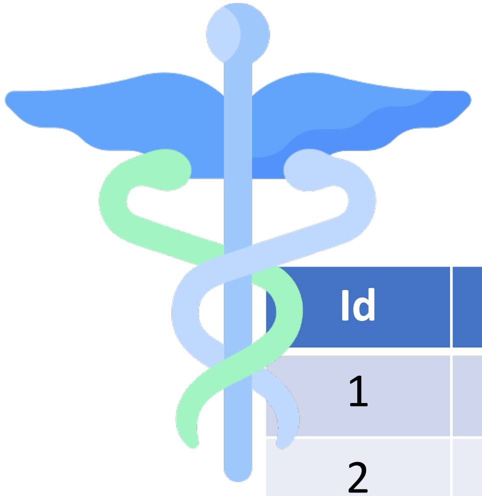
CS848
Oblivious RAM

Sujaya Maiyya

Slides partially acquired from Prof. Amr El Abbadi

Data encryption to achieve privacy?



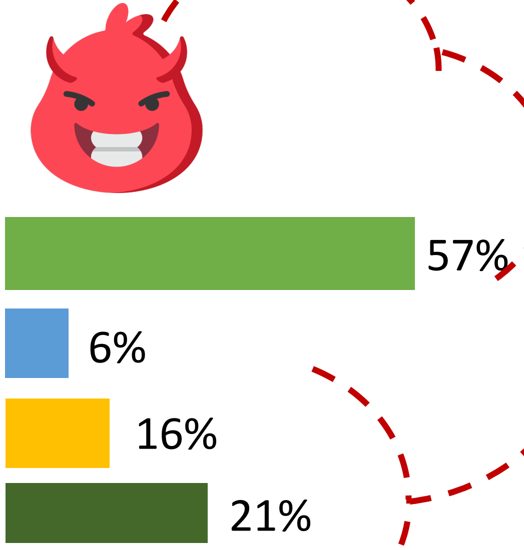
Id	Medicine
1	Humira
2	Januvia
3	Tivicay
4	Herceptin

Honest-but-curious adversary

Id	Medicine
X12	S6C…23
2SD	1NW…SJ
D45	3G8…SO
F4A	DJW…O8

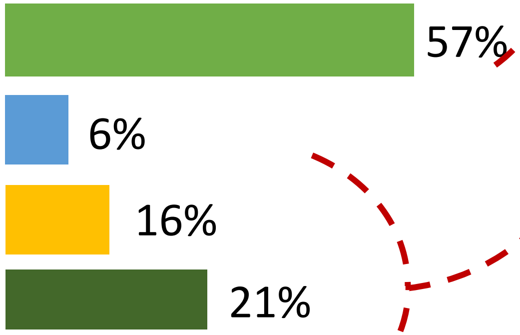
Encryption is **not** sufficient for data privacy

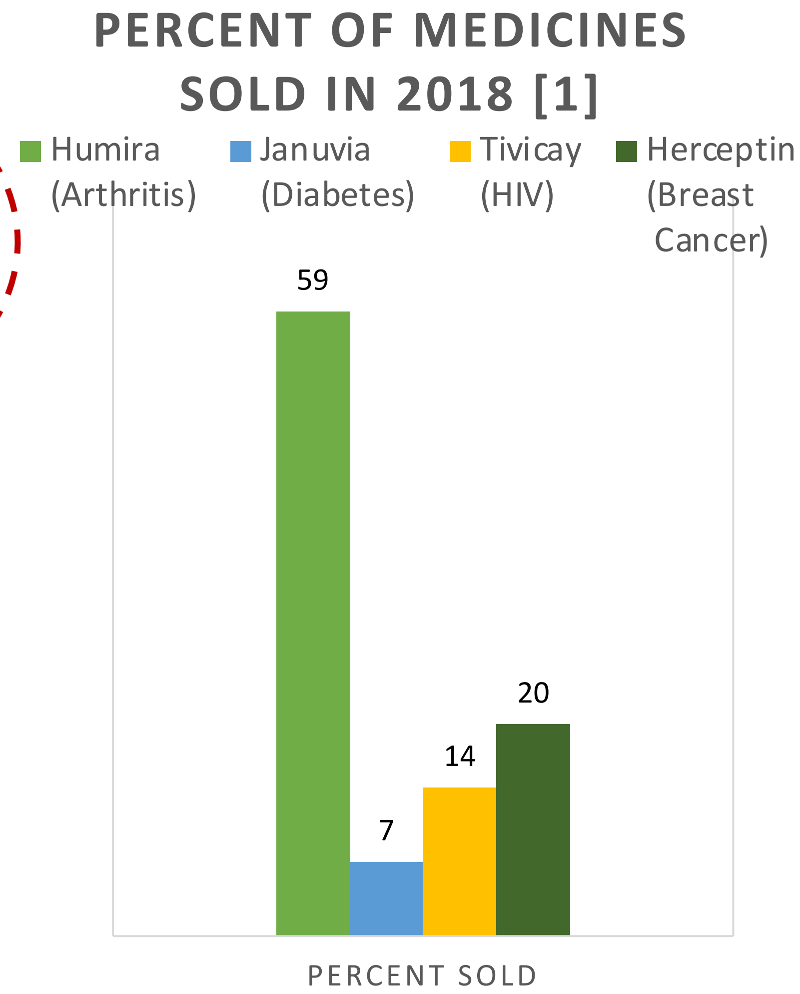
Id	Medicine
X12	S6C…23
2SD	1NW…SJ
D45	3G8…SO
F4A	DJW…O8



Encryption is **not** sufficient for data privacy

Id	Medicine
X12	S6C…23
2SD	1NW…SJ
D45	3G8…SO
F4A	DJW…O8

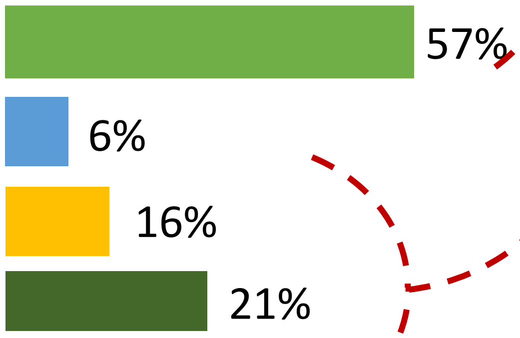


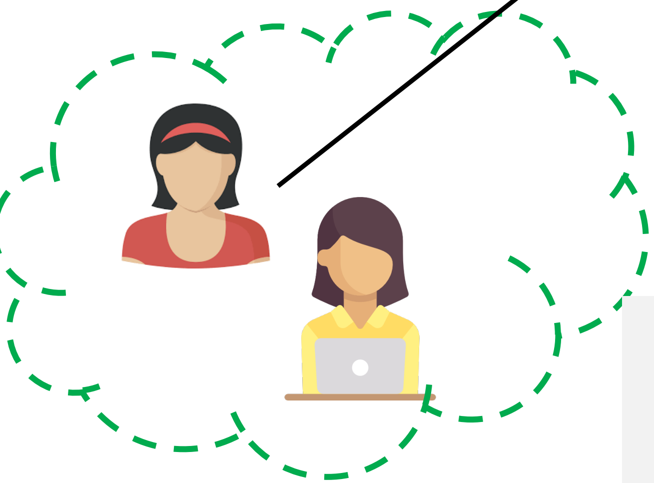


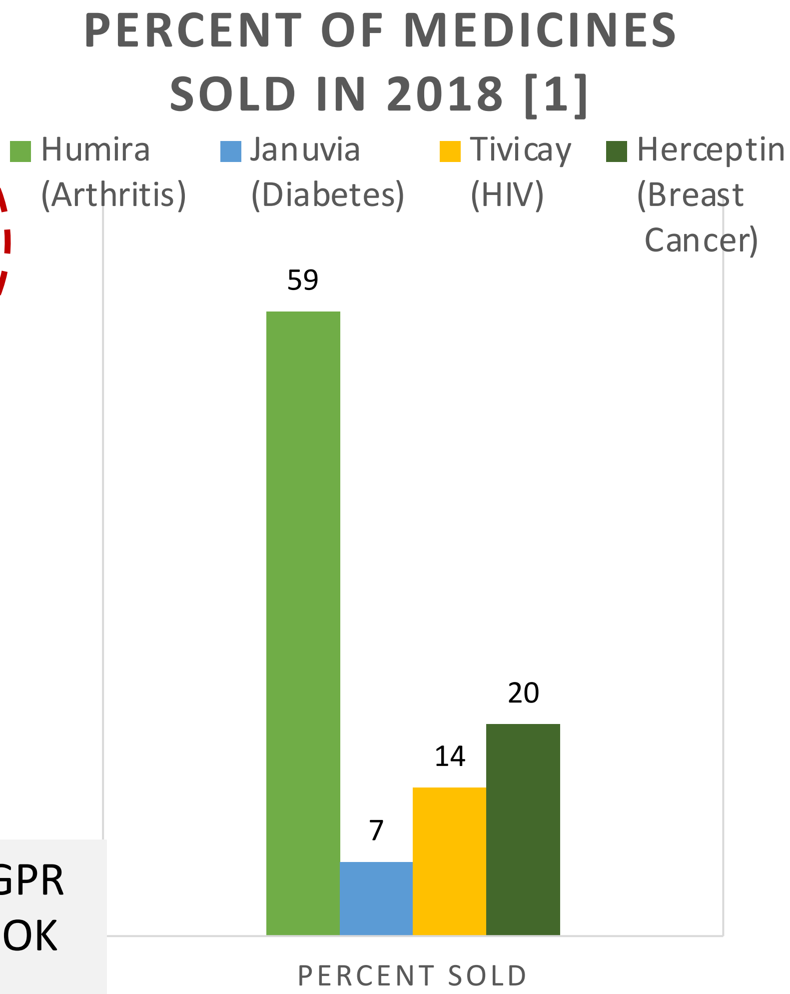
[1] https://truecostofhealthcare.org/pharmas-50-best-sellers/

Encryption is **not** sufficient for data privacy

Id	Medicine
X12	S6C…23
2SD	1NW…SJ
D45	3G8…SO
F4A	DJW…O8







Access Pattern Attacks

Many practical attacks: [IKK NDSS'12], [NKW CCS'15], [CGPR CCS'15], [KKNO CCS'16], [GLMP S&P'19], [KPT S&P'19], [OK Security'21], [OK Security'22]

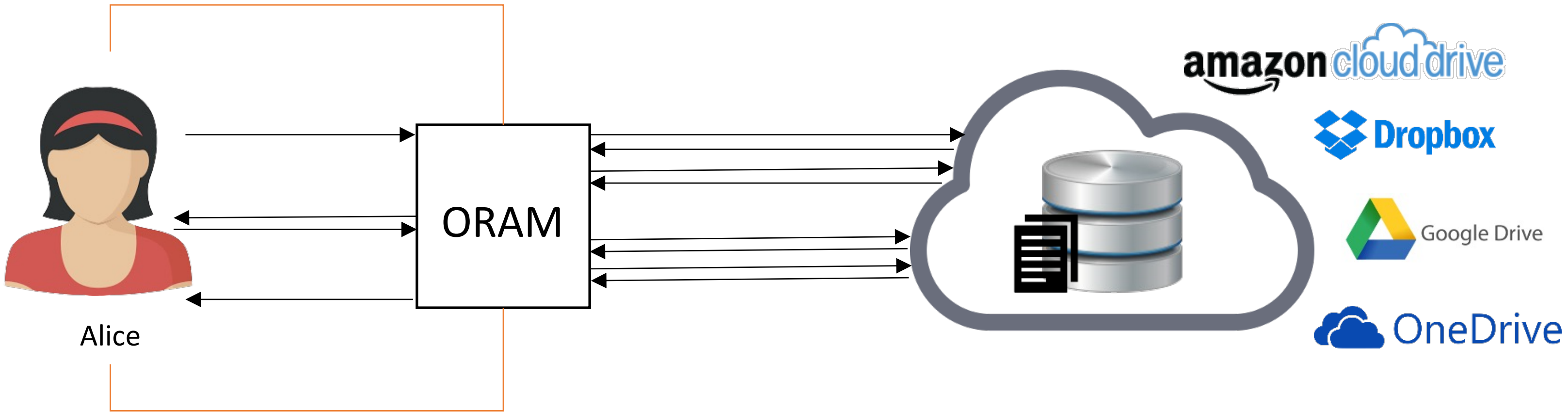
[1] https://truecostofhealthcare.org/pharmas-50-best-sellers/

Workload independence

to protect against these attacks by hiding…

- which data is being accessed
- how old it is (when it was last accessed)
- whether the same data is being accessed
- access pattern (skewed vs. uniform)
- whether the access is a read or a write

Random accesses ensures workload independence



Goal: Oblivious Access

Translate each logical access
to a sequence of random-looking accesses

OBLIVIOUS RAM (ORAM)

Initially proposed by [Goldreich and Ostrovsky, JACM'96]

ORAM provides workload independence

- Clients wish to outsource data to an untrusted cloud storage
- Honest-But-Curious cloud can control & observe network & cloud storage
- Keep the **data** and **access pattern** private

Client 1

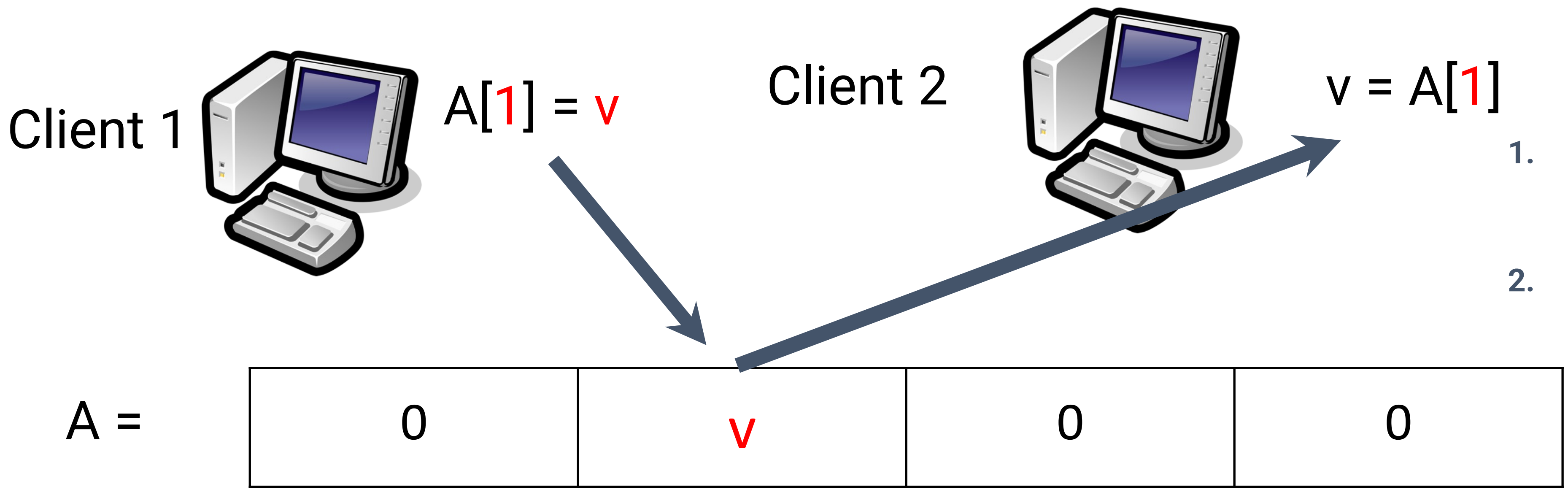
Client 2

A =

0	0	0	0

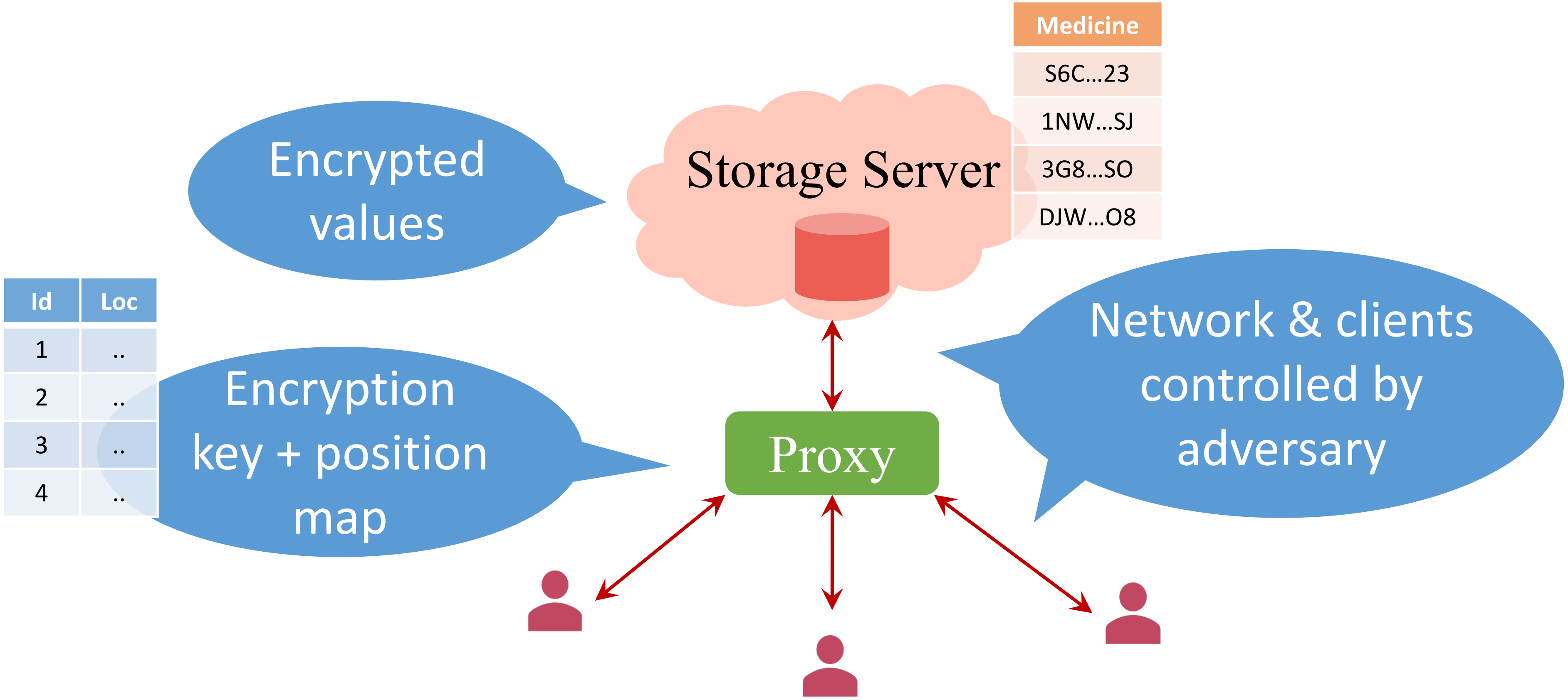
ORAM provides workload independence

- Clients wish to outsource data to an untrusted cloud storage
- Honest-But-Curious cloud can control & observe network & cloud storage
- Keep the **data** and **access pattern** private

