浙江大学

不经意随机访问机综述

面向信息技术的沟通技巧—作业 3

Zhu Jingsen
朱璟森
3170104166

May 26, 2019

Contents

Abstract

With the development of cloud computing, more and more people and companies choose to store their data in cloud servers, which cause potential safety hazards. Despite the use of strong encryption schemes, adversaries can still learn valuable information regarding encrypted data by observing the data access patterns. In solution, one can hide the access patterns, which may leak sensitive information, using Oblivious RAMs (ORAMs). There are many implementations of ORAMs to hide the access patterns. A main issue remains to decrease the time and bandwidth costs. In this review, we list several implementations of ORAMs and summerize their performences.

Keywords: Access pattern, ORAM, cloud computing

1 背景与定义

1.1 背景—访问模式与数据安全 [1, 2]

随着云计算、云服务的发展，越来越多的数据被存储在云端服务器上。然而，一旦服务器遭到恶意攻击，或者不可信的云服务提供商恶意获取服务器信息，用户的隐私信息就会泄露。传统的手段是对数据内容进行加密, 密钥由用户个人持有. 用户通过上传、下载密文数据来保护个人的隐私信息不被泄露。

然而，加密只能保证数据内容的机密性，由于 CPU 与内存之间的内存总线 (Address Bus) 是无法加密的，攻击者通过客户端访问的数据块地址序列（访问模式），依旧能够推断出其它的隐私信息。如文献 [3] 中，攻击者可以通过搜集访问模式，可以通过统计的手段推断出 80% 已加密的搜索请求。而在逆向工程领域 [4, 5]，内存地址的访问模式可以帮助攻击者分析出程序的结构，例如，在程序运行的过程中，攻击者可以观察到对内存位置 100,101，X（102 或 103），104 重复出现，他就可以推断出这是一个包含着条件分支的循环。

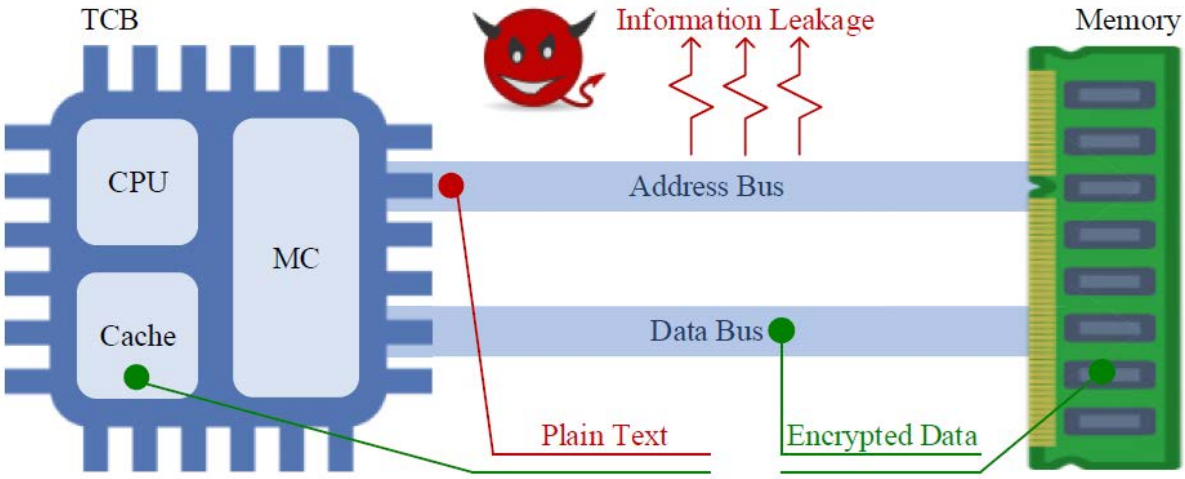

Figure 1: 内存总线上访问模式的泄露

Goldreich 与 Ostrovsky 在论文中 [6] 提出了不经意随机访问机 (Oblivious RAM, abbr. ORAM) 技术。这一技术的目的是隐藏对真实数据块的访问，使得攻击者不能区分每一次访问是真实的还是随机的。不经意随机访问机应用于云存储系统可以有效的防止攻击者利用访问模式获取隐私信息，减小数据存储系统的攻击面，打破了单纯使用传统加密方式来保护数据隐私的系统框架，为用户提供更完善的安全存储服务。

但是，不经意随机访问机也会带来额外的开销，比如：为了隐藏访问模式，需要对多个数据块进行访问，这增大了客户端与服务器之间的带宽，客户端需要更大的缓存空间来存储从服务器端返回的额外数据块。所以，不经意随机访问机的实用性还面临很大的挑战。

1.2 ORAM 安全性定义

为保护和混淆内存的访问模式，ORAM 保证了在存储器中的任意数据块不会永久驻留在某一个物理地址中，这确保了任意两次访问不会产生关联。同时，ORAM 将每一次读写访问 (access) 细化成一次读取加一次写回的原子操作 (operation)，其中读问转化成读取内容再写回相同内容，写访问转化成读取内容再写回更新后的内容，使得攻击者不能够区分具体的访问方式。ORAM 能很好地保护

