Computer Security Symposium 2019
21 - 24 October 2019

複数キーワードによる検索が可能なOblivious RAM

市川 敦謙[1,a)]　濱田 浩気[1]

概要：Oblivious RAM (ORAM) は信頼できない外部サーバ等にクライアントがデータを保存し，また安全にアクセスを行える技術である．既存の ORAM はデータへのアクセス (検索) に用いるキーワードが 1 種類かつデータ毎に固有でなくてはならないという課題があった．本論文はこれを解決し，複数キーワードによる検索，および重複するキーワードによる検索が可能な ORAM を提案する．

キーワード：Oblivious RAM，複数キーワード検索

Oblivious RAM for Conjunctive Keyword

Atsunori Ichikawa[1,a)]　Koki Hamada[1]

Abstract: Oblivious RAM (ORAM) allows a client to securely access to data that stored in an untrusted server. Existing ORAMs require indices to be only one value for one record, and be unique. We propose an ORAM that enables each record to have two or more indices, or a non-unique index.

Keywords: Oblivious RAM, multi-index.

1. 背景

Oblivious RAM (ORAM) [3], [5], [6], [8], [9], [10] は，RAM(Random Access Memory) マシンをシミュレートする暗号学的プリミティブとして Goldreich らにより考案された [6]．これは単に安全な検索可能外部ストレージとして用いることができる他，秘密計算のインスタンスとして活用 [9], [10] するなど幅広く利用できる技術である．

ORAM はデータへの検索履歴 (アクセスパターン) を秘匿することを目的としており，この性質を用いれば，例えばマルチパーティ計算における配列アクセスの高速化 [11] や条件分岐の効率化 [7] といった，他技術との組み合わせにおいても有用となる技術である．

しかし既存の方式はいずれも，各データの索引 (検索キーワード) が単一かつ固有でなくてはならないという制約があり，例えば外部ストレージのようなユースケースを想定した場合，利便性に課題が残っていた．

そこで，本研究ではこの制約から脱し，単一でなく複数の索引を 1 つのデータに対して紐付けられるような方式を提案する．またその方式を拡張し，索引の重複が可能な方式を提案する．

2. 準備

2.1 RAM モデルと ORAM

RAM モデルにおいて，RAM マシンはランダムアクセス可能な記憶領域と，記憶領域へのアクセスを制御する CPU により構成される．RAM マシンは記憶領域に索引 v とデータ x の組 (v, x) を保存し，アクセスクエリ $(op, v, \widetilde{x})$ に従いデータの読み書きを行える．$op \in \{\mathsf{Read}, \mathsf{Write}\}$ は読み書きを定める命令であり，$\widetilde{x}$ は書き込み命令により書き込まれるデータである．v はデータ固有のキーワード，あるいはメモリアドレスそのものを表し，この値を元に RAM マシンは位置 v のデータ x を参照する．

以下では，多項式長のクエリの列 $\mathbf{Q} = ((op_1, v_1, \widetilde{x}_1), \ldots, (op_l, v_l, \widetilde{x}_l))$ を仮想アクセスパターンと呼ぶ．また，仮想アクセスパターン $\mathbf{Q}$ に従い参照さ

[1] NTT セキュアプラットフォーム研究所
NTT Secure Platform Laboratories
[a)] atsunori.ichikawa.nf@hco.ntt.co.jp

れる実体のアドレス列 $\widetilde{\mathbf{Q}}$ を実体アクセスパターンと呼ぶ．RAM マシンにおいては常に $\mathbf{Q} = \widetilde{\mathbf{Q}}$ である．

ORAM は RAM マシンのシミュレータであり，記憶領域の役割を (外部) サーバが，CPU の役割をクライアントがそれぞれ務める．ORAM の目的は，サーバの視点における実体アクセスパターンと仮想アクセスパターンの独立性を保ったまま，RAM マシンのシミュレーションを実現することである．