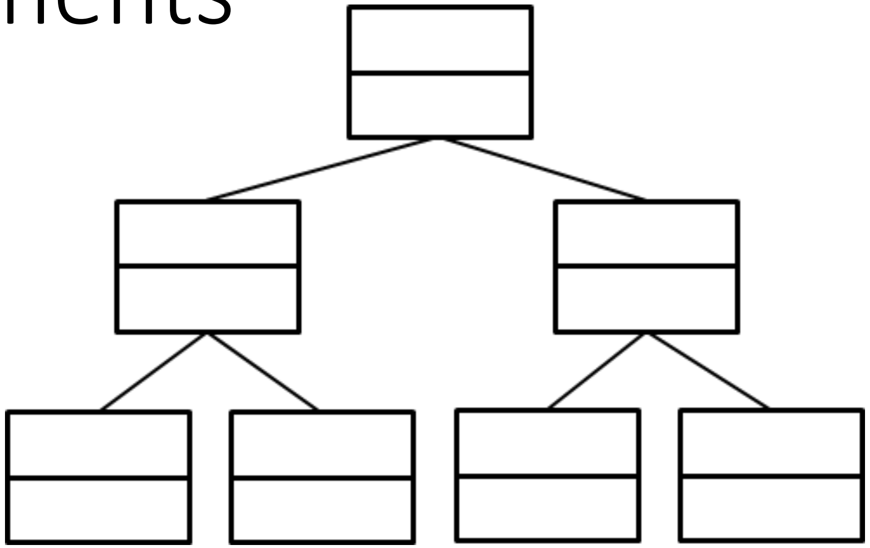


1. The cloud never learns about **v**
2. The cloud never learns that clients accessed **index 1**

Typical (but not all) ORAM architecture



Tree-based ORAM Developments



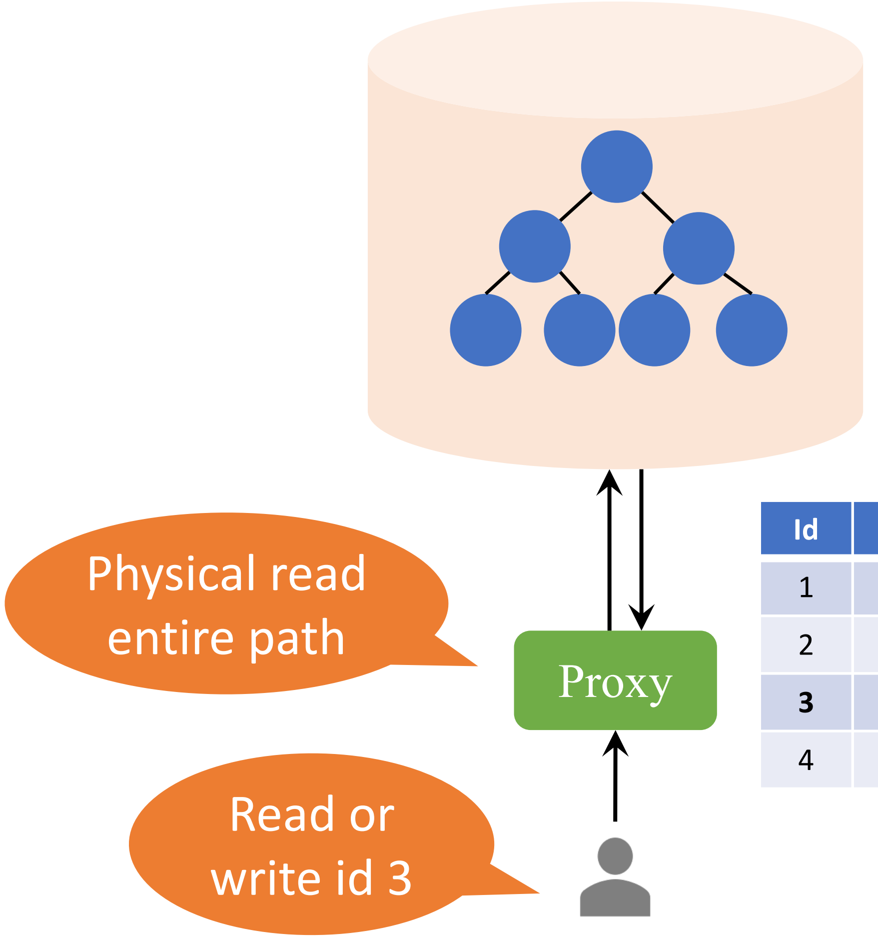
- While other forms ORAM constructions exist, most are theoretical in nature

A practical and famous solution

- Path ORAM: an extremely simple oblivious RAM protocol [Stefanov et al. CCS'13]

1000 ft overview of ORAM (PathORAM[1])

Step 1. Read path

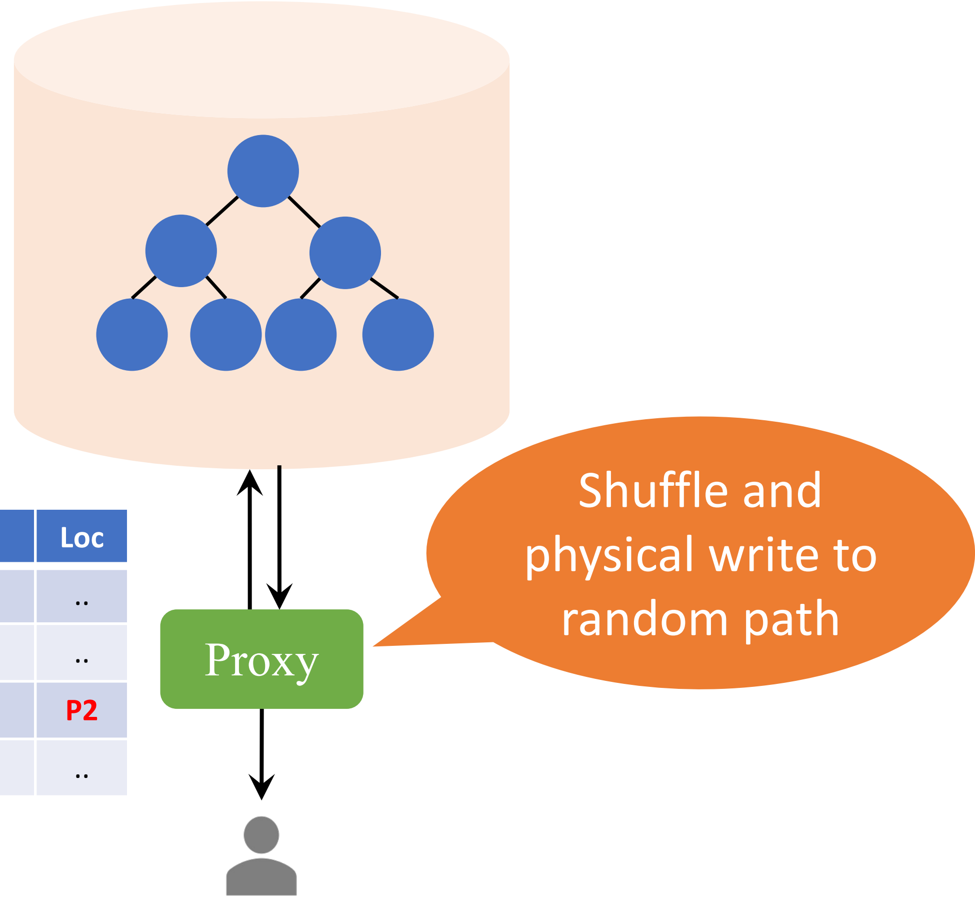


Id	Loc
1	..
2	..
3	**P1**
4	..

Path id changes to another random path after each access to 3

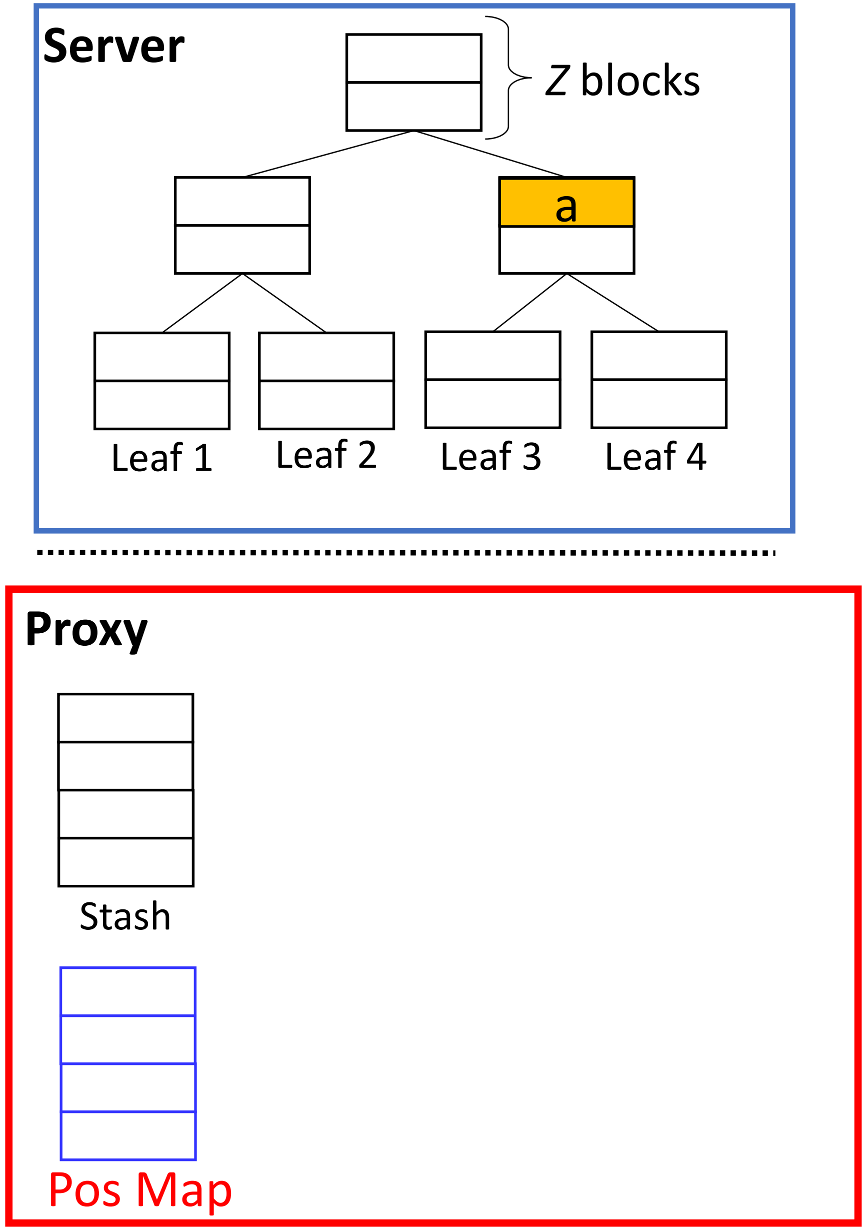
Step 2. Shuffle and Write path

Id	Loc
1	..
2	..
3	**P2**
4	..



[1] E. Stefanov, et al. "Path ORAM: an extremely simple oblivious RAM protocol." *Proceedings of the 2013 ACM SIGSAC*. 2013.

Path ORAM [Stefanov et al. CCS'13]

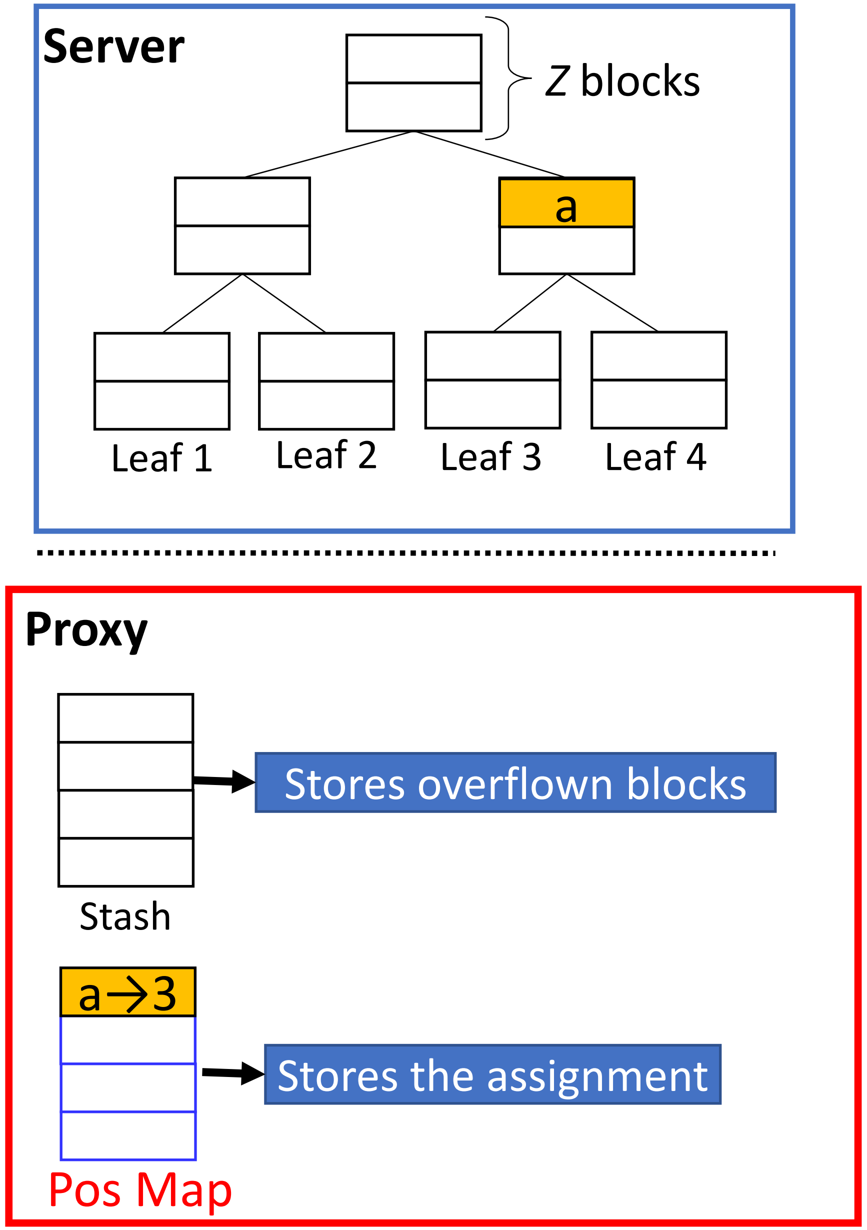


Storage is organized as a binary tree

Every access to a random path

Items randomly re-assigned after every access

Path ORAM [Stefanov et al. CCS'13]

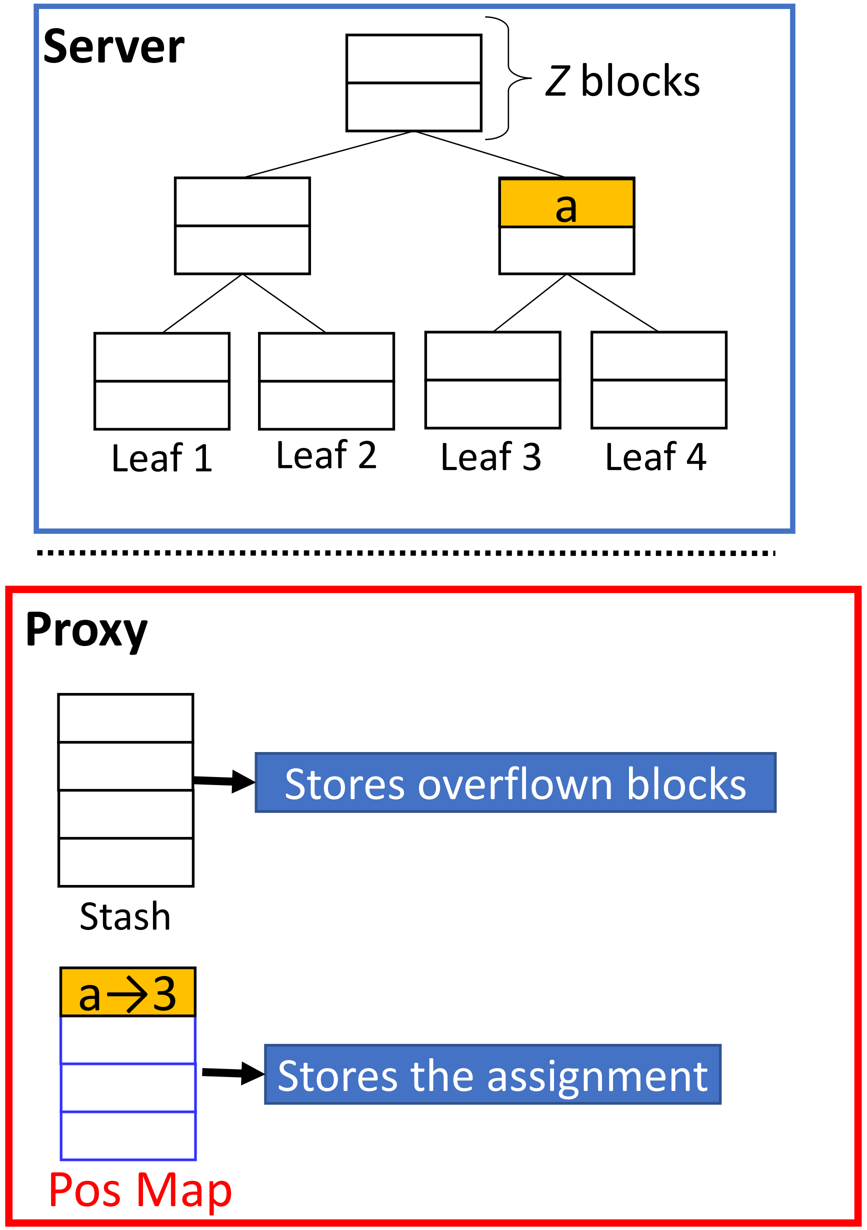


Storage is organized as a binary tree

Every access to a random path

Items randomly re-assigned after every access

Path ORAM [Stefanov et al. CCS'13]