1. 访问数据块的位置
2. 数据块请求的顺序
3. 对相同数据块的访问频率
4. 具体的读写访问方式

下面我们对 ORAM 的安全性进行正式的定义：

定义 1.1. ***ORAM* 安全性**
设 $\vec{y}$ 为长度为 M 的一系列数据操作序列，$\vec{y} = (op_1, id_1, block_1), \ldots, (op_M, id_M, block_M)$，$op_i$ 为读操作 $read(id_i)$，其读取标识符为 id_i 的块；或写操作 $write(id_i, block_i)$，将新数据 $block_i$ 写入到标识符为 id_i 的块中。如果 $op_i = read(id_i)$，那么 $block_i = null$。
给定操作序列 $\vec{y}$，设其经过 *ORAM* 加密后的访问模式为 $A(\vec{y})$，该 *ORAM* 体制是安全的当期仅当对任意两个相同长度序列 $\vec{y}, \vec{y'}$，其访问模式 $A(\vec{y}), A(\vec{y'})$ 计算上不可区分。

为了让读写操作不可区分，标准的解决方案是始终采用“先读后写”的操作，不需要写入的时候通过写入无效信息 (dummy write) 来进行混淆。

1.3 发展脉络

不经意随机访问机的概念最早起源于 RAM (random access machine) 模型，RAM 是一种重要的计算仿真手段。在这个模型中，处理器通过对存储器的读写来实现程序的执行。上个世纪 80 年代，为了隐藏程序对内存的访问模式来避免软件的逆向工程，Goldriche 等人在此基础上提出了 ORAM [4]，开启了 ORAM 的研究历程。不经意随机访问机面临的最大问题是性能开销大。纵观不经意随机访问机的发展。其设计模型大致可以分为 5 类：简单模型、平方根模型 [7,8]、层次模型 [7]、分区模型 [9] 和树状模型 [10,11]。不同的设计模型表示服务器存储数据块的数据结构不同，用户通过对服务器 ORAM 的访问来获取所需的数据块，这些模型设计的目标主要还是提高性能。本文主要介绍后 4 种模型及其变种，并简介 ORAM 的最新研究进展。

2 常见 ORAM 体制研究进展

2.1 ORAM 常用的技术

2.1.1 不经意排序

不经意排序 *(Oblivious sort)*，又称为茫然排序，是指这样一种排序算法，其排序过程对数据操作的顺序是预定义、固定的，如冒泡排序是典型的茫然排序，而选择排序则不是 (顺序依赖于数据中每一轮的最大值)。时间复杂性较好的茫然排序有 $O(n \log^2 n)$ 的 Batcher 排序网络和 $O(n \log n)$ 的 AKS 排序网络，然而后者的常数项 (≈ 6100) 远大于前者 (≈ 0.5)，因此实用中通常采用理论复杂度更高的前者。

2.1.2 不经意随机排列

不经意随机排列 *(Oblivious random permutation)*，在许多 ORAM 体制对数据的预处理与混洗 (shuffle) 中用到。我们通常采用 Knuth 洗牌算法：

Algorithm 1 Knuth shuffle algorithm

```
1: procedure SHUFFLE(A, N)
2:     for i ← 0 to N − 1 do
3:         j ← random(0, N − 1)
4:         SWAP(A_i, A_j)
5:     end for
6: end procedure
```

2.2 平方根模型

Square Root ORAM 是第一个在文献中提出的 ORAM 模型 [7]，分为 *Basic Square Root ORAM (Basic-SR)* 及其改进 *Interleave Buffer Shuffle Square Root ORAM (IBS-SR)*。

2.2.1 Basic-SR

在这个 ORAM 中，服务器的结构如下：服务器共储存 $N + 2\sqrt{N}$ 个数据块，其中 N 个为加密过的真实数据块，$\sqrt{N}$ 个为用于混淆的伪数据块 (dummy block)，最后 N 个为缓冲区 (shelter)。
Basic-SR 的算法流程如下：

1. 对前 $N + \sqrt{N}$ 个数据块，也就是真实数据块与伪数据块一起进行随机排列。
2. 扫描缓冲区 $\sqrt{N}$ 个数据块
   - j 在缓冲区中，则对缓冲区进行一次伪数据块读写
   - j 不在缓冲区中，则从 j 的位置 $\pi(id_j)$ 中读取数据并写入伪数据块
3. 进行第二次缓冲区扫描，如果是写操作就在缓冲区中更新数据，读操作就写入伪数据块。
4. 当计数器 $count$ 增加到 $\sqrt{N}$ 时，就要对所有数据块进行不经意排序，否则访问模式就会有泄漏的风险。

