OblivShare: Towards Privacy-Preserving File Sharing with Oblivious Expiration Control

Yanjun Shen[1,2], Xingliang Yuan[1], Shi-Feng Sun[3], Joseph K. Liu[1(✉)], and Surya Nepal[2]

[1] Department of Software Systems and Cybersecurity, Faculty of Information Technology, Monash University, Melbourne, Australia
joseph.liu@monash.edu

[2] Data61, CSIRO, Canberra, Australia

[3] Shanghai Jiao Tong University, Shanghai, China

Abstract. People have personal and/or business need to share private and confidential documents; however, often at the expense of privacy. Privacy aware users demand that their data is secure during the entire life cycle, and not residing in clouds indefinitely. A trending feature in industry is to set download constraints of shared files - a file can be downloaded for a restricted number of times and/or within a limited time framework. Metadata privacy becomes concerning with web services and applications providing such additional level of security control but not hiding the metadata. There is no prior research focusing on privacy-preserving expiration control, hence we propose OblivShare, a privacy-preserving file sharing scheme to proactively fill the gap. The scheme is based on ORAM for secure computation that 1) supports file expiration at users' control, 2) hides expiration metadata from the server, 3) server is fully oblivious of file access pattern and expiration state of a file. We demonstrate that our protocol has a complexity poly-logarithmic to the number of files while achieving privacy of metadata.

Keywords: E2EE file sharing · Metadata privacy · ORAM · Secure computation

1 Introduction

Users sharing files with other users over the Internet are common practices today. However, data leakages and mass surveillance projects [16,27] have drawn public attention of the vulnerability and sensitivity of personal data, and in turn promoted privacy awareness of users. Further, existing regulations and acts to protect personal data [19,31], also impose on service providers to grant individuals control over their private information. Therefore, sharing files securely and privately is becoming a fundamental requirement.

In order to achieve secure file sharing, systems and services have been developed to support end-to-end encryption (E2EE) [29], using which, a user encrypts file content before it leaves their device and only authorised users are able to decrypt the file. However, E2EE does not appear to fully protect the privacy of user or file metadata, and a file can stay in servers indefinitely. Recent innovative services [7,17] provide impermanence of data store on top of E2EE, which offers extra security control to users over the files they share: setting files to expire after a certain amount of time or number of downloads. On the one hand, such services incorporate two most desired features, **E2EE** and **ephemeral**, which meets personal needs of more secure connections and intimate sharing; on the other hand, limitations are also apparent: 1) Users send expiration control metadata (expiration metadata for short in the rest of the paper), i.e. download number and time limits to check if a file has expired, to servers in plaintext, which can be used to deduce the popularity and sensitivity of specific file(s). 2) Expiration control is at servers' hand and users have to fully trust a service to honestly check if a file has expired.

Inadequate discussion has since occurred to understand the privacy of expiration metadata. Therefore, we aim to propose a new protocol to solve this emerging problem with practical values. This is a first attempt to focus on secure file expiration control, and the proposed protocol has not yet been implemented in real cloud environment. Security and performance analysis are provided in the paper, and we consider real experimental evaluation to illustrate the performance in the future.

1.1 Motivation

"If you have enough metadata, you don't really need content", "we kill people based on metadata" [22,23]. Sharing a file resembles calling or messaging someone from the perspective of metadata exposure, hence metadata privacy in file sharing is also concerning. While increasing service providers provide expiration control on top of E2EE, a gap exists in both industry and academia. To illustrate the motivation of hiding expiration metadata and oblivious file sharing, we present some privacy issues even with E2EE file sharing systems.

Sensitivity Derived from Expiration Metadata. Alice is an oncologist, and shares files with patients and other contacts in an E2EE system. Alice shares medical records with her patients and sets each to expire after 1 download, and other files without expiration conditions. With knowledge of the expiration metadata, a curious server learns that Alice shares some files with strict access, hence deduces they are sensitive. Bob is a patient of Alice and downloads his report from the system. With Alice's identity and the sensitivity of the file, the server thus infers Bob is suspected to have cancer without decrypting the report.

1.2 Summary of Contributions

We now propose OblivShare, a secure and ephemeral file-sharing system that for the first time provides users with advanced and oblivious expiration control. OblivShare puts forward a new framework of a file-sharing scheme that not only supports comprehensive file expiration control, but is also expiration-metadata-private and oblivious. This is a generic solution that can be integrated

Table 1. Overview of techniques to achieve the goals.

Goal	Technique
Expiration metadata privacy	Secret sharing
Oblivious expiration control	Secure two party computation
Oblivious file sharing	ORAM
User IP addresses	Anonymous network, e.g., Tor

into file sharing services to address metadata privacy issues. To understand our contributions, we now outline the main challenges OblivShare aims to address.

Challenge 1: how to achieve expiration control over protected expiration metadata? We define expiration metadata as: 1) User-set download constraints, i.e., download number and time limits to facilitate expiration control. 2) Internal download state, i.e., current download count used to compute expiration control outcome. Users are not able to download a file if it has expired. To the best of our knowledge, whereas many scholars focus on protection of general security control metadata in file sharing such as user identity and access pattern, there is no prior research aiming to prevent leakage of expiration metadata, hence a gap exists to address such expiration-metadata-privacy.
Challenge 2: how to make download requests of a specific file indistinguishable from servers? Only hiding the expiration control process, outcome and metadata is not sufficient, as a server can still infer that a file has expired if the specific file has not been accessed for a long time. A server not fully oblivious of the file sharing process learns which file is accessed for each download and can reasonably deduce the expiration metadata.

Contribution. OblivShare supports E2EE meanwhile protects the expiration metadata through the entire course with oblivious file access and expiration control. Our goals and techniques are summarized in Table 1. Overall, our contributions are:

1. We are the first to address metadata privacy issues in file sharing systems that support expiration control. User-defined download constraints are hidden from servers through the entire course of file upload, sharing and download. Internal download state is also protected by secret sharing between servers, therefore the servers cannot directly learn file expiration status.
2. We use synchronised tree-based ORAMs to store both file content and metadata, which hides file access patterns from servers, hence the servers cannot distinguish which file and how many times is requested so as to deduce file expiration status and further expiration metadata.
3. We are the first to use secure computation for oblivious expiration control, which not only guarantees that a single server cannot manipulate the expiration control result, but is also efficient to implement using garbled circuits.
4. We also approve that our scheme has negligible extra computation and communication overhead on top of a primitive ORAM file sharing system, which requires one interaction with users hence not sacrifices user experience.