Storage is organized as a binary tree

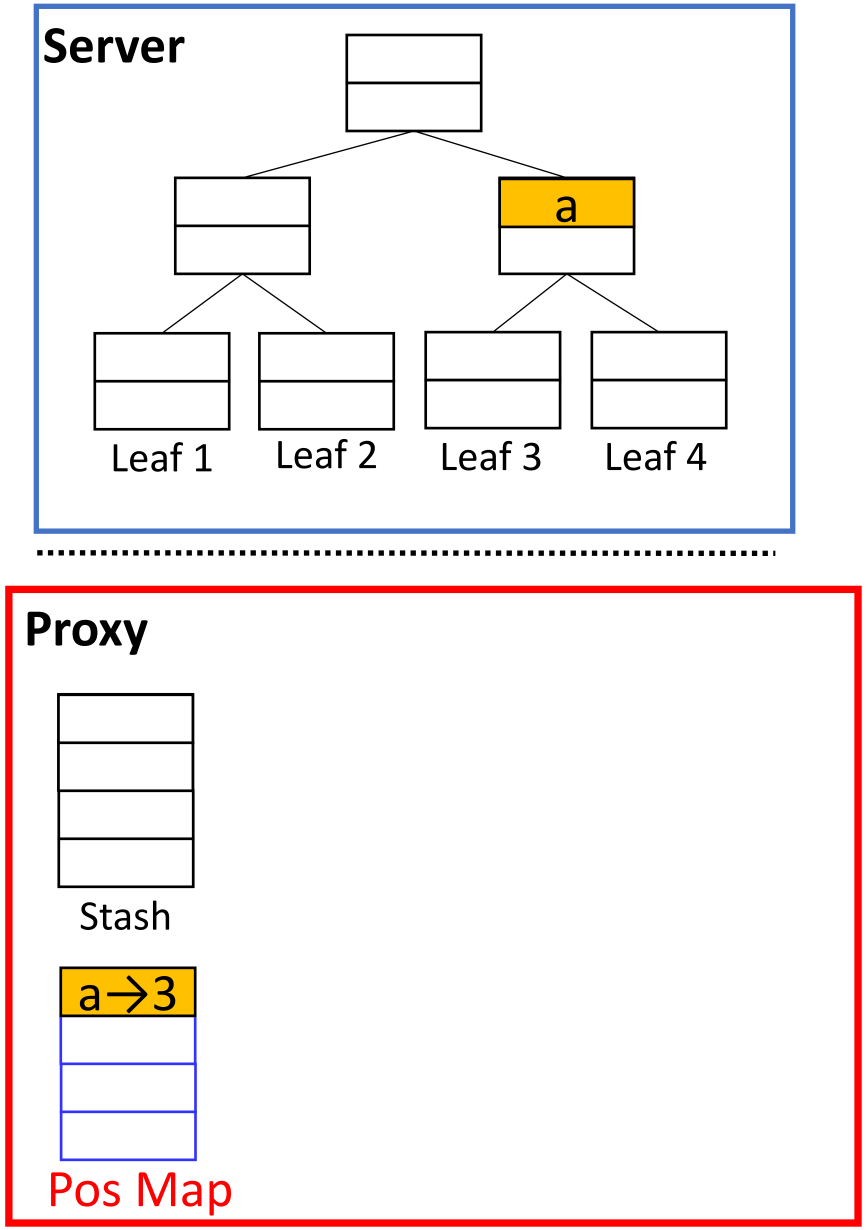
Every access to a random path

Items randomly re-assigned after every access

Possible to outsource position map recursively

But need many rounds of communication

Path ORAM

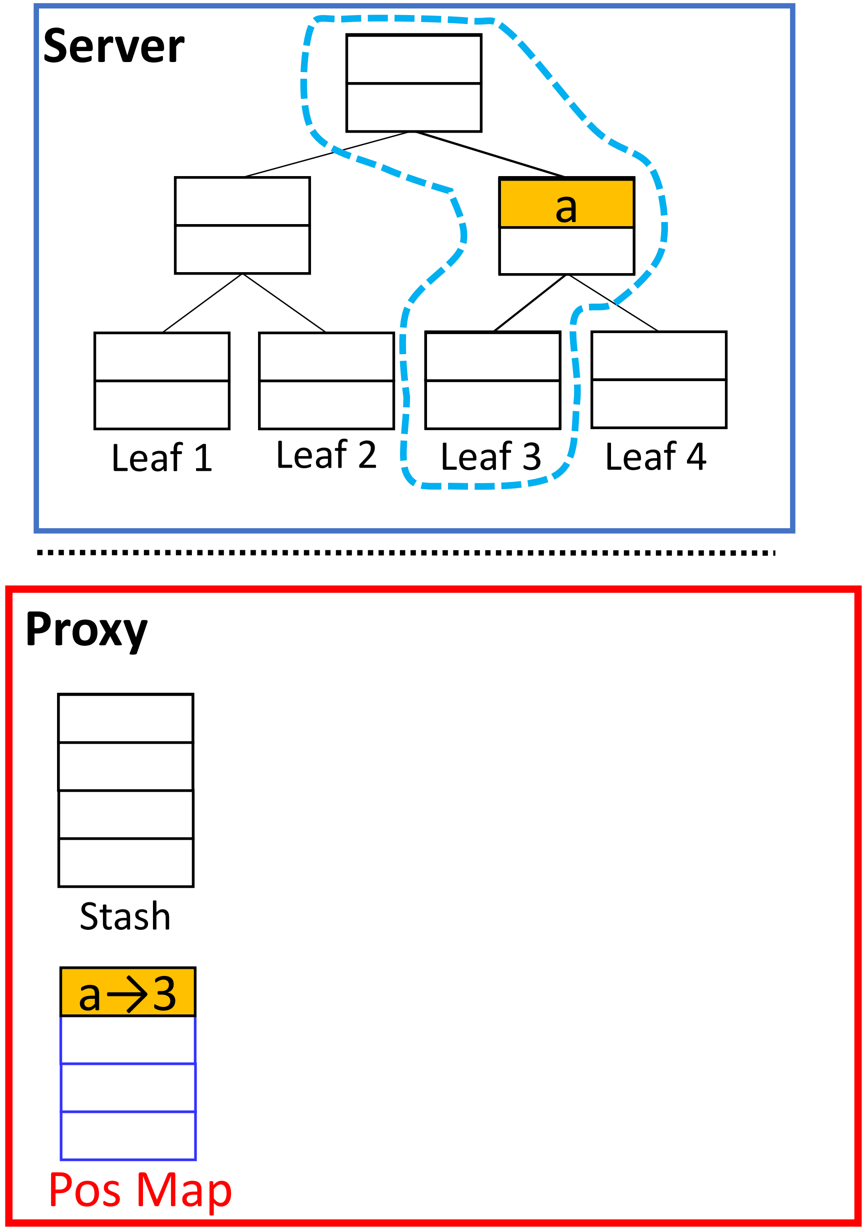


Read/Write block a

1) Read path

- Fetch associated path
- Read/Modify block
- Assign block to a new random path in position map
- Move all read blocks to *stash*

Path ORAM

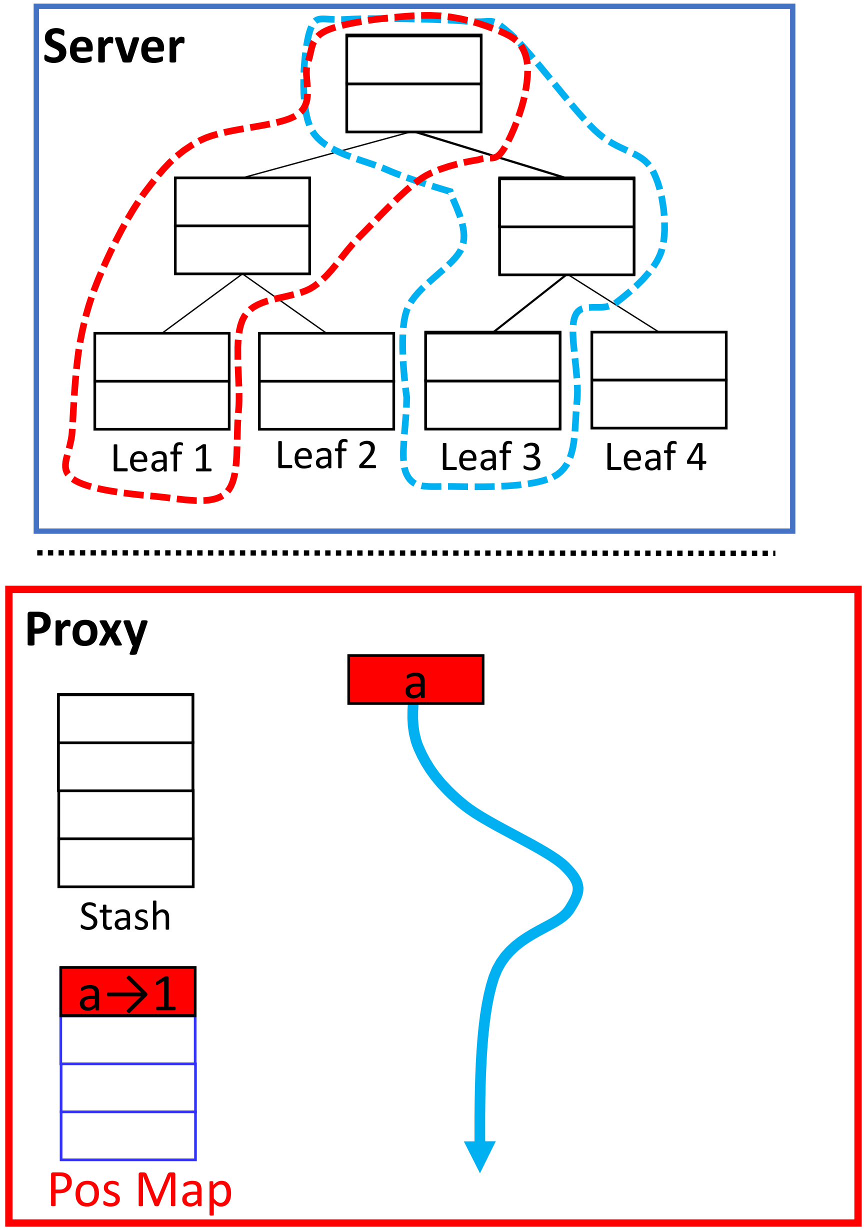


Read/Write block a

1) Read path

- Fetch associated path
- Read/Modify block
- Assign block to a new random path in position map
- Move all read blocks to *stash*

Path ORAM



Read/Write block a

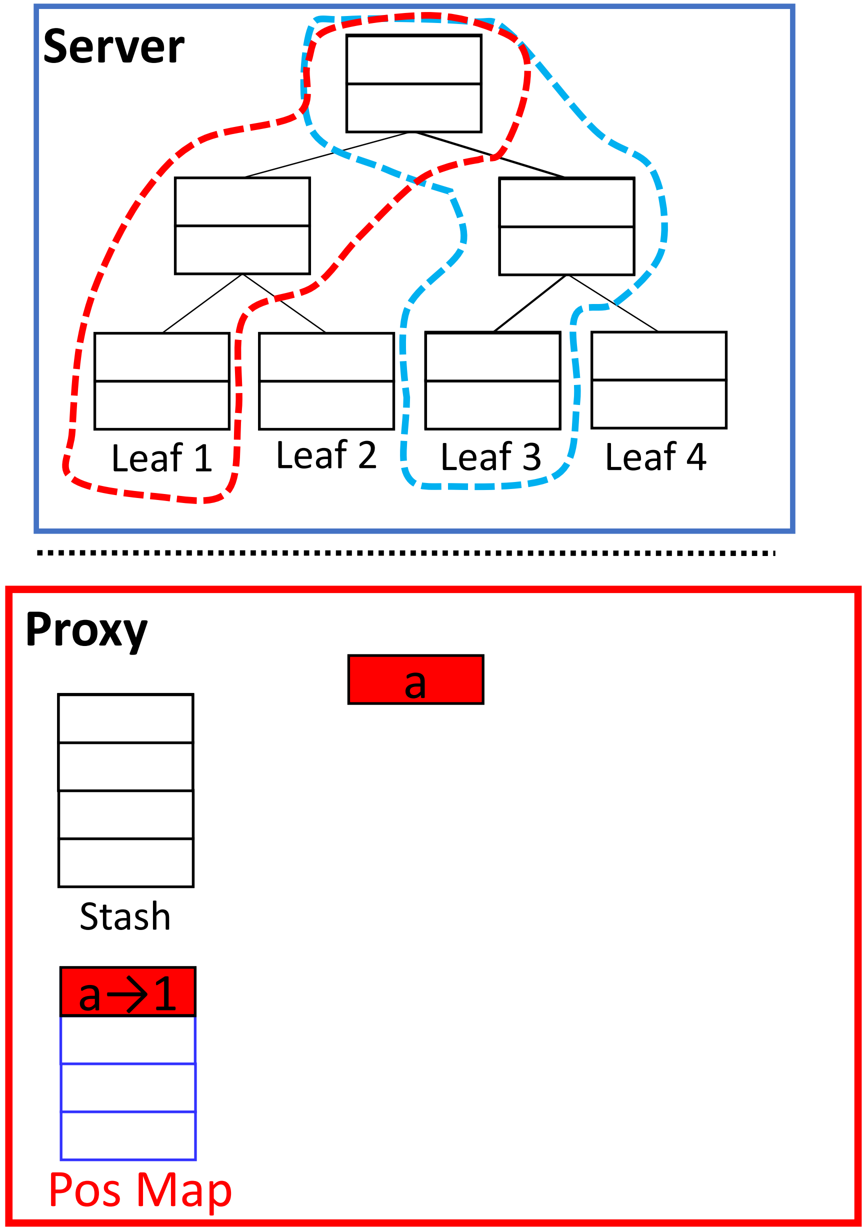
1) Read path

- Fetch associated path
- Read/Modify block
- Assign block to a new random path in position map
- Move all read blocks to *stash*

2) Flush

- Push every block to the lowest non-full node that intersects with its assigned path (otherwise→stash)

Path ORAM



Read/Write block a

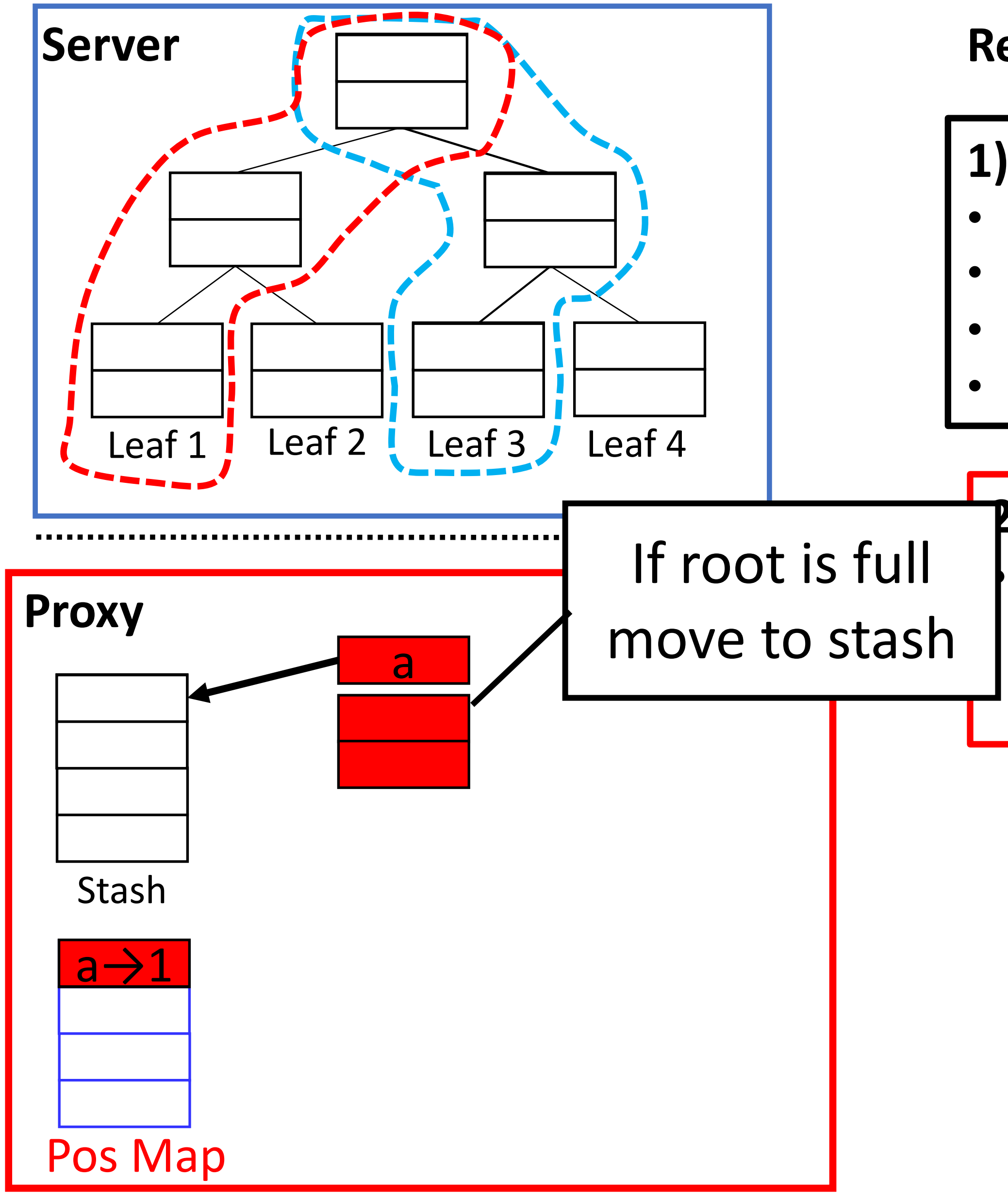
1) Read path

- Fetch associated path
- Read/Modify block
- Assign block to a new random path in position map
- Move all read blocks to *stash*

2) Flush

- Push every block to the lowest non-full node that intersects with its assigned path (otherwise→stash)