定義 2.1. *(略式)*. Ram を *RAM* マシンとする．*(確率的)RAM* マシン Oram が以下の *2* 点を満たすならば，Oram は Ram に対応する *oblivious RAM* である，と呼ぶ．

- 長さ l の任意の仮想アクセスパターン $\mathbf{Q}$ に対し，Ram と Oram が $1-negl(l)$ 以上の確率で等しい出力を行う．
- 長さ l の任意の仮想アクセスパターン $\mathbf{Q}, \mathbf{R}$ における，Oram の実体アクセスパターン $\widetilde{\mathbf{Q}}, \widetilde{\mathbf{R}}$ が計算量的に識別不可能である．

後者の条件については次のようにも帰着できる：実体アクセスパターン $\widetilde{\mathbf{Q}}$ と，それをシミュレートする乱択アルゴリズムを計算量的に識別できない．

また，ORAM の効率評価においては，データ操作の際に必要となる通信量のオーバーヘッドと，個々のデータ (およびその付加情報) のビット長の 2 つが評価軸とされる[*1]．データ操作に際しては，決められた数回のクエリ毎に 1 回，重い処理 (データ配列の再構成など) が必要になる方式が多く，そのため複数クエリを通した償却コストを評価に用いる．

以下では，データとその付加情報を合わせた塊を「データブロック」と呼び，そのビット長をブロックサイズと呼ぶ．また，(償却) 通信量またはコストと呼ぶ場合には操作中に通信されるビット長を，(通信量の) オーバーヘッドと呼ぶ場合には操作中に通信されるブロックの数を表すものとする．

2.2 ハッシュ

ハッシュ (hashing) とは，複数のデータをランダムな位置に格納するテーブル (配列) を作成し，また検索キー (索引) によってテーブルからデータを取得するためのアルゴリズム群である．

本論文では通常のハッシュと oblivious hashing と呼ばれるハッシュの 2 種類について，概要を紹介する．なお，本論文ではハッシュをブラックボックスとして扱い，特定の方式を指定しないものとするが，具体例として cuckoo hashing [5], [8] と two-tier hashing [1], [3] の 2 つを効率の評価に用いる[*2]．

$\mathcal{H}$ が次の 3 つのアルゴリズムを含む時，これを (n, μ, s)-ハッシュ[*3]であると呼ぶ：鍵生成 $\kappa \leftarrow \mathcal{H}.\mathbf{Gen}(n)$，テーブル生成 $(T, S) \leftarrow \mathcal{H}.\mathbf{Build}(\kappa, X)$; $|T| = \mu, |S| = s, |X| = n$，参照 $P \leftarrow \mathcal{H}.\mathbf{Lookup}(\kappa, v)$．ここで，$T$ はハッシュテーブルと呼ばれるサイズ $\mu(\geq n)$ の配列，S はスタッシュと呼ばれるサイズ s の配列，$X = ((v_1, x_1), \ldots, (v_n, x_n))$ はテーブルへ格納されるデータ列である．また P はデータバケツと呼ばれる，少数のデータが格納できる配列であり，P の中に探索対象 (v, x) が存在する．

さらに，$\mathcal{H}$ が次の 2 点を満たすならば，これを oblivious hashing と呼ぶ．

- 任意の鍵とデータの組 (κ, X) および (κ', X') に対し，$\mathcal{H}.\mathbf{Build}$ を実行した際の (実体) アクセスパターンが識別不可能である．
- 相異なる l 個の索引の列 $\mathbf{v}$，および $\mathbf{v}'$ に対し，$\mathcal{H}.\mathbf{Lookup}$ を実行した際の出力が計算量的に識別不可能である．

2.3 Privacy Information Retrieval

Privacy Information Retrieval (PIR) とは，外部サーバ等に保存されたデータ列 $(x_1, \ldots, x_n)$ に対し，$i \in [n]$ [*4]をサーバに対して秘匿したまま x_i を取得する技術である．

