# Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage

Peter Williams, Radu Sion, Bogdan Carbunar

*Presented by*: **Mehmud Abliz**

# Introduction

- Networked storage, such as email storage and storage portals, needs protection.
  - Assurance of data confidentiality and integrity
  - Moreover, access pattern privacy
- Private Information Retrieval (PIR) protocols are impractical.
  - They are less time-efficient than trivially transferring the entire database.
- Oblivous RAM (ORAM)
  - 100~1000 seconds per data access

# Related Work – Oblivious RAM

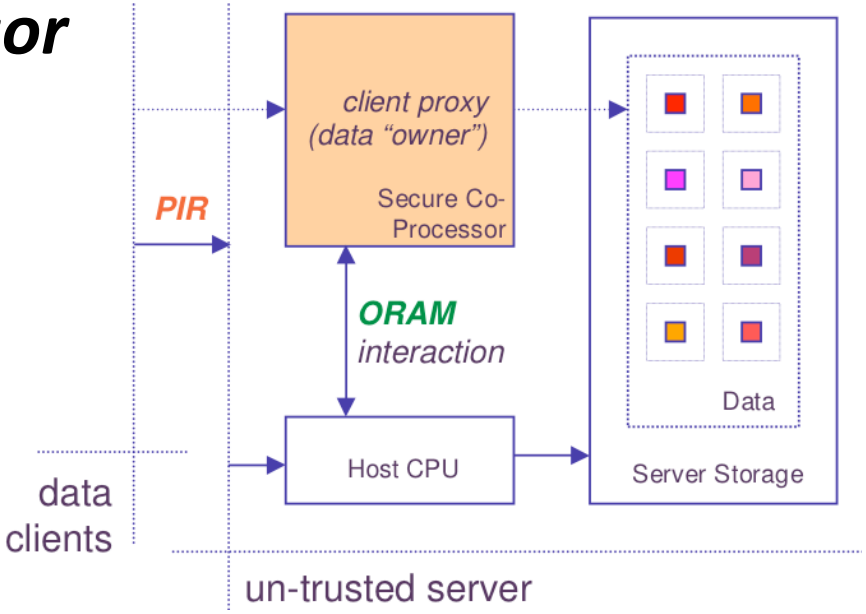
- Database model
  - A set of ***n*** encrypted blocks
  - Supports **read**(*id*), **write**(*id*, *newvalue*)
  - Data is organized into $log_4(\boldsymbol{n})$ levels as a pyramid; level ***i*** consists of up to $4^i$ blocks; ***block address*** is determined by a hash function; each ***Bucket*** may contain 0 to log ***n*** blocks.
- ORAM Reads
  - Hash function tells one bucket to visit at each level;
  - Client scans indicated buckets until target block is found
  - Continue by randomly scanning one bucket per level, until all levels are scanned
  - Re-encrypt query result, and place it on the top level

# Oblivious RAM (cont.)

- ORAM Writes
  - Almost identical, **except** the *newvalue* will be inserted into the top-level, instead of query result.
- Reads and writes are indistinguishable. ***Why?***
- Level Overflow
  - Top level emptied into 2nd level and 2nd level is then re-encrypted; 2nd level emptied into 3rd …
  - A sorting network is used to re-order the blocks
  - Each re-order process fills all partially empty buckets with fake blocks.
- Computational complexity: $O(\log^4(\boldsymbol{n}))$
- Williams et al. provide a variant with $O(\log^2(\boldsymbol{n}))$

# Related Work – Secure Hardware-aided PIR

- Difference between PIR and ORAM
  - In PIR, data isn't necessarily originated by the user
- Secure Hardware aided-PIR
  - Using ***secure co-processor*** as a trusted server-side ORAM client proxy
- Complexity: *O*(*sqrt*(n))



