CAOS

Ordean, Mihai; Ryan, Mark; Galindo Chacon, David

DOI:
10.1145/3322431.3325101

Document Version
Peer reviewed version

CAOS: Concurrent-Access Obfuscated Store

Mihai Ordean
University of Birmingham, UK
m.ordean@cs.bham.ac.uk

Mark Ryan
University of Birmingham, UK
m.d.ryan@cs.bham.ac.uk

David Galindo
University of Birmingham and
Fetch.AI, Cambridge, UK
d.galindo@cs.bham.ac.uk

ABSTRACT

This paper proposes Concurrent-Access Obfuscated Store (CAOS), a construction for remote data storage that provides access-pattern obfuscation in a honest-but-curious adversarial model, while allowing for low bandwidth overhead and client storage. Compared to other approaches, the main advantage of CAOS is that it supports concurrent access without a proxy, for multiple read-only clients and a single read-write client. Concurrent access is achieved by letting clients maintain independent maps that describe how the data is stored. Even though the maps might diverge from client to client, the protocol guarantees that clients will always have access to the data. Efficiency and concurrency are achieved at the expense of perfect obfuscation: in CAOS the extent to which access patterns are hidden is determined by the resources allocated to its built-in obfuscation mechanism. To assess this trade-off we provide both a security and a performance analysis of CAOS. We additionally provide a proof-of-concept implementation[1].

CCS CONCEPTS

• **Security and privacy → Management and querying of encrypted data**; *File system security*;

KEYWORDS

data obfuscation; concurrent-access obfuscated store; access pattern

1 INTRODUCTION

Cloud computing has become an attractive solution for data storage. Unfortunately, current cloud computing architectures do not provide sufficient and reliable security for private and sensitive data. Even when encryption is used, malicious servers and operators can learn user access patterns and derive information based on them (e.g., data accessed more often can be assumed to be more important) [7].

One cryptographic primitive specifically designed to hide access patterns is Oblivious RAM (ORAM). This primitive was introduced by Goldreich and Ostrovsky [12, 13] for the purposes of preventing software reverse engineering by hiding a program's access patterns to memory. The issue has since become important in the context of cloud computing, where clients and data-store servers often reside in different trust domains and trust between them cannot always be established. Modern ORAM schemes [10, 16, 22] are seen as viable options of addressing this problem. However, in real-life scenarios, even the best ORAMs can prove to be impractical [15], mainly because of the high bandwidth requirements and/or client storage constraints. Another major limitation of modern ORAM constructions is that they are mainly restricted to having a single-client that connects to the data-store server. This is because data in the store is accessed through a client maintained local structure (i.e. a map). Migrating from this model has proven difficult. Even small deviations [20, 23], such as allowing multiple clients to access the store through a proxy that acts as the single-client have been shown to have vulnerabilities [17].

As ORAMs have been difficult to use in real-life, other specialised, and more efficient security primitives have been developed in the context of privacy preserving access to cloud-stored data e.g. searchable encryption (SE) schemes. SE uses either symmetric keys [8, 9, 18] or public keys [19] and allows clients to securely search cloud stored databases through precomputed ciphertexts called *trapdoors*. SE schemes have low computational requirements from clients and are bandwidth efficient. However, prior work has shown that searchable encryption schemes leak significant amounts of information about their encrypted indexes when using attacks which combine access-pattern analysis, background information about data stored, and language-based word frequency knowledge [14]. Incorporating changes and updates to the searched database is also a difficult process. Often schemes require the whole index to be regenerated for any the new information added [8, 9]. Finally, SE schemes are restricted to search operations, actual data retrieval needs to happen through a private information retrieval protocol [6] or an ORAM.

As such, an ideal system would have the general applicability and access-pattern privacy of ORAM (cloud storage which hides access patterns), and bandwidth efficiency and concurrent access capabilities similar to those of SE schemes. In this paper we take steps towards this direction by proposing a new design for a general-purposed secure storage with concurrency and bandwidth efficiency. However, the privacy guarantees we provide are not absolute. Instead, our protocol requires that users provision resources for access-pattern obfuscation, and the security guarantees depend on how much of these resources are available.

1.1 Contributions

This paper proposes Concurrent-Access Obfuscated Store (CAOS), a storage access protocol that can hide data access frequency and access patterns, while allowing for concurrent data access. Our main focus when designing CAOS is to obtain a bandwidth-efficient protocol that supports concurrency by design and that is able to provide a customizable amount of data and access-pattern privacy. Our main contributions are as follows:

(1) **Obfuscated access patterns.** We propose a secure access protocol for remote data storage which is able to hide access patterns. Our construction requires at least one of each of the following two types of clients: a *regular client* which stores data, and an *obfuscation client* which hides client's access patterns. Maximum privacy is achieved as long as at least one obfuscation client behaves honestly.

[1] Available: https://github.com/meehien/caos

(2) **Concurrent access.** We provide, to our knowledge, the first concurrent-access protocol with access-pattern hiding properties that does not require a trusted third party (e.g. proxy). Our concurrent access protocol is applicable in scenarios with multiple readers, but can cope with having a single writer.
(3) **Small and constant bandwidth.** For all clients with read-write/read-only access, our protocol requires a constant bandwidth that is independent of the size of the store. This is possible because we separate the regular access clients and the security responsible clients (i.e. the obfuscation clients). The bandwidth requirements for interacting with the store are also small. In our current instantiation a single block of data requires a constant two blocks to be transferred.
(4) **Security and performance analysis.** We give a game-based definition of data and access-pattern privacy for CAOS-like protocols against honest-but-curious storage servers. Furthermore, we apply this new definition to CAOS and prove it secure. Last but not least we report on the theoretical and observed performance of our protocol thanks to our proof-of-concept implementation.

2 CONCURRENT-ACCESS OBFUSCATED STORE (CAOS)

CAOS is a protocol for storing data securely by encrypting it and anonymizing (read or write) access patterns. CAOS allows users to trade storage space and security for concurrency and bandwidth efficiency.

Data elements. In CAOS data is partitioned into blocks of equal size. Each block of data is uniquely identified by a client using a *block id (bid)*. Storing a block remotely involves encrypting the contents of the block and then placing the resulting ciphertext at a random location in the store's memory. We refer to these locations at the store's memory as *positions*. The size of the store is measured in the number of positions it has available for storing blocks. Clients can store the same block at multiple positions and keep track of where their data is located by maintaining a *map* which links block ids to positions.

Clients. There are two types of clients in CAOS: regular clients (RC), which may be read-write (RW) or read-only (RO); and obfuscation clients (OC). Both RC and OC access the store directly and independently from each other.

Regular clients are the main users of the store. They have low bandwidth and local storage requirements, as they only have to store a map. However, accesses done by these clients do leak information about access patterns. Obfuscation clients are the clients that provide security. These clients are able to provide access-pattern obfuscation for themselves as well as RCs. For that purpose OCs use of a buffer which is stored locally in addition to a map. The size of the OC's local buffer and the OC's bandwidth requirements are proportional to the speed of obfuscation.

Access-pattern obfuscation. Our definitions derive from existing access-pattern security definitions in ORAM [21]. Intuitively, the ORAM definitions require that no information should leak with regards to: (1) which data is being accessed, (2) the frequency of accesses, (3) the relation between accesses, (4) whether access is read or write, and (5) the age of the data. ORAM constructions maintain invariants to ensure that no information is leaked regardless of how many times the store is accessed. CAOS maintains the requirement that no information is leaked for cases (1)-(5), but does not provide guarantees for each individual access operation. Instead, CAOS provides security guarantees for *access sequences* that involve both regular clients and obfuscation clients. Our security definitions for content and access-pattern security in CAOS are detailed in Section 5.

Concurrency. CAOS allows multiple clients to access the store simultaneously and independently from each other. Achieving concurrency in CAOS is not a trivial task. This is because each client's access operation randomly changes the contents of store, and these changes are only stored locally to that client. Thus, CAOS needs to address two problems: (1) to synchronise locally stored client maps in an efficient manner, and (2) how to allow multiple clients to change the store simultaneously and in a way that does not result in data loss for other clients.

Syntax. In the following we draw on the above and give the syntax of our CAOS protocol. Alternative variants of CAOS are possible if adhering to this syntax. We say that an (n, N) store S is a collection of n data blocks written to N store positions such that $n < N$.