本研究では特に，distributed point function [4] と呼ばれる，ある特定の値を入力した時のみ正規の出力を与え，それ以外では 0 を出力するような関数を用いた PIR [2] を利用する (詳細は省略する)．

3. 先行研究：複数サーバによる ORAM

本章では，本研究の前提となる既存の oblivious RAM として [9] を紹介する．[9] は，効率のよい階層的 ORAM である [8](および [3]) に対し，複数サーバを用いた構造 [10] を適用し，さらに PIR [2] を導入した方式である．

階層的 ORAM とは，その名の通り複数のハッシュテーブルが階層的に積み上がり，ピラミッド状のデータ構造を成している ORAM である．多くの方式では，最上段には一定のサイズをもつ配列が置かれ，以下 (上から数えて)$i = 1, \ldots, L$ 段目にはハッシュテーブルが置かれる．i 段目に置かれたハッシュテーブルは，最上段配列および $1, \ldots, i-1$ 段目の全てのデータが格納できる．なお全てのテーブルは安全性のため oblivious hashing により作成される．

[8] は階層的 ORAM の 1 つであり，以下の構造を持つ．

[*1] より詳細には，クライアントの記憶領域やサーバのローカル計算量なども評価対象であるが，既存/提案方式間で大きな差が生じないため省略する．

[*2] それぞれの詳細は省略する．

[*3] 以下，既存/提案方式の説明においては簡単のためパラメータを省略し，単にハッシュと呼ぶ．

[*4] 以下では，$[n] = \{1, \ldots, n\}$ とする．

- k, d をパラメータとする．最上段配列は k 個までのデータブロックを格納でき，各 i 段目には kd^{i-1} 個までのデータブロックを格納するハッシュテーブルが $d-1$ 個置かれる．

- データアクセスの際は，最上段配列から初め，各段のハッシュテーブルへ索引 v によりアクセスする．ただし，一度データ (v, x) が発見されれば，以後のテーブルへはダミーの索引でアクセスする．読み出されたデータは最上段配列の空いている場所へ再配置される．

- 最上段配列，および i 段目までのハッシュテーブルが全て埋まった場合には，含まれるデータを全て $i+1$ 段目の空いているハッシュテーブルへ再配置する (この操作を再シャッフルと呼ぶ)．空いているハッシュテーブルは，実行されたクエリの総数 t により一意に判別できる．i 段目以下のデータの総数は $k + \sum_{j=1}^{i}(d-1) \times kd^{j-1} = kd^i$ であるため，$i+1$ 段目のテーブル 1 つに全て格納できる．

[10] は，階層的 ORAM の各階層を 2 台のサーバで互い違いに所持する方式である．別の言葉を用いれば，各 i 段目のハッシュテーブルに対する役割を「所持する役」「再シャッフルする役」で分担する手法である．例えば，ある i 段目のテーブルを所持しない (すなわち i 段目へのデータアクセスに関与しない) サーバは，代わりに i 段目のテーブルを作成する役割を果たす．この方式はサーバ 2 台が結託しない限り，通常のハッシュを用いても oblivious hashing と同等の安全性を獲得しうるという利点がある．ただし $\mathcal{H}$.**Build** および $\mathcal{H}$.**Lookup** の識別不可能性のため，索引 v と「エポック」と呼ばれるテーブル固有の値 e を擬似ランダム関数 $\mathbf{PRF}_s$ で写像した，「タグ」と呼ばれる値をデータに紐付け，これを索引の代わりに用いる必要がある．全てのハッシュテーブル関して，エポックはクライアントが所持し，サーバへ秘匿する．

[9] はここに加え，各 i 段目を所持するサーバを 2 台に増やし，テーブルを複製して所持させることで，PIR によるデータ取得を可能としている．この実現方法は [9] にて 2 つ提案されているが，ここでの紹介は割愛する．

4. 複数の索引で検索可能な ORAM ～PIR を用いない場合～

以下に，前章にて紹介した複数サーバによる ORAM を拡張し，複数の種類の索引を用いたデータアクセスを可能とする ORAM を提案する．本章では，まず PIR を用いないシンプルな構成法を述べる．