Path ORAM

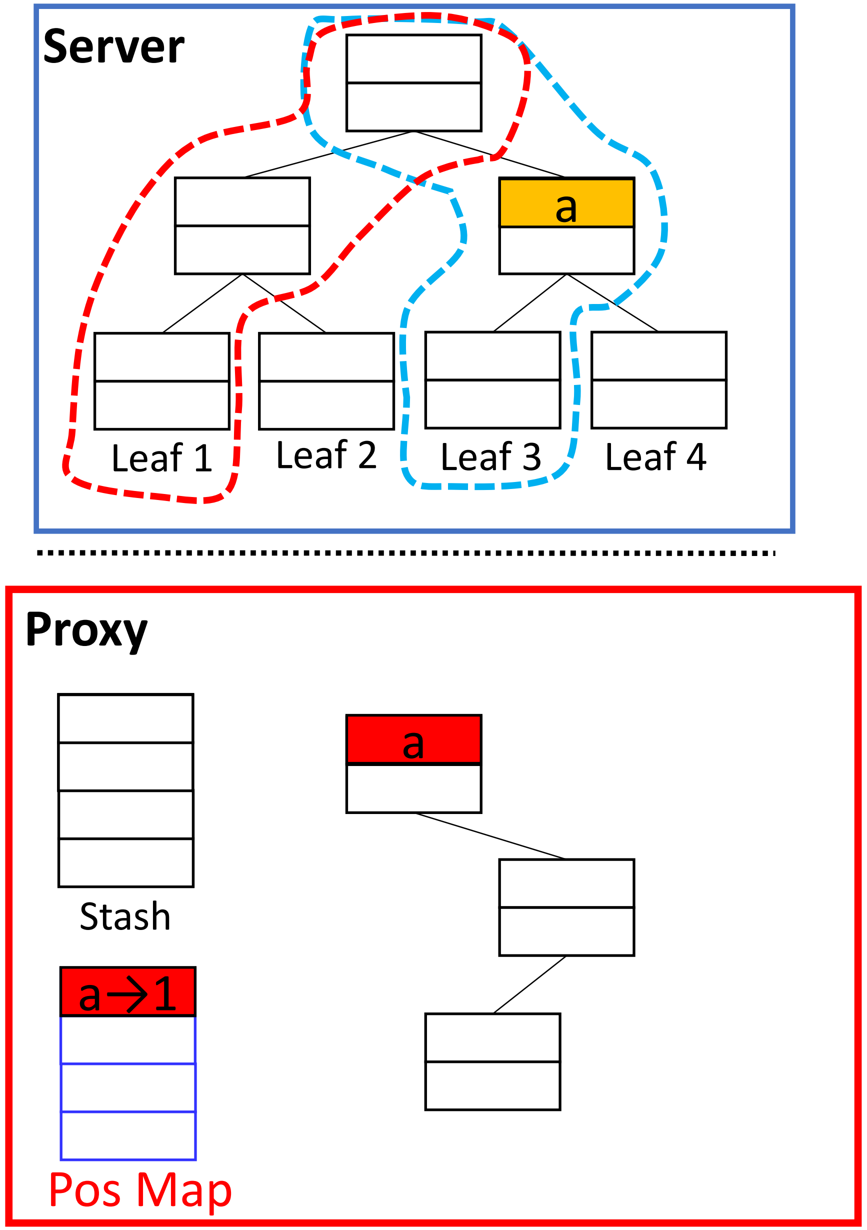


Read/Write block a

1) Read path
- Fetch associated path
- Read/Modify block
- Assign block to a new random path in position map
- Move all read blocks to *stash*

2) Flush
- Push every block to the lowest non-full node that intersects with its assigned path (otherwise→stash)

Path ORAM



Read/Write block a

1) Read path

- Fetch associated path
- Read/Modify block
- Assign block to a new random path in position map
- Move all read blocks to *stash*

2) Flush

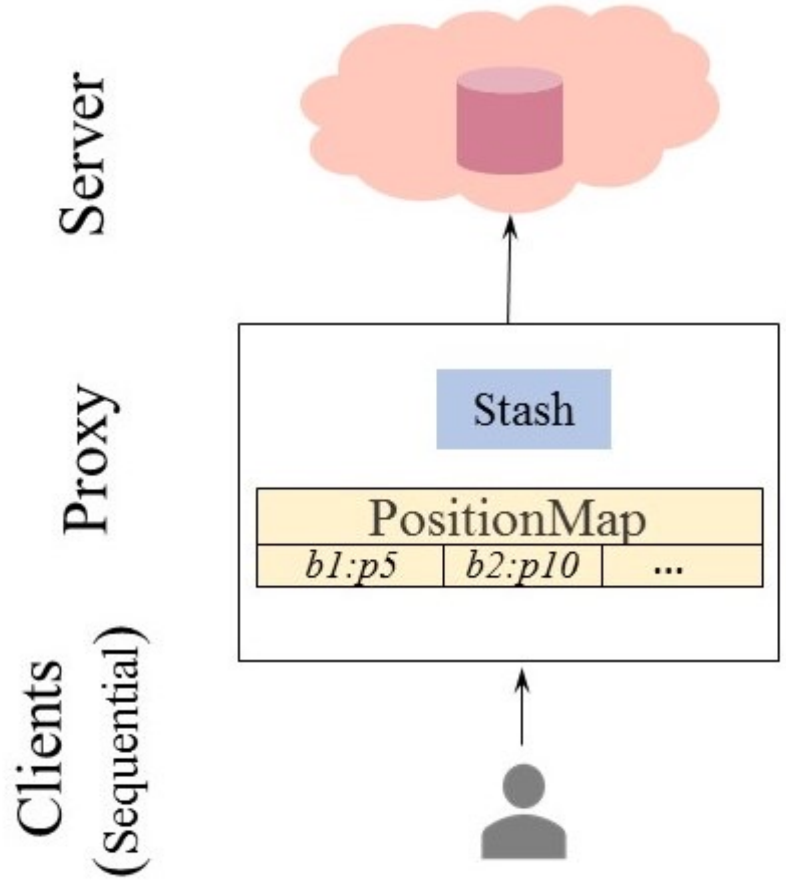
- Push every block to the lowest non-full node that intersects with its assigned path (otherwise→stash)

3) Write-back

- Re-encrypt w/ fresh randomness

PathORAM

- Steps to access block *B*:
  1. Fetch path *P* containing block *B* from Server
  2. Update requested block *B* (if write)
  3. Answer Client Request
  4. Assign block *B* to random path
  5. Flush path *P*
  6. Writeback to server



Does PathORAM provide workload independence?

Say a client requested block *b* stored in path *p*. From an adversary's perspective

- Which data is accessed? → One of the Z**logN* objects accessed
- When was *b* last accessed? → Only knows when *p* was last accessed, not when *b* was last accessed
- Did 2 subsequent requests access *b*? → Only knows two random paths *p* and *p'* being accessed in subsequent requests
- Access pattern (uniform or skewed)? → Observes accesses to random paths
- Is *b* read or written? → Each path is read and then written with fresh encryption

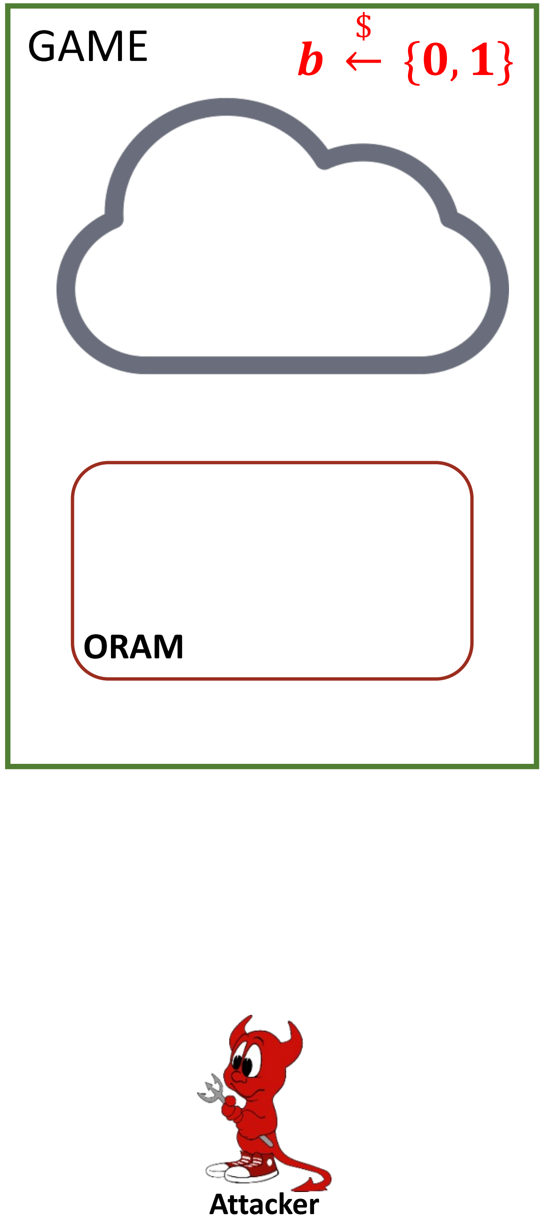
Yes! PathORAM provides workload independence!

ORAM – Security

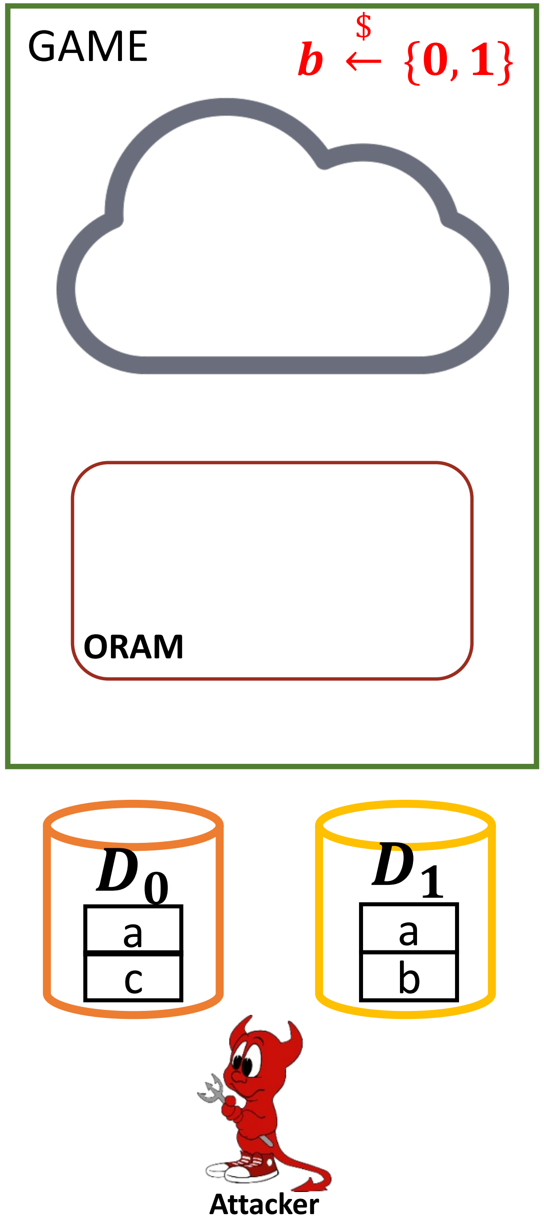
- Let $A = \{(\mathrm{op}_1, \mathrm{bid}_1, \mathrm{val}_1), \ \ldots (\mathrm{opm}, \mathrm{bidm}, \mathrm{valm})\}$ represent a sequence of accesses $op_i \ \in \{read, write\}$, bid_i is the block identifier, and val_i is either updated value writes or null for reads

- An ORAM scheme is secure if given two such sequences A_0 *and* A_1 and the system executed A_i, the adversary cannot guess which sequence was executed with probability >> 1/2