Definition 2.1. CAOS consists of a tuple of five PT algorithms $\mathcal{O} = (\mathsf{KGen}, \mathsf{INIT.STORE}, \mathsf{INIT.OC}, \mathsf{ACCESSRW}, \mathsf{ACCESSOC})$ over an (n, N) store S:

$k \leftarrow \mathsf{KGen}(1^\lambda)$: is a setup probabilistic algorithm run by the RW client. It takes as input the security parameter λ and outputs a secret key k.

$S \leftarrow \mathsf{INIT.STORE}(DB, N, k)$: is a deterministic algorithm run by the RW client to initialize the data store. It takes as input a database $DB = (B_0, \ldots, B_{n-1})$ of n data blocks, encrypts each block under key k, and distributes them between the total number N of store positions.

$buf, S \leftarrow \mathsf{INIT.OC}(S, k)$: is a deterministic algorithm run by the OC to initialize itself. It requires access to an initialized store S and its encryption key k and creates the internal buffer of the obfuscation client.

$ret, S \leftarrow \mathsf{ACCESSRW}(B, op, d, S, k)$: is a probabilistic algorithm that RCs run to access a store. It takes as input the bid B to be accessed, the operation $op \in \{read, write\}$, the data d to be written if $op = write$, and the store S and its key k. When the client runs this algorithm, some positions on the server are read, and others are written. It returns the block read or an acknowledgement for the write operation, and the new state of the store S.

$buf, S \leftarrow \mathsf{ACCESSOC}(buf, S, k)$: is a probabilistic algorithm run by the OC to access a store. It takes as input a local data structure buf that acts as a buffer, a store S and a key k. The algorithm alters the OC's buffer of the obfuscation client. Additionally, when the obfuscation client runs this algorithm, some positions in the store are read, while others are written. This changes the mapping between blocks and positions.

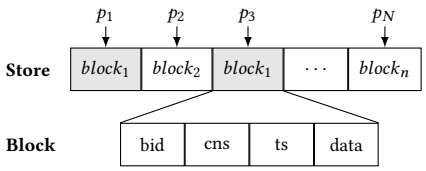


Figure 1a: Server data structures in CAOS. The server redundantly stores n equally-sized encrypted blocks at N memory locations, $n < N$. The memory locations are addressable through unique positions ids $p_1, \ldots, p_N$. Each $block$ stored contains the data intended for storage (i.e. $block.data$) and a small amount of metadata (i.e. $block.bid$, $block.ts$ and $block.cns$) that helps with map synchronization between concurrent clients.

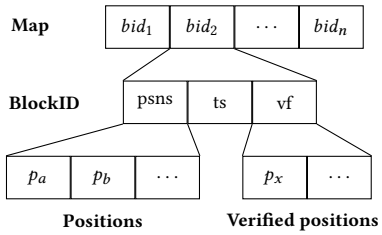


Figure 1b: Client local data structures in CAOS. The client stores a linear map indexed by block ids (i.e. bid). For block id bid the client stores a list of server positions (i.e. $bid.psns$) from where the block can be retrieved. The map also keeps some metadata about each block id (i.e. $bid.ts$ and $bid.vf$) that helps with synchronisation between concurrent clients.

3 EFFICIENT ACCESS-PATTERN OBFUSCATION IN CAOS

This section describes CAOS. We begin with an overview of the protocol and we will follow up with details about the corresponding algorithms and discussing a proof of concept implementation. The complete source-code is available at [1].

3.1 Overview

Access pattern obfuscation. In CAOS we achieve access-pattern obfuscation for sequences of access operations (see Section 5). This is a weaker security guarantee than that used in other works [12, 13, 21], where obfuscation is achieved for each single access operation. In return, our construction allows for concurrency and is more practical.

In CAOS hiding the type of access (read or write) and the age of the data is done by joining both the read and the write operations into a single access function, ACCESSRW. This prevents the adversary from learning when data is read or written, and when new data is added to the store, with the exception of the initial provisioning of the store done by running INIT.STORE.

CAOS uses a locally stored *map* per client to keep track of which store positions contain which blocks. By setting the size of the store to be larger than the size of the data to be secured, the algorithm ACCESSRW can create redundancies through re-encrypting and duplicating blocks from the store and assign them to random free-positions. We use the term *free-positions* to refer both to store positions which have never been written, and to positions whose corresponding blocks have at least one redundancy (i.e. blocks that are stored in two or more places). By allowing regular clients (RCs) to access the same data from multiple store positions we are able to partially obfuscate details about the frequency with which specific data is being accessed, and about the relationship between subsequent accesses. We say partially because, even though data is duplicated to random positions, the adversary can still connect these positions to the initial position from where the duplication process began. To address this issue we use obfuscation clients (OCs). These are read-only clients that use the ACCESSOC function to access the store similarly to RCs. The difference is that OCs maintain a local buffer which is used to store the contents of the positions received from the store. When an OC performs a store access, it writes (i.e. duplicates) in the store a block read from its buffer. As such, blocks that are duplicated by OCs are not linked to current store blocks and do not leak any access-pattern information.

Data structures. In CAOS each store block contains the following: data to be stored *block.data*, a block identifier *block.bid*, a consolidation field *block.cns* that indicates the number of clients that know that a block is stored at a position, and a timestamp *block.ts* of when the data was last changed (cf. Fig 1a).

Client local maps are indexed by the block identifier *bid*, and contain the following: *bid.psns* enumerates the positions in the store from which the block *bid* is available, *bid.ts* stores most up-to-date timestamp observed, and *bid.vf* stores positions p observed by the map holder (i.e. client) to have *block.cns*=$|Clients|$, where $Clients$ is the set of all clients engaged in our protocol (cf. Fig 1b).

Concurrency. In CAOS access-pattern obfuscation is achieved through shuffling, thus achieving efficient concurrency with direct access for all clients represents a significant challenge. This is especially difficult because in CAOS each client maintains its own map and syncs it with the store *independently* from other clients during ACCESSRW or ACCESSOC operations. In order to prevent data loss, i.e. that a client looses track of the current data in the store, we ensure that "for each data block, there exists a valid position that is known to all clients". Maintaining this invariant has lead to two design constraints: (1) we require that for each single block accessed two positions are read and two positions are written on the server store, and (2) CAOS can only handle a single read-write client that works concurrently with other read-only and/or obfuscation clients. These restrictions are further discussed in Section 5.2.

Shared knowledge between clients is tracked using *block.cns* and *bid.vf*. All clients start from the same version of the map, which is afterwards maintained independently by each one. The protocol requires clients to signal each other when they perform changes to their local maps (i.e. when reassigning a position or when changing the data in a block). Because the client only has access to one position per block during an access operation, the change produced by the client will be localised to that particular position in the store. The problem is that without any additional signalling other clients

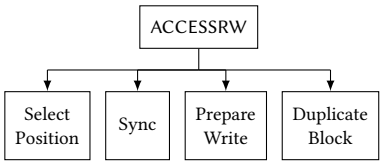


Figure 2: Client read-write access function. The function ACCESSRW **performs four actions: (1) selects the position used to retrieve a block from the store, (2) synchronises local map with metadata from retrieved blocks, (3) prepares a block to be written back by ensuring that the write operation is possible, and (4) attempts to duplicate one of the retrieved blocks onto the position of the other.**

who are not aware of the change have no way of assessing whether the block stored at a specific position is the correct one (as indicated by their map).

We indicate shared knowledge about a position as follows. Whenever a client makes a change to a block, the value *block.cns* is set to 1, meaning that only one client, the one that made the change is currently aware of the change. When other clients access this block they can become aware that a reassignment has taken place by comparing the *block.bid* value stored in the block with the value they were expecting according to their local map. If the values do not match the client will infer that the position used to retrieve the block has been reassigned, and will update their local map accordingly. Similarly, by comparing timestamp data from the local map *map[bid].ts* and from the retrieved block *block.ts* clients can determine if the block's data was updated.

Once a client becomes aware that a block has been reassigned to a new position p (and has performed changes in its local map) it increments *block.cns* by 1 to signal clients that it is aware of the change. When the *block.cns* value is equal to the number of clients then all the clients can safely assume they have the same view about what block is stored at position p.

Next we continue by specifying CAOS algorithms.

3.2 Read-write (and read-only) client access

Read-write (RW) clients, such as email SMTP servers, and read-only (RO) clients, such as email readers, perform reading and writing through the function ACCESSRW (cf. Fig 2). We introduce the main function first with calls to several sub-functions that implement required functionality. Each sub-function is subsequently described.

