Brief Announcement: Towards Security and Privacy for Outsourced Data in the Multi-Party Setting

Matteo Maffei    Giulio Malavolta    Manuel Reinert    Dominique Schröder
Saarland University & CISPA
Saarbrücken, Germany
{maffei, malavolta, reinert, schroeder}@cs.uni-saarland.de

ABSTRACT

Cloud storage has rapidly acquired popularity among users, constituting a seamless solution for the backup, synchronization, and sharing of large amounts of data. This technology, however, puts user data in the direct control of cloud service providers, which raises increasing security and privacy concerns related to the integrity of outsourced data, the accidental or intentional leakage of sensitive information, the profiling of user activities and so on.

We present GORAM, a cryptographic system that protects the secrecy and integrity of the data outsourced to an untrusted server and guarantees the anonymity and unlinkability of consecutive accesses to such data. GORAM allows the database owner to share outsourced data with other clients, selectively granting them read and write permissions.

GORAM is the first system to achieve such a wide range of security and privacy properties for outsourced storage. Technically, GORAM builds on a combination of ORAM to conceal data accesses, attribute-based encryption to rule the access to outsourced data, and zero-knowledge proofs to prove read and write permissions in a privacy-preserving manner. We implemented GORAM and conducted an experimental evaluation to demonstrate its feasibility.

Categories and Subject Descriptors

K.6.5 [Management of Computing and Information Systems]: Security and Protection

Keywords

Oblivious RAM; ORAM; GORAM; Cloud Storage; Privacy-enhancing Technologies

1. INTRODUCTION

Cloud storage has rapidly acquired a central role in the digital society. It is convenient for users, who can smoothly backup their data, access them from computationally limited devices, synchronize them across multiple locations, and share them with other parties. Typical examples are data storage services such as Dropbox, Microsoft SkyDrive, and Google Drive, or music and app synchronization services such as iCloud or Amazon. Despite its convenience, cloud storage poses a number of security and privacy issues.

The first problem is related to the integrity of user data. The server has the possibility to modify client’s data, which can be harmful for several reasons: for instance, a storage provider could change the client for space it is not using by adding fake data, or save on storage space by deleting old backup copies of large files, which are likely not to be accessed by the user.

A second problem is related to the secrecy of user data, which are often sensitive and, thus, should be concealed from the server. For instance, a client accessing a medical database in search for symptoms of and remedies against a certain disease would probably like to keep both the queried documents private. As shown in the literature [10, 12], even the capability to link consecutive accesses to the same file can be exploited by the server to learn sensitive information: hence the obliviousness of data accesses is another fundamental property for cloud storage services.

Furthermore, modern cloud storage services (e.g., Google Docs) offer the database owner the possibility to share some of the documents with other clients, yet in a controlled manner, i.e., selectively granting them read and write permissions. Data sharing complicates the enforcement of integrity properties, since clients might be malicious and try themselves to break the integrity of user data. Additionally, access control is in seeming contradiction with the obliviousness property: how can the server check if the client is authorized to read or write a certain file if the server is not supposed to learn which file the client is accessing?

Related Work. Oblivious RAM (ORAM) [5] has recently proved an effective technique to hide the data as well as the access patterns of a client from an untrusted server in storage outsourcing services [3, 12, 1, 6]. Some of these protocols further guarantee the integrity of user data against a malicious server [16, 13]. Standard constructions, however, are not suitable for a multi-client setting, since clients would need to hold the ORAM key, a top-level key used to encrypt data on the server’s side, which would allow them to read and also potentially disrupt the entire database.

A few recent works have started to tackle this problem. Franz et al. introduced the concept of delegated ORAM [4]. The idea is to encrypt and sign each entry with a unique set of keys, initially only known to the database owner: giving a client the decryption key (resp. the decryption and signing
keys) suffices to grant read (resp. write) access to that entry. This solution, however, has the drawback of forcing the database owner to track the individual client accesses and to come back periodically to re-shuffle the ORAM according to the access history in order to enable further unlinkable ORAM accesses. Furthermore, revoking access for a single client requires to change (and distribute) the capabilities of all other users that have access to that file. Goodrich et al. [7] developed an ORAM construction for the multi-client setting, but their scheme considers only trusted clients and does not allow clients to verify the integrity of outsourced data. Huang and Goldberg presented a protocol for outsourced private information retrieval (PIR) [9], which is obtained by layering a PIR scheme on top of an ORAM data layout. This solution is efficient and conceals client accesses from the database owner, but it does not give clients the possibility to update data, as required in collaborative applications. Furthermore, it assumes \( \ell \) non-colluding servers, due to the information theoretic multi-server PIR. Despite the significant progress in this field, none of the existing approaches constitutes a practical solution to the problem of sharing data stored on the cloud with potentially distrustful clients.

Our Contributions. We tackle this long-standing problem by presenting GORAM (Group ORAM), the first ORAM construction that allows the database owner to verify the integrity of outsourced data and, at the same time, to share these data with other clients, selectively granting them access as well as write permissions, without the need for the database owner to manage the consistency of the database. Data accesses are oblivious and the server does not learn any information about them. The fundamental idea is to combine ORAM [14] to conceal data accesses, attribute-based encryption [11] to rule the access to outsourced data, and zero-knowledge proofs [5, 2] to prove read and write permissions in a privacy-preserving manner; finally, we demonstrate the feasibility of our approach with an experimental evaluation.