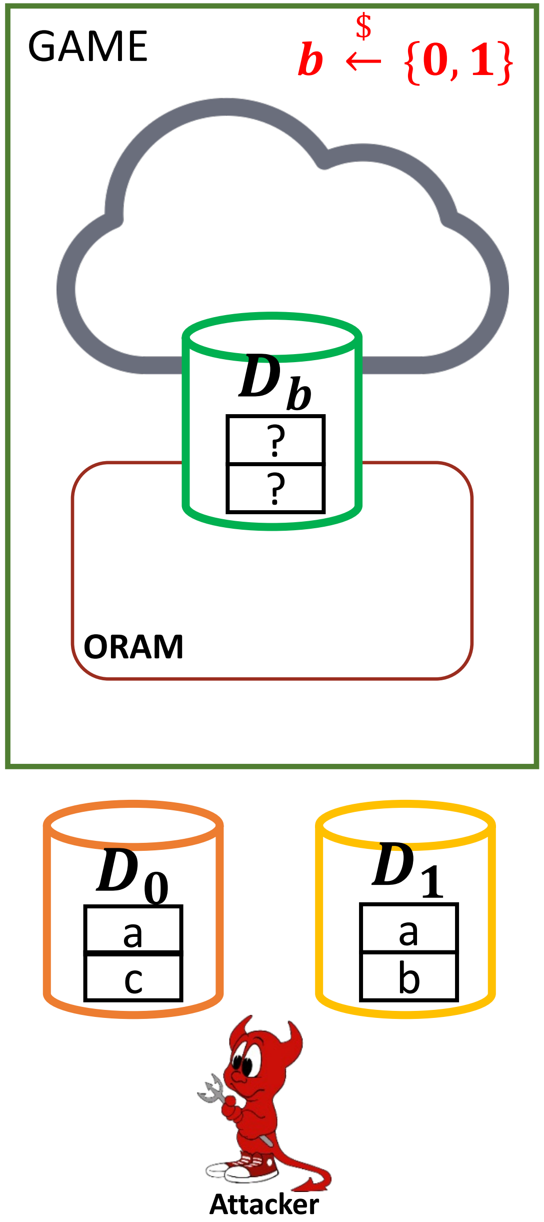
ORAM - Security



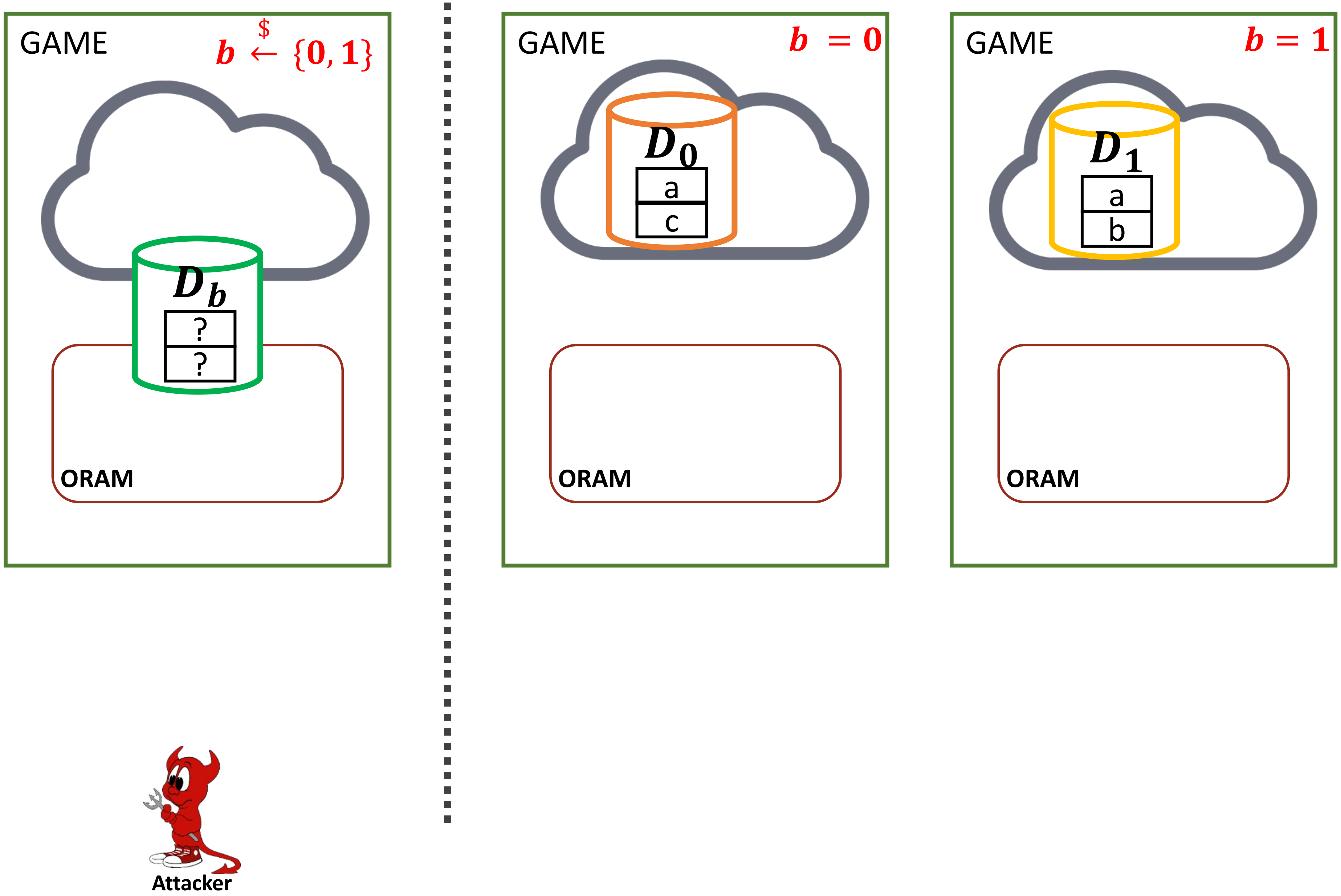
ORAM - Security



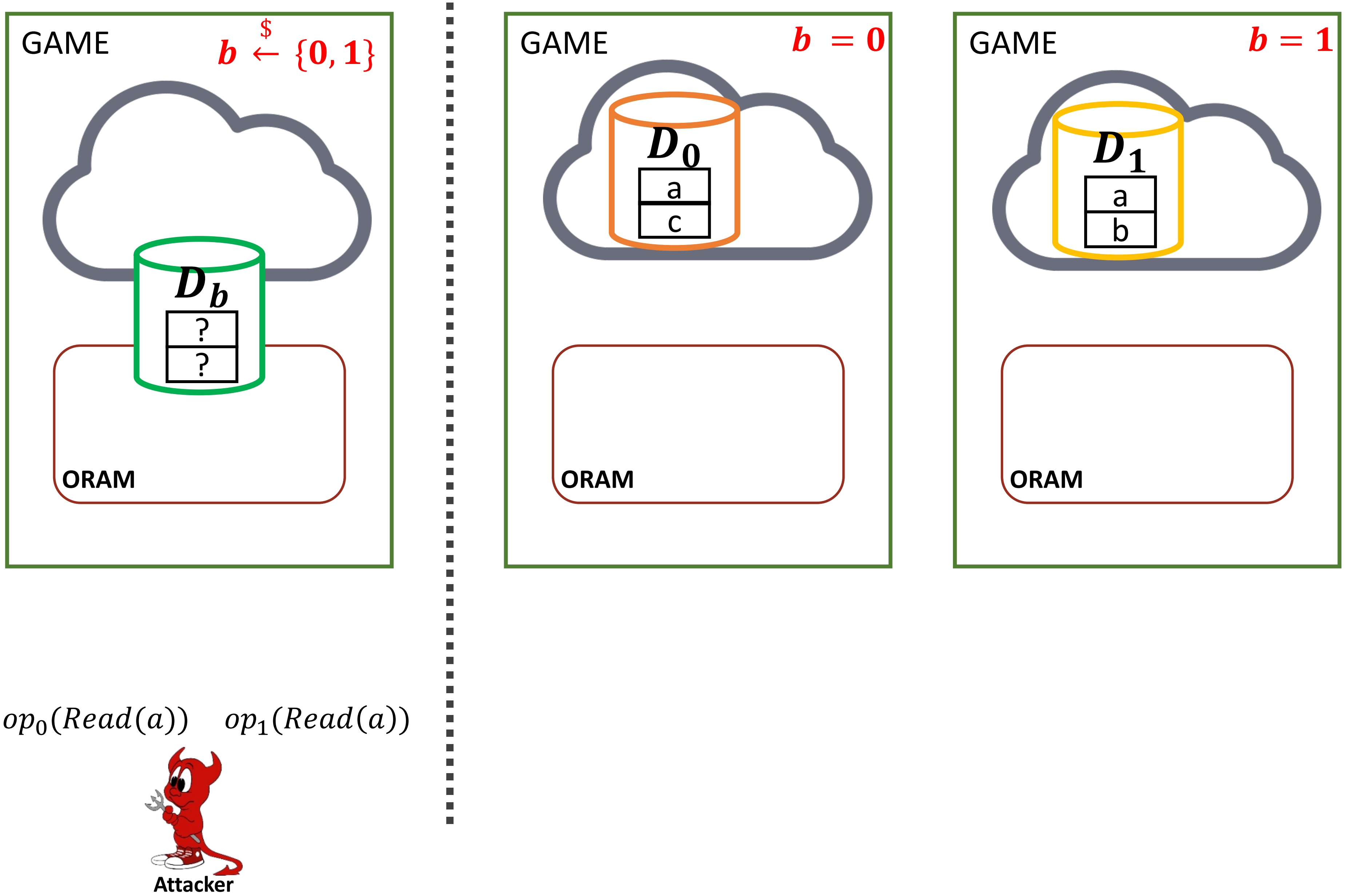
ORAM - Security



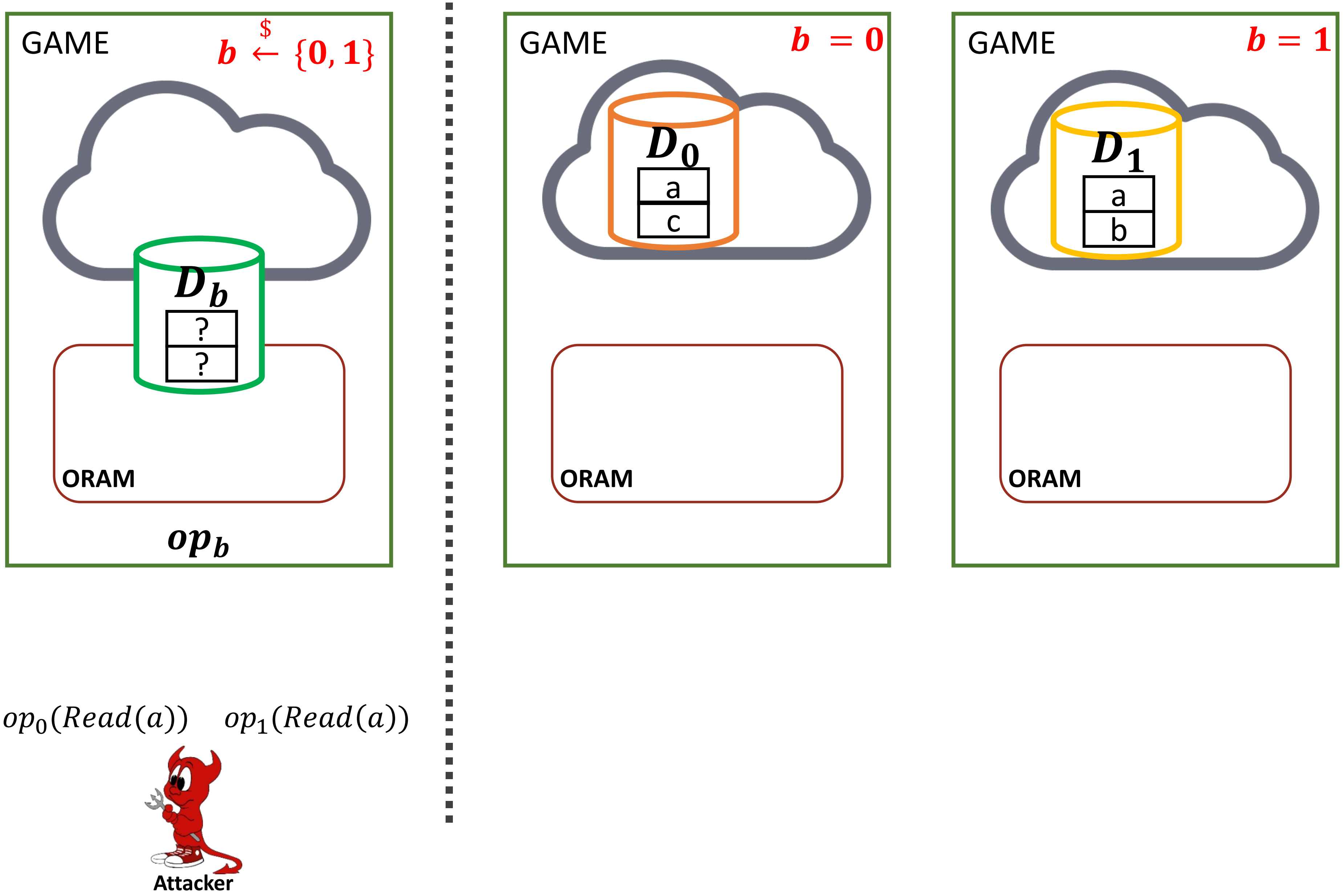
ORAM - Security



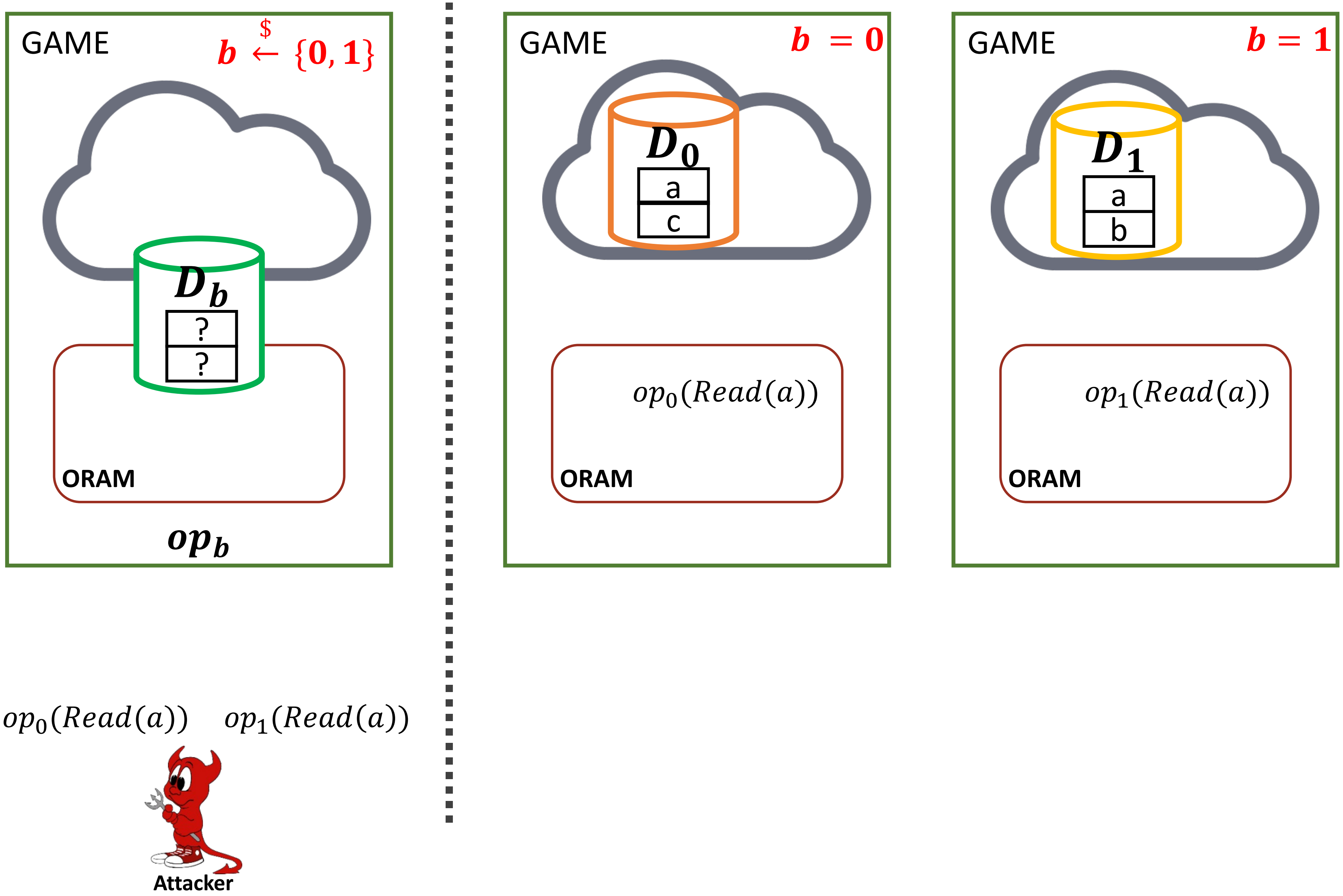
ORAM - Security



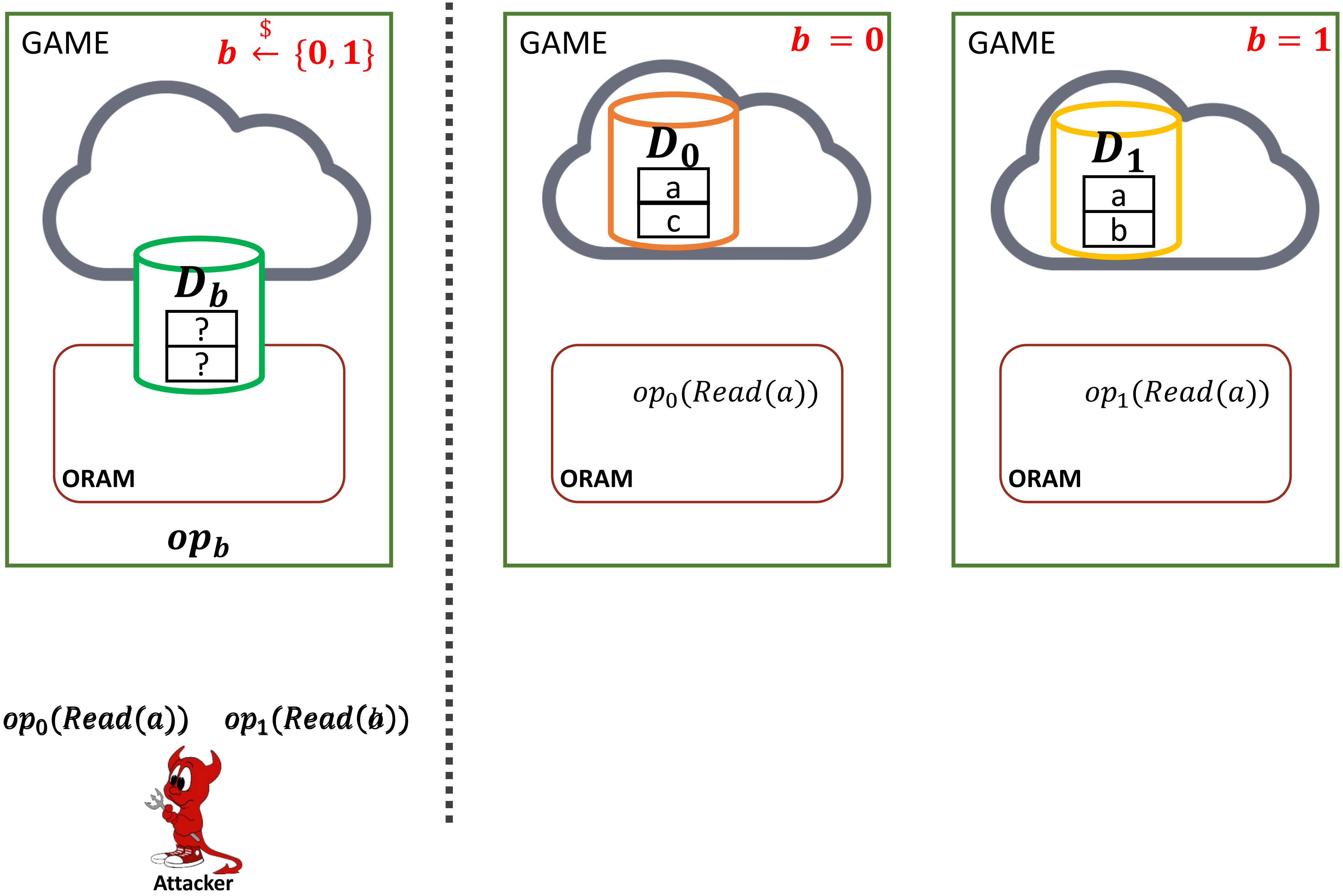
ORAM - Security



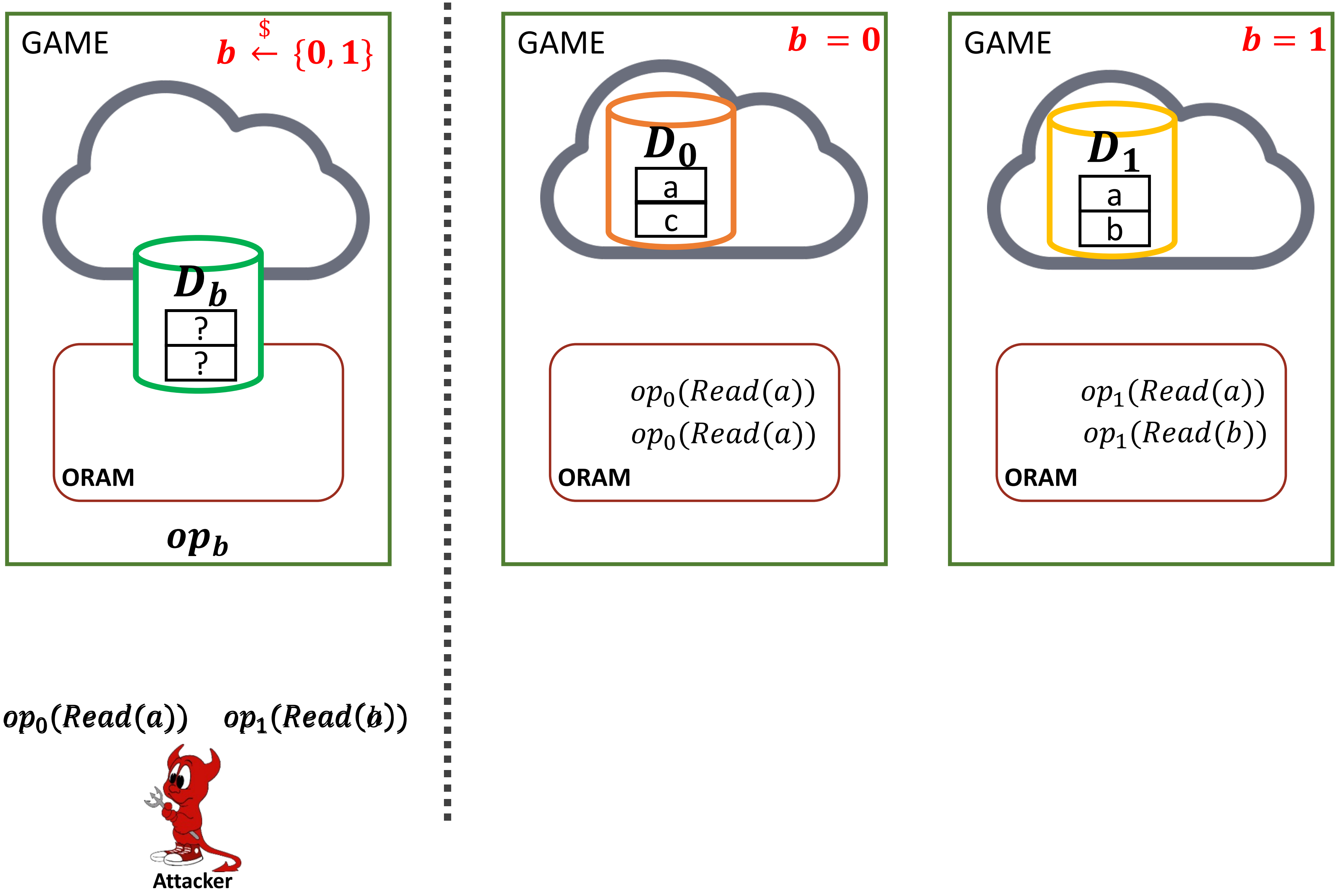
ORAM - Security



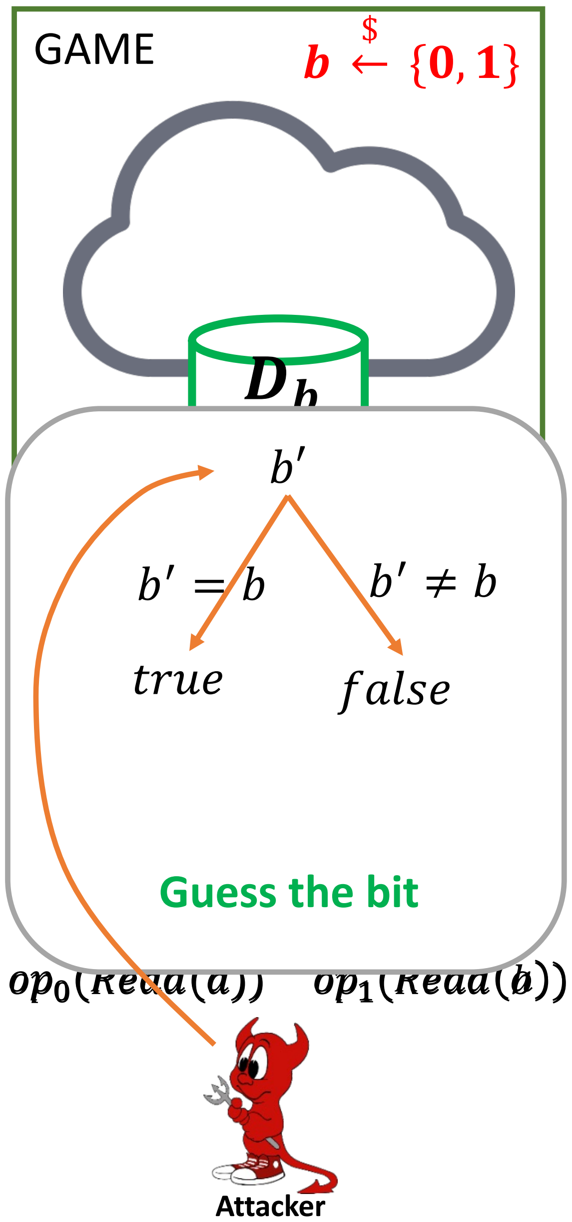
ORAM - Security

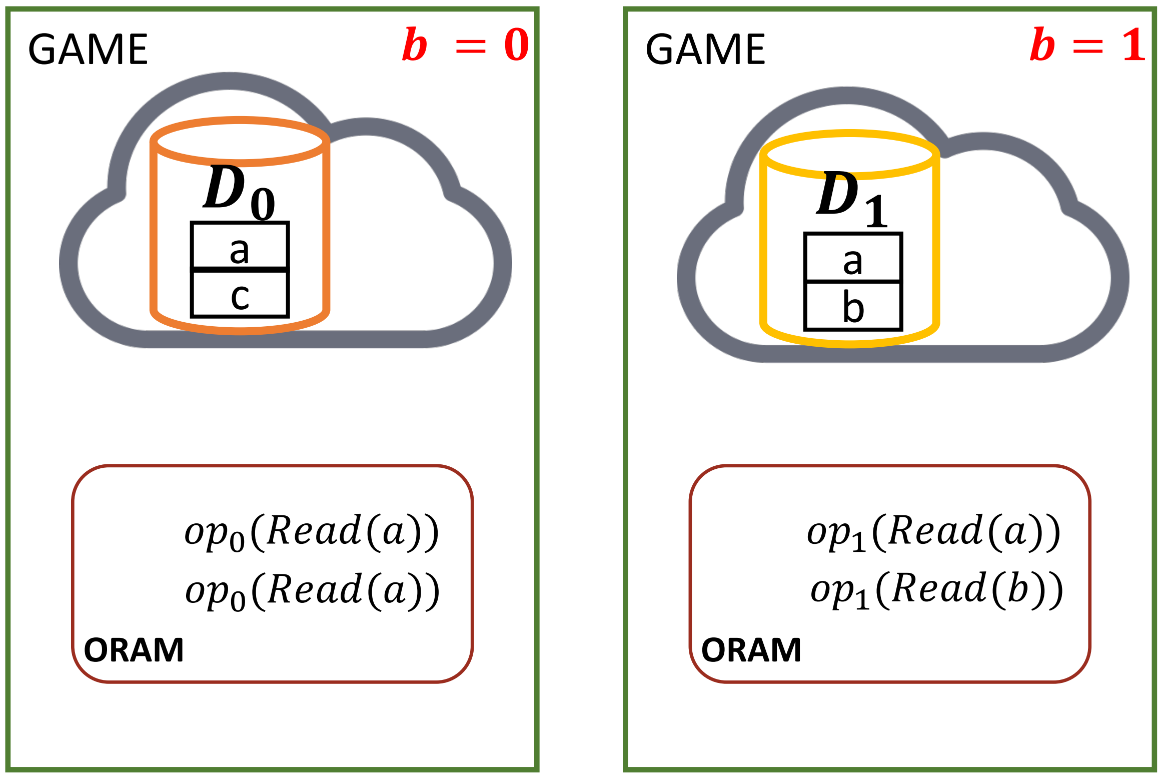


ORAM - Security



ORAM - Security





A secure ORAM has

$$\Pr(A \diamond G \rightarrow true) = \frac{1}{2} + negl$$

i.e., adversary has negligible advantage is guessing bit *b*

Two observations on PathORAM

- Bandwidth overhead: ***2*Z*logN*** → Depends on *Z*
- The *online* rounds of communication b/w client and server: **2 rounds**
  - Even for read reqs, need an online write step
- Can these two limitations be improved?

RingORAM [Ren et al. Usenix Security'15]

Goals:

