一种具体高效的分层不经意随机访问机 FutORAMa 的复现

摘要

不经意随机访问机 (Oblivious RAM, ORAM) 是一种用于隐藏内存访问模式的通用技术。这是许多安全计算应用程序背后的基本任务。尽管已知的 ORAM 方案提供了最佳的渐近复杂性，尽管付出了巨大的努力，其具体成本对于许多有趣的应用来说仍然昂贵得令人望而却步。目前最先进的实用 ORAM 方案仅适用于较小的存储器 (Square-Root ORAM 或 Path ORAM)。这项工作基于最近在 ORAM 方案 (PanORAMa 和 OptORAMa) 渐近复杂性方面的突破，提出了一种新颖、具体、高效的 ORAM 构造。本文工作通过放宽对恒定本地内存大小的限制，将这些结构引入实际有用的方案领域。对于一组合理的内存大小 (例如，1GB、1TB) 和相同的本地内存大小，本文的设计比原始 Path ORAM 的实现提供了至少 6 到 8 倍的改进。作者认为，这是基于完整分层 ORAM 框架的 ORAM 的第一个实际实现。在这个份工作开展工作之前，人们认为基于 ORAM 的分层结构在实践中本质上过于昂贵。本文提出一个新颖的设计并提供广泛的评估和实验结果。

关键词： 不经意随机访问机；分层 ORAM；紧密压缩

1 引言

云服务变得越来越流行。云计算最早也是最基本的应用之一是云存储，客户端可以将数据库卸载到远程服务器，然后访问数据库并进行查询。自 20 世纪 80 年代初以来，此类服务已得到广泛使用。基于云的范例具有多种优势，包括改善能源消耗、灵活的资源利用以及各种其他优化的工作流程和环境考虑。然而，所有这些优势都在数据安全和隐私方面带来了巨大的成本。现在众所周知并被广泛接受的是，仅在将数据库条目上传到云之前对其进行加密并不能保证隐私 [3]。事实上，访问模式本身可能揭示有关底层数据或在数据上执行的程序的重要信息。为了减轻此类攻击，大家不仅希望能够加密底层数据，还希望能够 “扰乱” 观察到的访问模式，使它们看起来与数据无关。实现这一目标的算法工具称为 Oblivious RAM (ORAM)。

ORAM 在 Goldreich 和 Ostrovsky [7,8] 的开创性工作中引入，是一种概率机器，其内存访问不会透露有关其执行的程序或数据的任何信息。ORAM 结构通过排列存储在服务器上的数据块并定期重新排列它们来实现此目的。虽然 Goldreich 和 Ostrovsky 最初的用例是用于软件保护，但 ORAM 已成为设计各种密码系统的核心工具，包括云计算设计、安全处理器设计、多方计算协议等。

为了投入实际的使用，ORAM 的设计必须 “高效”。ORAM 是否高效通常通过其带宽开销来衡量：即，与原始非遗忘实现相比，在遗忘模拟中必须访问多少数据项。Goldreich 的原

始工作表明，对数开销对于受限类别的 ORAM 结构（球和箱模型中的结构以及信息理论上安全的结构）是必要的。Larsen 和 Nielsen [10] 的突破性结果取消了这些限制，表明对于任何结构来说，对数开销都是不可避免的。当前普遍认为：

(a) 基于路径 ORAM 的方案实际上是有效的，尽管渐近地不是最优的；

(b) 基于分层 ORAM 的方案可以渐近最优，但实际上表现不佳。在这项工作中，作者通过证明基于分层 ORAM 的方案可以变得实用，从而改变了这种认识。原文的主要贡献是使用 [1] 的渐近最优 ORAM 方案的思想，实际有效地实现了分层 ORAM。