4.1 準備

初めに，本方式におけるデータは $(v_1, \ldots, v_m, x)$ と表されるとする．各 $v_p (p \in [m])$ は索引，x はデータ本体である．

本章では，任意の 2 つのデータ $(v_1^{(1)}, \ldots, v_m^{(1)}, x^{(1)})$, $(v_1^{(2)}, \ldots, v_m^{(2)}, x^{(2)})$ について $v_p^{(1)} \neq v_p^{(2)}$ が成り立つ，すなわち同じ種類に属する (同じ添字番号の) 索引はそれぞれ重複を持たないとする[*5]．

クエリ (仮想アクセスパターン) は，索引の列 $(v_1^q, \ldots, v_m^q)$ と $op, \widetilde{x}$ によって表される．ただし任意の $p \in [m]$ について，$v_p^q = \perp$ としてクエリすることができるとする．既存の ORAM とクエリの構造が異なるため，厳密には安全性定義の拡張が必要であるが，既存の安全性定義における v を長さ m のベクトルと見なせば全く同様の議論が可能であるため，本論文では省略する．

また，N を ORAM へ格納されるデータ数 (の最大値)，t を総クエリ回数のカウンタ，k, d をそれぞれ定数とし，$L = \log_d N$ とする．$\mathcal{H}$ を oblivious でない通常のハッシュ，$\mathbf{PRF}_s$ を擬似ランダム関数とする．

4.2 索引が 2 種類の場合の構成

まずは簡単のため，各データは索引を 2 種類まで持つ，すなわち (v_1, v_2, x) と表せるとする．

この構成では，2 台のサーバ $\mathcal{S}_0, \mathcal{S}_1$ と 1 台のクライアントにより ORAM の読み出し・書き込みプロトコルが実行される．$\mathcal{S}_0, \mathcal{S}_1$ はデータ格納用のハッシュテーブル $T_{i,j}$ を，階層 i ごとに交互に持つ．$i = 1, \ldots, L$, $j = 1, \ldots, d-1$ とし，$T_{i,j}$ は i 段目における j 番目のテーブルであることを示す．ここでは一般性を失わず，偶数段のテーブルを $\mathcal{S}_0$ が，奇数段のテーブルを $\mathcal{S}_1$ が持つと仮定する．この $T_{i,j}$ は第 1 索引 v_1 を用いてデータを探索するために使用される．

本方式ではさらに，$T_{i,j}$ と対となるハッシュテーブル $U_{i,j}$ を用いる．$U_{i,j}$ は $T_{i,j}$ を所持していない側のサーバ (例えば i が偶数なら $\mathcal{S}_1$) が持つ．$U_{i,j}$ は第 2 索引 v_2 を用いて対応する第 1 索引 v_1 を取得する目的に使用され，これにより v_2 単体での検索 (すなわち $v_1 = \perp$ であるクエリ) を可能とする．

詳細な提案方式を Algorithm 1,2 に示す．

Algorithm 1 はデータの読み書きを行う手続きである．クライアントはローカルに保存されたエポック $e_{i,j}^T, e_{i,j}^U$ とクエリ索引 v_1^q, v_2^q を用いてタグを生成し，ハッシュテーブルへの検索を行う．

その際，まず初めにテーブル $U_{i,j}$ に対して検索を行うことで，第 2 索引 v_2^q と対応する v_1 を (もしあれば) 手に入れる．その後，v_1^q もしくは v_1 を用いることで $T_{i,j}$ の探索を行う．ただし検索の過程で一度データがヒットしたならば，以後の検索は全てダミーを用いて行われる．