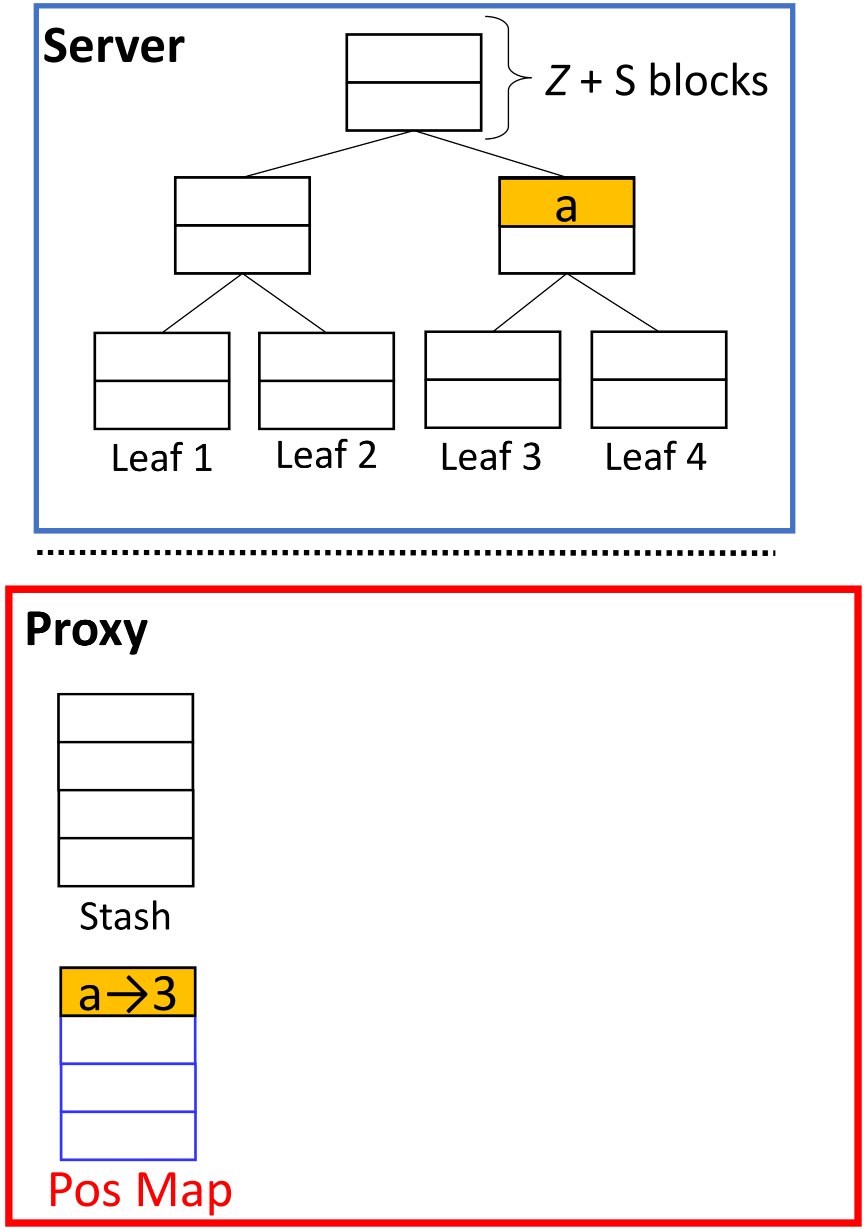
1. Eliminate the ORAM bandwidth's dependence on *Z*

   How?
   Read exactly one block per bucket along the path

2. Reduce online communication rounds to 1

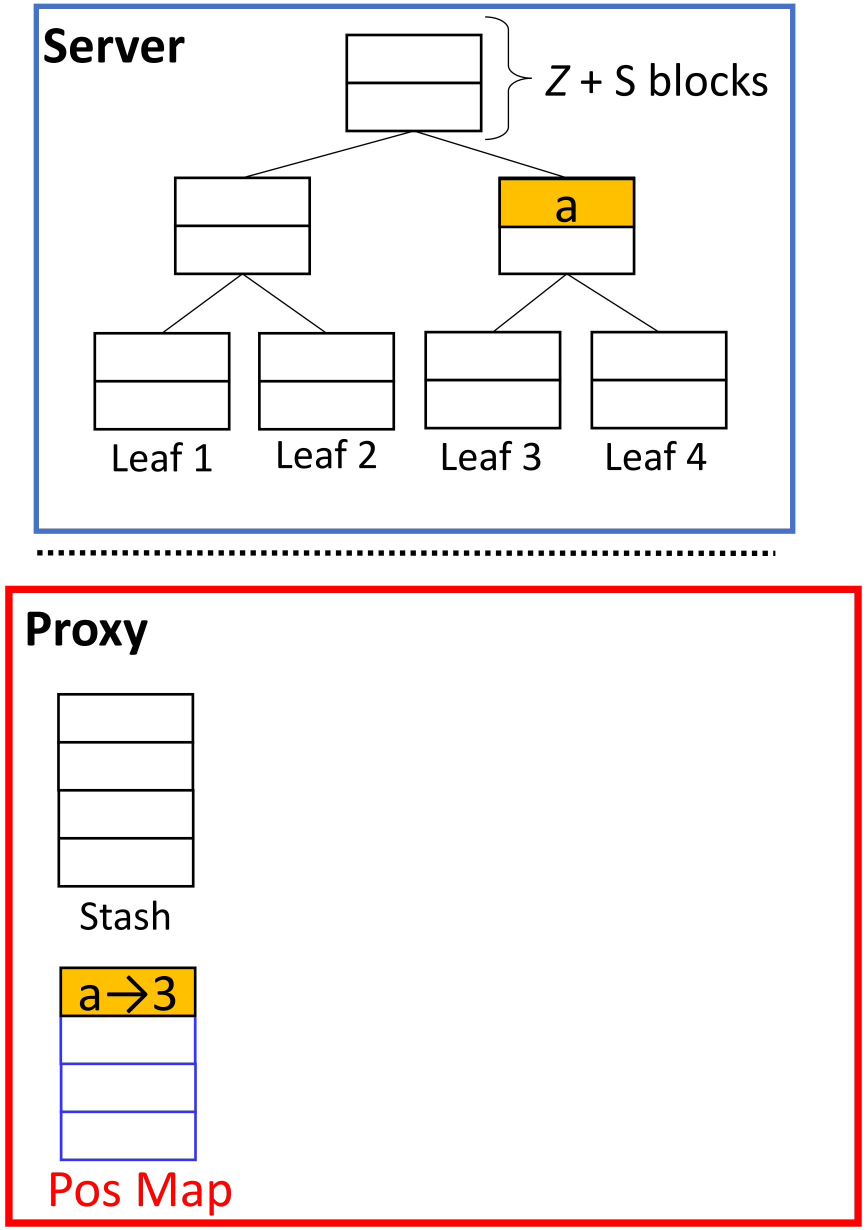
   How?
   Only read path for each client request, buffer writes, and write path back in an offline step

Ring ORAM



Each bucket stores at most *Z* real blocks and at least *S* dummy blocks

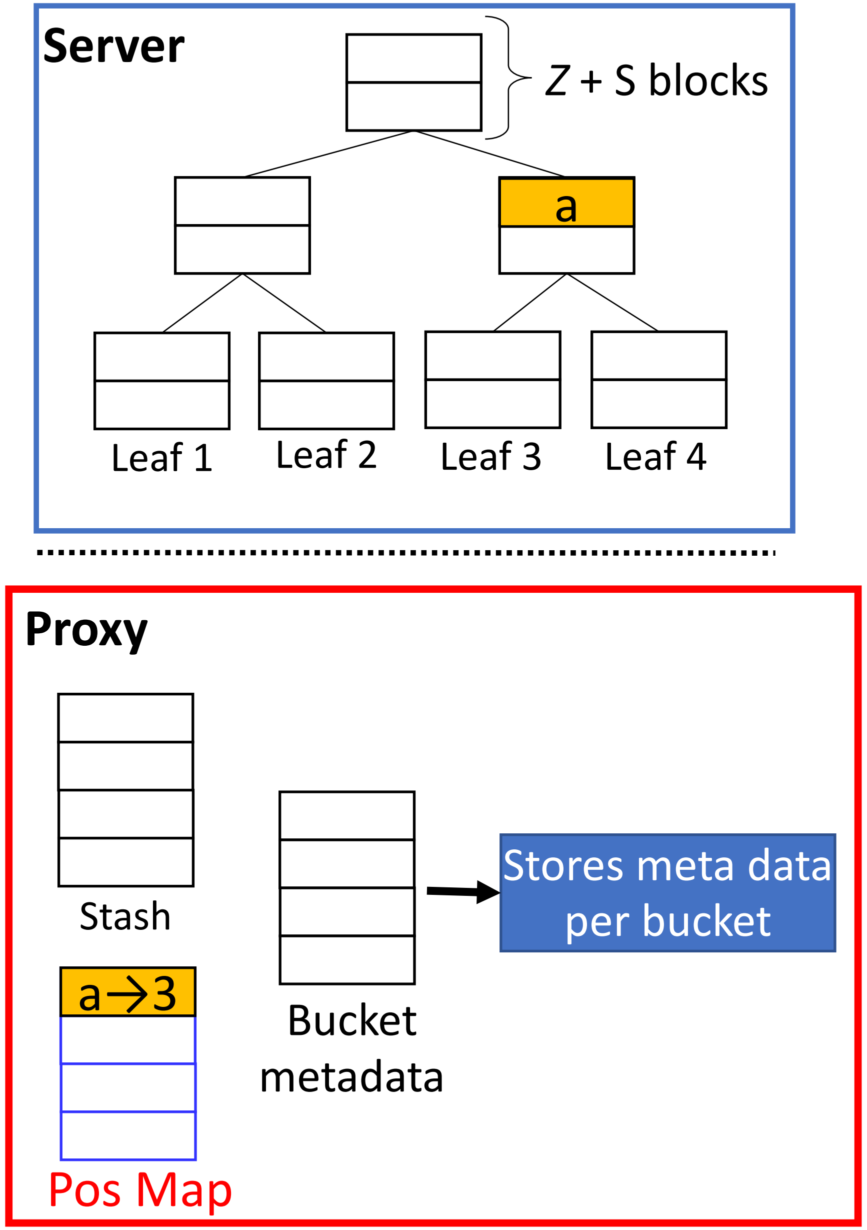
Ring ORAM



Each bucket stores at most *Z* real blocks and at least *S* dummy blocks

Every access to a random path reads only one block per bucket

Ring ORAM



Each bucket stores at most *Z* real blocks and at least *S* dummy blocks

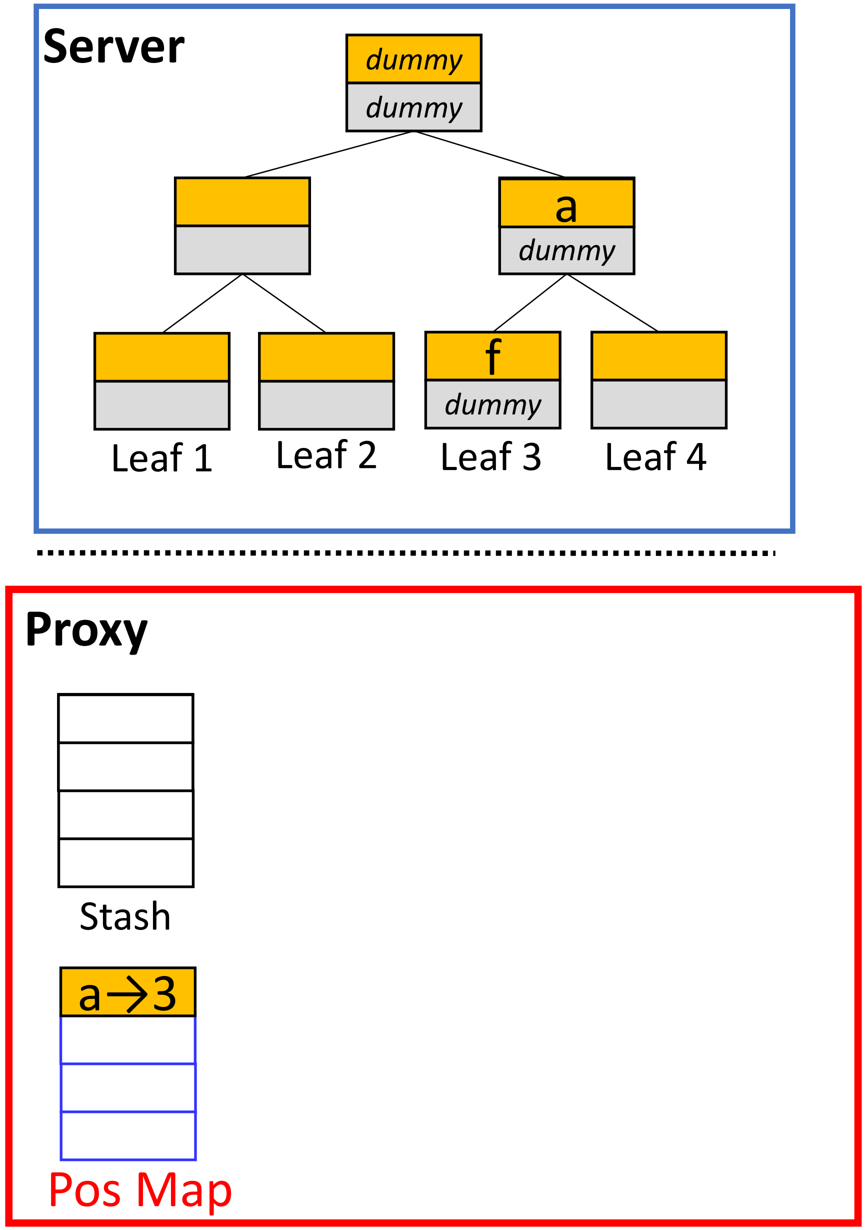
Every access to a random path reads only one block per bucket

Bucket metadata stores info on

1. *count:* how many times is this bucket accessed
2. *valid:* which of the *Z+S* blocks are not yet accessed
3. *addr:* ids of real blocks in a bucket