Table 2. Secure file sharing services.

Product	E2EE	Time limit	Number limit	Hide expiration metadata	Oblivious server
OblivShare	✓	✓	✓	✓	✓
Firefox Send [17]	✓	✓	✓	✗	✗
DropSecure [7]	✓	✓	Future	✗	✗
SendSafely [24]	✓	✓	✗	✗	✗
WhatsApp [34]	✓	Future	✗	✗	✗
Digify [8]	✗	✓	✗	✗	✗
Dropbox [6]	✗	✗	✗	NA	✗

2 Related Work

2.1 Existing Secure File Sharing

Ephemeral content sharing is a highly pursued feature in industry [7,12,17,24, 29,34,37]. With certain expiration control, users are confident that what they share is only accessible to dedicated users for limited time or number of times, and never stay in a server for longer than necessary and become a vulnerability later.

Table 2 compares several existing secure file sharing applications or web services. We organise the comparison by the following properties: 1) Does it support E2EE? 2) Does it support file expiration? 3) Does it hide expiration metadata? 4) Is the server oblivious of file access and expiration control if applicable?

[7,17] and [24] claim to offer zero-knowledge E2EE. [7] (premium) provides client-side encryption that keeps a public key protected encryption key in a key sever (isolated from the file storage server), and only a recipient's private key can decrypt the encryption key. [24] uses OpenPGP encryption and the file encryption key consists of a server secret (generated by the server) and a client secret (generated by the sender). Services such as [6] and [8], though do not support E2EE, but provide an addition layer of password security on top of server-side encryption. A user can double encrypt files or folders by setting a password, and share it to recipients outside the service. [34] also offers E2EE for file attachments and has been developing its "Expiring Messages" feature.

Our solution is aiming to address the security weakness of existing systems mentioned in Sect. 1 with a good balance of desired features and cost.

2.2 ORAM for File Storage

Oblivious RAM (ORAM) [10] is an attempt to hide a user's access pattern from service providers meanwhile supporting extra operations. Traditional ORAM schemes usually have worst-case communication complexity linear to their capacity and block size even with amortized communication cost [18], and their single client setting [10,18,25] is not suitable for file sharing. Multi-user ORAM

Table 3. Notation

Notation	Description
λ	ORAM's statistical security parameters
ts_U	A timestamp that denotes the upload time
ts_D	A timestamp that denotes the download time
ts_{Exp}	A timestamp that denotes the expiration time, that is $ts_U + t$
D	An array of data content stored in ORAM
x	An positive integer that denotes a file index in ORAM, up to the ORAM file number bound, and $D[x]$ is the data stored in ORAM
Exp	An expiration policy that stores download constraints and state indexed by x
$[s]$	A secret share of s
N	The number of real data blocks in ORAM
h	The height of the ORAM tree, that is $\lceil log_2 N \rceil$
θ	A threshold of timestamp difference that is accepted by OblivShare

schemes are promising designs that can be applied in file sharing, but unfortunately, very few of such works exist. Among those that support file sharing, GORAM [14] is a system that guarantees anonymity of users and obliviousness of data access; but it does not protect the owner of a file. PIR-MCORAM [15] is a multi-user ORAM-based file sharing system, but has a very high overhead hence liner worst-case complexity. There are other ORAM schemes that focus on malicious users but do not readily support file sharing [1,11].

At the best of our knowledge, none of the existing ORAM schemes, either hide access patterns and/or user identities or not, with linear or poly-logarithm complexity, has addressed expiration control. With OblivShare, we propose an efficient secure file sharing scheme that not only achieves lightweight system design on top of ORAM (we present performance analysis in Sect. 5 that proves OblivShare has poly-logarithmic complexity), but also enables expiration control while hiding expiration metadata.

3 Preliminaries

OblivShare makes black box use of secure two party computation, and also follows ORAM paradigm for metadata and file storage.

Notation. We define parameters, entities, denotations in OblivShare in Table 3.

3.1 Secure Computation

The Millionaires' Problem first described by Yao [35] enables to solve the following problem: Alice and Bob have their own secret inputs, which are their wealth x^A and x^B million, respectively. Yao's protocol enables that Alice and Bob can compute a function $f(x^A, x^B) \longrightarrow (y^A, y^B)$ such that Alice learns only its function output y^A while Bob knows only y^B, i.e., who is richer, and nothing else about the other party's wealth.

Since Yao's secure computation protocol was proposed, researchers have advanced a number of variations and extensions to address different scenarios. Recent secure multiparty computation (MPC) solutions includes private sorting [13], private computational geometry [26], private voting [30], and private data mining [2,9] etc.

3.2 ORAM

OblivShare deploys ORAM for oblivious data storage and retrieval. More specifically, we use ORAM for secure computation [5,32,36], so as to ensure oblivious data access in MPC applications. There is a class of tree-based ORAM schemes [25,28,32] that are efficient for practical implementations especially in MPC, among which, we consider Circuit ORAM [32] as an appropriate scheme for our setting because of its competitive performance. Comparing to schemes like SqrtORAM [36] and Floram [5], Circuit ORAM client has complexity that is poly-logarithmic to the number of files, and also reduces the circuit size comparing to Path ORAM [28] and SCORAM [33]. Circuit ORAM is a tree-based ORAM. To store N files, Circuit ORAM constructs a binary tree with height $h = \lceil log_2 N \rceil$. The tree is composed of tree nodes, each of which has three blocks with fixed block size; apart from that, it also has a stash (up to the stash size bound) that temporarily stores blocks that will be later evicted to the tree. Each block either stores the data of a file or is left empty. To store a file $D[x]$ in a file array D, a block contains the file index x, the file data $D[x]$, and its position that is the path from leaf to root. If a block is cached in the stash, the block stores the corresponding path that the block will be evicted onto. The file index x and its corresponding path p constitute a position map. We adapt Metal's protocol [3] as a underlying primitive for efficient and oblivious data access in S2PC.

Read from ORAM. To read a file, the two servers first check the file's leaf label (hence corresponding path) in the position map, then search for the block with the file index via a linear scan over both the stash and path. The servers then read the file block stored in the block. After reading the file, they randomly assign a new path to this block, put it back into the stash, and update the position map accordingly.

Write to ORAM. To write a file, the steps are similar until when the two servers add the block into the stash, and they replace it with the data to write provided by the user.