实验证明，本文使用新颖的 ORAM 实现，除了实现渐进最优的分层 ORAM 方案，其表现还远优于基于 Path ORAM 的构造。

2 相关工作

人们在开发高效的 ORAM 方案方面投入了大量的精力，产生了两种主要的 ORAM 设计方法。

2.1 Path ORAM

Path ORAM 起源于 Stefanov 等人的开创性工作。这种方法被认为是实用的，因为隐藏常数非常小 [13]。然而，它使用多对数本地内存大小，开销为 $O(log2N)$。由于其对中等内存大小的具体效率，Path ORAM 式系统已被广泛实现和部署。Ascend 安全处理器、Phantom 安全处理器、GhostRider 系统以及实用混淆项目都采用了 Path ORAM 的版本。最近，安全消息传递公司 Signal 宣布使用 Path ORAM 为 enclave 提供更快的层 [4]，标志 ORAM 进入实际应用的新阶段。Path ORAM 以二叉树结构对外部存储进行组织，其中树中的每个节点为一个可存放 Z 个数据块的存储桶，树的高度为 L。每个数据块通过加密数据和一个包含程序地址、路径 ID 或叶节点标签和初始化向量（IV）的标头加以保护。ORAM 控制器将标准内存访问转换为符合 ORAM 的序列，包含位置映射（PosMap）、小型高速缓存（Stash）、地址转换和加密/解密单元。PosMap 记录逻辑地址至路径 ID 的对应，而 Stash 在访问期间暂存数据块。通过在每次访问后随机更改数据块路径 ID，ORAM 确保了访问模式的匿名性。

2.2 分层 ORAM

分层 ORAM 源自 Goldreich 和 Ostrovsky 的原始工作。连同 PanORAMa [12] 和 OptORAMa [1] 的许多突破性的重要想法和优化，该技术可用于获得渐近最优的 ORAM 结构。具体来说，对于大小为 N 的存储器，OptORAMa 使用由 $O(1)$ 个存储器字组成的本地存储器，每个存储器字的大小为 $O(logN)$，并且它们的 ORAM 具有 $O(logN)$ 开销。然而，当前普遍认为基于 ORAM 的分层构造实际上效率很低。事实上，OptORAMa 构造中的隐藏常数是 2^{228}），即使进行了各种优化，它仍然保持在数万的数量级，这使得它对于任何合理的情况来说效率都非常低。分层 ORAM 的简并版本，如平方根 ORAM [6]，已被实现，但由于带宽开销较大，它们的结构在实践中只能支持相当小的内存大小（例如，$N = 2^{14}$）。

OptORAMa 最明显的障碍是他们开发和使用的紧密压缩算法。该算法依赖于具有特定属性的扩展图的存在。他们表明了这种膨胀器的存在，但其具体尺寸（大致）与宇宙中原子的数量成正比。另一个看似固有的障碍是他们（在多个地方）对不经意的 Cuckoo 哈希的依赖。

虽然非不经意的布谷鸟哈希很实用，但不经意的布谷鸟哈希的构建更具挑战性，并且与实际相关还很远。通过这个工作可以知道如何设计这样一个具有合理渐近成本的哈希范式；然而，OptORAMa 使用了大量的不经意排序调用，以不经意的方式执行重要的图算法，在实践中实施此类程序的成本太高。

2.3 其他工作

由于现有方案（包括基于 Path ORAM 的方案）对于各种具体任务来说效率有些低下，因此系统方面的工作重点是考虑渐近性较差但隐藏常数较小的 ORAM 结构。研究表明，对于较小的内存大小，这种权衡可能会带来更好的实际效率。ORAM 还被用作高效安全多方计算协议的中心构建块（理论和实践）。例如，戈登等人 [9] 表明 ORAM 可用于在亚线性（摊销）时间内执行两方计算。此外，较多工作中提出了针对多方计算的 ORAM 渐进性的各种改进。为了简化部署，开发了新的语言和编译器来帮助以无缝的方式设计和转换算法。