# Assumed System Model

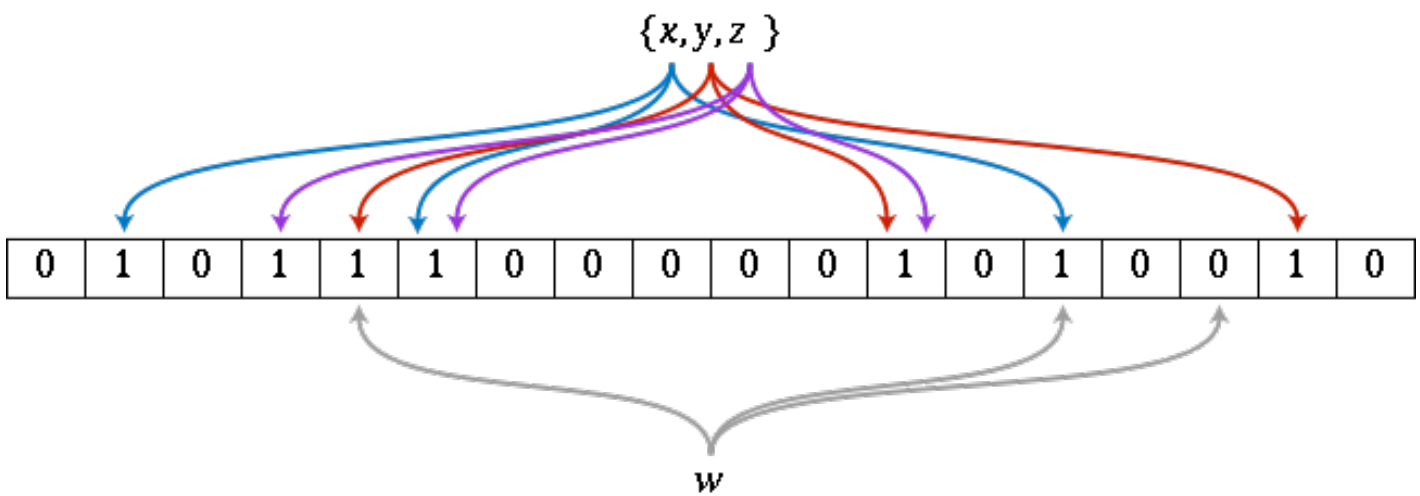
- System
  - data is stored on remote server, and accessed through the server's online query interface.
  - Data is composed of equal-sized blocks
- Client
  - *O*(***sqrt***(n)) temporary storage
- Adversary
  - Desire to illicitly gain info about the stored data
  - Attempt to cause data alteration

# Proposed Solution Overview

- Use one collision-free Bloom filter at each level, instead of the fixed-sized hash bucket in ORAM.
- Bits of the Bloom filter are encrypted.
- At each level, the per-level Bloom filter is queried first, and a *fake data item* is queried if the target item is not in that level.
- After finding the target item, it is moved into the top level.
- Encrypted Bloom filter avoid hash collision, thus removes the log ***n*** computation of hash collision handling.

# Bloom Filter Refresher

- A bit array of ***m*** bits
- ***k*** different hash functions
  - each maps some set element to one of the m array positions
- Add an element
  - feed it to each of the ***k*** hash functions to get ***k*** array positions; Set the bits at all these positions to 1.
- Query an element
  - feed it to each of the ***k*** hash functions to get ***k*** array positions. If any of the bits at these positions are 0, the element is not in the set.