The main ACCESSRW function detailed in Algorithm 1 involves the following 8 main steps:

(1) *Select positions* (line 2,3). First, select a position (line 2) to read/write a block as indicated by the input value *bid*. Then select a random position to sync to the local map (line 3). The block requested via *bid* will also be copied to the random position if necessary conditions are met (see Algorithm 5). The *SelectPosition*() function interacts with the locally stored map data structure and selects a random position for a given block id *bid*. If no *bid* is supplied a completely random position is chosen from the set of known store positions.
(2) *Read positions from store* (line 4). Retrieve the block indicated by *bid* and the random block using previously selected positions and decrypt them.

Algorithm 1: Main CAOS access function.

Input: block id, operation, data (for write operation), store, client map_c, store key
Output: data (for read operation), store, client map_c

```
1  function ACCESSRW (bid, op, data, S, map_c, k)
2      req_p ← SelectPosition(bid, map_c);
3      cpy_p ← SelectPosition(null, map_c);
4      (req_blk, cpy_blk) ← READ(req_p, cpy_p, S, k);
5      (req_blk, map_c) ← Sync(req_blk, req_p, map_c);
6      (cpy_blk, map_c) ← Sync(cpy_blk, cpy_p, map_c);
7      if (op = READ) then
8          if (req_blk.bid = bid) then
9              out ← req_blk.data;
10     if (op = WRITE) then
11         (req_blk, out) ←
             PrepareWrite(bid, req_blk, data);
12     (cpy_blk, map_c) ←
         DuplicateBlock(req_blk, cpy_blk, cpy_p, map_c);
13     S ←WRITE(S, k, req_blk, cpy_blk, req_p, cpy_p);
14     return (out, S, map_c);
```

(3) *Sync* (lines 5,6). Synchronize the metadata from the blocks retrieved with the local client map.
(4) *Return read data* (lines 7-9). Prepare to return data if the operation is *READ* and the block retrieved is correct. If the block retrieved has a different *bid* than the one requested the client has to re-run ACCESSRW. The protocol guarantees that every client has a valid position for every block.
(5) *Replace block data* (lines 10,11). If the operation is *WRITE* replace the current data in *req_blk* with the input data *data*.
(6) *Duplicate block* (line 12). Attempt to shuffle store data by coping the *req_blk* to position *cpy_p*, thus increasing the number of positions for *req_blk* and reducing the available positions for block *cpy_blk* (see Algorithm 5).
(7) *Write blocks to store* (line 13). Encrypt *req_blk* and *cpy_blk* and write them at positions *req_p* and *cpy_p*.
(8) *Return* (line 14). Return *out* as requested data if *op* = *READ* and operation was successful; return *null* otherwise. Return input *data* as *out* if *op* = *WRITE* and operation was successful (*null* otherwise).

3.2.1 Sync function. This function (see Algorithm 2) enables CAOS clients to work concurrently, without having to synchronize their maps to access the store. We do require that once, during the setup phase, clients share the store key and a map from one of the other participating clients. Once a copy of the map has been obtained each client can maintain its own version of the map i.e. map_c.

Intuitively this works as follows: after a block has been retrieved from the store, the client needs to establish if local knowledge about the block is correct, i.e. *bid* of the retrieved block and the position used to retrieve it are correctly linked, and the time-stamp of the block is not older than the time-stamp stored locally. As such, the

Algorithm 2: Sync metadata between a store block and the local map.

Input: block, position, client map
Output: block, client map

```
1  function Sync (block, p, map_c)
2      if (block.ts < map_c[block.bid].ts) then
3          remove p from map_c[block.bid].psns and
             map_c[block.bid].vf;
4          block.bid ← free;
5          block.ts ← current_time;
6          block.cns ← 1;
7      else
8          if (block.cns < |clients|) then
9              if (block.ts > map_c[block.bid].ts) then
10                 clear map_c[block.bid].psns and
                     map_c[block.bid].vf;
11                 set map_c[block.bid].ts to current_time;
12             if (p ∉ map_c[block.bid].psns) then
13                 move p to map_c[block.bid].psns;
14                 block.cns ← block.cns + 1;
15         if (block.cns = |clients|) then
16             add p to map_c[block.bid].vf;
17     return (block, map_c);
```

block can be in multiple states of which only the following require a map modification: (1) old data – the block is marked as free; (2) wrong position, data up-to-date – update bid/position association; (3) new data – update bid/position association and local time-stamp.

Old data in a block is detected by comparing the *block.ts* value with the locally stored $map_c[block.bid].ts$ value, where *block.bid* is the bid of the block being synchronised. Old data is thus easy to detect as each *bid* entry from the map has a single time-stamp (i.e. $map_c[block.bid].ts$): the newest time-stamp observed over all store accesses. In this case the position used to retrieve the block can be used for writing or duplication. If new data is detected, the local time-stamp is updated and positions pointing to the old version of the block are removed from $map_c[block.bid].psns$ and $map_c[block.bid].vf$. They can now be reused for writing or duplication.

If a wrong position is detected then map_c is updated with correct bid-position association (the position thought to point to the requested *bid* is moved to the correct $map_c[block.bid].psns$). Any modification to a client's local map is communicated to other by incrementing *block.cns*.

3.2.2 PrepareWrite function. Writing or retrieving a block to/from the store is not guaranteed to succeed. The PrepareWrite function ensures that all necessary conditions are met before overwriting the data of a block. A block can only be replaced if the id of the block requested matches the id of the block fetched.

In the event that the block fetched is different than the block requested data can still be replaced but only if the data in the fetched block was old. Replacement of data inside the block is indicated to other clients by resetting the block.cns value to 1 (i.e. only the current client knows about it) and updating the block.ts to current epoch time. We give the pseudo-code for this function in Appendix E, while Algorithm 4 presents the PrepareWrite function in detail.

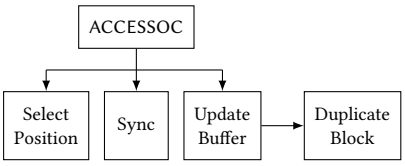


Figure 3: ACCESSOC **performs three actions: (1) selects and retrieves two blocks from the store, (2) synchronizes local map with metadata from retrieved blocks, and (3) attempts to replace retrieved blocks with buffered stored blocks.**

3.2.3 DuplicateBlock. As previously mentioned, data shuffling is performed through duplication, with two purposes: (1) increasing the number of available positions for blocks accessed more often; (2) reducing the available positions for the less accessed blocks.

The *DuplicateBlock* operation performs the necessary checks and attempts to reassign positions from one block to another while ensuring that clients still have at least one position for each block. The difficulty of this operation resides in the fact that each client has a slightly different version of the map and can only become aware of changes after they have happened (often with significant delay). In order to allow clients to attain a shared view about a position we require that positions do not get reassigned while their block.cns value is below the maximum number of clients. The single exception to this rule is when a client has reassigned a position, decides to reassign it again and no other client has noticed the reassignment. This is allowed because it will not affect the convergence time for that position. The specifics are presented in Appendix E, Algorithm 5.

3.3 Obfuscation client access

In CAOS data shuffling is performed on a single position at a time. This process leaks a lot of information to the store server because every run of the ACCESSRW function links two positions together with high probability (i.e. ideally, we require that the *DuplicateBlock* function is successful every time).

To mitigate this we use a separate obfuscation client function ACCESSOC as presented in Algorithm 3. This function is identical to the ACCESSRW function as far as the interaction with the store is concerned: two blocks are read from two server positions and two blocks are written to the same positions.

The purpose of the OC is to prevent the store from linking the positions that are duplicated thorough the regular ACCESSRW function. This is done through the use of an OC locally stored buffer. ACCESSOC places the blocks read from the store into its buffer, while blocks chosen at random from those currently stored in the buffer are written back to store. The replacement procedure is bound to the same constraints as when blocks are duplicated by regular clients, so we use the same DuplicateBlock functionality to ensure these are enforced. A detailed version of our method is presented in Algorithm 3.

Algorithm 3: Obfuscation client main access function.