伪代码如下：

Algorithm 2 Basic-SR ORAM

$p \leftarrow 0$
while true do
  $p \leftarrow p + 1$
  Perform an oblivious random permutation π of the blocks in the first $(N + \sqrt{N})$ locations (data + dummy blocks)
  for $count \leftarrow 1$ **to** $\sqrt{N}$ **do**
    $j \leftarrow (p-1)\sqrt{N} + count$
    if id_j in shelter location x $(N + \sqrt{N} + 1 \leq x \leq N + 2\sqrt{N})$ **then**
      Read a dummy block from $\pi(N + count)$
      Write back this dummy block
    else ▷ id_j not in shelter
      $x \leftarrow -1$, read id_j from $\pi(id_j)$
      Write a dummy block to $\pi(id_j)$
    end if
    for $i \leftarrow 1$ **to** $\sqrt{N}$ **do** ▷ scan the shelter
      Read shelter block $N + \sqrt{N} + i$
      if $N + \sqrt{N} + i = x$ **then**
        if $op_j = write$ **then**
          Write $block_j$ to $N + \sqrt{N} + i$
        else ▷ $op_j = read$
          Write the same data back to $N + \sqrt{N} + i$
        end if
      else if $i = countx = -1$ **then**
        if $op_j = write$ **then**
          Write $block_j$ to $N + \sqrt{N} + i$
        else ▷ $op_j = read$
          Write the same data back to $N + \sqrt{N} + i$
        end if
      else
        Write a dummy block to $N + \sqrt{N} + i = x$
      end if
    end for
  end for
  Perform an oblivious sort on all $(N + 2\sqrt{N})$ locations
end while

平方根模型以 $\sqrt{N}$ 次访问为一轮。在每一轮访问中，保证前面每个数据块的位置只会被访问一次，之后被访问的数据块就会被载入缓冲区 shelter 中。而对缓冲区 shelter 的查找与访问都会扫描整个缓冲区，以保证不会泄露访问模式。每一轮访问结束之后，所有数据就要重新排序混洗。

Basic-SR ORAM 模型，对每一轮访问 ($\sqrt{N}$ 次)，摊还时间复杂度由不经意排序的复杂度决定，为 $O(\sqrt{N}\log^2 N)$(Batcher 排序网络) 或 $O(\sqrt{N}\log N)$(AKS 排序网络)。

2.2.2 IBS-SR

IBS-SR 是对平方根模型的一个简要改进 [8]，服务器储存 N 个数据块和 $\sqrt{N}$ 个伪数据块，而对客户端有 $O(\sqrt{N})$ 的内存空间复杂度。

IBS-SR 的改进在于，使用哈希函数，每次对哈希映射到的 $\sqrt{N+\sqrt{N}}$ 个数据块进行混洗并更换哈希映射，以代替 Basic-SR 中 $O(N\log^2 N)$ 的不经意排序操作，该混洗操作称为 *Interleave Buffer Shuffle (IBS)*。从而突破了不经意排序的性能瓶颈，使得 ORAM 的摊还时间复杂度降为 $O(\sqrt{N})$。

2.3 层次模型

Hierarchical ORAM 是文献 [7] 中提出的另一种 ORAM，相比 Square Root ORAM，Hierarchical ORAM 的效率比较高。

Basic Hierarchical ORAM (Basic-HR) 将服务器存储空间划分为 $L=\lceil \log_2 N\rceil$ 层，每层 $\ell(1\le \ell\le L)$ 中有 2^ℓ 个桶，则每层的桶的个数恰好是上一层中的两倍。每 2^ℓ 次操作后，第 ℓ 层就会被混洗。初始化的时候，要为每一层选择一个 hash 函数，用于决定数据块应放入哪个桶中。