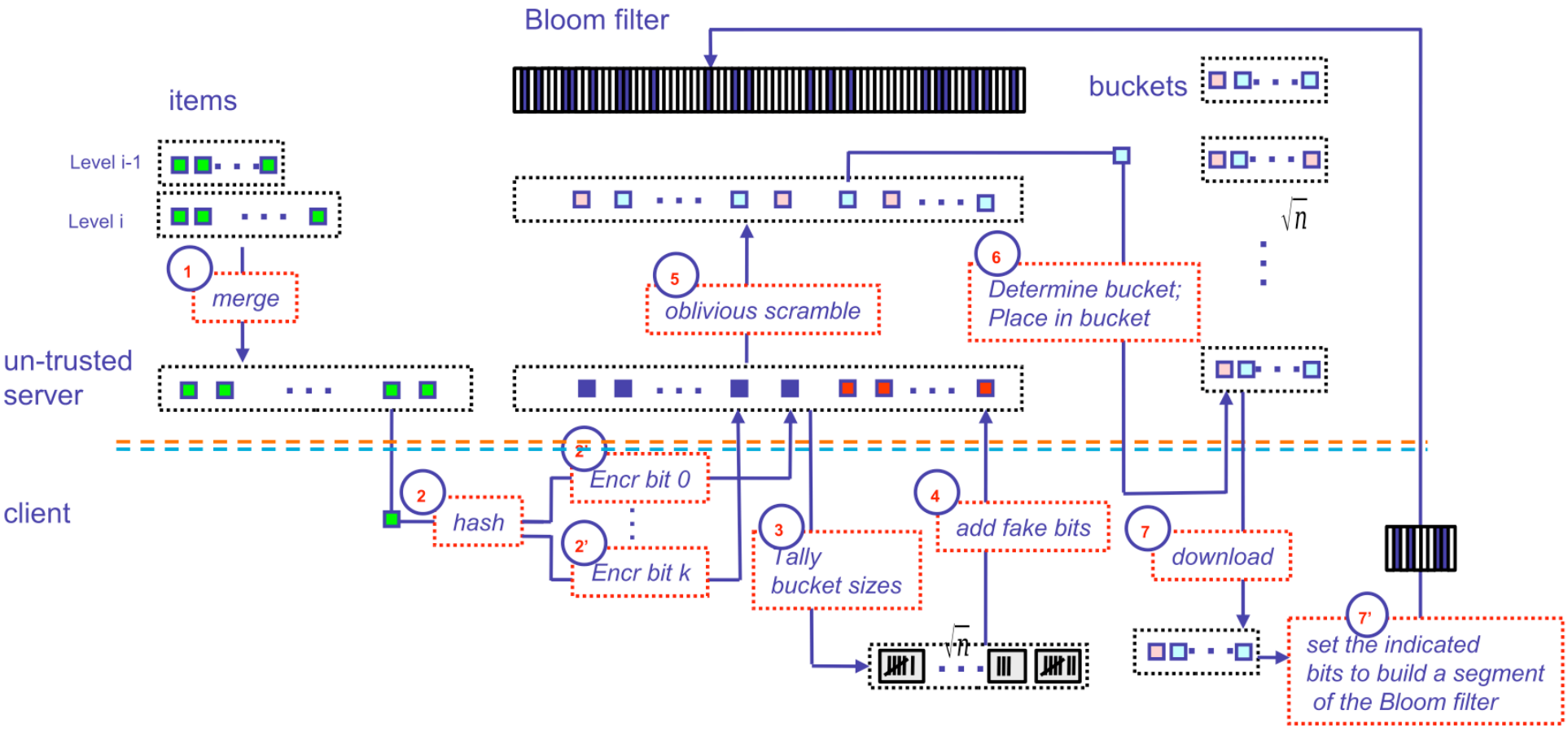
# Querying a Data Item

```
query(x : id)
  server : Server; #server stub
  bits : int[]; #bit values in Bloom filter
  label, fakeLabel : int[]; #search labels
  fakeAccCtr : int[]; #per level access counter
  found : bool;
  K : int[]; # secret key
  v : Object; # value for name x
found, v := scanServerItemCache(x);
for (i := 1; i < log_4 n; i++) do
    fakeAccCtr(i)++;
    fakeLabel := hash(i, "data", Gen(i), fakeAccCtr(i), K);
    if(found = false) do
        label := hash(i, "BF", Gen(i), "x", K);
        bits := server.getBloomFilter(i, label);
        if(decrypt(bits) = "11..1") do
            label := hash(i, "data", Gen(i), "x", K);
            v := server.getNRemove(label); found := true;
        else server.getNRemove(fakeLabel) fi;
    else
        label := hash(i, "BF", Gen(i), fakeAccCtr(i), K);
        server.getBloomFilter(i, label);
        server.getNRemove(fakeLabel);
    fi
itemCache.append(x, v);
return (x, v);
end.
```

# Initial Database Construction and Handling Level Overflows

- Database construction and handling level overflow uses the same process.
- Reshuffle overview
  - When item cache is full, it is emptied into level 1; when level 1 is full it is emptied into level 2, and so forth.
  - When **level *i-1*** is dumped into **level *i***, a new Bloom filter is constructed for level *i*, and data items in level *i* is scrambled.