Stash Eviction. Circuit ORAM performs a stash eviction for each read and write operation, at which stage, blocks cached in the stash are evicted to the ORAM tree to prevent stash overflowing. We do not elaborate the eviction algorithms of Circuit ORAM in detail here, but will illustrate rearrangement steps that are relevant to OblivShare. A generic Circuit ORAM data access operation is provided in Algorithm 1.

Algorithm 1: ORAMAccess

```
1 Input $op, idx, data$
2 Output $returnData$
    1. $label \leftarrow PositionMap[idx]$
    2. $\{idx||label||returnData\} \leftarrow ReadnRemove(idx, label)$
    3. $PositionMap[idx] \leftarrow UniformRandom(0, ..., N-1)$ //update position map
    4. if $op = "read"$ then
        (a) $data \leftarrow retrunData$
    5. $stash.add(\{idx||PositionMap[idx]||data\})$
    6. Evict()
    7. Outputs $returnData$
```

3.3 Synchronised Inside-Outside ORAM Trees

We identify that the **synchronising inside-outside ORAM trees** technique used by METAL [3] is suitable for OblivShare. As has been introduced in Sect. 3.2, taking Circuit ORAM as an instance, each block in an ORAM tree contains the file index x, the file data $D[x]$, and its position. METAL, however, splits position map (i.e. index and path position) and actual file data, and stores them in two ORAM trees separately: one tree contains files' indices and positions stays inside S2PC procedures because it is small while the other tree that stores actual file contents stays outside S2PC. The two trees are maintained synchronised during initialisation and after each data access so that the file identifier and content can be found at the same position in the two trees. By doing so, the position of a file can be processed and revealed securely and efficiently in S2PC without loading large file data, and the block fetching and eviction of the actual file data are achieved by two protocols to keep the trees re-synchronised. METAL uses a secret-shared doubly oblivious transfer protocol to ensure that servers fetch the actual file data after revealing the position (in secret shares), and a distributed permutation protocol to track the movement of blocks after eviction and apply the rearrangement to according positions, without any servers learning the actual file's position.

In what follows, we provide some background knowledge of METAL's techniques relevant to our setting and describe more details in Appendix A.

Secret-Shared Doubly Oblivious Transfer. In order to get the actual file block outside S2PC, the two servers first process and reveal the file position inside S2PC, which means that the i-th block on the path p stores the position map and file data respectively in two ORAM trees. The S2PC then generates a list of keys for all the blocks on the path and outputs all these keys to Server 1 that stores the ORAM of actual file data, and Server 2 receives only one key corresponding to the actual file location i. Server 1 then needs to encrypt all the file blocks on the path p using the corresponding keys in order and re-randomise the encrypted blocks before sending them to Server 2. Server 2 uses its key received from the S2PC and decrypt blocks received from Server 1 to obtain the i-th block without either server getting the actual file location.

Distributed Permutation. As has been mentioned in Sect. 3.2, stash eviction is called after every read or write after fetching a data block in ORAM. *Distributed permutation* [3,36] captures the rearrangement of blocks, which is used when putting the read block into stash before eviction and later evicting stash blocks to selected paths. The two servers in S2PC generate a permutation of an array of blocks, including the blocks in the stash, the block read and the block to write; then secret shares the permutation; and apply permutation shares accordingly. The result of the protocol is that the two ORAMs store the permuted blocks in the same location as per the updated file position map. By following this protocol, neither server learns the new position of the file after eviction, and neither server knows which permutation, read or write, is performed.

4 System Overview

In OblivShare, a data owner encrypts a file and sets the file to expire at certain conditions before uploading. OblivShare stores both the cipher file and expiration policy in a secure manner. When a recipient makes a download request to OblivShare, OblivShare first performs expiration control over download constraints and download state, then sends the cipher file to the recipient if the file has not expired. To understand how OblivShare fulfils these operations securely, we present an overview of OblivShare's design, threats and security goals.

4.1 System Architecture

A high-level framework of OblivShare is illustrated in Fig. 1, which consists of two servers, a data owner and multiple clients (in this paper, client is used interchangeably with recipient):

- **Owner** sends upload requests to OblivShare, and shares the file index and file encryption key embedded in a URL to recipients via secure channels.
- **Recipient(s)** sends download requests to the servers, and receives results from OblivShare as per expiration check.
- **Servers** each takes its share of the requests as inputs to the S2PC, and together run S2PC procedures and send the outputs from S2PC to the recipients. The servers also keep updated ExpCtrlORAM and DataORAM, which is explained in detail in Sect. 5.1.

OblivShare incorporates two major components: **OblivExp** for expiration Control and **OblivData** for file access. **OblivExp** is placed in front of **OblivData** to conduct expiration control. A client's request first arrives at **OblivExp**, which checks whether the requested file has expired or not inside the S2PC by the two servers. If no, the request is sent to **OblivData** for a file access. If yes, the request is also dispatched to make the expiration control result indistinguishable to the servers, but in a manner to access dummy data instead. This is a loose description, and detailed construction is elaborated in Sect. 5.

OblivExp updates ExpCtrlORAM after each access and the changes to blocks as a result of stash eviction are applied to DataORAM during **OblivData** via synchronisation between ExpCtrlORAM and DataORAM.

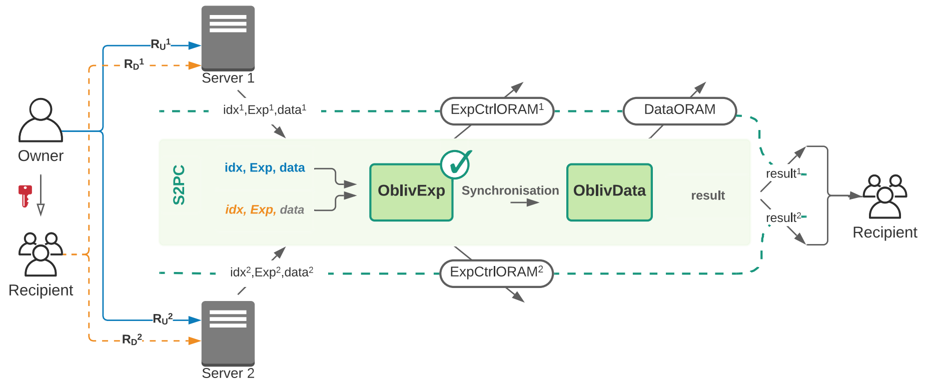

Fig. 1. High level framework. A client sends its secret-shared request of file access to the two servers. The request is reconstructed and executed in S2PC.

4.2 Threat Model

Assumptions. OblivShare makes the following assumptions:

At Least One Server is Honest. An attacker can compromise one of the servers in the two party secure computation while the other is not.