与本文工作最接近的工作是 Zahur 等人的工作 [15], 优化并实现了分层 ORAM 的简并版本。具体来说，它们实现了单级分层 ORAM，允许使用 $O(\sqrt{N})$ 本地内存和相同的带宽开销来访问大小为 N 的内存。扎胡尔等人表明，对于相对较小的 N 值，他们的方案通过使隐藏常数比后者小得多，从而击败了 Path ORAM [14] 的最先进变体。然而，由于他们的系统有 $O(\sqrt{N})$ 带宽开销，因此不适合需要大内存的应用程序。（事实上，在他们的实验中，考虑的最大内存大小是 $N \leq 2^{14}$。）相比之下，本文的 ORAM 只有对数带宽开销，因此即使内存非常大（例如 1TB 甚至 1PB），依然可以发挥不错的性能。

3 本文方法

实现 ORAM 方案的实用版本非常重要，因为后者强烈依赖于重型机械，目前不知道如何将其引入现实系统领域。最明显的障碍是他们开发和使用的紧密压缩算法。该算法依赖于具有特定属性的扩展图的存在。另外，基于 Path ORAM 的方案在效率方面有一个额外的对数因子，但隐藏常数相当小；为了与 Path ORAM 对比，本文需要设计一种方案，其中隐藏常量小于内存大小的对数因子，对于合理的内存大小（例如，为了不经意地模拟大小为 1GB 的内存，需要一个 < 30 的常量）。在后续测试中，原文实现了这一目标，并获得比 Path ORAM 显著的加速。

原文进行了非常有用的观察，OptORAMa 中的大部分工作都是由于客户端只有 O(1) 块（安全）本地内存的限制。在很多激励场景中，并不需要如此强烈的要求。例如，当用户将 1TB（= 240 字节）的数据委托给云时，需要 1MB（= 220 字节）的本地存储是合理的。本文引入一个参数 Z，它捕获本地内存的大小，并将其视为 logN 的小多项式，其中 N 是内存的大小。原文认为，假设 Z 大致为内存大小的平方根是合理的。1MB（220 字节）足以存储 1TB（240 字节），32MB（225 字节）足以存储 1PB（250 字节），1GB（230 字节）足以存储 1EB（260 字节），这样的比率对于许多环境来说都是合理且合适的。

3.1 不经意的紧压缩算法

OptORAMA 的一个核心构建模块是一种不经意的紧压缩算法。在这个问题中，首先输入是一个数组，其中一些元素被区分，目标是输出一个具有相同元素集的数组，但所有区分

的元素都出现在开头。这可以看作是对不经意排序的放松，其中元素与 1 位键相关联。研究证明，如果元素是不可分的，那么稳定的不经意紧压缩需要对大小为 n 的输入进行 $(nlogn)$ 操作。[11] 的工作展示了一种非稳定的不经意紧压缩，其工作时间为 O $(nloglogn)$ 且错误概率可忽略不计。OptORAMa [1] 展示了一种确定性的不经意算法，该算法是渐近最优的并且只需要 O(n) 的工作；然而，隐藏常数很大（根据，至少为 2^{228}）。OptORAMA 的算法相当复杂，并使用扩展图和打包技巧。该常数在 [5] 中得到了改进，但仍然是数万，使其远未达到实际用途。

原文介绍了一种具有线性开销的不经意的紧密压缩算法，这个是第一个重要的结果，并假设客户端可以存储 $Poly\ logn$ 个单词（与 [1] 中的 $O(1)$ 相反）。该方案获得了渐进最优的线性开销，而且隐藏常数相对较小，约为 18。更重要的是，原文的 ORAM 的设计方式是不在其过程中使用不经意的紧压缩。具体来说，整个 ORAM 仅对数组使用压缩，其中（1）恰好有一半的元素被区分（并且对手知道这一事实，并且算法可以安全地“泄漏”它）；（2）输入数组是使用对对手隐藏的排列进行随机洗牌。以前从未研究过具有两种松弛的压缩，因为它不知道是否足以满足 ORAM。受到一般紧压缩算法的启发，作者获得了一种新的、极其高效的紧压缩算法在上述松弛下是安全的。具体来说，对于大小为 n 的输入数组，它需要 $4.2n$ 带宽。