なお，クエリされたダミーではない索引 v_1^q, v_2^q と，ORAM へ保存されている実データの索引 v_1, v_2 との間に片側不一

[*5] $v_p^{(1)} = v_q^{(1)}, v_p^{(1)} = v_q^{(2)}; p \neq q$ は成り立ちうる．

致が発生してしまった場合 (すなわち $v_1^q \neq v_1$ かつ $v_2^q = v_2$ など)，次のいずれかの動作が可能である．

i) 片側一致を許容し，データを読み出す．

ii) 不正なクエリと見なしてデータを読み出さない．

Algorithm 1 の記述では，このうち i) に相当する動作を行っている．ii) を行う場合には，まず Algorithm 1 の 5〜10 行目で発見された (v_2, v_1) に対して $v_1^q = v_1$ の判定を行い，さらに 11〜16 行目で発見された $(v_1, (v_2, x))$ に対し，改めて $v_1 = v_1^q (\text{or } l) \wedge v_2 = v_2^q$ の判定を行えばよい．その際，不正なクエリと判定された場合は，ORAM の安全性のため $(v_1^q, v_2^q, m) \leftarrow (\text{“}dummy\text{”} \circ t, \text{“}dummy\text{”} \circ t, \text{“}dummy\text{”})$ とする．これらは全てクライアントによってのみ制御される動作であり，サーバの視点から動作選択の識別は不可能である．

Algorithm 2 は，クエリ k 回ごとに一度実行されるハッシュテーブル生成アルゴリズムである．最上層配列と i 段目以下の全てのハッシュテーブルが埋まった際に，その全てのデータを新たなハッシュテーブル $T_{i+1,j}$ へと格納する．また，$T_{i+1,j}$ に対応する $U_{i+1,j}$ も新たに作成する．

初めに両サーバはクライアントを介して全てのデータを 1 つの配列に統合し，空のデータを削除しつつテーブル $T_{i+1,j}$ のシャッフル役である $\mathcal{S}_{1-z}$ へ渡す．同時にクライアントは組 (v_2, v_1) を抽出し，後にテーブル $U_{i+1,j}$ のシャッフル役である $\mathcal{S}_z$ へ渡す．両サーバはそれぞれ担当の $T_{i+1,j}, U_{i+1,j}$ を生成し，クライアントを介して対岸のサーバへと受け渡す．

4.3 安全性

Algorithm 1,2 について，以下が成り立つ[*6]．ここでは，便宜上全ての $T_{i,j}$ の集合を階層 T，$U_{i,j}$ の集合を階層 U と呼ぶ．

補題 4.1. 各アルゴリズムの実行中，全てのステップにおいて，$v_p = \text{“}empty\text{”}$ を除く任意の $(v_p, *)$ は必ず，階層 T, U それぞれに最大 *1* つだけ存在する．

証明．v_p は実データの索引・ダミー・空のいずれかに属する．このうち，実データの索引については，Algorithm 1 において階層 T ないし U をまず探索することで重複を回避できている．既に T ないし U に v_p が存在する場合，必ず探索により発見され当該のデータが最上段へ再配置されるためである．また，ダミーの v_p についてはクエリカウンタ t により重複が起こらない． □

補題 4.2. 全ての階層，全てのテーブルにおいて，*1* つのテーブルに同じ v_p がクエリされることはない．

証明．v_p がダミーである場合，クエリカウンタ t により二度同じ値がクエリされることはない．もし v_p がダミーでないならば，データ発見以前に探索された全てのテーブル $T_{g,h}$ に v_p がクエリされたこととなる．しかし以後，データは最上段配列へと配置され，再シャッフルが起こるまで下層のテーブルへ格納されることはない．よって $T_{g,h}$ の再構成 (再シャッフル) が起こるまでは，常に $T_{g,h}$ よりも上層で v_p のデータが発見されるため，再度 v_p がクエリされることはない (ダミーがクエリされる)． □

定理 4.3. 一方向性関数とハッシュ $\{\mathcal{H}_i\}_{i=1}^{L}$ が存在し，かつサーバ同士が結託しないという仮定のもとで，提案方式は *ORAM* である．