Key Secrecy. A client does not reveal the URL with the key to adversaries.

Out-of-Band Communication. A data owner in OblivShare shares a URL with recipient(s) through third party secured channels of their own control, such as *Telegram* [29] and *Signal* [21]. OblivShare only uses such out-of-band communication once at the sharing stage, which is a common practice of other secure file sharing systems [6, 17], keeping all other activities within OblivShare.

Anonymous Network. In order to hide other metadata during file sharing, OblivShare assumes the clients communicate with servers in an anonymous manner that does not reveal network information via existing tools such as Tor [4] or secure messaging [29] based on decentralised trust.

Secure Communication. Each client establishes secure connections with each server, e.g., Transport Layer Security, so that data in transition are secured.

OblivShare does not address denial-of-service attacks.

Threats. OblivShare considers the following threats:

1. A server can see the expiration metadata of a file. It enables the server to learn data sensitivity and popularity of the file, also deduces other valuable information of encrypted data, which has been explained in Sect. 1.
2. A server on its own has control over its internal download state metadata, hence can forge the state, e.g. a small download count or an expiration timestamp that never expire.

3. A server can observe the file access pattern, hence the server is able to learn which specific file is accessed and the number of times the file has been accessed. If a file no longer receives download request, the server can deduce that the file has expired hence infer the user-set expiration metadata.
4. An attacker controlling a client tries to compromise the security of a file that has expired.
5. A recipient can forge its download timestamp so as to make an invalid download pass the expiration control check.

Security Goals. We now present security goals of OblivShare with respect to the threats given in Sect. 4.2.

1. **Expiration metadata privacy.** OblivShare ensures expiration metadata is totally at a data owner's control, and not visible in transit or at rest on either server.
2. **File confidentiality.** OblivShare ensures that neither server learns the actual file content; further, a compromised client cannot access the encrypted file and decrypt the content after the file has expired.
3. **Oblivious expiration control.** OblivShare ensures both servers are oblivious of the expiration control process. Though OblivShare does not prevent a server from manipulating its download state or a client forging its timestamp, the S2PC procedure for expiration control will fail if it detects compromised inputs to the S2PC. Hence such attack gains no information and little value.
4. **Oblivious file sharing.** OblivShare ensures neither server learns access patterns so that the servers are not able to infer if a specific file has expired hence expiration metadata.
5. **Download timestamp integrity.** OblivShare ensures that the download timestamp is independent of a recipient's input, but is controlled in S2PC.
6. **General metadata protection.** Recall the anonymous network assumptions OblivShare makes in Sect. 4.2, users' IP addresses are garbled when communicating with a server, and OblivShare addresses general metadata privacy in file sharing.

OblivShare guarantees the goals above based on common cryptographic assumptions. However, OblivShare does not address denial-of-service (DoS) attacks, which means OblivShare does not prevent a dishonest server from denying a valid download request even if the time has not expired or the number of downloads permissible has not been exceeded.

5 Detailed Construction

Note that in Circuit ORAM, the linear search of the file index happens within the S2PC, the real data is too large to process. We identify that METAL's synchronised ORAM trees [3] benefits our design to reduce the data accessed inside the S2PC. Below we introduce building blocks of OblivShare. In the following, we present our protocol assuming each file is an single file block for simplicity, but in practice, an uploaded file consists of multiple file blocks and is padded to have the same size.

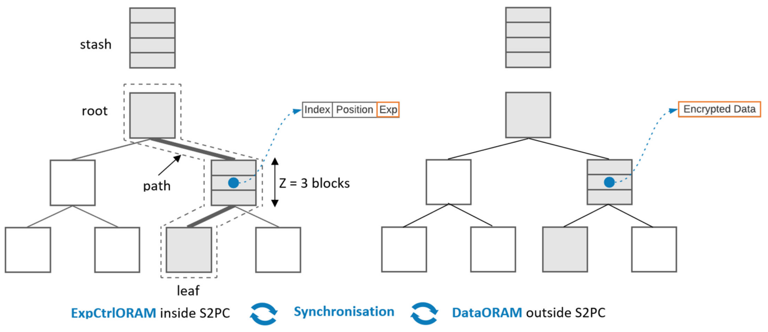

Fig. 2. OblivShare has two tree ORAMs that store small metadata and large real data respectively and synchronised in ExpCtrlORAM and DataORAM.

5.1 Synchronised ORAM Trees

OblivShare has two ORAM trees, an ExpCtrlORAM tree to store the metadata in a recursive manner and a DataORAM tree to store the actual file data. The two trees are synchronised so that the content and the metadata of a file are at the same location in DataORAM and ExpCtrlORAM.

ExpCtrlORAM is a set of trees that recursively stores small metadata, including: 1) the position map; 2) the expiration metadata, i.e. download constraints set by the data owner and the current download state. The ExpCtrlORAM is secret shared with two servers, and will be completed loaded and accessed inside the S2PC. OblivShare uses the standard recursive technique [3,32] to store the metadata in ExpCtrlORAM, and for simplicity purpose, we refer to the last tree when using ExpCtrlORAM in the rest of the paper.

DataORAM resembles ExpCtrlORAM's last tree but only stores the data (encrypted file). The DataORAM tree is stored on Server 1, and only relevant portion of the data structure will be loaded into the S2PC.

Figure 2 shows the ORAM structure in OblivShare and how two ORAMs are synchronised by storing corresponding metadata and data at the same location.

The read and write operations of Circuit ORAM still suffice data retrieval and update in ExpCtrlORAM but no longer fulfil fetching data from and putting data into DataORAM. OblivShare follows MTEAL's *secret-shared doubly oblivious transfer* (SS-DOT) [3] protocol to fetch the i-th block on path p from DataORAM. At the end of SS-DOT, Server 2 obtains the fetched block, i.e. the i-th block (encrypted under ElGamal) in the array, without either server learning i. We describe the details relevant to OblivShare in Appendix A.1. OblivShare also follows *distributed permutation* to make sure ExpCtrlORAM and DataORAM are re-synchronised after each eviction by tracking and permuting block movements. At the end of *distributed permutation*, Server 1 stores the permuted file blocks in DataORAM in the same location as metadata blocks in ExpCtrlORAM. By following this protocol, neither server learns the new location of the evicted block. We give more details of how permutation is created, shared and applied in Appendix A.2.