3.2 遗忘的哈希表

分层 ORAM 由哈希表的层次结构（也称为“级别”）组成，其大小呈几何级数增加。最大的层可以容纳大约 N 个元素，即内存的大小。对 ORAM 的访问会转化为对每个级别的查找，并将找到的项目向上移动到最小级别。每当一个级别已满时，其内容基本上就会在层次结构中向下移动。一个重要的事实是，级别是通过固定的时间表合并和重建的，该时间表取决于访问次数。本文主要遵循 OptORAMA 的流程，但对上述流程的工作方式进行了一些关键修改，接下来将重点介绍其中的一些修改。

OptORAMA 中的层次构建如下。令 n 表示输入数组中的元素数量。n 个元素被（明确）路由到随机箱，然后每个箱中元素的（秘密）次线性部分被路由到称为溢出堆的二级结构。其余的箱称为主要箱。每个主要箱的大小都是多对数的，溢出堆的大小是 n 的次线性大小。OptORAMA 付出了巨大的努力来实现对主要箱的访问，并且他们的解决方案涉及非常昂贵的工具（包括不经意的 Cuckoo 散列、压缩和各种 SIMD 打包技巧）。作者利用本地存储足够大来存储整个主要垃圾箱这一事实，从而完全避免了这种复杂性。同时，作者使用不经意的布谷鸟散列，但构建过程是在本地内存中处理的，因此具体效率很高。

溢出堆的处理很大程度上与 OptORAMA 类似。由于它的大小是次线性的，因此实际上可以负担得起使用具有对数开销的已知哈希技术。与 OptORAMA 的主要且最显着的区别在于每个哈希表的提取操作。构造过程需要偶尔需要合并一些层次，为此，首先需要收集每个表中保留的元素，提取操作正是这样做的。当想要确保提取的元素是随机排列的时，实现这个操作就变得很重要（这是在整个构造过程中应该保持的不变量）。事实上，在 OptORAMA 中，作者只是调用通用压缩和洗牌算法。然而，这些原语实际上效率很低，因此本文希望避免使用它们。本文使用了一种具体有效的提取过程，该过程在非常高的水平上逆转了构建操作。这个想法以类似的形式 PanORAMa [12] 出现，但利用本地内存很大的事实进一步优化它。

3.3 散布

渐近最优 ORAM 构造中的另一个核心构建块是能够将两个随机打乱的数组分散到一个随机打乱的数组中，也就是说，它实现了以下抽象：

（1）输入：大小分别为 n0、n1 的数组 $A = A0\|A1$。假设输入数组中的每个元素都适合 O (1) 个内存字；

（2）输出：大小为 $n0 + n1$ 的数组 B，包含 $A0\|A1$ 中元素的随机排列。

上述过程背后的想法是不需要生成所有 $(n0 + n1)!$ 可能的输出；相反，该算法仅生成 $C_{n_0}^{n_0+n_1}$ 个输出作为可能的 Aux 阵列的数量。然后它根据 Aux 数组不经意地移动球；如果 $Aux[i] = 0$，则输出数组中的第 i 个位置将包含 A_0 中的元素（无需替换）。同样，如果 $Aux[i] = 1$，那么在第 i 个位置将有一个来自 A_1 的元素。可能的 Aux 阵列数量为 $\binom{n_0+n_1}{n_0}$；给定输入假设，其中两个输入数组被随机打乱，根据需要可能输出的实际数量为 $\binom{n_0+n_1}{n_0} \cdot n_0! \cdot n_1! = (n_0+n_1)!$。