2. OVERVIEW OF GORAM

Cryptographic Construction. We consider the following storage outsourcing scenario. A database owner \( O \) stores her data \( DB \) on an untrusted server \( S \). Moreover, she wants to share data with other clients \( C_1, \ldots, C_k \). In order to operate on the server, the clients should not synchronize with \( O \), since this would require \( O \) to be always online, which is not a realistic assumption. In order to manage the database, \( O \) has several functions at her disposal. She can read and write entries from and to the database, she can grant other clients access, and she can change the access permissions for single entries in the database. GORAM provides three different access modes: read access, read and write access, and no access. Clients which \( O \) shares data with can read and write database entries according to their access permissions.

The protocols supported by GORAM are depicted in Figure 1 and explained below. The database of size \( O(N) \) is a binary tree of depth \( O(\log N) \). An entry \( E_i \) has four components, which are all secured by a top-level public-key encryption layer: (1) a unique resource identifier \( u \), (2) an attribute-based encryption ciphertext \( c_{Auth} \) that regulates write access, (3) an attribute-based encryption ciphertext \( c_{Key} \) that contains a symmetric key and regulates the read access, and (4) the actual data \( c_{Data} \) that is secured by a symmetric-key encryption scheme. Attribute-based encryption (see, e.g., [11]) is a cryptographic primitive that encodes access policies in ciphertexts and secret keys. Intuitively, a user can only decrypt a ciphertext \( c \) if her secret key fulfills the policy encoded in \( c \). For the structure of \( E_i \), this means that, if the client has read access, she is able to decrypt \( c_{Key} \) and can extract the symmetric key that allows her to decrypt \( c_{Data} \).

In order to grant client \( C_j \) access, \( O \) hands over a capability \( c_{P_{j, cap}} \) that contains the top-level private key and two attribute-based secret keys, for read and write access control. Furthermore, \( O \) can set for each entry \( E_i \) in the database the access permissions separately. If she wants to give \( C_j \) read (resp. write) access to index \( i \), she sets \( c_{P_{j, cap}} \) (resp. \( c_{Auth} \)) such that \( C_j \) can decrypt it.

In order to read or write data from the database, a client downloads a path \( E_1, \ldots, E_k \) in the tree from the root to a leaf. After the client has read the entry that she was interested in (by inspecting the resource identifiers), she shuffles and re-randomizes the entries such that the entry that she read is in the top position. She then sends the shuffled and re-randomized path \( E'_1, \ldots, E'_k \) together with a proof \( P \) to the server. \( P \) is a zero-knowledge proof showing that the new path’s content has not changed except for the applied permutation and the randomness used in the encryption. \( S \) verifies \( P \) with respect to the new and the old path and replaces the entries accordingly. If, additionally, the client wants to write an entry that she has access to (which is now in the top position, i.e., \( E'_1 \)), she encrypts the new data with the key that she extracted from \( c_{Key} \). She then creates \( E''_1 \) by re-randomizing \( c_{Auth} \) and the top-level encryption of \( E'_1 \). Finally, she sends \( E''_1 \) and \( P' \) to \( S \). \( P' \) is a zero-knowledge proof showing that the client is able to decrypt \( c_{Auth} \) in \( E''_1 \) (this proves that she possesses a valid decryption key). \( S \) verifies \( P' \) and replaces \( E'_1 \) by \( E''_1 \).

Security and Privacy Properties. GORAM achieves the following properties:

Secrecy. The attribute-based encryption scheme ensures that only clients holding read permissions on a certain file can learn any information about its content.

Integrity. The zero-knowledge (ZK) proofs ensure that every database entry can be modified only by clients holding write permissions on it, as long as \( S \) is honest but curious. If \( S \) is malicious and colludes with a subset of users, our model still guarantees honest clients to notice whenever a data is
tampered, as long as corrupted clients have neither read nor write permissions on that data.

**Anonymity.** The ZK proofs further ensure that nobody can link a read or write operation to a particular client, within the set of users having access to the target entry.

**Obliviousness.** The ORAM scheme along with the ZK proofs ensures that the pattern of two subsequent accesses is uniformly distributed over the database entries, independent of the processed data and the performed operation.

3. **EXPERIMENTAL EVALUATION**

The computational complexity of our construction both on the client and on the server as well as the space needed on the client’s side is $O((G+B) \log N)$ where $B$ is the block size of entries in the database, $N$ is the size of the database, and $G$ is the number of client groups that have access to the database. Hence, our construction only adds $G$ to the original ORAM complexity. Our experimental results for the read and write protocols are depicted in Figure 2. Figure 2a shows the logarithmic growth in the client and server time while we increase the storage size $N$. Figure 2b shows the linear increase in client and server time while we grow the block size $B$. Interestingly enough, multi-threading keeps the time almost constant from 8KB to 64KB. Overall, the computation overhead is dominated by the re-randomization of the top-level public-key encryption and the computation of the zero-knowledge proofs. In order to optimize our implementation, we envision the usage of GPUs, which can speed up the computation time by a factor of 20 [17, 15].

4. **ACKNOWLEDGEMENTS**

This work was supported by the German research foundation (DFG) through the Emmy Noether program and by the German Federal Ministry of Education and Research (BMBF) through the Center for IT-Security, Privacy and Accountability (CISPA). Finally, we thank the reviewers for their helpful comments.

5. **REFERENCES**