Input: local buffer, obfuscation client map, store, key
Output: store, obfuscation client map

function ACCESSOC *(buf, S, map_{oc}, k)*
  $p_1 \leftarrow SelectPosition(null, map_{oc})$;
  $p_2 \leftarrow SelectPosition(null, map_{oc})$;
  $(blk_1, blk_2) \leftarrow$ READ(p_1, p_2, S, k);
  $(blk_1, map_{oc}) \leftarrow Sync(blk_1, p_1, map_{oc})$;
  $(blk_2, map_{oc}) \leftarrow Sync(blk_2, p_2, map_{oc})$;
  $(blk_1, buf) \leftarrow UpdateBuffer(blk_1, p_1, buf)$;
  $(blk_2, buf) \leftarrow UpdateBuffer(blk_2, p_2, buf)$;
  $S \leftarrow$WRITE$(S, k, blk_1, blk_2, p_1, p_2)$;
  return (S,map_{oc});

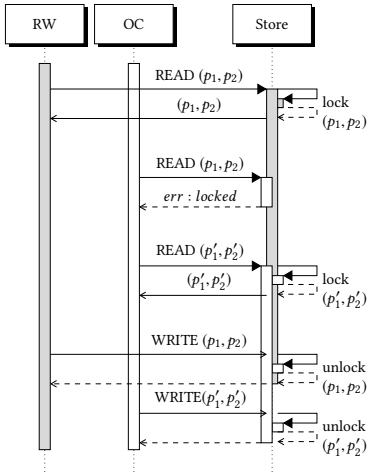


Figure 4: Concurrent access in CAOS. Upon receiving a $READ(p_1, p_2)$ message the server locks requested positions p_1 and p_2. If receiving a second request for p_1 and/or p_2, from a different client (e.g. the OC), the server will reply with an error. The second client can restart the protocol and request a new pair (e.g. p'_1 and p'_2). Positions are unlocked after the server processes the $WRITE$ position message (e.g. $WRITE(p_1, p_2)$ or $WRITE(p'_1, p'_2)$).

4 CONCURRENCY AND PARALLEL ACCESS IN CAOS

In CAOS concurrency is supported by design. Regular clients communicate with the store using two messages that are abstracted here as follows: $READ(p_1, p_2)$ and $WRITE(p_1, p_2)$. The message $READ(p_1, p_2)$ is used to request the two blocks located at positions p_1 and p_2 in the store. Upon receiving this message the server will lock p_1 and p_2 and will return the corresponding blocks to the client. The client will process the data, and will use the blocks metadata to update its local map (cf. Section 3). While the client processes the contents of the blocks the server will keep p_1 and p_2 locked. The positions p_1 and p_2 are unlocked when the server receives a $WRITE(p_1, p_2)$ message from the client that locked the positions. In the event the client is unable to send a $WRITE(p_1, p_2)$ message a timeout period is used to unlock them.

Recall that the client is interested in retrieving the content of only one of the positions, say p_1, where p_1 is chosen randomly from the set of positions associated to a given block id in the client's local map. The other position p_2 is chosen from the store at random and is used to duplicate the data from p_1. As such, a second client can request the same block using different positions. In the event that a second client requests the same positions p_1 and/or p_2 while they are locked, the server will inform the client of the lockout through an error message. By running ACCESSRW again, the second client can select new positions p'_1 and p'_2 to retrieve the requested block and a duplicate-destination block. A detailed instantiation of this procedure is presented in Fig 4.

A secondary benefit from supporting concurrency by design is that it allows *parallel access* to the store. Because each store access by CAOS functions only affects two positions at a time, regular clients (and OCs) can instantiate multiple ACCESSRW (or ACCESSOC) operations simultaneously for different blocks. This increases access speeds when the data required is stored in multiple blocks. It also affects the speed of the obfuscation process which can be increased or decreased by adjusting the bandwidth available.

5 CONTENT AND ACCESS-PATTERN PRIVACY FOR CAOS AND PROTOCOL CORRECTNESS

In this section we give a formal definition of content privacy and pattern access privacy for CAOS-like protocols. Next we analyse the privacy offered by our proposal CAOS. Finally, we present our invariants which ensure that our asynchronous concurrency protocol does not result in unintentional data loss.

5.1 Game-based privacy for CAOS

We start with the following definition.

Definition 5.1. The *access-pattern* induced by the i-th run of the ACCESSRW or ACCESSOC algorithms of a CAOS-like protocol O is the tuple $AP_i = (p_{r_1}, \ldots, p_{r_l}, p_{w_1}, \ldots, p_{w_m})$, consisting of the l positions read by the server and the m positions written by the server. The access-pattern induced by a sequence of queries is the combination of the patterns induced by the individual queries.

Attacker Model. When the algorithms ACCESSRW and ACCESSOC are run, the adversary learns the access-pattern induced by those queries. The attacker is given access to the setup algorithms INIT.STORE and INIT.OC, and thus knows the initial layout of the store and OC buffer.

Using the access-pattern definition above we define the security of a generic CAOS-like construction O against a *multiple query attacker* in the following.

Definition 5.2 (data and access-pattern privacy). CAOS *data and access-pattern privacy* is defined through a multiple query security experiment as follows (see also Fig 5). Let O = (KGen, INIT.STORE, INIT.OC, ACCESSRW, ACCESSOC) be a CAOS-like protocol over

$Exp^{b}_{\mathsf{m},\mathcal{A}}(O)$

$k \xleftarrow{R} \mathsf{KGen}(1^\lambda)$
$st_0 \leftarrow (\mathsf{INIT.STORE}, \mathsf{INIT.OC})$
for $j \in \{1, \ldots, q\}$ **do**
  for $i \in \{0, \ldots, r-1\}$ **do**
    $st_i \leftarrow \mathsf{ACCESSOC}(S, k)$
  endfor
  $(aux_j, B_{0,j}, B_{1,j}, op_{0,j}, op_{1,j}, d_{0,j}, d_{1,j}) \leftarrow \mathcal{A}_{1,j}(st_0, \ldots, st_{r-1})$
  $st_{r,j} \leftarrow \mathsf{ACCESSRW}(B_{b,j}, op_{b,j}, d_{b,j}, S, k)$
endfor
$b' \leftarrow \mathcal{A}_2(aux_1, \ldots, aux_q, st_{r,1}, \ldots, st_{r,q})$
return b'

Figure 5: CAOS data and access-pattern privacy game for a multiple query adversary $\mathcal{A} = (\mathcal{A}_{1,1}, \ldots, \mathcal{A}_{1,q}, \mathcal{A}_2)$.

an (n, N) store S, λ a security parameter, r be number of rounds the OC is run, and $b \in \{0, 1\}$. Let $\mathcal{A} = (\mathcal{A}_{1,1}, \ldots, \mathcal{A}_{1,q}, \mathcal{A}_2)$ be a q-query adversary. Let DB be a database. We consider $\mathrm{Exp}^{b}_{m,\mathcal{A}}(O)$, a probabilistic experiment defined in terms of a game played between an adversary $\mathcal{A}$ and a challenger C, consisting of:

(1) *Setup.* C runs $\mathsf{KGen}(1^\lambda)$ to create a symmetric key k, and runs $(\mathsf{INIT.STORE}, \mathsf{INIT.OC})$ to initialize the store and the OC's internal buffer.
(2) Query. During each query j:
  (a) *Obfuscation.* C runs the obfuscation algorithm $\mathsf{ACCESSOC}(S, k)$, r times.
  (b) *Challenge.* $\mathcal{A}_1$ chooses two block id's, B_0 and B_1, two operations op_0 and op_1, and two data contents d_0, d_1 to be written if any of op_0 or op_1 is a write operation. C runs $\mathsf{ACCESSRW}(B_b, op_b, d_b, S, k)$.
(3) *Guess.* $\mathcal{A}_2$ computes a guess $b' \in \{0, 1\}$, winning the game if $b' = b$. The output of the experiment is b'.

The *advantage of a multiple query adversary* $\mathcal{A} = (\mathcal{A}_{1,1}, \ldots, \mathcal{A}_{1,q}, \mathcal{A}_2)$ against the *data and access-pattern privacy* of a CAOS-like protocol O is defined as:

$$\left|\Pr\left[\mathrm{Exp}^{0}_{\mathsf{m},\mathcal{A}}(O) = 1\right] - \Pr\left[\mathrm{Exp}^{1}_{\mathsf{m},\mathcal{A}}(O) = 1\right]\right|.$$

Let us argue that the security experiment above captures both content and access-pattern privacy for CAOS-like protocols. Data privacy against the server is captured by letting the adversary choose two equal-sized data blocks B_0 and B_1 (indexed by their block id's) before every ACCESSRW query. The read-write (read-only) client will call ACCESSRW on data block B_b, where $b \in \{0, 1\}$ is unknown to the adversary. On the other hand, access-pattern privacy against the server is captured by an adversarial choosing of the operation to be performed, i.e. either op_0 or op_1, depending on the value of b that is unknown to the adversary. During the experiment, the adversary $\mathcal{A}$ has full knowledge of the instructions run and the changes occurred in the store when the read-only (read-write) and obfuscation clients will be calling the algorithms ACCESSRW, ACCESSOC as defined by the corresponding protocol O. The attacker's advantage as defined above measures how well the adversary does in gaining knowledge on b by querying ACCESSRW, ACCESSOC a number j of times. We obtain the following result:

Theorem 5.3. *CAOS has content and access-pattern privacy, i.e. the advantage of any multiple query adversary against the privacy of O is negligible in the security parameter λ and the number of OC rounds r.*

See Appendix C for a proof.