4 复现细节

4.1 与已有开源代码对比

复现的工作已有源代码。源码中未包含具体 Path ORAM 的访问操作，本文相较于源代码添加了 Path ORAM 的具体实现，同时将源代码整合方便阅读，构造出基于计数以及真实的访问之间的输入选择，为完善实验数据对比提供更多可能。

4.2 实验参数介绍

本文将构造中的块大小设为 32 字节。

（1）逻辑存储器的总大小：它代表 ORAM 的总容量。由于块为 32 字节，因此大小为 X 的总逻辑内存，意味着球的数量 N 实际上是总大小除以 X。

（2）Z：箱的大小。本文在整个构建块中使用相同的尺寸（例如，箱压缩、散布、箱放置等）。Z 代表元素的数量，实际消耗的内存为 Z · 32 字节。

（3）ϵ：表示哈希表中移动到溢出堆的元素的比例。

（4）stashBound：这是在 Bin 内用于不适合其主 Cuckoo 哈希表的元素的存储大小。选择相对于 N 的箱大小 Z，经过证明很少的元素会以非常高的概率进入存储区。具体在实验中，使用 stashBound=9。

4.3 参数设置分析

（1）关于 ϵ 的大小。在溢出堆上，算法运行时间与 $n \cdot \epsilon log(n/Z)$ 成正比。因此，理论上，需要使用大约 $1/log\lambda \approx 1/logn$ 的 ϵ 来实际消除 $log(n/Z)$ 因子。然而，在表 4 中显示，虽然可以使 ϵ 更小，但确实发现了更小的开销。后续展示了不同大小的总逻辑内存大小在 $\epsilon = 1/10$、1/15 和 1/20 时的开销。至于往返，由查找引起的轮次（即 $logN/Z$ ）是主导因素，对构建过程的影响几乎可以忽略不计（$(logN/Z \cdot (6(1+\epsilon) + \epsilon log(\epsilon N/Z)) + 4.5)/Z$ ）

（2）关于 Z 的大小。[2] 期望的最好的开销是 $11.5 \cdot logN/Z$ 。因此，本文还展示了可以达到此开销的 Z、ϵ 的那些参数，权衡在于是否增加本地内存。

5 实验结果分析

本文实验测试是按如下方式进行：

(1) 初始化 ORAM 以分配 N 个块；

(2) 从 ORAM 中访问 N 个随机块进行读/写操作。这保证了底层至少被访问 N 次，并且至少被重建一次。

5.1 服务器上被访问的块数量 (平均)

本小节展示了一个实验，其中使用 Z=131,220 和 ϵ=1/10。该表显示了访问逻辑内存上的元素时服务器上正在访问的块数（平均）。然后将实施结果与预期的理论分析进行比较。

表 1. 一次请求需要访问的平均块数

内存大小	实际	理论
1GB	157	139
10GB	236	212
100GB	296	285
1TB	351	359
10TB	455	461

根据表 1，在 ORAM 的实际实现中，实际的结果与理论略有不同，主要是由于四舍五入级别数、每个级别中的箱数以及其他实现细节造成的。例如，当在关卡的构建中将球扔进干净箱时，我们必须读取箱中有多少元素，以便将元素写入箱中的正确位置。当内存较大时，实现甚至比数学分析稍好一些，因为理论分析中使用了上限约简了一些数。

5.2 不同块大小的带宽测试

表 2. 不同块大小的带宽测试

块的大小	复现方案		Path ORAM	
	往返读写	带宽	往返读写	带宽
32B	21.9	10.9KB	20	85.2KB
256B	17.8	67KB	10	136.6KB
1KB	15.9	229.6KB	6	392.2KB
4KB	13.7	772KB	4	1.2MB
256KB	7.7	24.4MB	2	48MB
1MB	6.2	111MB	2	172MB