Algorithm 2: OblivExp.Upload

```
1 Input of S1: $[x]^1, [Exp]^1, [ExpCtrlORAM]^1$
2 Input of S2: $[x]^2, [Exp]^2, [ExpCtrlORAM]^2$
3 Output: $[ExpCtrlORAM]^1, [ExpCtrlORAM]^2$
    1. $x \leftarrow [x]^1 \oplus [x]^2, Exp \leftarrow [Exp]^1 \oplus [Exp]^2, ExpCtrlORAM \leftarrow$
       $[ExpCtrlORAM]^1 \oplus [ExpCtrlORAM]^2$; //reconstruct
    2. $p \leftarrow PositionMap[x]$; //select a path
    3. $(ExpCtrlORAM^{'}, \perp) \leftarrow ORAMAccess(ExpCtrlORAM, x, Exp)$,
       $ExpCtrlORAM \leftarrow ExpCtrlORAM'$; //the distribute permutation protocol will be
       invoked before and after ExpCtrlORAM's stash eviction
    4. $[ExpCtrlORAM]^1 \leftarrow \$, [ExpCtrlORAM]^2 \leftarrow ExpCtrlORAM \oplus [ExpCtrlORAM]^1$;
       //secret share $ExpCtrlORAM$
    5. Output to $[ExpCtrlORAM]^1$ to S1 and $[ExpCtrlORAM]^2$ to S2 respectively.
```

5.2 OblivExp for Expiration Control

Upload a File. During the initialisation stage at the Data Owner's end, it generates an expiration policy Exp locally, which is a description of file download constraints and download state that is shared among the two servers. For example, a policy of a file that expires after 10 downloads and on "21 June 2021 21:21:21" has a policy "File Index x : 10, 21-06-2021T21:21:21, 0" where x is the file index, 10 is the download count (e.g. expire after 10 downloads) chosen by the Data Owner, t_{Exp} is the expiration timestamp derived based on upload timestamp (e.g. 18 June 2021 21:21:21) and download time setting (e.g. expire after 3 days) chosen by the Data Owner, and 0 is the initial download count. The Data Owner also encrypts a file using a secret key before the file is sent to Server 1. Algorithm 2 shows the secure computation during Upload.

Remark. After ExpCtrlORAM's stash eviction, evicted blocks are at new locations but the blocks in DataORAM are not rearranged. To synchronise DataORAM, we use *distributed permutation protocol* by applying the same rearrangement to data blocks in DataORAM. We will elaborate how OblivData ensures the data blocks in DataORAM is still synchronisation in Sect. 5.3.

Download a File. When a client requests a file download by a file index, the two servers search the file index in ExpCtrlORAM, then retrieve the path p and the expiration policy Exp of the file following primitive ORAM read process. The two servers then access DataORAM on the client's behalf and return the file block back to the client via secret shares if the expiration control check passes; otherwise a dummy instead. Note that the S2PC locates the i-th block on the path p in ExpCtrlORAM is the block for file x by a linear search, hence can access the encrypted data of file x in the i-th block of the same path p in DataORAM due to the synchrony between two ORAMs.

During the expiration control check, the two servers inside the S2PC determine a mutually agreed download timestamp (e.g. agree on a deviation threshold θ and then take a mean of the two timestamps from each server), and run the S2PC to compare download constraints to internal download state.

Algorithm 3: OblivExp.Download

```
1 Input of S1: $[x]^1, [ts_D]^1, [ExpCtrlORAM]^1$
2 Input of S2: $[x]^2, [ts_D]^2, [ExpCtrlORAM]^2$
3 Public Input: $\theta$
4 Output: $[ExpCtrlORAM]^1, [ExpCtrlORAM]^2$, $[i]^1$, $[i]^2$
    1. $ts_D \leftarrow agree(ts^1_D, ts^2_D, \theta)$; //agree on a download timestamp
    2. if $ts_D = \perp$ then stop. //the procedure stops if the agreement fails
    3. $x \leftarrow [x]^1 \oplus [x]^2, ExpCtrlORAM \leftarrow [ExpCtrlORAM]^1 \oplus [ExpCtrlORAM]^2$;
    4. $locked \leftarrow Exp(status)$; //check the current lock status
    5. if $locked = TRUE$ then stop.
    6. else $locked \leftarrow TRUE$; //change the lock status to locked
    7. $(ExpCtrlORAM`, \{p, Exp\}) \leftarrow$
       $ORAMAccess(ExpCtrlORAM, x, \perp), ExpCtrlORAM \leftarrow ExpCtrlORAM`$;
       //run an ORAM read operation to get the expiration policy
    8. $i = search(x, p)$; //determine the $i$-th location on path $p$ that stores $Exp$
    9. $r = isValid(Exp(count), Exp(number), Exp(ts_{Exp}), ts_D)$; //expiration check
    10. if $r = TRUE$ then $Exp(count) += 1$; //increments the current download count
    11. else $i \leftarrow |stash| + 3h + 1$; //add a dummy at the end of the array and point $i$ to it
    12. $locked \leftarrow FALSE$ and $Exp(status) \leftarrow locked$; //reset the lock status
    13. Generate $[ExpCtrlORAM]^1$,
        $[ExpCtrlORAM]^2 \leftarrow ExpCtrlORAM \oplus [ExpCtrlORAM]^1$
    14. Generate $[i]^1$, $[i]^2 \leftarrow i \oplus [i]^1$
    15. Output $[ExpCtrlORAM]^1$, $[i]^1$ to S1 and $[ExpCtrlORAM]^2$, $[i]^2$ to S2 respectively.
```

The two servers in the S2PC also update lock status of a file requested on the fly to indicate if the file is being accessed. The file is locked until the data, either the encrypted file or a dummy, has been successfully returned to the client.

To ensure the servers do not know if a file has expired, the S2PC appends a dummy block, and secret share the location i related to this dummy block instead of the actual block if a file has expired (step 11 in Algorithm 3).

Remark. Note that the two servers cannot simply fetch the i-th block in DataORAM, after revealing i in ExpCtrlORAM as the location i is related to the block history, i.e. a location i that is closer to the root level of the ORAM is more likely to have been accessed and evicted recently, and vice versa [20].

To make Upload and Download indistinguishable, expiration policy is constructed as {File Index: download number, expiration timestamp, download count, download timestamp, lock status}, hence {File Index: download number, expiration timestamp, 0, $\perp$, FALSE} for Upload and {File Index: $\perp$, $\perp$, $\perp$, download timestamp, $\perp$} for Download.

5.3 OblivData for File Access

In what follows, we show how the two servers in combination fetch and put a file, which is the same for each ORAM Upload and Download.

