifier())
action: ALLOW
}
rule OwnerAccessItsMerkleTree {
description: "Grant owner r/u/c to its Merkle tree"
participant(t): "Owner"
operation: READ,UPDATE, CREATE
resource(v): "MerkleRoot"
condition: (v.owner.getIdentifier() == t.getIdentifier())
action: ALLOW
}
rule ClientAccessMerkleRoot {
description: "Grant client read access to Merkle trees"
participant: "Client"
operation: READ
resource: "MerkleRoot"
action: ALLOW
}
```