証明．n 個のクエリ $q_1, \ldots, q_n$ を仮定し，各サーバが Algorithm 1,2 の実行の過程で，結託せずにクライアントとシミュレータの識別を試みるとする．

データアクセスの際に各サーバが受信するのはタグ $\tau_{i,j}^{\{T,U\}} \leftarrow \mathbf{PRF}_s(e_{i,j}^{\{T,U\}}, v_p^q)$ か，またはシミュレータが生成するランダムな値である．$e_{i,j}^{\{T,U\}}$ がテーブル毎に固有かつ再シャッフルの際に再生成されること，および補題 2. から，サーバが受信する全てのタグは相異なる $(e_{i,j}^{\{T,U\}}, v_p^q)$ により生成される．よって擬似ランダム関数の識別不可能性からシミュレータとクライアントの識別は困難．

次に再シャッフルの実行時，各サーバが得られる情報について考える．

初めに，あるテーブル $T_{i,j}$ または $U_{i,j}$ を再シャッフルする側のサーバの視点を考える．再シャッフル時に得る情報は，ランダムに並べ替えられ再暗号化されたデータ (実データまたはダミー) と，それに紐付けられた新規のタグである．タグの生成には新規に選ばれたエポックと索引 v_p が用いられ，補題 1. より全ての v_p が固有の値を持つことから，タグとランダムな値との識別は不可能である．また，再シャッフルされるデータの個数は t から一意に定まるため，シミュレータとクライアントの識別を補助する情報は得られない．したがって再シャッフルを行うサーバにはシミュレータの識別は困難であると言える．

次に，再シャッフルを行わない側のサーバについてだが，こちらは他のサーバが作成した暗号文の列 (ハッシュテーブル) を受け取るのみである．ハッシュテーブルに含まれるデータの個数と空の個数は t より一意であるため，シミュレータがこれと区別のつかないテーブルを生成することは容易である．

以上により，Algorithm 1,2 の実行において，結託しない各サーバはクライアントとシミュレータを識別不可能であると示された． □

4.4 3 種以上の索引への拡張

各データが $2 < m$ 個の索引を持つ，すなわち $(v_1, \ldots, v_m, x)$ と表される場合の，提案方式の拡張方法を示す．

基本的なアイデアは前述のテーブル $U_{i,j}$ を，索引の種類

[*6] 本論文では，正当性についての議論は省略する．

Algorithm 1 データアクセス

Require: $\mathcal{S}_0, \mathcal{S}_1$ はそれぞれデータ階層の最上層に長さ $k/2$ の配列と，スタッシュ S^T, S^U を持つ．以後，i 段目のハッシュテーブル $T_{i,j}$ を $\mathcal{S}_{i \mod 2}$ が，$U_{i,j}$ を $\mathcal{S}_{i-1 \mod 2}$ が持つ $(j = 1, \ldots, d)$．クライアントは各ハッシュテーブルに対応するエポック $e^T_{i,j}, e^U_{i,j}$ を持つ．