5.2 Invariants in CAOS

Different clients hold different position maps, and therefore it is possible that a client's position map will be out of date. Being a little bit out of date is not a problem: if a client seeks a block at a position and finds a different block there, the client can detect that, and seek the block at another position. However, we need to ensure that a client will always eventually find the block. We therefore prove the following invariant: *for every client and every block, the client has a valid position for the block in its map.* See Appendix D for a proof.

6 PERFORMANCE AND IMPLEMENTATION

The security of CAOS derives from the fact that the obfuscation client (OC) runs in between the accesses made by the regular clients. Intuitively, the more runs of OC, the more secure the system. In this section, we derive the expected number of rounds of OC needed to get perfect security (i.e. a situation in which the adversary has no knowledge of what block is in what position). In our algorithm, OC processes two positions per round. For simplicity, in this section, we consider an OC that processes a single position per round. Intuitively, the performance of the two-round OC is no worse than a one-round OC.

Theorem 6.1. *The expected number of rounds of OC needed to obtain perfect privacy for a (n, N) store is at most $2+s+(n-s)\log(n-s) + N \log N$, where s is the OC's buffer size.*

Since N is much greater than n and s, this is dominated by $N \log N$. Intuitively, perfect privacy cannot be achieved with less than N rounds, since every store position needs to be updated. See Appendix B for a proof.

6.1 Performance

We next report on the performance of our instantiation of CAOS in terms of storage and bandwidth, and present measured data from a non-optimized implementation of our protocol [1] for a 64GB store size.

Read-write client. Client storage in CAOS is limited to the client map and the two blocks from the store retrieved by the ACCESSRW. These, however, are only stored only for the duration of the method run.

Our bandwidth requirements are constant: two blocks are accessed for each requested data block from the store. Because the storage server only locks the positions requested for the duration of a client access, multiple blocks can be requested simultaneously in a parallel manner.

Obfuscation client. The OC accesses the store the same way the RW client does, hence the bandwidth requirements per block are the same. The OC requires additional storage than the read-write client in the form of a local buffer which is needed for the obfuscation process. In Appendix B we show how to compute r_1 the number of rounds required to obtain secure output from the OC for any given buffer size s. We note that the rate given by Eq. 1, Appendix B is valid for a single positions processed per round. Given that the OC behaves identical to the read-write client implies that the rate is in fact doubled.

Similarly, one can also compute an upper bound on the number of rounds needed to bring the store from a completely insecure state (i.e. adversary knows the contents of each position) to a secure state using Eq. 2, Appendix B.

Storage server. In order for our CAOS protocol to guarantee that blocks can be shuffled between positions it is required that each block is stored redundantly on a number of positions equal to the number of clients using CAOS. However, if the additional space is not available when the store is initialised, our protocol will automatically adjust itself allowing the size of the store to be increased gradually.

Implementation. We have implemented CAOS in C++ using the OpenSSL-1.0.2.k for the encryption operations (concretely we use AES-128 in CBC mode) and Protocol Buffers[2] for network message serialisation. Our setup consists of a server which exposes a block-store API with direct access to disk. The server's API is addressable through positions. In addition to the server we implemented a RW client and an OC client which are able to run ACCESSRW and ACCESSOC respectively.

On an Intel i5-3570 CPU at 3.40GHz running ArchLinux we have instantiated a secure store of 64GB with a redundancy factor $C = 2$ (i.e. every block is stored on two positions resulting in $N = 2n$). We measured the read/write speed to the store, client storage and server storage. Our results are presented in Table 1.

	Protocol O Sct. 3	Instantiation with $n = 10^6$
Store capacity	$n \cdot \|block\|$	64 GB
Server storage requirements	$C \cdot n \cdot \|block\|$, $C > 1$	128 GB $C = 2$
RW/RO client storage requirements	$c \cdot n$ ($c \ll 1$)	55 MB
RW/RO client bandwidth	$ps_1 \cdot 2 \cdot \|block\|$	$1 \cdot 2 \cdot 64$KB
OC buffer size	$s \cdot \|block\|$ $(2 \leq s \leq n)$	655 MB $(s = 10^4)$
OC bandwidth	$ps_2 \cdot 2 \cdot \|block\|$	$1 \cdot 2 \cdot 64$KB
$\frac{\text{client storage}}{\text{store capacity}}$		0.08%
$\|block\|$		64KB
Measured speed		131 KB/s per thread

Table 1: Storage and bandwidth requirements to store of n data blocks of size $|block|$ bytes in our CAOS instantiation. c, C are constants that are implementation-dependent and ps_1 and ps_2 are the number of parallel sessions run by the client and by the OC respectively. Figures are given in a generic notation (middle column), and as obtained from our own non-optimized implementation of the protocol (right column).

7 CONCLUSION

In this paper we have proposed Concurrent-Access Obfuscated Store (CAOS), a cloud storage solution that is able to provide access-pattern obfuscation. We have presented our concurrent access protocol that allows multiple read-only clients to simultaneously access a CAOS store. We have proved that the concurrent access does not result in data loss. We have also proven the security of our protocol for any buffer size s used by the obfuscation client given a sufficient numbers of rounds r. Finally, we have shown that security provided by the CAOS protocol is proportional to the resources provisioned for access-pattern obfuscation, namely the buffer size s and the number of rounds r. To our knowledge, there is no existing work that uses an obfuscation client to hide access patterns. However several works have been influential in the development of CAOS, and we list them in Appendix **??**.

REFERENCES

[1] [n. d.]. CAOS Souce Code. https://github.com/meehien/caos

[2] David Aldous and Persi Diaconis. 1986. Shuffling cards and stopping times. *The American Mathematical Monthly* 93, 5 (1986), 333–348.

[3] David Aldous and Jim Fill. 2002. Reversible Markov chains and random walks on graphs.

[4] Fredrik Backåker. 2012. *The Google Markov Chain: convergence speed and eigenvalues.* Ph.D. Dissertation. Master Thesis, Uppsala University, Sweden.

[5] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. 1998. Relations Among Notions of Security for Public-Key Encryption Schemes. In *Advances in Cryptology - CRYPTO '98*, Vol. 1462. Springer, 26–45.

[6] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on.* IEEE, 41–50.

[7] Cloud Security Alliance. 2016. *The Treacherous 12 - Cloud Computing Top Threats in 2016.* Technical Report.

[8] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. [n. d.]. Searchable symmetric encryption: Improved definitions and efficient constructions. *Proceedings of the 13th ACM Conference on Computer and communications security (CCS'06)* ([n. d.]), 79–88.

[9] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. [n. d.]. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology (CRYPTO '13)*. 353–373.

[10] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion ORAM: A constant bandwidth blowup Oblivious RAM. In *Theory of Cryptography*. Springer, 145–174.

[11] Persi Diaconis, James Allen Fill, and Jim Pitman. 1992. Analysis of top to random shuffles. *Combinatorics, probability and computing* 1, 02 (1992), 135–155.

[12] Oded Goldreich. 1987. Towards a theory of software protection and simulation by Oblivious RAMs. In *STOC*. ACM, 182–194.

[13] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on Oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[14] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.. In *NDSS*, Vol. 20. 12.

[15] Seny Kamara. 2014. Applied Crypto Highlights: Restricted Oblivious RAMs and Hidden Volume Encryption. http://outsourcedbits.org/2014/12/09/applied-crypto-highlights-restricted-oblivious-rams-and-hidden-volume-encryption/.

[16] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants count: practical improvements to Oblivious RAM. In *USENIX Security 15*. 415–430.

[17] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Rachel Lin, and Stefano Tessaro. 2016. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *Security and Privacy (SP), 2016 IEEE Symposium on*. Oakland.