Note: Bucket metadata actually stored at server

Ring ORAM



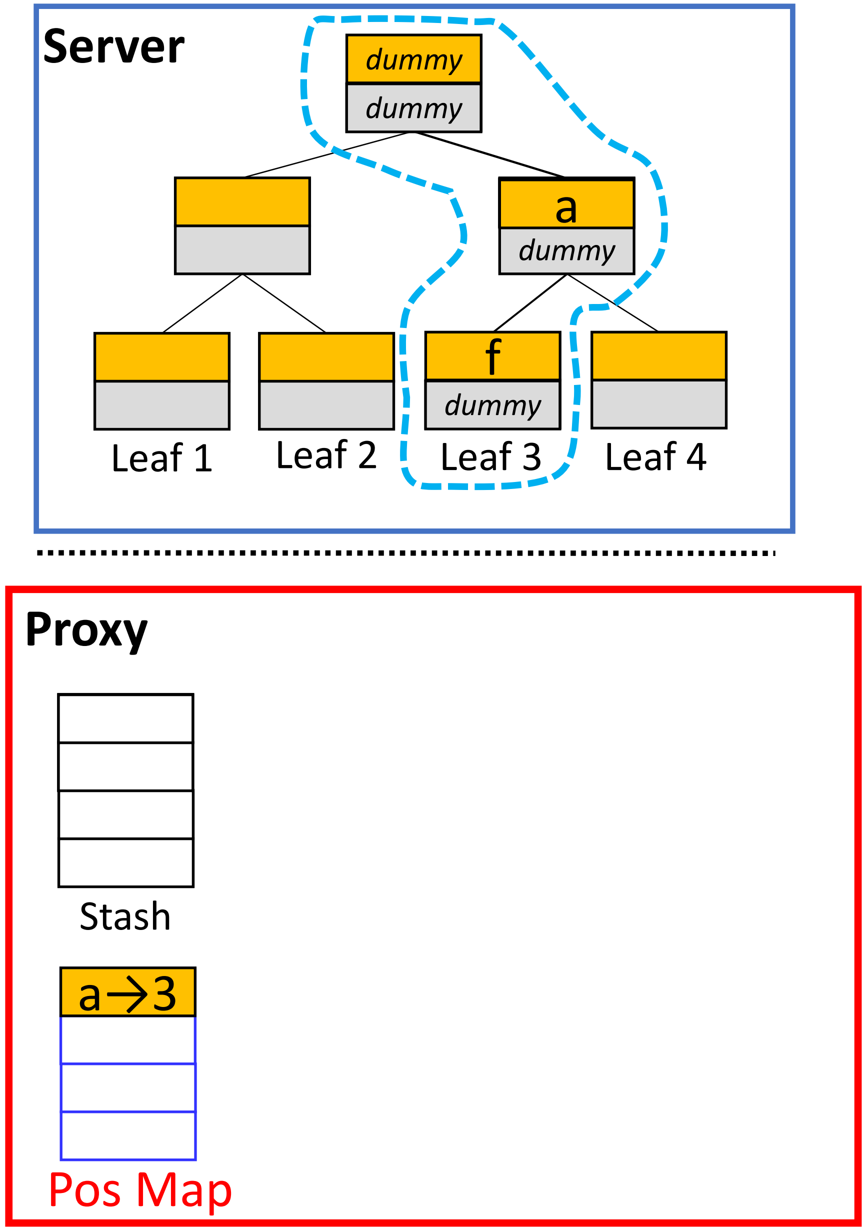
1) Read path

For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*

Assign block to a new random path in position map

Ring ORAM



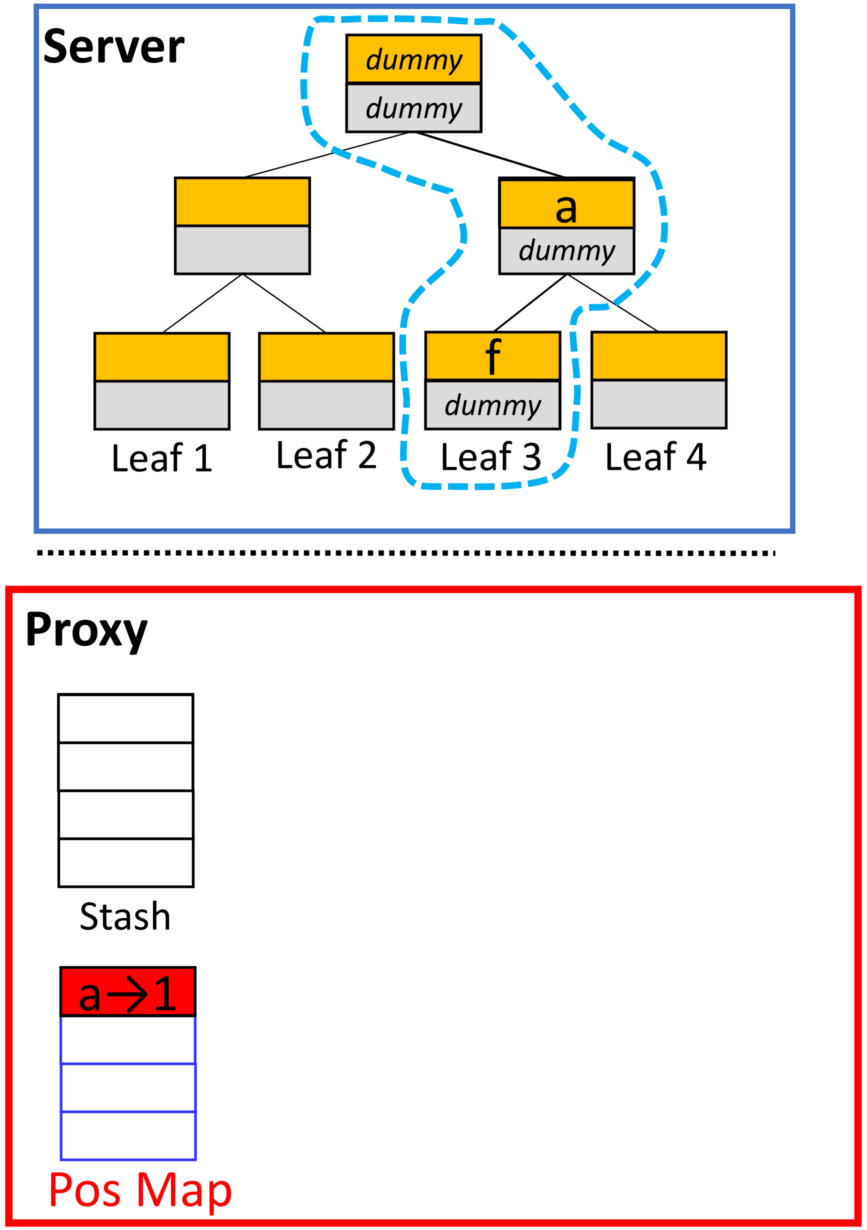
1) Read path

For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*

Assign block to a new random path in position map

Ring ORAM



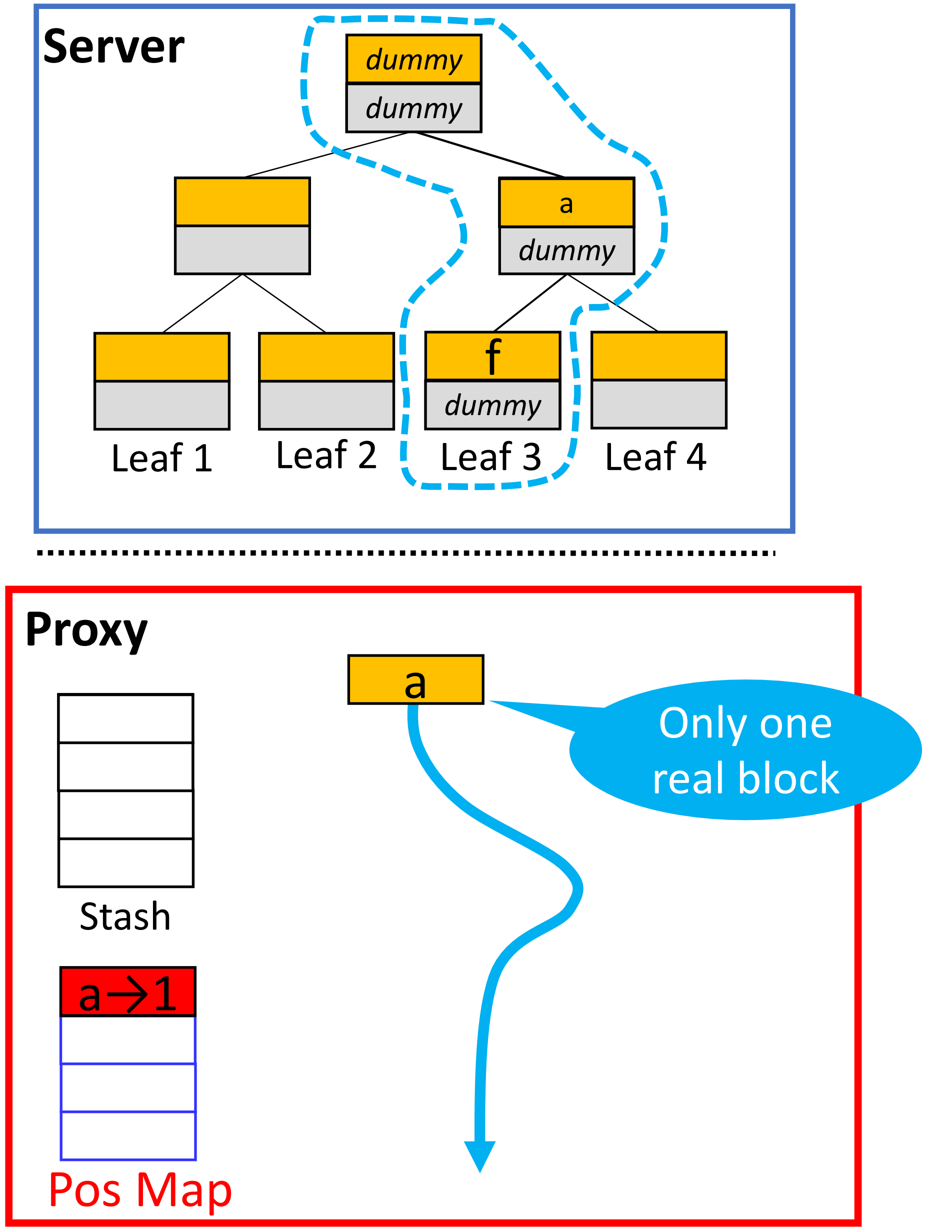
1) Read path

For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*

Assign block to a new random path in position map

Ring ORAM



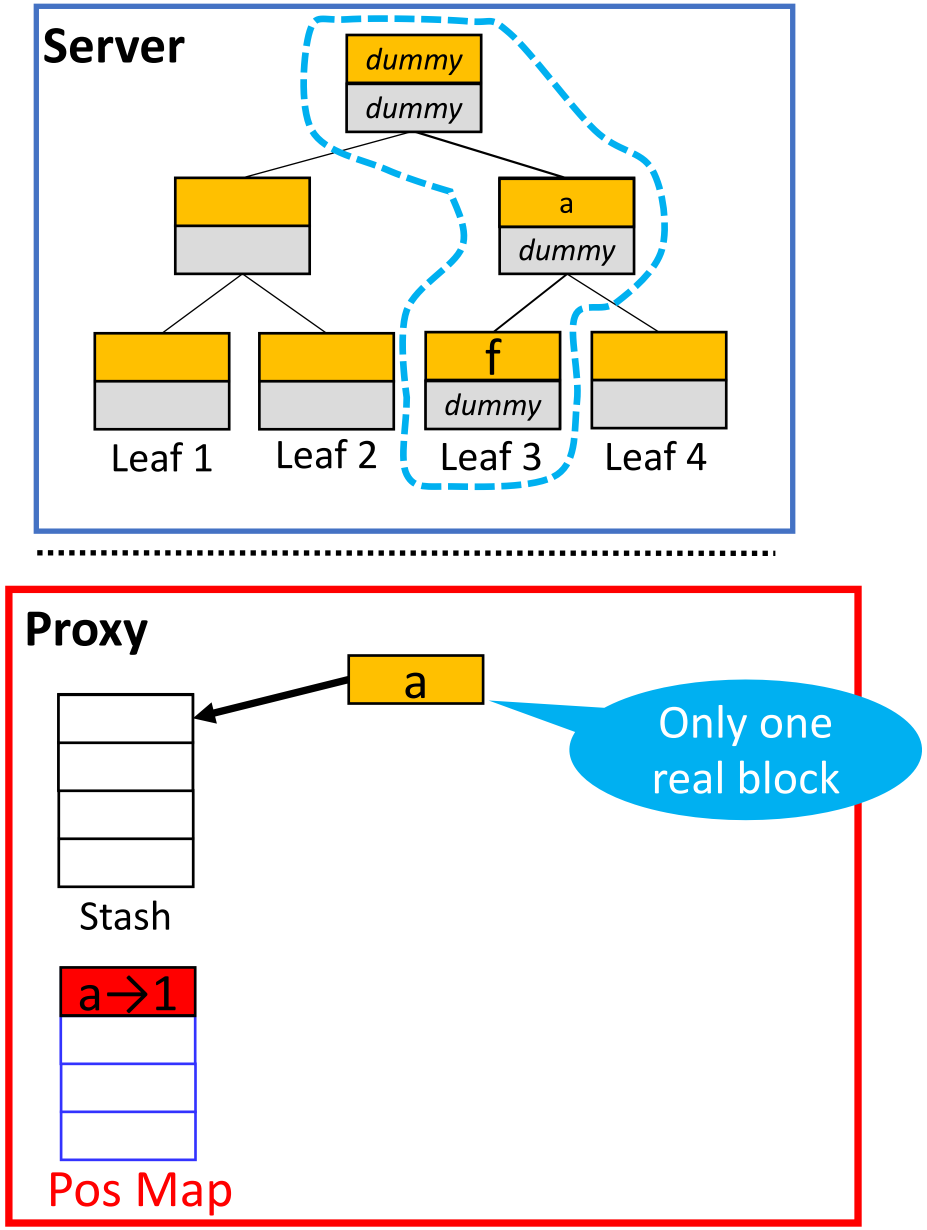
1) Read path

For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*

Assign block to a new random path in position map

Ring ORAM



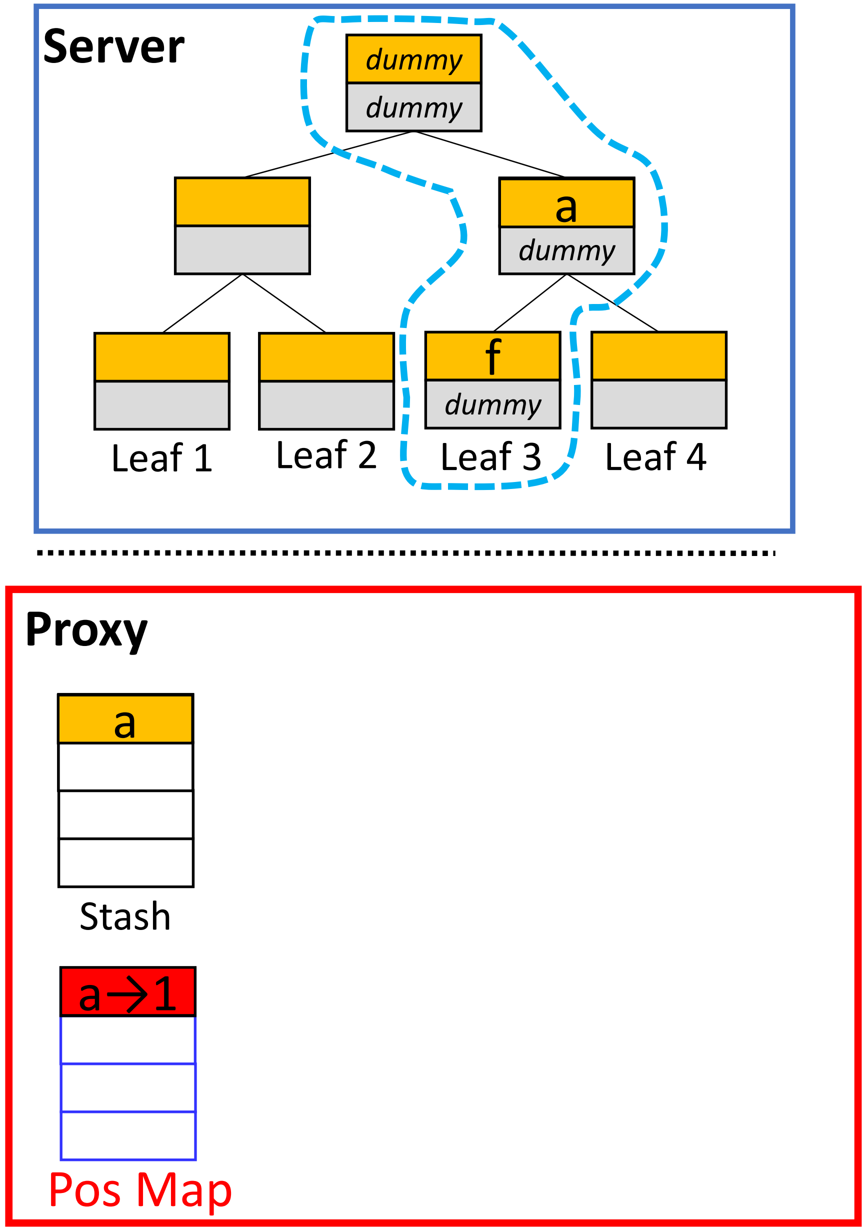
1) Read path

For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*

Assign block to a new random path in position map

Ring ORAM



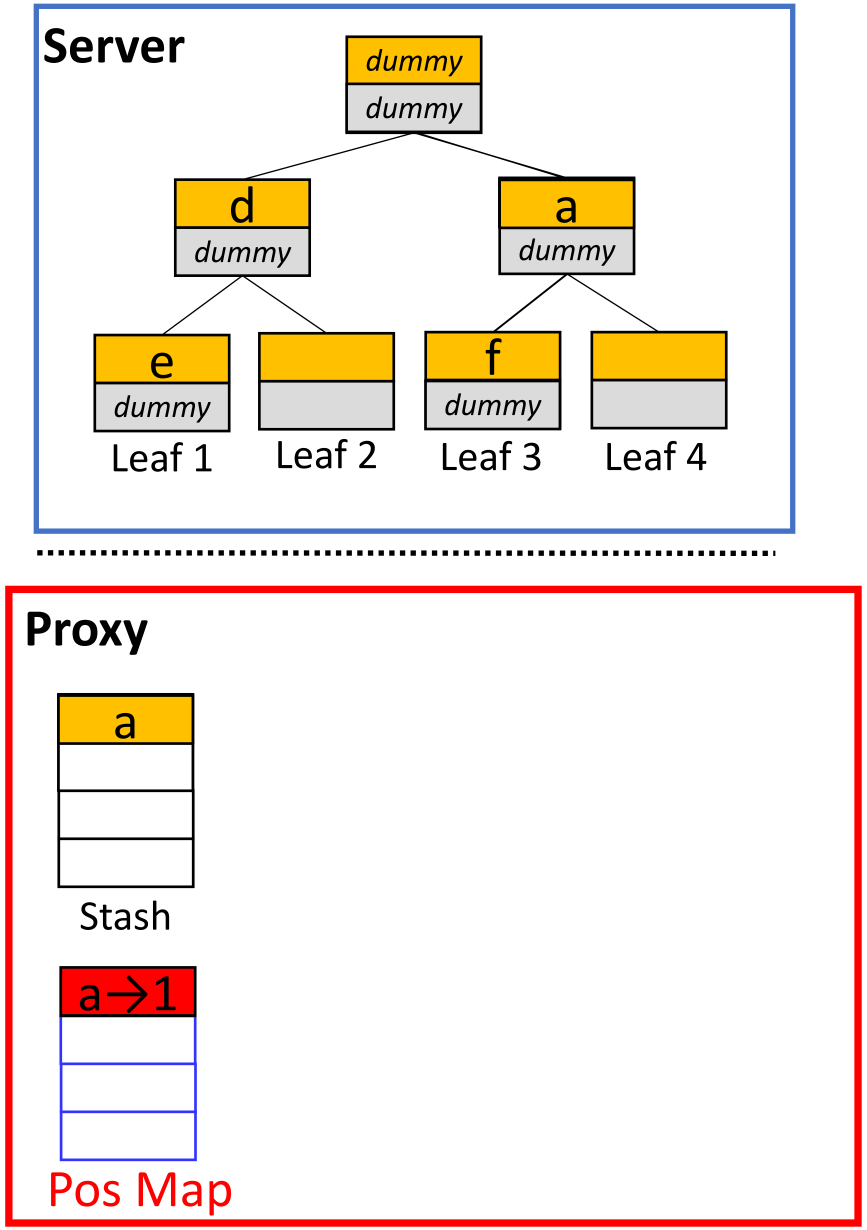
1) Read path

For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*

Assign block to a new random path in position map

Ring ORAM

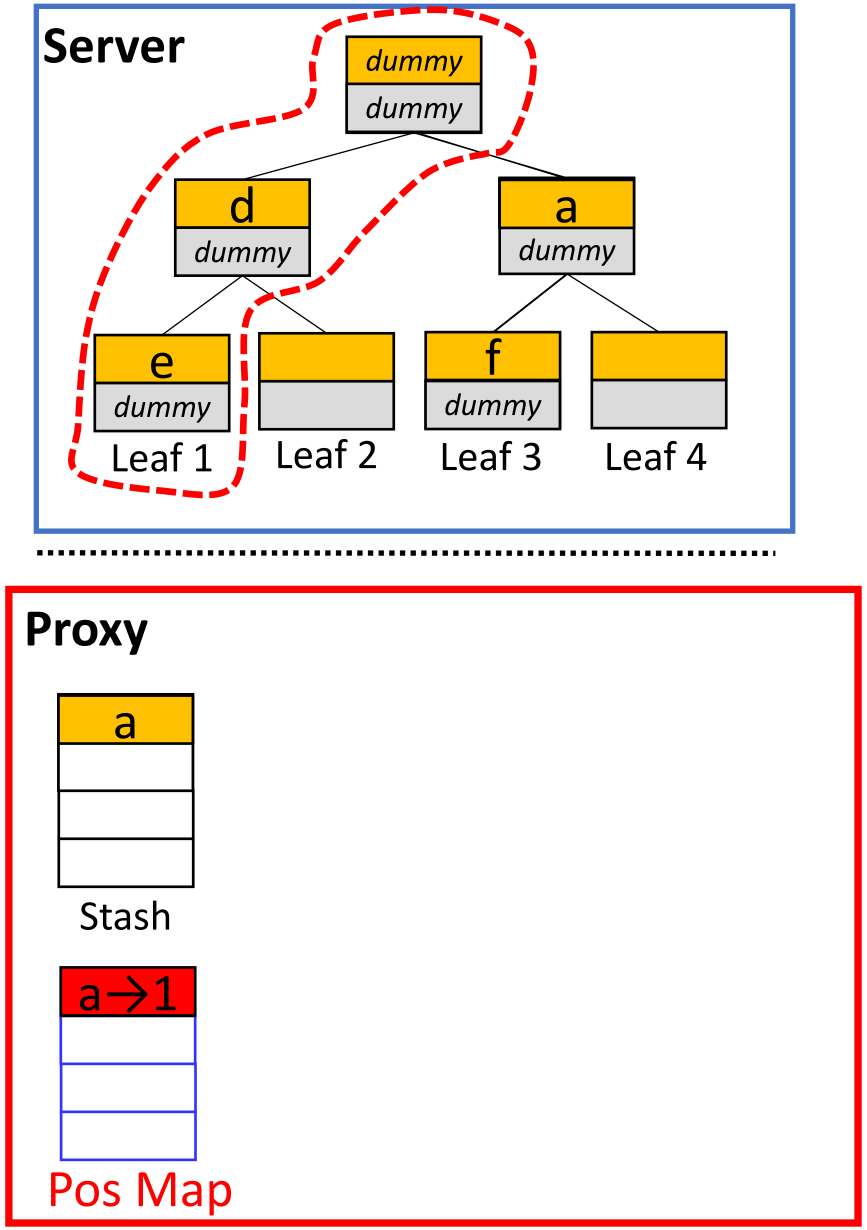


1) Read path

For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*
- Assign block to a new random path in position map

Ring ORAM



1) Read path

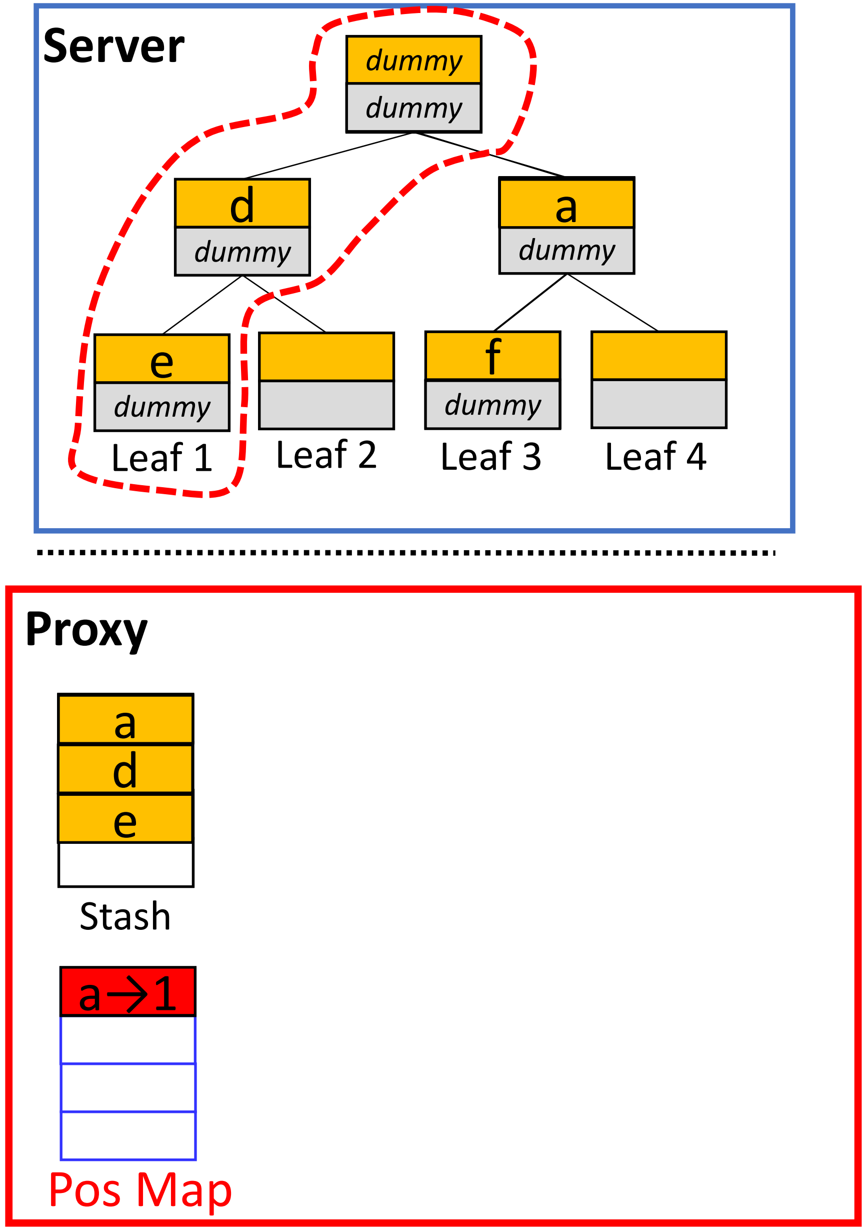
For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*
- Assign block to a new random path in position map

2) Evict

- After *A* read paths, in a deterministic order pick the next path to evict
- For each bucket, read all remaining valid real blocks (if < Z, read dummy) to *stash*
- Write each bucket from *stash* and reset all metadata

Ring ORAM



1) Read path

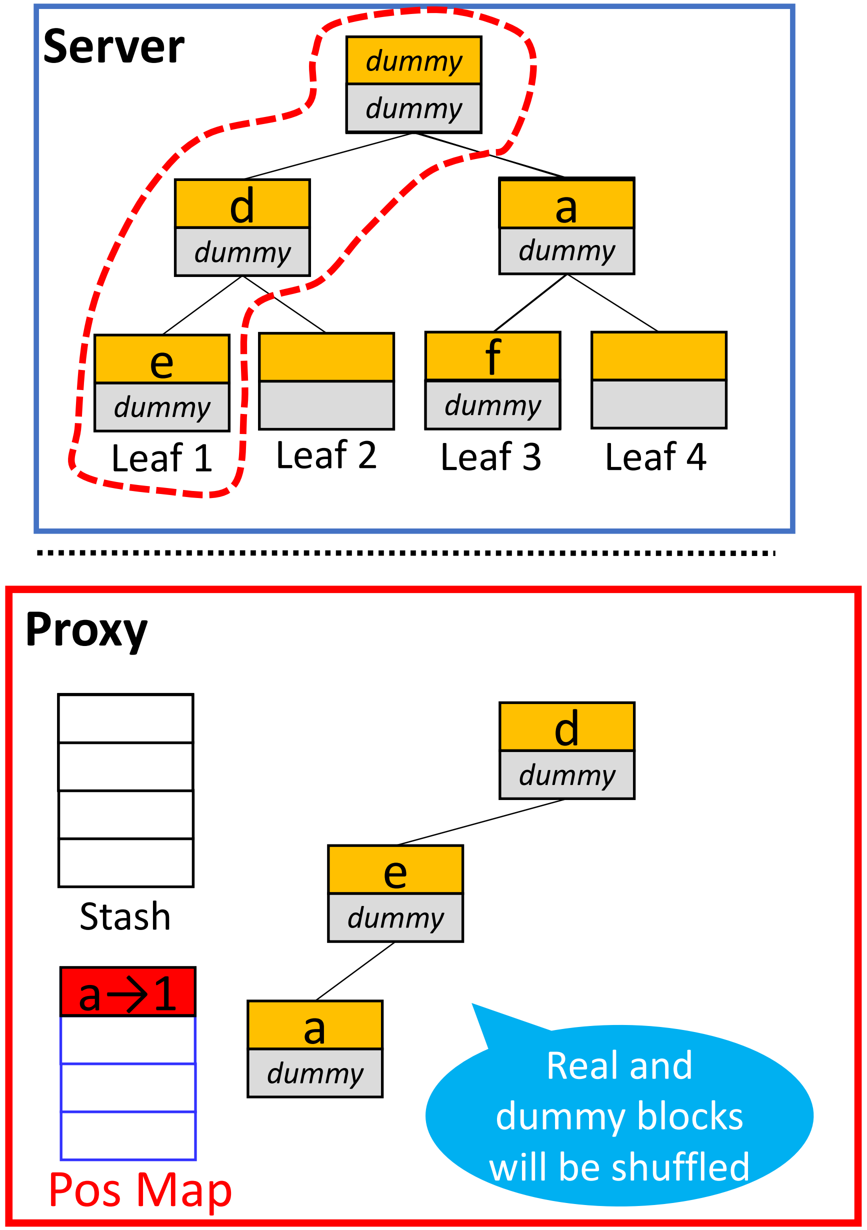
For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*
- Assign block to a new random path in position map

2) Evict

- After *A* read paths, in a deterministic order pick the next path to evict
- For each bucket, read all remaining valid real blocks (if < Z, read dummy) to *stash*
- Write each bucket from *stash* and reset all metadata

Ring ORAM



1) Read path

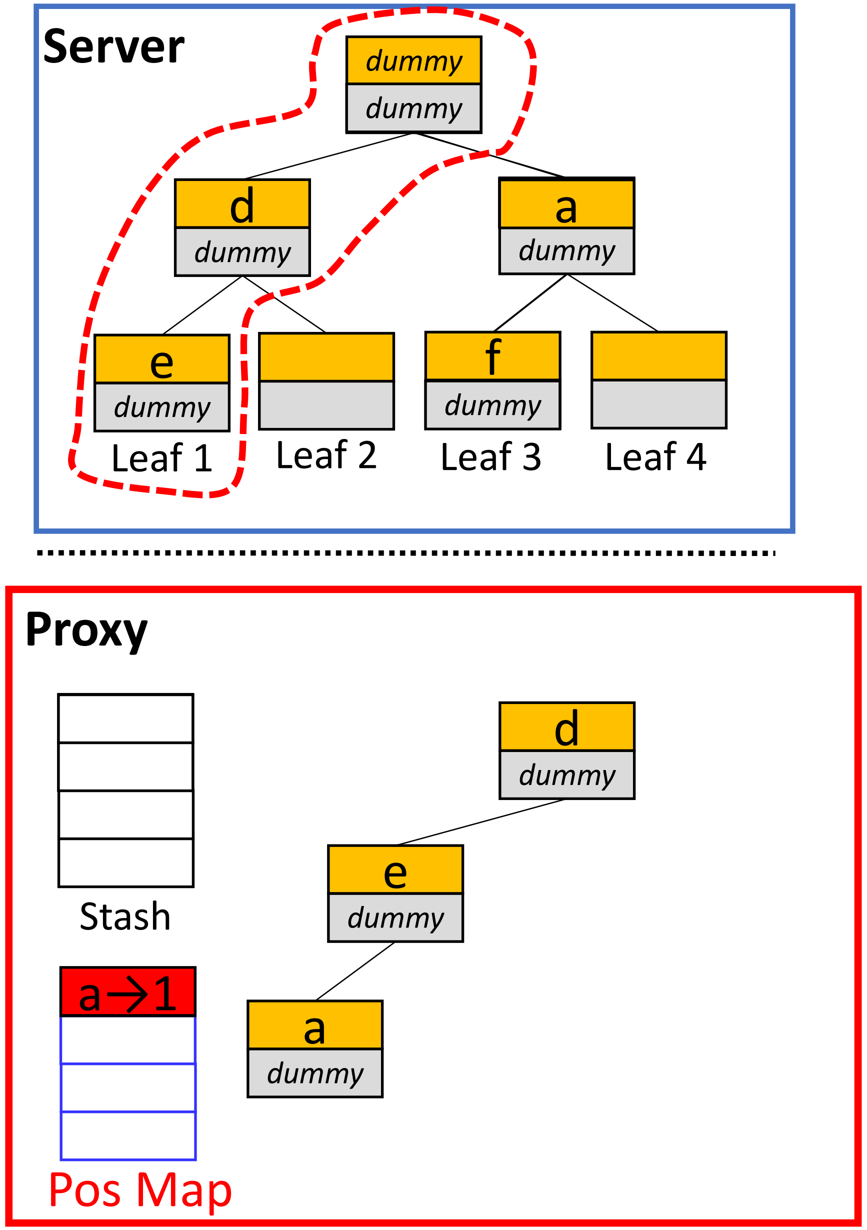
For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*