1: クライアントはデータ格納用の一時記憶領域 ϕ, π を用意し，$\phi =$ "$dummy$" $\circ\, t, \pi =$ "$dummy$" と初期化する．
2: 2 値フラグ found $\leftarrow 0$ と初期化する．
3: クエリの中に $v^q_p = \perp$ があれば $v^q_p \leftarrow$ "$dummy$" $\circ\, t$ とする．以後クエリと合致するデータ (v_1, v_2, x) が発見された際には $v^q_p \leftarrow v_p$ とする．
4: $\mathcal{S}_0, \mathcal{S}_1$ から最上層配列の要素 (v_1, v_2, x) を順に受け取る．$v_1 = v^q_1, v_2 = v^q_2$ となる要素があれば $\pi \leftarrow x$, found $\leftarrow 1$ とする．ただしダミークエリは一致判定から除外する．
5: $\mathcal{S}_0, \mathcal{S}_1$ からスタッシュ S^U の要素 (v_2, v_1) を順に受け取る．found $= 0$ かつ $v_2 = v^q_2$ ならば $\phi \leftarrow v_1$, found $\leftarrow 1$ とする．
6: 各テーブル $U_{i,j}$ に対して繰り返す：
7: もし found $= 0$ ならば，タグ $\tau^U \leftarrow \mathbf{PRF}_s(e^U_{i,j}, v^q_2)$ とする．
さもなくば $\tau^U \leftarrow \mathbf{PRF}_s(e^U_{i,j},$ "$dummy$" $\circ\, t)$ とする．
8: クライアントは $\mathcal{S}_{i-1 \mod 2}$ に τ^U を送信し，$\mathcal{S}_{i-1 \mod 2}$ はデータバケツ $P_{i,j} \leftarrow \mathcal{H}_i.\mathbf{Lookup}(\tau^U, \kappa^U_{i,j})$ を得る．
9: クライアントはバケツ $P_{i,j}$ の要素 (v_2, v_1) を順に受け取り，再暗号化して返送する．
ただし $v_2 = v^q_2$ ならば，$\phi \leftarrow v_1$, found $\leftarrow 1$ とした上で，v_2 を "$Hashdummy$" $\circ\, t$ で上書きして返送する．
10: 繰り返し終了
11: $\mathcal{S}_0, \mathcal{S}_1$ からスタッシュ S^T の要素 $(v_1, (v_2, x))$ を順に受け取る．found $= 0$ の時，$v_1 = v^q_1$ となる要素があれば $\pi \leftarrow x$, found $\leftarrow 1$ する．found $= 1$ の時，$v_1 = \phi$ となる要素があれば $\pi \leftarrow x, l \leftarrow$ "$dummy$" $\circ\, t$ とする．
12: 各テーブル $T_{i,j}$ に対して繰り返す：
13: もし found $= 0$ ならば，タグ $\tau^T \leftarrow \mathbf{PRF}_s(e^T_{i,j}, v_1)$ とする．
さもなくば $\tau^T \leftarrow \mathbf{PRF}_s(e^T_{i,j}, \phi)$ とする．
14: クライアントは $\mathcal{S}_{i \mod 2}$ に τ^T を送信し，$\mathcal{S}_{i \mod 2}$ はデータバケツ $Q_{i,j} \leftarrow \mathcal{H}_i.\mathbf{Lookup}(\tau^T, \kappa^T_{i,j})$ を得る．
15: クライアントはバケツ $Q_{i,j}$ の要素 $(v_1, (v_2, x))$ を順に受け取り，再暗号化して返送する．
found $= 0$ かつ $v_1 = v^q_1$ ならば，$\pi \leftarrow x$, found $\leftarrow 1$ とし，v_1 を "$Hashdummy$" $\circ\, t$ で上書きして返送する．
found $= 1$ かつ $v_1 = \phi$ ならば，$\pi \leftarrow x, \phi \leftarrow$ "$dummy$" $\circ\, t$ とし，v_1 を "$Hashdummy$" $\circ\, t$ で上書きして返送する．
16: 繰り返し終了
17: 必要ならば，読み出した π を他の値で上書きする．
18: $\mathcal{S}_0, \mathcal{S}_1$ のスタッシュおよび最上層配列を順に呼び出し，v^q_1, v^q_2 と合致するデータがあれば，データ本体 x を π によって上書きする．
19: 上記のデータ上書きが起こらなかった場合は (v^q_1, v^q_2, π) を，起こった場合は ("$dummy$" $\circ\, t$, "$dummy$" $\circ\, t$, "$dummy$") を，最初に現れた空データに対して上書きする．
20: クエリカウンタ t をインクリメントし，もし t が k の倍数であれば次に示す再シャッフルを行う．