[2] https://developers.google.com/protocol-buffers

[18] Seny Kamara and Charalampos Papamanthou. 2013. Parallel and Dynamic Searchable Symmetric Encryption.
[19] Elaine Shi, John Bethencourt, TH Hubert Chan, Dawn Song, and Adrian Perrig. [n. d.]. Multi-dimensional range query over encrypted data. In *IEEE S&P 2007*. 350–364.
[20] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 253–267.
[21] Emil Stefanov, Elaine Shi, and Dawn Song. 2012. Towards practical Oblivious RAM. 20 (2012), 12.
[22] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings ACM Conference on Computer & communications security*. 299–310.
[23] Peter Williams, Radu Sion, and Alin Tomescu. [n. d.]. Privatefs: A parallel oblivious file system. In *Proceedings of ACM CCS 2012*. 977–988.

A CAOS USE-CASE EXAMPLE

Alice wishes to move her 1TB encrypted email store to the cloud, for easy access from her three devices: a work computer, a laptop and a mobile device. She also requires that her devices have simultaneously access and access-pattern protection.

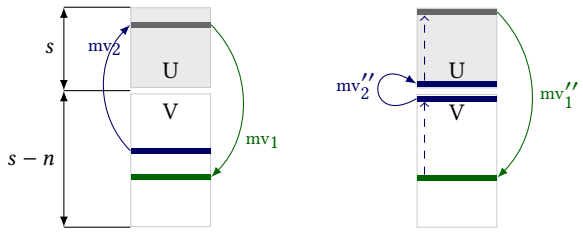
One option for Alice is to use an ORAM scheme [10, 16, 22] as they provide both data encryption and access pattern security with $O(\log N)$ efficiency. With ORAM, Alice can expect to download and upload an average of about 468KB of data[3] for each 30KB email message she wants to access.

To enable simultaneous access for her devices she can use a proxy-based ORAM scheme [17, 20]. This would allow Alice's devices to use less bandwidth, because they would interact with the proxy, however, the bandwidth between the proxy and the store will remain similar (i.e. [20] requires 30MB/s to provide 1MB/s access to the store, [17] requires a transfer of 256KB to access a 4KB store block). Unfortunately, the high bandwidth and computational requirements might force Alice to place the proxy in the cloud, an option she does not like because then access patterns to the proxy would leak information.

An alternative option for Alice would be to use CAOS. In CAOS her devices would have simultaneous and direct access to the store. Access-pattern obfuscation would be done by OCs which can be placed either in the cloud or on the premises because they access the store independently from her devices. However, access operations in CAOS randomize the store, and Alice's devices and OCs would require some form of synchronisation between them to be able to access it.

One naïve CAOS implementation which solves the synchronisation problem is to separate a part of the store, share it between devices and OCs, and use it to log the changes. During each access, clients would be able to process the log and update their access information (i.e. maps). However, if Alice were to stop using any one of her 3 device for just a month, upon resuming use, that device will have to process around 46MB worth of log entries[4] only to access a 30KB index, as the log grows linearly with the number of accesses. Security would also be affected. At the very least the variable network traffic required to retrieve the log reveals the period when the device was offline.

[3]For a 1TB ORAM store which uses 4KB blocks and 4 blocks per bucket, the size of transferred data is between 104KB and 832KB, depending on how the data is laid out in the server's memory and the ORAM scheme used.
[4]This number assumes that a log entry is 1KB, and that the remaining 2 devices will only access a 30KB email index that uses 8 blocks, 6 times per hour, 16 hours per day for 30 days. The number does not account for any OCs doing access-pattern obfuscation, in which case the number would be much higher.



(a) OBS shuffle (b) OBS″ shuffle

Figure 6: The OBS shuffle (a) consists of two moves: mv_1, which moves a random card from partition U into a random location of partition V; and mv_2, which moves a random card from partition V into the vacated place of partition U. In the OBS″ (b) shuffle mv''_1 moves the top card in U into a random location of V; and mv''_2 which moves the top card in V into the bottom place of U.

B PERFORMANCE ANALYSIS

In the following we give a proof for Theorem 6.1.

Proof. The number of rounds of OC needed is the sum of:

(1) The number r_1 of rounds needed for the buffer contents to be randomised (this is needed only the first time OC is run, when the adversary knows the contents of the buffer);
(2) The number r_2 of rounds needed for every position in the store to have been overwritten by the buffer.

To calculate r_1, consider an array of size n containing all the blocks. Let U be the sub-array containing the first s blocks of the array (these represent the contents of the OC buffer), and V the remaining $n - s$ blocks (representing those that are not in the buffer). In each round of OC, a random block B from U is selected and moved to V (this corresponds to evicting a block from the buffer), while a random block from V is moved to the former place of B in U. We seek the expected number r_1 of runs which completely randomises the array.

This situation is similar to shuffling a deck of n cards, using a "top-to-random" shuffle, a well studied method [2, 3, 11]. In a top-to-random shuffle one repeatedly takes the top card of the deck and inserts it into a random position. In the following we will argue that our shuffle, OBS (Fig 6a), is no worse than a "top-to-random" shuffle.

In OBS the deck is partitioned into U and V. It takes a random card from U and inserts it randomly in V, and takes a random card of V and inserts it into the place vacated in U. Intuitively, this is no worse than the shuffle OBS′ which simply takes a random card in U and inserts it in V, and takes the top card in V and inserts it at the bottom of U. Now consider the shuffle OBS″ (Fig 6b) which is like OBS′ but instead of taking a random card in U, we take the top card in U. Again, OBS′ is no worse than OBS″, and therefore OBS is no worse that OBS″.

To calculate the expected number of rounds of OBS″ needed, we proceed similarly to [2]. Consider the bottom card of the deck. One has to wait on average $n - s$ rounds before a card is inserted

below it. Then one has to wait $(n-s)/2$ more rounds before a card is again inserted below the original bottom card. Continuing in this way, the number of rounds needed for the original bottom to get to the U/V threshold is

$$(n-s) + \frac{n-s}{2} + \frac{n-s}{3} + \cdots + \frac{n-s}{n-s} \leq 1 + (n-s)\log(n-s).$$

A further s rounds are needed for the original bottom to progress from the U/V threshold to the top of the deck. Thus, the total number of rounds is $(n-s)\log(n-s)+s$. Since OBS is no worse than OBS$''$, we have

$$r_1 \leq 1 + (n-s)\log(n-s) + s. \tag{1}$$

Calculation of r_2 is similar. The first of the N positions is overwritten in one round. For the second position to be overwritten, we have to wait $N/(N-1)$ rounds; this is a bit longer, because we may accidentally overwrite the first one again. The ith position requires us to wait $N/(N-i)$ rounds. Thus, all the positions are overwritten after an expected

$$r_2 = 1 + \frac{N}{N-1} + \cdots + N \leq 1 + N\log N \tag{2}$$

rounds. Therefore, $r_1 + r_2 \leq 2 + (n-s)\log(n-s) + s + N\log N$. □

C ANALYSIS OF CAOS CONTENT AND ACCESS-PATTERN PRIVACY

Our aim now is to analyze privacy offered by CAOS against a multiple query adversary. To this end, we need to make use of two lemmas, starting with the following lemma.

Lemma 1. *Let S be the (n, N) CAOS-like store with $N = \{p_0, \ldots, p_{N-1}\}$ positions. Suppose that our* ACCESSOC *is run r times. Let $no(p_i)$ be the event that the position p_i was not overwritten during the r rounds. The probability that, after r runs of the* ACCESSOC *method, at least one of the N positions has not been overwritten (i.e. at least one position survived being overwritten) is:*

$$p_{N,r} = \Pr\left[\mathsf{no}(\mathsf{p}_0) \vee \ \ldots \vee \mathsf{no}(\mathsf{p}_{N-1})\right] = \\ = \sum_{i=1}^{N} (-1)^{i+1} \cdot \binom{N}{i} \cdot \left(\frac{N-i}{N}\right)^r \tag{3}$$

In order to increase its advantage in the privacy game, the adversary uses as leverage its knowledge of what blocks are in which positions in the store, and what blocks are in the OC buffer. As OC runs, the adversary might try to track the movement of blocks. We can model the adversary's knowledge of what is inside the OC buffer as a probability distribution.

Suppose the buffer size is s. The OC buffer contains at most one occurrence of a given block id. Our goal in the following is to describe the evolution of the adversary's knowledge as a probability distribution. We start by recalling that, at each step, OC selects a block to be added to the buffer. If the selected block id is already in the buffer, then the set of block ids in the buffer is not changed. But if the selected block is not currently in the buffer, then one of the buffer's slots is chosen (with uniform probability) and the block id in that slot is evicted, in order to add the selected one.
Therefore, the buffer contents will be X after such a step if:

(1) The buffer contents were already X, and an element in X was selected; or
(2) For some $x \in X$ and $e \notin X$, the buffer contents was $(X - \{x\}) \cup \{e\}$, and x got selected for adding to the buffer, and e was chosen for eviction from the buffer and gets put back in the store where x had been.