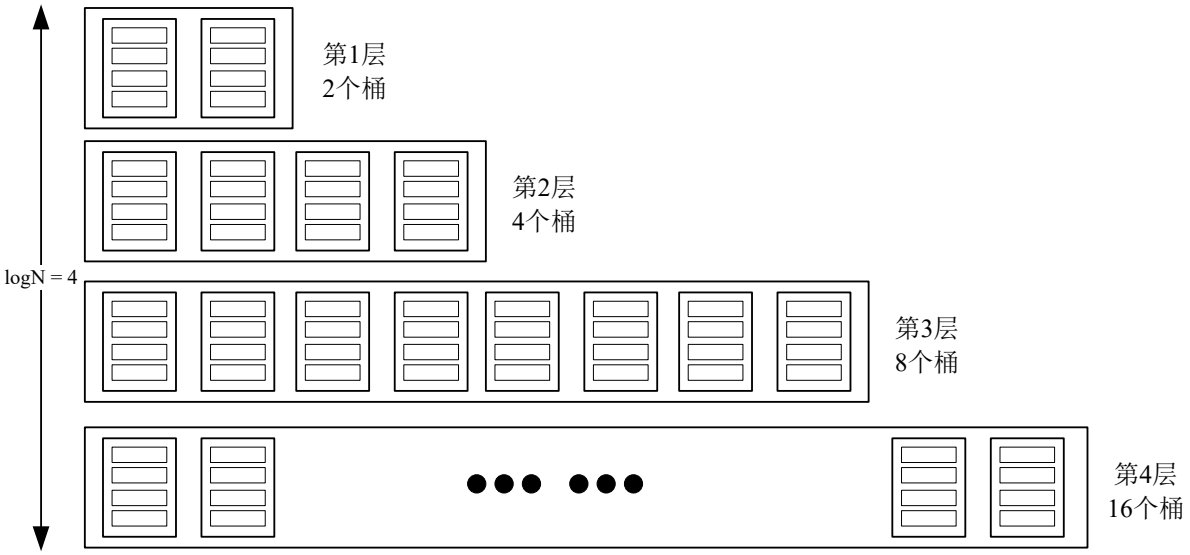

Figure 2: $L=4$ 时 Hierarchical ORAM 的数据结构

Basic-HR 算法流程如下：

1. 从第 1 层到第 L 层，逐层根据哈希函数扫描数据块 id_j 可能在的桶。如果在之前的层中已经找到，则随机选一个桶来读写。如果在该层找到，则读取后写回一个伪数据块。如果没有找到，则读取并原样写回该桶的数据块。
2. 所有层搜索完毕后，将数据块 id_j 写入第一层。
3. 每进行 2^ℓ 次操作，就要将 ℓ 层与 $\ell+1$ 层数据块混洗，并且哈希函数要重新选取。

伪代码如下：

Algorithm 3 Basic-HR ORAM

$j \leftarrow 0$
while true do
  $j \leftarrow j + 1$
  for $\ell \leftarrow 1$ **to** L **do**
    if block id_j hasn't been found **then**
      Scan hash bucket to find id_j
      if id_j found **then**
        Write a dummy block back
      else
        Write the original block back
      end if
    else ▷ id_j has been found
      Read and write a random block in level ℓ
    end if
  end for
  if j is odd **then**
    Write block id_j into 1st bucket in level 1.
  else
    Write block id_j into 2nd bucket in level 1.
  end if
  $d \leftarrow \max\{1 \le x \le L : j \bmod 2^x = 0\}$
  for $\ell \leftarrow 1$ **to** d **do** ▷ shuffle every 2^ℓ operations
    Pick a new hash function for level $\ell + 1$
    Shuffle data in level ℓ and level $\ell + 1$ using the new hash function.
  end for
end while

复杂度分析上，云端服务器的空间开销为 $O(N \log N)$，客户端空间开销为 $O(1)$。摊还时间复杂度由不经意排序的复杂度决定，为 $O(\log^4 N)$(Batcher 排序网络) 或 $O(\log^3 N)$(AKS 排序网络)。与平方根模型相比有显著提升，但仍开销较大。

2.4 分区模型

TP-ORAM 是文献 [9] 中提出的一种对层次模型的改进，其核心思想为将服务器分区 (partition)，将单个大小 N 分割为 $\sqrt{N}$ 个小型 ORAM，每个分区相当于一个 ORAM 的黑盒子，并不依赖具体的内部实现。

TP-ORAM 中，分为 $P = \sqrt{N}$ 个分区，每个分区采用层次模型，因此每个分区内有 $L = \log_2 \sqrt{N} + 1 = \frac{1}{2}\log_2 N + 1$ 层。与 Basic-HR 一样，第 ℓ 层有 2^ℓ 个块。为了防止随机误差导致的可能溢出，第 L 层有 $2^L + \epsilon = 2\sqrt{N} + \epsilon$ 个区块。通过简单的几何级数求和可以知道，每个分区一共有 $4\sqrt{N} - 2 + \epsilon$ 个区块，而在具体实现 [9] 中，被进一步放宽为 $4.6\sqrt{N}$。除了分块内存，服务器中还存在 $O(\sqrt{N})$ 大小的缓冲区 (stash)，stash 被分为 P 块，每块对应服务器内存的一个分区。另外还有一个映射表 (position map) 来决定每一个数据块 id_j 被分配到哪个分区中。