Fetch Data from DataORAM. In Sect. 5.2, we already provide a solution of indistinguishable file fetch regardless of expiration status by appending a dummy block. After OblivExp completes the expiration control, either passed or failed, it proceeds the request to OblivData that fetches a block, either a real data block or a dummy depending on the expiration control result, in the form of different values of the location i. As has been briefed in Sect. 5.1, at the end of *SS-DOT*, Server 2 received ElGamal cipher-texts of the data block at the i-th location on path p in DataORAM, with neither server aware of i. Upon ElGamal decipher, the result is either the actual file content or the dummy encrypted under a file encryption key (shared by a data owner to dedicated recipients during the share stage), which is finally returned to the recipient who can further decrypt the result. Algorithm 4 in Appendix A.1 shows how the *SS-DOT* protocol works in OblivData.

Remark. Note that in Algorithm 4, j is independent of i as a result of shuffle, hence Server 2 is not aware of i all through the course.

Evict Data to DataORAM. After ExpCtrlORAM's stash eviction, positions of blocks are updated in ExpCtrlORAM, hence OblivData needs to ensure the corresponding real data blocks in DataORAM are rearranged in the same manner. To guarantee that the two ORAMs are still synchronised, OblivShare tracks the block movements in ExpCtrlORAM and apply the same changes to DataORAM. We use *Distributed Permutation* again during this stage following a similar manner of METAL to re-synchronising trees after each eviction.

Algorithm 5 in Appendix A.2 demonstrates how the re-synchronisation is achieved by tracking the movement of blocks in ExpCtrlORAM and applying the same permutation to DataORAM, hence Server 1 stores the blocks in the corresponding locations.

5.4 Security Guarantees

We now present security guarantees of OblivShare with respect to the goals given in Sect. 4.2.

1. **Expiration metadata privacy.** OblivShare hides expiration metadata from both servers yet is able to enforce the expiration control by using secret sharing. The standard secret sharing technique ensures the shares of the expiration metadata are of the same length hence indistinguishable, and each share reveals no information about the secret (line 1 in Algorithm 2). Also, OblivShare uses ElGamal encryption for data blocks stored in ORAM, which prevents the leakage of sensitive expiration metadata from the servers during the entire course.
2. **File confidentiality.** Data owner of OblivShare encrypts a file before uploading the file to servers and only shares the private key to authorised clients. In addition, all data blocks in ORAM are ElGamal encrypted hence the servers cannot decrypt the actual file content without non-trivial computation. OblivShare also prevents invalid access to expired file by returning a

dummy instead of the encrypted file (ensured by line 11 in Algorithm 3) so that a compromised client cannot retrieve a file that has expired even with the private key.

3. **Oblivious expiration control.** During OblivExp and OblivData, OblivShare uses S2PC protocols to perform expiration control (line 7–11 in Algorithm 3 and the first stage of SS-DOT that samples random keys in line 2(a)–2(d) in Algorithm 4). The security of S2PC guarantees neither server learns or tampers the expiration control result.
4. **Oblivious file sharing.** The obliviousness of ORAM guarantees that file access patterns are hidden from the servers.
5. **Download timestamp integrity.** (line 1–2 in Algorithm 3) The security of S2PC guarantees neither server learns the input timestamp of the other hence cannot modify the actual timestamp or fabricate a new timestamp that is used in the following expiration control operation (line 9 in Algorithm 3).
6. **General metadata protection.** OblivShare makes the anonymous network assumptions in Sect. 4.2 that users' identities and their online activity are encrypted during client-server communications through existing secure tools [4,29]. OblivShare also encrypts metadata such as file name, size, type in the same way as it does for a file hence addresses general metadata privacy in file sharing.

Non-guarantees. As stated in Sect. 4.2, OblivShare does not address DoS attacks by a server, neither protects from malicious server(s), which means OblivShare does not guarantee the availability of a file if a dishonest server denies a valid download request.

5.5 Performance

We consider the system supports N files in total (for simplicity, each file is a single block hence N data blocks) with block size S in DataORAM. As a result of METAL's synchronised inside-outside ORAM trees, the cost for accessing small metadata blocks in ExpORAM is negligible and considered as constant comparing to accessing large data blocks in DataORAM [3]. We use $\mathcal{O}_\lambda(\cdot)$ to present the complexity, while N_{block} is polynomially bounded by λ. We parameterise to have $\frac{1}{N^{\omega(1)}}$ failure probability that is the same as Circuit ORAM [32].

The amortised computational cost of Circuit ORAM is $\mathcal{O}_\lambda((S+\log^2_N)\log N)\cdot\omega(1)$, and OblivShare has minimal additional cost on top of Circuit ORAM. The file access, i.e., read and write operations in OblivShare's `Upload` and `Download` are indistinguishable and have the same cost that includes the cost of Circuit ORAM, SS-DOT and distributed permutation. During `Download`, OblivShare's expiration control incurs additional cost.

The cost for creating download timestamp (line 1–2 in Algorithm 3) is $\mathcal{O}_\lambda(1)$. Expiration control is independent of data blocks in DataORAM and has constant cost $\mathcal{O}_\lambda(1)$ (line 3–12 in Algorithm 3). The total cost of SS-DOT, including Server 1 fetching blocks (line 1(a) in Algorithm 4), the S2PC generating keys (line 2(d)

Table 4. Total computational complexity for `Upload` and `Download` stages.

Stage	Computational complexity
`Upload`	$\mathcal{O}_\lambda((S+\log_N^2)\log N+\log N)\cdot\omega(1)=\mathcal{O}_\lambda((S+\log_N^2)\log N)\cdot\omega(1)$
`Download`	$\mathcal{O}_\lambda((S+\log_N^2)\log N+\log N+1)\cdot\omega(1)=\mathcal{O}_\lambda((S+\log_N^2)\log N)\cdot\omega(1)$

in Algorithm 4), Server 1 encrypting blocks (line 3(b) in Algorithm 4), and the maximum cost of Server 2 decrypting blocks (line 4(b) in Algorithm 4), is linear to the number of blocks fetched on the path and in the stash (with constant size) hence is $\mathcal{O}_\lambda(\log N)$. Distributed permutation also has total cost linear to the blocks on the paths and in the stash (line 1 in Algorithm 5), therefore $\mathcal{O}_\lambda(\log N)$.