# Encrypted Bloom Filter Construction



# Oblivious Merge Scramble Algorithm

- The algorithm is used for scrambling both Bloom filter positions and data items.
- The algorithm is essentially a **merge sort**, except
  - a random number generator is used in place of a comparison
  - the array is recursively divided into segments, and multiple segments are merged in parallel.
  - ***Oblivious Merge Step*** is performed on each segment.

# Oblivious Merge Step

```
oblivious_merge_step(A_1[], ...A_r[])
  B : array[n]; #new remote destination buffer of size n
  s := 2c√(n/r); #size of local queues
  for (i := 1; i ≤ r; i++) do
      q_i := new queue[s];
      for (x := 1; x ≤ s/2; x++) do
          q_i.enqueue(decrypt(A_i.readNextItem()));
  #at this point each queue has s/2 items
  for (x := s/2; x ≤ n + s/2; x++) do
      if (x ≤ n) then
          for (i := 1; i ≤ r; i++) do
              q_i.enqueue(decrypt(A_i.readNextItem()));
      fi
      # now we have read r items; time to output r items
      for (i := 1; i ≤ r; i++) do
          v := randomlyChooseWhichArray();
          t := q_v.dequeue();
          B.writeNextItem(encryptWithNewNonce(t));
end.
```

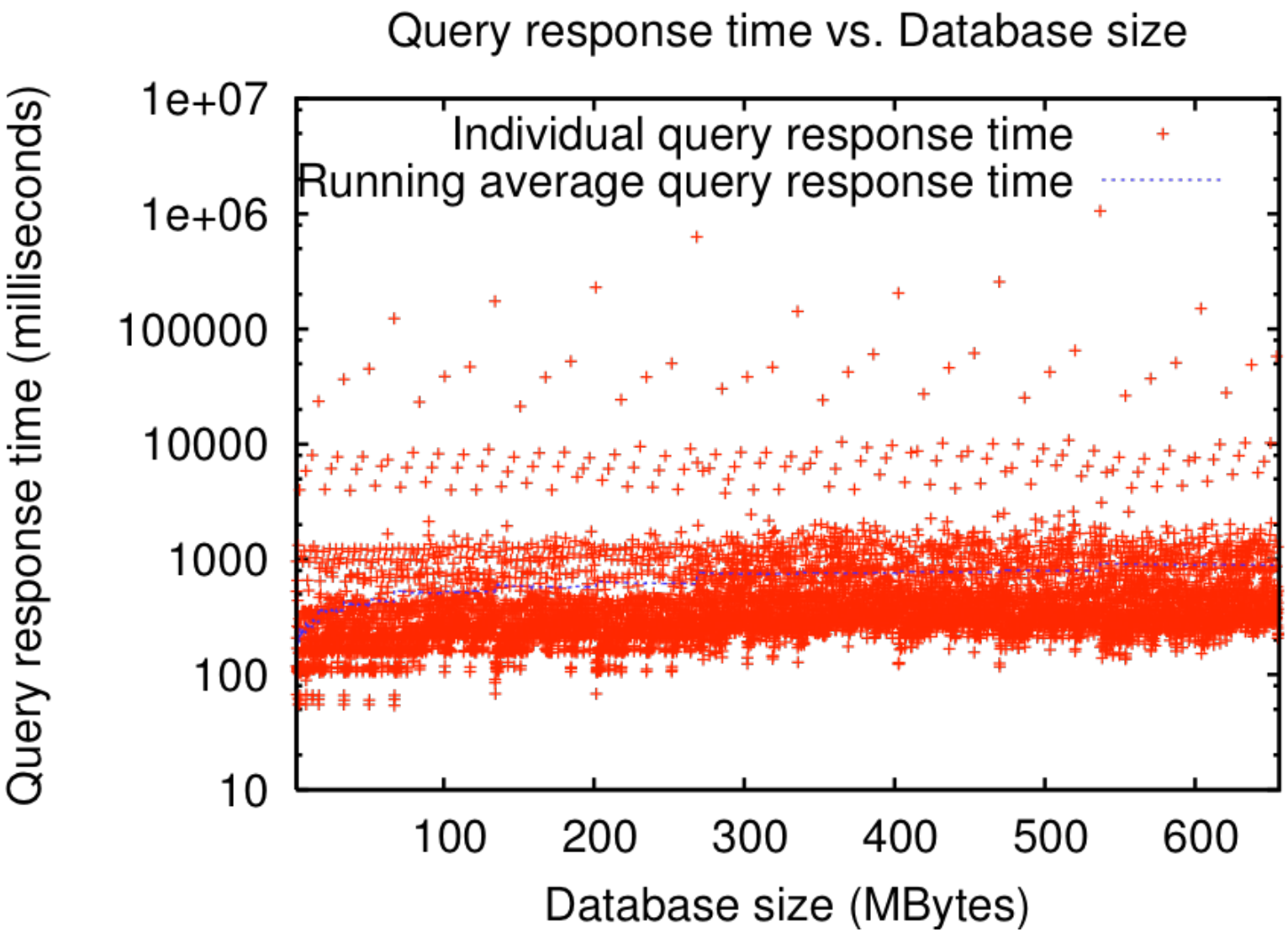
# How Access Privacy is Achieved

- No item is ever queried twice using the same label.
  - Achieved by removing an item once it is found, and placing it in the item cache.
- Access patterns are indistinguishable.
  - Access all levels and retrieve one data item at each level. Add one item to the cache.

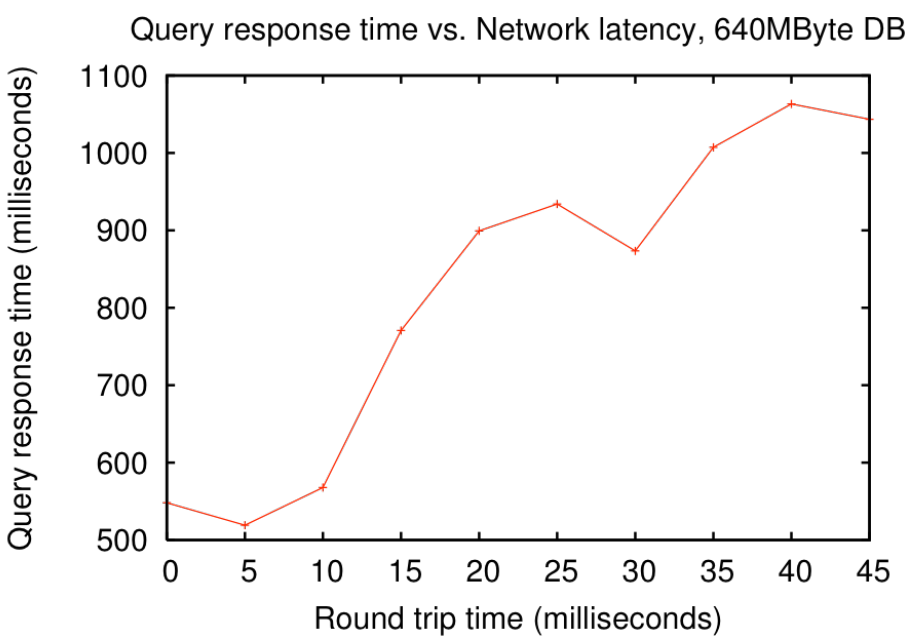
# Correctness and Integrity

- To detect tampering by the storage provider
  I. Compute MAC for each data item and bloom filter, and it at the client
  II. Unique version labels for each item
  III. Incremental, collision-resistant ***commutative checksum*** for item set in each level.
    - $\boldsymbol{H(\mathrm{B}) = \prod h(\mathrm{b}_i) \bmod p}$

# Performance Evaluation



# Impact of Network and DB Size




# Strength & Weaknesses

- Strength
  - Provide significantly more efficient private information retrieval framework.
- Weakness
  - Equal-sized blocks can lead to waste of storage space
  - Complex, hard to implement.