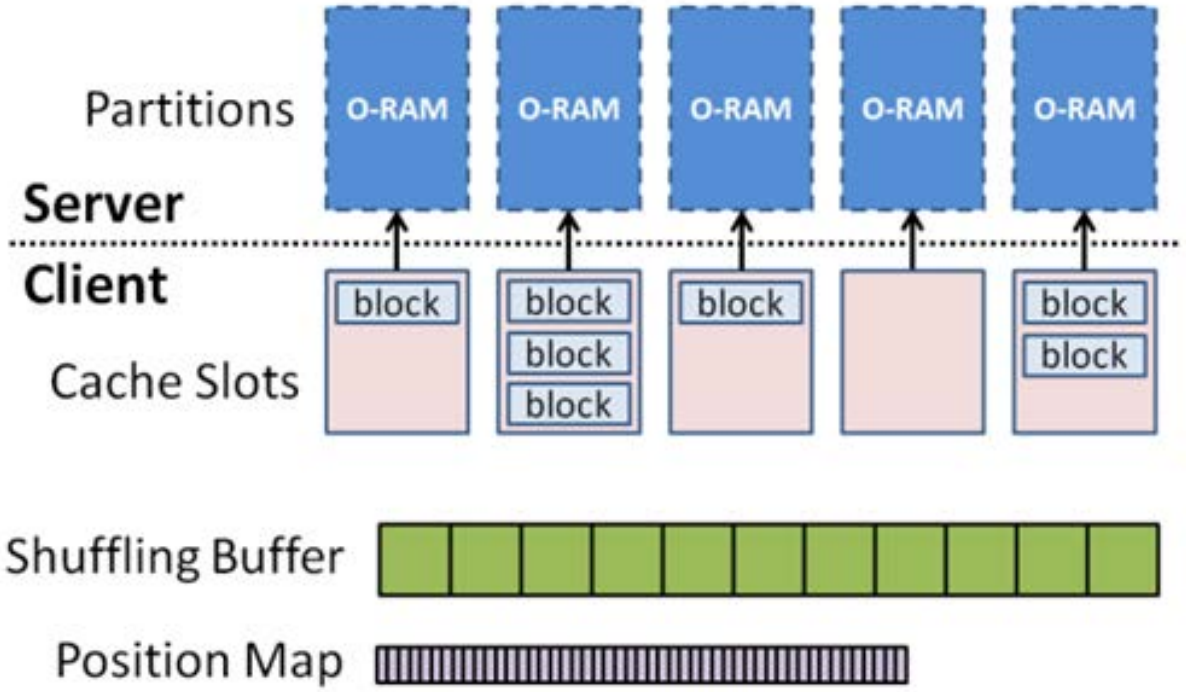

Figure 3: 分区模型的数据结构

TP-ORAM 的主要流程如下：

1. 通过映射表获取 id_j 所在的分区。先访问缓冲区，如果找到就读取出来并对内存分区进行一次伪访问；否则从内存分区中访问 id_j。
2. 随机选择另一个分区编号 r，更新映射表并将 id_j 放到新的缓冲区中。
3. 进行两次驱逐操作。

伪代码如下：

Algorithm 4 TP-ORAM

$j \leftarrow 0, s \leftarrow 1$
while true do
  $j \leftarrow j + 1$
  $p \leftarrow position[id_j]$
  if block id_j found in $stash[p]$ **then**
    Read and delete id_j from $stash[p]$
    Read a dummy block from partition p
  else ▷ id_j not in stash
    Read id_j from partition p
  end if
  $r \leftarrow random(1, P)$
  $position[id_j] \leftarrow r$
  add id_j into $stash[r]$
  EVICT(p) ▷ Piggy-backed eviction
  SEQUENTIALEVICT(ν) or RANDOMEVICT(ν)
end while

驱逐操作 (eviction) 分为两个过程，第一个 eviction 过程是捎带式 (Piggy-backed) 的，也就是客户端执行一次访问之后，就会执行一次驱逐操作。驱逐操作将缓冲区中的一个元素“驱逐”回内存分区中，并对缓冲区写入一个伪数据。

第二步称为背景式 (Background) 驱逐，取决于一个频率参数 ν，每次数据访问之后，都会执行 ν 次 eviction 操作。如果 $\nu < 1$ 说明多次访问之后才会执行一次背景式 eviction 操作。而背景式 eviction 操作有两种策略，一种是顺序驱逐，另一种是随机驱逐。

Algorithm 5 Evict

```
1: procedure EVICT(p)
2:     if stash[p] is empty then
3:         Write a dummy block to partition p
4:     else
5:         Write a block from stash[p] to partition p and remove it from stash[p]
6:     end if
7: end procedure
```

Algorithm 6 Sequential and Random Evict

```
1: procedure SEQUENTIALEVICT(ν)
2:     num ← D(ν)        ▷ D is a prescribed distribution
3:     for i ← 1 to num do
4:         count ← count + 1
5:         EVICT(count)
6:     end for
7: end procedure
```

```
1: procedure RANDOMEVICT(ν)
2:     num ← D(ν)
3:     for i ← 1 to num do
4:         r ← random(1, P)
5:         EVICT(r)
6:     end for
7: end procedure
```

复杂度分析上，云端服务器的空间开销为 $O(N)$，客户端空间开销为 $O(\sqrt{N} + \frac{N}{B})$，$O(\sqrt{N})$ 为缓冲区而 $O(\frac{N}{B})$ 为映射表 (B 为每个数据块的大小)。摊还时间复杂度可以低至 $O(\log N)$，最坏情况为 $O(\sqrt{N})$。分区模型相比于传统的模型而言有着显著的提升，但最坏情况依然开销较大。

2.5 Basic Binary-Tree ORAM

文献 [10] 中提出了一种全新的树状模型，它不需要进行不经意排序和混洗操作，可以将 ORAM 操作的**最坏时间复杂度**控制在对数多项式以内，完成了一次显著的突破。

Basic Binary-Tree ORAM (BB-ORAM) 将整个服务器存储看做一棵平衡二叉树，而每个数据块映射到二叉树的每个叶节点。同时，客户端需要维护一个映射表 (position map) 来确定每个数据块所在的节点。