Table 4 summaries the above cost and the total computational complexity of OblivShare for both `Upload` and `Download` is $\mathcal{O}_\lambda((S+\log_N^2)\log N)\cdot\omega(1)$. `Upload` and `Download` have the same computational complexity and communication complexity following Metal's protocol [3], which is linear to the file size S and poly-logarithmic to the number of files N.

6 Conclusion

We propose OblivShare, a lightweight privacy-preserving file sharing scheme that for the first time protects expiration metadata together with file access patterns from servers meanwhile ensures oblivious expiration control by adopting cryptography protocols like secure computation and ORAM. We prove that our protocol can achieve its security goals without additional cost that the computation and communication complexity is poly-logarithmic to the number of files. The current framework focuses on semi-honest thread models and we consider malicious security setting as future work. Corresponding prototype implementation and evaluation will also be part of the future work to prove practicality of the proposed protocol.

A METAL's Synchronised Inside-Outside ORAM Trees

A.1 Secret-Shared Doubly Oblivious Transfer

Let N be an array of the blocks in the stash and the $3h$ blocks on path p:

1. The two servers inside S2PC, generate n keys $k_1,\ldots,k_n$ such that S1 receives as output all these keys, and S2 receives only k_i. $n=|stash|+3h+1$.
2. For each $j\in 1,\ldots,n$, S1 uses k_j to encrypt 0 and m_j to obtain cipher-texts z_j and c_j respectively. S1 shuffles all the (z_j,c_j) pairs and sends them to S2.
3. S2 uses k_i to decrypt the first cipher-text of each pair: only one z_j, will decrypt to 0. It then decrypts the corresponding c_j and hence obtain m_i.

Algorithm 4: OblivData.Fetch

```
1 Input of S1: $[i]^1, p, DataORAM$
2 Input of S2: $[i]^2$
3 Output: $m_i$
    1. S1:
        (a) $blocks \leftarrow Fetch(DataORAM, p)$. //fetch all blocks in stash and on path $p$
    2. S2PC:
        (a) $i \leftarrow [i]^1 \oplus [i]^2$
        (b) $blocks.Append(\perp)$; //add a dummy block at the end of the array
        (c) $n \leftarrow |blocks| + 1$; //so that $n = |stash| + 3h + 1$
        (d) for $j = 1$ to $n$ do $k_j \xleftarrow{\$} (0, 1)^l$; //generate $n$ keys
        (e) Outputs $k_1, \ldots, k_n$ to S1 and $k_i$ to S2.
    3. S1:
        (a) $M \leftarrow \{\}[n]$ //initialise an array to store the encrypted pairs
        (b) for $j = 1$ to $n$ do $(z_j, c_j) \leftarrow Enc_{k_j}(0, m_j))$; $M.add((z_j, c_j))$;
        (c) $M.Shuffle()$;
        (d) Sends $M$ to S2.
    4. S2:
        (a) $found \leftarrow FALSE$;
        (b) $p \leftarrow 1$ while $p \leq n$ and $!found$ do
            i. $(z_p, c_p) \leftarrow M[p-1]$; $z_p^{'} \leftarrow Dec_{k_i}(z_p)$;
            ii. if $z_p^{'} = 0$ then $found \leftarrow TRUE$; $m_p \leftarrow Dec_{k_i}(c_p) = m_i$; //$m_i$ is the $i$-th
                block on path $p$ in DataORAM
            iii. $p++$;
        (c) Outputs $m_i$.
```

A.2 Distributed Permutation

Recall that Circuit ORAM selects two paths during eviction, hence we need to track the movement of blocks in the stash and on the two paths, which has $|stash| = 6h - 3$ blocks.

Before each eviction, OblivShare appends a number tracker from 1 to $|stash| = 6h - 3$ to each block on the stash and two paths in ExpCtrlORAM inside S2PC. After the ExpCtrlORAM's stash eviction, the protocol extracts the trackers and construct an array of the numbers. Note that some numbers no longer exist as the attaching blocks are removed during the eviction. In order to generate a permeation of the same $|stash| = 6h - 3$ elements, OblivShare searches for the missing trackers using a linear scan and fill in the empty slots with unused numbers.

Below, we present how the two servers in secure computation put a block into the DataORAM's stash before eviction:

1. The S2PC places the following in an array: the blocks in the stash, the block read, and the block to write, which has $(|stash| + 2)$ blocks.
2. The S2PC finds that the k-th block of the stash is vacant, then generates a permutation σ^{read} or σ^{write}, which exchanges the k-th block with the read block for σ^{read} or the block to write for σ^{write}. As a result, the correct block is inserted into the stash (i.e. the first $|stash|$ blocks of the permuted array).

Algorithm 5: OblivData.Sync

```
1 Input of S2PC: $M$
2 Output: $M^{'}$
    1. for $i = 0$ to $|stash| + 6h - 4$ do $M[i].Append(i + 1)$; //attach a tracker to each
       block before ExpCtrlORAM's stash eviction
    2. $M^{'} \leftarrow ExpCtrlORAM.Evict()$; //extract trackers after stash eviction
    3. $trackers \leftarrow \{\}$; //initialise an array to store the missing trackers
       //do a linear scan to find numbers in $\{1, 2, \ldots, |stash| + 6h - 3\}$ that are missing
    4. for $t = 1$ to $|stash| + 6h - 3$ do
        (a) $found \leftarrow FALSE$;
        (b) $k \leftarrow 0$ while $k \leq 18$ and $!found$ do
              i. if $M^{'}[k] = t$ then $found = TRUE$;
             ii. else $k{+}{+}$;
        (c) if $!found$ then
              i. $trackers.add(i)$;
        (d) $t{+}{+}$
       //do a linear scan to fill missing trackers into the empty slots
    5. $r \leftarrow 0$
    6. for $j = 0$ to $|stash| + 6h - 4$ do
        (a) if $M^{'}[j] = \perp$ then $M^{'}[j] = trackers[r]$; $r{+}{+}$; //locate the empty slots and
            fill in missing trackers
        (b) $j{+}{+}$
    7. $\sigma \leftarrow Permutation.Gen(M, M^{'})$ //generate a permutation $\sigma$ so that $M^{'} = M \circ \sigma$
    8. $\sigma^1 \leftarrow \$$ //sample a random permutation
    9. $\sigma^2 \leftarrow \sigma \circ (\sigma^1)^{-1}$ //composition of $\sigma$ and inversion of $\sigma^1$
   10. Outputs $\sigma^1$ to S1 and $\sigma^2$ to S2;
   11. S1:
        (a) Re-randomise the cipher-texts of $blocks$;
        (b) $M^1 = M \circ \sigma^1$; //apply $\sigma^1$ to $M$
        (c) Sends $M^1$ to S2.
   12. S2:
        (a) Re-randomize the cipher-texts of the blocks in $M^1$;
        (b) $M^2 = M^1 \circ \sigma^2 = M^{'}$; //apply $\sigma^2$ to $M^1$
        (c) Sends $M^{'}$ to S1.
```

3. The S2PC secret shares the permutation (σ^{read} or σ^{write}) into two permutations σ^1 and σ^2, when $\sigma^2 = \sigma \circ (\sigma^1)^{-1}$. $\circ$ denotes composition of permutation and $\sigma \circ (\sigma)^{-1}$ is the identity permutation.
4. S1 re-randomise the blocks, apply the permutations σ^1, and sends the permuted blocks to S2.
5. S2 re-randomise the blocks received, apply the permutations σ^2, and sends the permuted blocks back to S1.
6. S1 stores the permuted blocks in the corresponding location in DataORAM.