- Assign block to a new random path in position map

2) Evict

- After *A* read paths, in a deterministic order pick the next path to evict
- For each bucket, read all remaining valid real blocks (if < Z, read dummy) to *stash*
- Write each bucket from *stash* and reset all metadata

Ring ORAM



1) Read path

For each bucket in path

- From *valid* and *addr*, either read real block or a valid dummy block
- Invalidate the read block in *valid*
- Increment *count*
- Assign block to a new random path in position map

2) Evict

- After *A* read paths, in a deterministic order pick the next path to evict
- For each bucket, read all remaining valid real blocks (if < Z, read dummy) to *stash*
- Write each bucket from *stash* and reset all metadata

3) Early reshuffle

- If a bucket is accessed *s* times, read all valid real blocks, permute, and write back
- Reset metadata for the bucket

Security arguments for Ring ORAM

1. Read path leaks no information
   - For each access, a random path is read
   - For each bucket, a random offset is read

2. Evict path leaks no information
   - Every *A* accesses, a deterministically chosen path is read
   - Each bucket reads *Z* blocks
   - Path written back

3. Early shuffle leaks no information
   - After *S* accesses to a bucket, *Z* blocks are read
   - Bucket is written back

Limitations of Path and Ring ORAM

- Both are sequential
  - TaoStore by Sahin et al. S&P'16 [Jan 25th]
- They both require a proxy to be practical
  - ConcurORAM by Chakraborti et al. NDSS'19 [Jan 30th]
- They do not support transactions or complex queries
  - Obladi by Crooks et al. OSDI'18 [Feb 1st]
  - ObliDB by Eskandarian et al. VLDB'19 [Mar 12th]
- Neither is fault tolerant
  - QuORAM by Maiyya et al. Usenix Security'22 [Feb 6th]
- Neither is scalable
  - ObliviStore by Stefanova et al. S&P'13 (not reading)
  - Snoopy by Dauterman et al. SOSP'21 [Mar 14th]

Conclusion

- Access patterns leak information
- Need workload independence
- Databases using ORAM ensure workload independence
- PathORAM: a highly efficient tree-based ORAM
  - Simple abstraction & easy to implement
- RingORAM: optimizes PathORAM by reducing online bandwidth cost