2.6 Path-ORAM

Path-ORAM 是文献 [11] 中对 BB-ORAM 的一种改进，其思想同样采用树状模型，而在实现细节上有所优化。

与 BB-ORAM 一样，Path-ORAM 把整个服务器存储看做一棵高为 $L = \lfloor \log N \rfloor$ 的二叉树，树上的每个节点上有一个大小为 Z 的桶 (bucket)，其中含有真实数据块与伪数据块。每一个叶子节点 $x \in \{0, 1, \ldots, 2^L - 1\}$ 可以唯一确定一条从根到叶子的路径，令 $\mathcal{P}(x)$ 为路径上所有桶的集合，特别地，$\mathcal{P}(x, \ell)$ 表示路径上第 ℓ 层的桶。另外，客户端上有一块 $O(\log N)$ 的缓冲区 (stash)，以及需要维护一块 $O(\frac{N}{B})$ 的映射表。Path-ORAM 的算法流程如下：

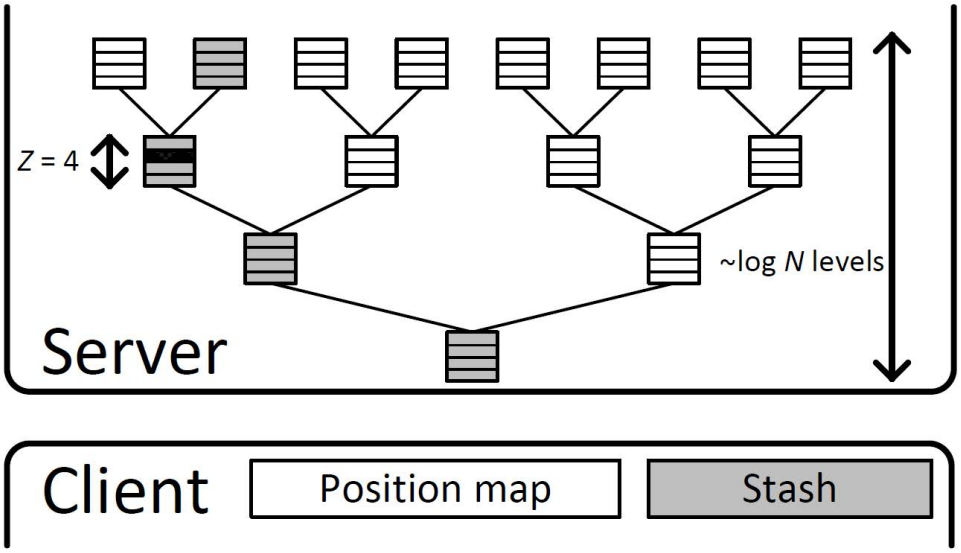

Figure 4: Path-ORAM 的树状结构

1. 通过映射表获取 id_j 所对应的叶节点 x，并更新映射表，将 id_j 随机重新映射到另一个叶节点上。
2. 将路径 $\mathcal{P}(x)$ 上的所有桶全部读取到缓冲区中
3. 如果是写操作，则写入 $block_j$
4. 将缓冲区中的数据尽可能写入回 $\mathcal{P}(x)$ 中，其中一些本来不映射到 x 的数据也可能写入到第 ℓ 层，只要 $\mathcal{P}(x,\ell)=\mathcal{P}(x',\ell)$，即 x 路径与 x' 路径在第 ℓ 层之前部分重合，那么该数据也可以写入到 $\mathcal{P}(x,\ell)$ 中。

这样，在服务端看来，客户端每次访问都是读取并写入一条随机的路径，因此访问模式不会被泄露。伪代码如下：

Algorithm 7 Path-ORAM

$j \leftarrow 0, s \leftarrow 1$
while true do
  $j \leftarrow j+1$
  $x \leftarrow position[id_j]$
  $position[id_j] \leftarrow random(0, 2^L-1)$
  for $\ell \leftarrow 0$ **to** L **do**
    $S \leftarrow S \cup \mathcal{P}(x,\ell)$ ▷ S denotes the stash
  end for
  data ← Read id_j from S
  if $op_j = write$ **then**
    $S \leftarrow (S - \{(id_j, data)\}) \cup \{(id_j, block_j)\}$ ▷ update old data to $block_j$
  end if
  for $\ell \leftarrow L$ **to** 0 **do**
    $S' \leftarrow \{(a', data') \in S : \mathcal{P}(x,\ell) = \mathcal{P}(position[a'],\ell)\}$ ▷ S' denotes the datas to write back in level ℓ
    $S' \leftarrow$ Select min$(|S'|, Z)$ from S'
    $S \leftarrow S - S'$
    Write S' into bucket $\mathcal{P}(x,\ell)$
  end for
end while

Path-ORAM 原理简单，却极其高效，而二叉树的良好特性使其复杂度比之前的模型有了明显的提升。服务端空间需求为 $O(N)$，而客户端的空间需求为 $O(\log N + \frac{N}{B})$，即缓冲区加上映射表的大小。