表 2 展示了对于 1TB 大小的 ORAM，Z = 131,220 且 $\epsilon = 1/10$ 时的带宽在不同块大小时的表现结果。结果表明，无论是复现方案还是 Path ORAM 方案，随着数据块的增加，虽然往返读写次数会有所降低，但是带宽随之变大。因此如何确定块的大小将对性能有重要影响。总体来看，复现方案的效果明显超越 Path ORAM。

5.3 单个元素访问的平均字节数

表 3. 单次访问字节数对比

逻辑内存大小	Path ORAM	复现方案
1GB	27.2KB	4.93KB
10GB	45.5KB	7.38KB
100GB	67.0KB	9.25KB
1TB	82.9KB	10.96KB
10TB	113.4KB	14.23KB

表 3 是复现方案与 Path ORAM 的比较。报告访问不同逻辑内存大小的单个元素访问的平均字节数。对比数据不难发现，复现方案比 Path ORAM 的效果要好很多，在大内存下性能差距更大，复现与原文的效果是几乎一致的结果。

5.4 带宽和往返次数

表 4. 不同 ϵ 对性能的影响

ϵ	带宽	大小 (KB)
TotalMem=1GB, Z=131,220,LocalMem=8MB		
1/5	178	5.75
1/10	141	4.42
1/15	130	4.18
TotalMem=1TB, Z=262,440,LocalMem=16MB		
1/5	475	15.29
1/10	334	10.75
1/15	293	9.41
TotalMem=1PB, Z=393,660,LocalMem=24MB		
1/5	960	30.19
1/10	628	19.79
1/15	519	16.34

关于 ϵ 的大小。在溢出堆上，算法运行时间与 $n \cdot \epsilon log(n/Z)$ 成正比。因此，本文需要使用大约 $1/log\lambda \approx 1/logn$ 的 ϵ 来实际消除 $log(n/Z)$ 因子。在表 4 中展示了不同大小的总逻辑内存大小在 $\epsilon = 1/10$、$1/15$ 和 $1/20$ 时的开销。虽然 ϵ 可以逐渐变小，但复现结果显示出更小的开销。

表 5. 真实访问时间对比

内存大小	**Path ORAM(μs)**	**复现方案 (μs)**
100MB	1000	384

5.5 时间对比

前面对于大型的内存，都是只记录了访问请求的计数，未能真正在 cpu 中处理，表 5 展示了根据原文结构复现的真实 Path ORAM 结果，通过真实的内存访问测算出时间，可见复现方案比 Path ORAM 依然显示出合乎预期的优势。

6 总结与展望

本方案给出了一个渐进最优的分层 ORAM 设计，一改渐进最优与分层 ORAM 难以兼得的普遍观点。该实验的分层次 ORAM 思想对于 Path ORAM 的进一步优化着启发性指引。对于树状结构的 ORAM 依然具备分层次的概念，如此一来可以减少片上存储的空间开销。目前已经有部分工作在对这一方面进行优化，在未来，我们可以进一步对 Path ORAM 的架构深入探讨。

参考文献

[1] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: optimal oblivious ram. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30*, pages 403–432. Springer, 2020.

[2] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. Futorama: A concretely efficient hierarchical oblivious ram. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 3313–3327, 2023.

[3] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 668–679, 2015.

[4] Graeme Connell. Technology deep dive: Building a faster oram layer for enclaves, 2022.

[5] Sam Dittmer and Rafail Ostrovsky. Oblivious tight compaction in o (n) time with smaller constant. In *International Conference on Security and Cryptography for Networks*, pages 253–274. Springer, 2020.

[6] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.

[7] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194, 1987.

[8] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[9] S Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 513–524, 2012.

[10] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *Annual International Cryptology Conference*, pages 523–542. Springer, 2018.

[11] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the n log n barrier for oblivious sorting? In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2419–2438. SIAM, 2019.

[12] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious ram with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882. IEEE, 2018.

[13] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious {RAM}. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, 2015.

[14] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.

[15] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root oram: efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 218–234. IEEE, 2016.