Obladi: Oblivious Serializable Transactions in the Cloud

NATACHA CROOKS
MATTHEW BURKE
ETHAN CECCHETTI
SITAR HAREL
RACHIT AGARWAL
LORENZO ALVISI

Laboratory for Advanced Systems Research

Department of Computer Science at The University of Texas at Austin
This talk

Obladi

a cloud-based transactional key-value store
that supports ACID transactions
but hides from the cloud what, when, and how data is accessed
Why Obladi – Cloud Privacy Concerns

Applications are moving to the cloud

Applications store sensitive information

Cloud storage means sharing data with an untrusted party

Cloud services can be the target of hacking, subpoena
Protecting sensitive information

Electronic Health Record (EHR) systems
- store/manage patient data
- underpin large hospitals

Molly: Medical Record 1
Desmond: Medical Record 2
Protecting sensitive information

Use encryption to hide contents of the data

Still leaking information about what data is being accessed

Still leaking information about when data is being accessed
Guaranteeing *obliviousness*

Hiding access patterns (*obliviousness*)

- **what** data is being accessed
- **when** data is being accessed
- **how** data is being accessed
How to maintain functionality?

Large body of work on analytical queries

but no way to run ACID transactions obliviously

This talk:

How to obliviously and efficiently implement serializable ACID transactions on top of untrusted cloud storage
Security Guarantees

The adversary should learn no information about

1. the **data accessed** by ongoing transactions
2. the **type of operations** in ongoing transactions
3. the **size** of ongoing transactions
4. the **outcome** of ongoing transactions
Threat Model

Obladi adopts the **trusted proxy model**

- **Clients** communicate with a **trusted proxy**.
- **Untrusted Cloud Storage**

- Doctors communicating over hospital LAN
- Cloud storage (Dynamo, S3, etc.) accessed over WAN
Failure Model

Obladi assumes clients and proxy *can fail*

But that cloud storage is **reliable**
Obladi’s security in a nutshell

**Workload Independence**

Obladi ensures that the request pattern sent to the untrusted cloud is *independent* of ongoing transactions.
The paradox of transactions

Transactions make guaranteeing obliviousness **harder**

Isolation and durability **add structure** to read/write operations

Transactions make improving efficiency **easier**

ACID must hold at **commit time** only
Oblivious RAM [Goldreich1996]

Obladi builds on **Oblivious RAM (ORAM)**

ORAM hides access patterns for read and write operations by making requests to untrusted storage independent of workload
ORAM from 1000 feet

Generate physical read/write requests from logical operations

Send requests to (encrypted) dummy data to hide what is being requested
Challenges of Transactional ORAM

ORAM guarantees workload independence for read/write operations. How can we preserve workload independence but also

1) Guarantee Isolation and Atomicity? 
   No concurrency control

2) Guarantee Consistency and Durability? 
   Write-back ordering for security vs for durability

3) Guarantee good performance? 
   Limited Concurrency
Delayed Visibility

Obladi centers its design around the notion of

delayed visibility

On the one hand, ACID guarantees apply only when transactions commit

On the other, commit operations can be delayed
The secret sauce: epochs

Obladi uses delayed visibility to partition transaction into fixed-sized epochs

Delays commit notifications until the epoch ends
The secret sauce: epochs

**ACID guarantees** only hold for committed transactions.

Enforce durability and consistency at **epoch boundaries** only.

---

**Epoch 1**
- B
- W(a)
- R(b)
- W(b)

**Epoch 2**
- B
- R(x)
The secret sauce: epochs

Within an epoch, Obladi executes transactions at the trusted proxy, buffering writes until epoch ends.
The secret sauce: epochs

Delayed visibility improves **performance**

1. Reduces number of requests sent to ORAM
   Only write the last version of every key

2. Implement multi-versioned concurrency control algorithm on top of single-versioned ORAM
   Better support for read-only transactions
The secret sauce: epochs

Delayed visibility should not increase contention

Should allow transactions in the same epoch to see each other’s effects

Obladi chooses a concurrency control that optimistically exposes uncommitted writes to ongoing transactions
The secret sauce: epochs

The fixed structure of epochs helps guarantee **workload independence**.

ORAM observes the **same sequence** of reads followed by the buffered writes.

Epoch 1

<table>
<thead>
<tr>
<th>Reads</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(x)</td>
<td>PAD</td>
</tr>
<tr>
<td>R(y)</td>
<td>PAD</td>
</tr>
<tr>
<td>R(b)</td>
<td></td>
</tr>
<tr>
<td>R(a)</td>
<td></td>
</tr>
</tbody>
</table>
How to guarantee good performance?

Send batches of requests to ORAM

But ORAM constructions are largely sequential
Parallelising ORAM

How can we parallelise ORAM?

For **correctness**: parallelization should be *linearizable*

For **security**: parallelization should be *workload independent*
Parallelising ORAM

Recall: breakdown logical operations into physical read/writes to cloud storage
Guaranteeing linearizability

To ensure linearizability

Execute operations that do not have **data dependencies** in parallel

Data-dependent operations must be executed sequentially
Dependencies violate independence

Wait for data dependencies to be satisfied introduces **timing channels**

Only exist between real objects, not dummies

Delaying reads for real objects causes **delay**, dummy objects don’t
Introduces side-channel

Must wait for all potential data dependencies
Can exist between any pairs of reads and writes

Never secure to execute reads and writes in parallel
Delayed visibility to the rescue

Delayed visibility allows ORAM to be consistent at epoch boundaries only

Writes can be safely delayed to epoch end
Delayed visibility to the rescue

Separate ORAM execution into a **read phase** and a **write phase**

- **Read Phase**: reads all necessary blocks
- **Write Phase**: writes all necessary blocks
Delayed visibility to the rescue

Executing each phase in turn obscures data dependencies
Still allows high concurrency
How to guarantee durability?

Must ensure recovery to a **consistent state**
No partially executed transactions are included

Traditionally achieved through **redo/undo logging**
For **consistency**: pretend partial transactions never happened
For **security**: cannot “undo” what the adversary observed

May lead to access sequences that **violate** workload independence
More details in the paper

Durability and recovery logic details

Additional optimisations for performance

Discussion of our chosen ORAM construction: RingORAM [Ren15]

Formal proof of security
Evaluation

Applications

TPC-C
(10 Warehouses)

SmallBank
(1 million records)

FreeHealth
(7000 patients, 10 hospitals)

Baselines

Obladi
(Our system)

NoPriv Baseline
(Shares concurrency logic with Obladi)

MySQL 5.7 InnoDB Baseline
(Server co-located with clients)

c5.4xlarge AWS instances. 10 ms latency between proxy and storage
Performance Results: The Good

Obladi is slow, but not too slow

Between 5x and 9x lower throughput for contention-bottlenecked TPC-C and FreeHealth

12x lower throughput for resource-bottlenecked SmallBank
Performance Results: The Bad

Batching significantly increases latency

Up to 70x on TPC-C

Better on other applications because of smaller write batches
Performance Results: The Ugly

Performance is sensitive to good tuning of epoch size

If too low, transactions cannot finish
If too high, idle time
Performance Results: The Ugly

Performance is sensitive to good tuning of epoch size

If too low, transactions cannot finish
If too high, idle time

May reveal type of application!
Conclusion

Obladi, a cloud-based transactional key-value store that **obliviously** supports **ACID transactions** using **delayed visibility**

Any questions?
Backup