时间复杂度方面，由于每次操作只需遍历一条路径，则平均与最坏时间复杂度都与树的高度成正比，即 $O(\log N)$。带宽的消耗也优化到了对数级别，总带宽为 $O(\frac{\log^2 N}{\log B})$，平均带宽为 $O(\frac{\log N}{\log B})$，只要 $B = \Omega(\log^2 N)$，Path-ORAM 将比之前所有的 ORAM 模型拥有更低的复杂度 [11]。

3 总结与展望

3.1 ORAM 小结

3.1.1 优劣分析

表1中，列出了几种 ORAM 模型的优缺点。

ORAM Type	优点	缺点
Basic-SR	只需要访问缓冲区中的数据，带宽较低	(1) 需要复杂的混洗 (不经意排序) 操作，利用排序网络开销较大 (2) 平均带宽依然很高，不适用于具体应用
IBS-SR	不需要不经意排序操作，开销降低	(1) 相比 Basic-SR 增加了客户端空间开销 (2) 复杂度 $O(\sqrt{N})$ 不够理想
层次模型	(1) 在平方根模型的基础上优化了带宽 (2) 相比于平方根模型，层次模型考虑了每一个数据块被访问的频率	(1) 依然需要复杂的混洗操作 (2) 依据 hash 函数计算每一个数据块对应的数据块集合，容易导致 hash 值冲突; (3) 平均带宽依然很高，不适用于具体应用
分区模型	(1) 分布式 ORAM 可以适用于云存储场景 (2) 在层次模型上进一步优化了带宽	(1) 需要更大的客户端存储空间 (2) 每一个服务器依然需要复杂的混洗操作，同时，驱逐操作也相对困难
树状模型	相比于平方根模型和层次模型，极大地降低了复杂度	(1) 驱逐操作的复杂度很高 (2) 客户端需要存储一个映射表，增大了客户端的存储空间

Table 1: 几种 ORAM 模型的优缺点

3.1.2 性能小结

前文所述的是一些基本的 ORAM 架构，表2对它们的性能进行了小结与比较，其中包括计算时间复杂度 (摊还、最坏)，通讯开销 (摊还、最坏)，以及服务器与客户端上消耗的空间存储。注：这里假设不经意排序采用 $O(N \log^2 N)$ 的 Batcher 排序网络。

ORAM Type	Computation Overhead		Cloud Storage	Communication Overhead		Client Storage
	Amortized	Worst-Case		Amortized	Worst-Case	
Basic-SR	$O(\sqrt{N}\log^2 N)$	$O(N\log^2 N)$	$O(N)$	$O(\sqrt{N}\log^2 N)$	$O(N\log^2 N)$	$O(1)$
IBS-SR	$O(\sqrt{N})$	$O(N)$	$O(N)$	$O(1)$	$O(\sqrt{N})$	$O(\sqrt{N})$
Basic-HR	$O(\log^4 N)$	$O(N\log^3 N)$	$O(N\log N)$	$O(\log^4 N)$	$O(N\log^3 N)$	$O(1)$
TP-ORAM	$O(\log N)$	$O(\sqrt{N})$	$O(N)$	$O(1)$	$O(1)$	$O(\sqrt{N}+\frac{N}{B})$
BB-ORAM	$O(\log^2 N)$	$O(\log^2 N)$	$O(N\log N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(\frac{N}{B})$
Path-ORAM	$O(\log N)$	$O(\log N)$	$O(N)$	$O(\frac{\log^2 N}{\log B})$	$O(\frac{\log^2 N}{\log B})$	$O(\log N+\frac{N}{B})$

Table 2: 几种 ORAM 模型的性能小结

3.2 ORAM 研究进展

不同的 ORAM 模型各有优劣，其中，对当下的 ORAM 研究启发最大的是 Hierarchical ORAM (Basic-HR) 的层状结构、Path ORAM 的树状结构与 TP-ORAM 的分区思想。ORAM 的研究主要集中在减小 ORAM 的开销上，这是由 ORAM 本身的特点所决定的。ORAM 主要应用于内存或存储系统，如果读写开销与存储开销太大，就会严重限制其实际应用。因此，前文所述 ORAM 研究主要是通过设计新的数据结构与算法，来降低 ORAM 的开销。

在这些研究的基础上，当下研究注重于结合各种 ORAM 模型的特点，改进已有的 ORAM 模型的算法与数据结构，运用新开发的技术，从细节上优化 ORAM 的带宽与存储开销、减少交互轮数、提高效率等。

1. 如 [12] 中提出的 *Ring ORAM* 模型，对树状模型进行了进一步优化，其借鉴了 Partition ORAM (TP-ORAM 分区模型) 的驱逐操作的思想，设置驱逐频率常数 A，每 A 步后对路径和缓冲区进行一次驱逐操作，而不用像 Path-ORAM 那样每次都要讲 Stash 写回树中，首次使树状模型的复杂度独立于数据块大小 B，比 Path-ORAM 效率提升了 2.3 到 4 倍 [12]。
2. 而 ObliviStore [13] 则是一个高性能、分布式的 ORAM 存储系统，提出分布式 ORAM 定义的同时，也实现了异步 IO 操作。
3. 还有的模型 [14] 使用有效数据的冗余来代替伪数据块，通过重复读写不同位置上的冗余数据来实现访问模式的混淆，将时间开销缩小到 $O(1)$，然而其空间开销巨大 (取决于冗余的程度)。