References

1. Backes, M., Herzberg, A., Kate, A., Pryvalov, I.: Anonymous RAM. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9878, pp. 344–362. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45744-4_17
2. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multiparty computation for data mining applications. Int. J. Inf. Secur. **11**(6), 403–418 (2012). https://doi.org/10.1007/s10207-012-0177-2
3. Chen, W., Popa, R.A.: Metal: a metadata-hiding file-sharing system. IACR Cryptology ePrint Archive 2020/83 (2020)
4. Dingledine, R., Mathewson, N., Syverson, P.: Tor: the second-generation onion router. Technical report, Naval Research Lab Washington DC (2004)
5. Doerner, J., Shelat, A.: Scaling ORAM for secure computation. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 523–535 (2017)
6. Dropbox: Dropbox Business Security: A Dropbox whitepaper. Technical report, Dropbox (2019)
7. DropSecure: Enabling True File Transfer Security: How DropSecure safeguards your confidential data. Technical report, DropSecure (2019)
8. Fitzpatrick, K.: Password protect files with Digify passkey encryption (2019). https://help.digify.com/en/articles/747136-password-protect-files-with-digify-passkey-encryption
9. Fu, Z., Ren, K., Shu, J., Sun, X., Huang, F.: Enabling personalized search over encrypted outsourced data with efficiency improvement. IEEE Trans. Parallel Distrib. Syst. **27**(9), 2546–2559 (2015)
10. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, pp. 182–194 (1987)
11. Hamlin, A., Ostrovsky, R., Weiss, M., Wichs, D.: Private anonymous data access. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11477, pp. 244–273. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17656-3_9
12. Liang, K., Liu, J.K., Lu, R., Wong, D.S.: Privacy concerns for photo sharing in online social networks. IEEE Internet Comput. **19**(2), 58–63 (2015)
13. Liu, W., Wang, Y.B., Jiang, Z.T., Cao, Y.Z.: A protocol for the quantum private comparison of equality with χ-type state. Int. J. Theor. Phys. **51**(1), 69–77 (2012). https://doi.org/10.1007/s10773-011-0878-8
14. Maffei, M., Malavolta, G., Reinert, M., Schröder, D.: Privacy and access control for outsourced personal records. In: 2015 IEEE Symposium on Security and Privacy, pp. 341–358. IEEE (2015)
15. Maffei, M., Malavolta, G., Reinert, M., Schröder, D.: Maliciously secure multi-client ORAM. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) ACNS 2017. LNCS, vol. 10355, pp. 645–664. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61204-1_32
16. McMillan, R., Knutson, R.: Yahoo triples estimate of breached accounts to 3 billion (2017). https://www.wsj.com/articles/yahoo-triples-estimate-of-breached-accounts-to-3-billion-1507062804
17. Nguyen, N.: Introducing Firefox send, providing free file transfers while keeping your personal information private (2019). https://blog.mozilla.org/blog/2019/03/12/introducing-firefox-send-providing-free-file-transfers-while-keeping-your-personal-information-private/

18. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 502–519. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_27
19. Regulation, G.D.P.: Regulation (EU) 2016/679 of the European parliament and of the council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46. Off. J. Eur. Union (OJ) **59**(1–88), 294 (2016)
20. Roche, D.S., Aviv, A., Choi, S.G.: A practical oblivious map data structure with secure deletion and history independence. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 178–197. IEEE (2016)
21. Rösler, P., Mainka, C., Schwenk, J.: More is less: on the end-to-end security of group chats in Signal, WhatsApp, and Threema. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 415–429. IEEE (2018)
22. Rusbridger, A.: The Snowden leaks and the public (2013)
23. Schneier, B.: Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World. WW Norton & Company, New York (2015)
24. SendSafely: Powerful security that's simple to use (2019). https://www.sendsafely.com/howitworks/
25. Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_11
26. Shundong, L., Chunying, W., Daoshun, W., Yiqi, D.: Secure multiparty computation of solid geometric problems and their applications. Inf. Sci. **282**, 401–413 (2014)
27. Silverstein, J.: Hundreds of millions of Facebook user records were exposed on amazon cloud server (2019). https://www.cbsnews.com/news/millions-facebook-user-records-exposed-amazon-cloud-server/
28. Stefanov, E., et al.: Path ORAM: an extremely simple oblivious RAM protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 299–310 (2013)
29. Telegram: What is a secret chat in Telegram (2019). https://telegramguide.com/secret-chat-telegram/
30. Toft, T.: Secure data structures based on multi-party computation. In: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 291–292 (2011)
31. de la Torre, L.: A guide to the California consumer privacy act of 2018. SSRN 3275571 (2018)
32. Wang, X., Chan, H., Shi, E.: Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 850–861 (2015)
33. Wang, X.S., Huang, Y., Chan, T.H., Shelat, A., Shi, E.: SCORAM: oblivious RAM for secure computation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 191–202 (2014)
34. WhatsApp: WhatsApp encryption overview: technical white paper. Technical report, WhatsApp (2017)
35. Yao, A.C.C.: How to generate and exchange secrets. In: 27th Annual Symposium on Foundations of Computer Science (SFCS 1986), pp. 162–167. IEEE (1986)

36. Zahur, S., et al.: Revisiting square-root ORAM: efficient random access in multi-party computation. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 218–234. IEEE (2016)
37. Zuo, C., Shao, J., Liu, J.K., Wei, G., Ling, Y.: Fine-grained two-factor protection mechanism for data sharing in cloud storage. IEEE Trans. Inf. Forensics Secur. **13**(1), 186–196 (2018)