We view this as a system of states and transitions. A *state* is the pair (X, sel), where X is a set of block ids, and sel is a function from block ids to reals that abstracts the store. The value $\mathrm{sel}(x)$ is the probability that a position in the store chosen uniformly at random contains block id x. Since x must occur at least once in the store, and since every other block must also occur at least once, x appears at least once and at most $N-n+1$ times. Therefore, for any x, the value $\mathrm{sel}(x)$ is in the set $\{\frac{1}{N}, \frac{2}{N}, \ldots, \frac{N-n+1}{N}\}$.

There are two kinds of transition, corresponding to the two points above:

(1) For each $x \in X$: (X, sel) can transition to (X, sel), with probability $\mathrm{sel}(x)$.
(2) For each $x \in X$ and $e \notin X$:
$((X-\{x\})\cup\{e\}), \mathrm{sel}\left[\begin{smallmatrix} x \mapsto \mathrm{sel}(x) + \frac{1}{N} \\ e \mapsto \mathrm{sel}(e) - \frac{1}{N} \end{smallmatrix}\right])$ can transition to (X, sel), with probability $\mathrm{sel}(x) \times \frac{1}{s}$. (Here, the notation $\mathrm{sel}[x \mapsto expr]$ means the function sel but with x mapping to $expr$. The factor $\frac{1}{s}$ in the transition probability is the probability that e is evicted.)

This system is easily seen to be a Markov chain with eigenvalues $1 = |\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$. Moreover, this Markov chain describes a random walk in a strongly connected Eulerian digraph in which nodes represent the possible states (X, sel) and the edges represent the transitions between them.

Let us write $\mathrm{inb}_j(X)$ for the probability that, after the jth run, the set of block ids present in the buffer is X. We can state the following lemma.

Lemma 2. *Let $\mathrm{out}_j(B)$ be the probability that the block coming out of an OC with buffer size s at the j-th step is the block B. For all $j, j' > r$ and initial distributions inb_0, the probability $|\mathrm{out}_j(B) - \mathrm{out}_{j'}(B)|$ is at most*

$$\binom{n-1}{s-1} \cdot C \cdot |\lambda_2|^r$$

for some constant C and $\lambda_2 < 1$, and is therefore negligible in r.

Proof. Let P be the transition matrix for this Markov chain. Then P is stochastic, irreducible and aperiodic. Let π_j be the probability distribution after j steps. Then $\pi_j = \pi_0 \cdot P^j$. Let λ_1 and λ_2 be the largest and second-largest eigenvalues of P. Overwhelmingly likely, the enginvalues are all distinct. Since P is stochastic, $\lambda_1 = 1$ and $|\lambda_2| < 1$. Applying the Perron-Frobenius Theorem similarly to Backaker [4, Theorem 3 and Example 1], we have $P^r = P^\infty + O(|\lambda_2|^r)$, and therefore $\pi_j - \pi_{j'} = \pi_0 \cdot C'|\lambda_2|^r$ for $j, j' > r$ for some constant C'.

Let $\pi(X, \mathrm{sel})$ be the probability that the state is (X, sel) in the probability distribution π. Then $\mathrm{inb}_j(X) = \sum_{\mathrm{sel}} \pi_j(X, \mathrm{sel})$. Let $m_{j,j'} = \max_X(|\mathrm{inb}_j(X) - \mathrm{inb}_{j'}(X)|)$, where X ranges over sets of block ids of size s. Then, for $j, j' > r$,

$$\begin{aligned} m_{j,j'} &= \max_X\left(\textstyle\sum_{\mathrm{sel}} |\pi_j(X, \mathrm{sel}) - \pi_{j'}(X, \mathrm{sel})|\right) \\ &= \max_X\left(\textstyle\sum_{\mathrm{sel}} (\pi_0(X, \mathrm{sel}) \cdot C'|\lambda_2|^r)\right) \\ &= C'|\lambda_2|^r \cdot \max_X\left(\textstyle\sum_{\mathrm{sel}} \pi_0(X, \mathrm{sel})\right). \end{aligned}$$

Notice that $\text{out}_j(B) = \sum_{X|B\in X} \text{inb}_j(X)$. Therefore,

$$\begin{aligned}|\text{out}_j(B) - \text{out}_{j'}(B)| &= \sum_{X|B\in X} |\text{inb}_j(X) - \text{inb}_{j'}(X)| \\ &\leq \binom{n-1}{s-1} \cdot m_{j,j'} \\ &\leq \binom{n-1}{s-1} \cdot C \cdot |\lambda_2|^r\end{aligned}$$

where $C = C' \cdot \max_X(\sum_{\text{sel}} \pi_0(X, \text{sel}))$. □

Theorem C.1. *CAOS has content and access-pattern privacy, i.e. the advantage of any multiple query adversary against the privacy of CAOS is negligible in the security parameter λ and the number of OC rounds r.*

Proof. We bound the distinguishing advantage of any CAOS adversary through the following sequence of "game hops".

Game 1. This is the legitimate $\text{Exp}^0_{m,\mathcal{A}}(O)$ experiment. In particular, in the j-th challenge query C returns ACCESSRW($B_{0,j}$, $op_{0,j}$, $d_{0,j}$, S, k), for $j = 1, \ldots, q$.

We define the following series of experiments for $j = 1, \ldots, q$.

Game 2, j. Let $j \in \{1, \ldots, q\}$ be fixed. In this experiment, the queries $i = 1, \ldots, j$ are answered with ACCESSRW($B_{0,i}$, $op_{0,i}$, $d_{1,i}$, S, k), namely by performing operation $op_{0,i}$ on block $B_{0,i}$ with data $d_{1,i}$ (instead of data $d_{0,i}$). The remaining queries until the q-th query are responded normally, i.e. with ACCESSRW($B_{0,i}$, $op_{0,i}$, $d_{0,i}$, S, k) for $i = j+1, \ldots, q$.

Let us call this modified experiment $\text{Exp}^{2,j}_{m,\mathcal{A}}(O)$. One can easily see that

$$\left|\Pr\left[\text{Exp}^{2,\text{j}-1}_{\text{m},\mathcal{A}}(O) = 1\right] - \Pr\left[\text{Exp}^{2,\text{j}}_{\text{m},\mathcal{A}}(O) = 1\right]\right|$$

for $j = 1, \ldots, q$ is upper-bounded by the distinguishing advantage against the semantic security of the encryption scheme, which is negligible [5]. Trivially $\text{Exp}^{2,0}_{m,\mathcal{A}}(O) = \text{Exp}^0_{m,\mathcal{A}}(O)$.

Next, for $j = 1, \ldots, q$ we define a new sequence of experiments.

Game 3, j. Fix j. In this experiment, the queries $i = 1, \ldots, j$ are answered with ACCESSRW($B_{1,i}$, $op_{0,i}$, $d_{1,i}$, S, k), i.e. by running operation $op_{0,i}$ on block $B_{1,i}$ with data $d_{1,i}$ (instead of running it on block $B_{0,i}$). The remaining queries until the q-th query remain unchanged, i.e. they return ACCESSRW($B_{0,i}$, $op_{0,i}$, $d_{1,i}$, S, k) for $i = j+1, \ldots, q$. Let us call resulting experiment $\text{Exp}^{3,j}_{m,\mathcal{A}}(O)$.

We proceed to upper bound the probability that an adversary has in distinguishing access to two different blocks $B_{2,j}$ and $B_{1,j}$ in experiment $\text{Exp}^{3,j}_{m,\mathcal{A}}(O)$.

Consider the r runs of OC. Let $r_1 = r/2$; we will distinguish between the first set of r_1 runs, and the remaining $r - r_1$ runs. Now we distinguish between two situations that arise after the runs:

State A The adversary has observed that all the positions in the store got overwritten by a block id coming out of OC during the second set of $r - r_1$ runs.

State B The adversary has observed that not all the positions got overwritten; that is, at least one position survived being overwritten during the second set of $r - r_1$ runs.

When a store position is overwritten at step j by a block coming from OC, the adversary's probability distribution of what block is in that store position is $\text{out}_j(\cdot)$. If State A is observed, then the adversary's probability of distinguishing the block read in ACCESSRW is at most

$$\max_{B,i,i'} |\Pr\left[\text{S}_\text{i} \text{ contains B}\right] - \Pr\left[\text{S}_{\text{i}'} \text{ contains B}\right]|$$

which is at most $\max_{r_1 < j,j' < r} |\text{out}_j(B) - \text{out}_{j'}(B)|$. By Lemma 2, this value is at most $\binom{n-1}{s-1} \cdot S^r$.