3.3 存在问题

然而，虽然当下的 ORAM 研究已经取得了很大的进展，效率也有所提升，但依然有许多亟待解决的问题，这些问题限制了 ORAM 的实际应用，导致 ORAM 系统依然停留在理论研究上。

1. ORAM 的额外带宽开销依旧很大。如果要将 ORAM 部署于云环境上，则带宽开销会成为性能的主要瓶颈，严重影响 ORAM 的实用性。
2. ORAM 需要额外的客户端与服务端空间。ORAM 系统在服务端不仅存储真实数据块，还会存储伪数据块，这造成了存储空间的浪费。而 ORAM 系统在客户端也常常需要存储缓冲区、映射表等，造成客户端空间的开销。
3. ORAM 的访问操作需要与服务端进行多轮交互。与带宽开销相似，这个缺点也会影响其在云环境下的应用性能，同时也会让 ORAM 的具体实现变得复杂。
4. 当前支持分布式存储的 ORAM 极少，仅有少数 (如上文提到的 ObliviStore [13]) 支持异步访问、并发访问的安全性，但其开销较大，限制了实际应用。

3.4 研究展望

如前文所述，ORAM 技术最主要的问题就是开销巨大，这导致其难以应用于实际。因此当前的研究着重提高其效率，降低其开销。除此之外，要实际应用 ORAM 技术还需要考虑更多问题，如分布式系统的同步、异步访问等等。具体来说，未来的 ORAM 研究可以注重以下几个方面。

1. 在减小带宽开销方面还可以进行优化。云环境中，往往存储空间充足而传输带宽受到限制，如果带宽开销很大就会导致传输数据块的时间非常长，限制 OARM 的使用。当前，有的 ORAM(如 Onion-ORAM [15]) 能使带宽开销降低到 $O(1)$，但计算复杂度高。未来的研究可以尝试寻找其它的方式来达到 $O(1)$ 的带宽开销。
2. 在空间利用方面可以进行优化。当下的 ORAM 需要在服务端存储大量的伪数据块，这些伪数据块的内容没有意义，仅用于混淆真实数据块。但是，这样会造成空间的浪费。可以用利用伪数据块来存储一些信息，如 [14] 的冗余思想，使伪数据块的内容有意义，提高空间的利用效率。
3. 在分布式与并行访问方面进行研究。分布式与并行访问是云环境的一大特点，而真正实用的分布式 ORAM 系统还有待开发。
4. 对 ORAM 模型的进一步完善。ORAM 经典模型在上个世纪提出，这是一个简单的模型，只支持读写操作。未来可以针对实用性对 ORAM 模型进行进一步拓展，完善细节问题，如支持数据的删除与更新操作、满足分布式系统的一致性与可扩展性，实现日志功能等，从而推动 ORAM 系统的实际应用。

References

[1] Z. Chang, D. Xie, and F. Li, “Oblivious ram: a dissection and experimental evaluation,” *Proceedings of the VLDB Endowment*, 2016.

[2] 吴鹏飞, 沈晴霓, 秦嘉, 钱文君, 李聪, and 吴中海, “不经意随机访问机研究综述,” 软件学报, 2018.

[3] I. M. S, K. M, and K. M, “Access pattern disclosure on searchable encryption: Ramification, attack and mitigation,” *Proc Ndss*, 2012.

[4] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious rams,” *Journal of the ACM*, 1996.

[5] Z. X, Z. T, and L. H. H. S, “Hardware assisted control flow obfuscation for embedded processors,” *International Conference on Compilers*, 2004.

[6] R. Ostrovsky, “Efficient computation on oblivious rams,” *ACM Symposium on Theory of Computing*, 1990.

[7] O. Goldreich, “Towards a theory of software protection and simulation by oblivious rams,” *STOC*, 1987.

[8] D. Xie, G. L. B, Y. X, W. X, X. Y. Gao, and M. Guo, “Practical private shortest path computation based on oblivious storage,” *ICDE*, 2016.

[9] E. Stefanov, E. Shi, and D. X. Song, “Towards practical oblivious ram,” *NDSS*, 2012.

[10] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, “Oblivious ram with $o((\log n)^3)$ worst-case cost,” *ASIACRYPT*, 2011.

[11] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, “An extremely simple oblivious ram protocol,” *CCS*, 2013.

[12] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious ram," *USENIX Security*, 2015.

[13] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," *IEEE Symposium on Security and Privacy*, 2013.

[14] W. Liang, K. Bu, K. Li, J. Li, and A. Tavakoli, "Memcloak: Practical access obfuscation for untrusted memory," *ACSAC*, 2018.

[15] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, and E. S. Wichs, "Onion oram: A constant bandwidth blowup oblivious ram," *Theory of Cryptography Conference*, 2016.