The probability that state B is observed is p_{N,r_1} (this notation is defined in Lemma 1).

The probability of distinguishing is the probability of arriving in state A times the probability of distinguishing in state A, plus the probability of arriving in state B times the probability of distinguishing in state B. This is at most:

$$(1 - p_{N,r}) \times \binom{n-1}{s-1} \cdot S^r + p_{N,r} \times 1 \tag{4}$$

Since both S^r and $p_{N,r}$ are negligible in r, the probability and therefore the advantage of the adversary is negligible in r.

Let us define the last series of game hops, again for $j = 1, \ldots, q$.

Game 4, j. In this experiment, the queries $i = 1, \ldots, j$ are answered with ACCESSRW($B_{1,i}$, $op_{1,i}$, $d_{1,i}$, S, k), i.e. by running operation $op_{1,i}$ on block $B_{1,i}$ with data $d_{1,i}$ (instead of running operation $op_{0,i}$). The remaining queries until the q-th query remain unchanged, i.e. they return ACCESSRW($B_{1,i}$, $op_{0,i}$, $d_{1,i}$, S, k) for $i = j+1, \ldots, q$. Let us call resulting experiment $\text{Exp}^{4,j}_{m,\mathcal{A}}(O)$. Actually, it holds that

$$\Pr\left[\text{Exp}^{4,\text{j}-1}_{\text{m},\mathcal{A}}(O) = 1\right] = \Pr\left[\text{Exp}^{4,\text{j}}_{\text{m},\mathcal{A}}(O) = 1\right]$$

by construction. This is because ACCESSRW performs the same (read position followed by write position) both for read and write operations. It is easy to see that $\text{Exp}^{4,q}_{m,\mathcal{A}}(O) = \text{Exp}^1_{m,\mathcal{A}}(O)$.

Finally, by adding the probabilities obtained in each game hop, the statement of the theorem follows. □

D INVARIANTS

We prove the following invariant: *for every client and every block, the client has a valid position for the block in its map* (INV-2 below). In the following we will use $block[p]$ notation to denote the block contained at position p in the server store, and $map_C[B]$ to denote the entry for block id B in the map of client C.

We start with a simpler invariant, which is a useful lemma.

INV-1. The variable $block[p].cns$ on the server is at most equal to the number of clients that know that that block is at position p. More precisely: for all positions p,

$$\begin{aligned}block[p].cns = \\ \Big|\{C \in Clients \mid p \in map_C[block[p].bid].psns \\ \wedge\, map_C[block[p].bid].ts = block[p].ts\Big|\end{aligned} \tag{5}$$

The code maintains this invariant by linking any change to the local map with an operation on the *block[p].cns* value. These changes happen when the contents of the block is updated or deleted (by marking the block as *free*). The SyncPositons operation (see

Algorithm 2) updates a client's local map to reflect changes performed by other clients and frees old data. The *WriteBlock* and *DuplicateBlock* set the *block[p].cns* value to 1 in order to trigger map changes in other clients.

As a corollary, we have the following:

INV-1′. When $block[p].cns$ has the maxim value ($|Clients|$), then every client's local map contains the latest information about position p. More precisely, for each p:

$$block[p].cns = |Clients| \Rightarrow \forall C \in clients,\ p \in map_C[block[p].bid].psns \wedge \wedge\ map_C[block[p].bid].ts = block[p].ts \quad (6)$$

INV-2. For each block, a valid position is always known to all CAOS clients.

More precisely, for any block id B, there is a position p such that

$$block[p].bid = B \quad \wedge \quad p \in \bigcap_{C \in Clients} map_C[B].psns$$

Before we prove this invariant, we provide some intuition. To maintain INV-2, we use the $map[B].vf$ set stored in the client's map. This set tracks which are the positions p of a block that a client has observed to have a maximum value for $block[p].cns$. In order to prevent data loss, we require that (1) at any time each client can only reassign a single position, and (2) that at least one position still remains if all the clients decide to reassign one position.

The second part of the requirement (2) is easily achieved by checking that the size of $map[B].vf$ is bigger than the number of clients. We address the first requirement (1) by requiring each client mark the $map[B].vf$ set as empty whenever they reassign a position from it. This will prevent the client to reassign any consolidated positions until it re-learns their location.

Proof. We prove it for the case that there are two clients, say C and $\mathcal{D}$; as will be seen, the proof generalises intuitively to more clients. Suppose INV-2 is not an invariant; then there is a transition from a state st_3 in which INV-2 holds for a block id B, to a state st_4 in which it does not hold for B. Suppose client C reallocates the crucial position p in st_3 which is lost in st_4. Since st_3 satisfies INV-2, in st_3 $p \in map_C[B].psns$ and $p \in map_{\mathcal{D}}[B].psns$, and since st_4 does not satisfy INV-2, in st_4 $map_C[B].psns \cap map_{\mathcal{D}}[B].psns = \emptyset$. Using Algorithms 2 and 5, we see that the transition for C from st_3 to st_4 required $|map_C[B].vf| > 2$ in st_3, so suppose that $map_C[B].vf \supseteq \{p, q, r\}$ in st_3. Since $vf \subseteq psns$ (Algorithms 2 and 5), we have $\{p, q, r\} \subseteq map_C[B].psns$ in st_3.

Let st_0 be the state immediately after the previous reallocation by C. Then $map_C[B].vf = \emptyset$ in st_0. Since $q, r \in map_C[B].vf$ in st_3, there was a state between st_0 and st_3 in which $q.cns = 2$, and one in which $r.cns = 2$. Let st_1 be the most recent of those states. Either q or r was reallocated between st_1 and st_3, and by definition of st_0, that reallocation was done by $\mathcal{D}$. Consider the most recent reallocation by $\mathcal{D}$ done before st_3, say in a state st_2. Now we have states $st_0, st_1, st_2, st_3, st_4$ in temporal order. Based on Agorithms 2 and 5, $|map_C[B].vf| > 2$ in st_2, so say $map_{\mathcal{D}}[B].vf \supseteq \{t_1, t_2, t_3\}$. Then, in st_2 we have: $block[t_1].cns = block[t_2].cns = block[t_3].cns = 2$, and by INV-1′, $\{t_1, t_2, t_3\} \subseteq map_C[B].psns \cap map_{\mathcal{D}}[B].psns$. In st_3, $\mathcal{D}$'s transition has removed one element from $map_{\mathcal{D}}[B].psns$, and in st_4, C's transition has removed one element from $map_C[B].psns$. Therefore, in st_4, $map_C[B].psns \cap map_{\mathcal{D}}[B].psns$ is non-empty, contradicting our hypothesis. □

E ALGORITHMS

Algorithm 4: Write a block to the store.

Input: block id, block, data
Output: block, status

```
1  function PrepareWrite (bid, block, data)
2      if (block.bid = bid or block.bid = free) then
3          block.bid ← bid;
4          block.data ← data;
5          block.cns ← 1;
6          block.ts ← current_time;
7          status ← block.data;
8      else
9          status ← null;
10     return (block, status);
```

Algorithm 5: Duplicate a store block to a new position.

Input: source block, destination block, destination position
Output: destination block, client map

```
1  function DuplicateBlock (sblock, dblock, p, map_c)
2      if ((dblock.cns = |clients| and
         |map_c[dblock.bid].vf| > |clients|) or
3        (dblock.cns = 1 and |map_c[dblock.bid].psns| > 1))
4      then
5          dblock.bid ← sblock.bid;
6          dblock.data ← sblock.data;
7          dblock.ts ← sblock.ts;
8          dblock.cns ← 1;
9          clear map_c[dblock.bid].vf;
10         move p from map_c[dblock.bid].psns to
             map_c[sblock.bid].psns;
11     return (dblock, map_c);
```

Algorithm 6: Update the local buffer data structure.

Input: block, position, buffer, obfuscation client map
Output: block, buffer

```
1  function UpdateBuffer(blk, p, buffer, map_oc)
2      if (buffer not full) then
3          add blk to buffer;
4      if (buffer is full) then
5          buf_blk ←R buffer;
6          (blk, map_oc) ←
             DuplicateBlock(buf_blk, blk, p, map_oc);
7          if (buf_blk = blk) then
8              remove buf_blk from buffer;
9      return (blk,buffer);
```