Algorithm 2 再シャッフル：テーブル $T_{i+1,j}, U_{i+1,j}$ の作成

$\mathcal{S}_z$ が $T_{i+1,j}$ を，$\mathcal{S}_{1-z}$ が $U_{i+1,j}$ を持つものとする．$(z = i + 1 \mod 2)$
1: 両サーバは $g = 1, \ldots, i, h = 1, \ldots, d-1$ である全ての $U_{g,h}$ を削除する．
2: 両サーバは十分な大きさの配列を一時的に確保し，自身の持つ全ての $T_{g,h} (g \leq i)$，スタッシュ S^T および最上層配列の要素を格納する．
3: $\mathcal{S}_{1-z}$ は作成した一時配列に対してランダムな置換を行い，要素を 1 つずつクライアントへ送信する．
4: クライアントは送られる要素を再暗号化しつつ $\mathcal{S}_z$ へ送信し，$\mathcal{S}_z$ はこれを自身の一時配列へ結合する．
5: $\mathcal{S}_z$ は同様に配列へランダム置換を行い，要素を 1 つずつクライアントへ送信する．
6: クライアントは $T_{i+1,j}$ の新たなエポック $e^T_{i+1,j}$ を作成し，$\mathcal{S}_z$ から送られる "空ではない" データ $(v_1, (v_2, x))$ に対してタグ $\mathbf{PRF}_s(e^T_{i+1,j}, v_1)$ を付与する．この時には，ダミーデータは実データと同様に扱われ，空データのみが削除される．
7: クライアントはタグ付けしたデータ $(v_1, (v_2, x))$ を再暗号化して $\mathcal{S}_{1-z}$ へ送信する．また同時に，組 (v_2, v_1) を暗号化して $\mathcal{S}_{1-z}$ へ送る．
8: $\mathcal{S}_{1-z}$ はクライアントから，$d^i(k+s)$ 個のタグ付けされたデータ $(v_1, (v_2, x))$ と組 (v_2, v_1) を受け取る．まず $\mathcal{S}_{1-z}$ は $T_{i+1,j}$ を作成するため，ハッシュ鍵 $\kappa^T_{i+1,j} \leftarrow \mathcal{H}_{i+1}.\mathbf{Gen}(N)$ を生成し，$(T_{i,j}, S^T) \leftarrow \mathcal{H}_{i+1}.\mathbf{Build}(\kappa^T_{i+1,j}, X)$ を実行する (X はタグ付けされたデータ列)．もしハッシュテーブル作成に失敗した場合は新たな鍵を生成する (無視できるほど小さな確率で起こる)．
9: $\mathcal{S}_{1-z}$ はクライアントに向けてハッシュ鍵 $\kappa_{i+1,j}$ と，スタッシュ S^T へ格納された実データ数 σ を伝えた上で，$T_{i+1,j}, S^T$ の要素を 1 つずつ送信する．またさらに，(v_2, v_1) の配列にランダムな置換を施した上で，要素を 1 つずつクライアントへ送信する．
10: クライアントは $\mathcal{S}_z$ に $\kappa_{i+1,j}$ を送信し，その後 $T_{i+1,j}, S^T$ の要素を 1 つずつ再暗号化して送る．ただし，以下のデータのみ変更を加える．
(a) 最初の σ 個の空データは，("$Stashdummy$" $\circ\, r$, ("$Stashdummy$" $\circ\, r$, "$empty$")) とする．r は 1 回ごとにインクリメントする．
(b) その後の空データは空のまま，("$empty$", ("$empty$", "$empty$")) として暗号化される．
(c) S^T の空データは全て ("$Stashdummy$" $\circ\, r$, ("$Stashdummy$" $\circ\, r$, "$empty$")) とする．r は 1 回ごとにインクリメントする．
11: クライアントは続けて $U_{i+1,j}$ の新たなエポック $e^U_{i+1,j}$ を作成し，各 (v_2, v_1) に対してタグ $\mathbf{PRF}_s(e^U_{i+1,j}, v_2)$ を付与して $\mathcal{S}_z$ へ送信する．
12: $\mathcal{S}_z$ はクライアントから受信した $T_{i+1,j}$ を $i+1$ 段目・j 番目のテーブルとして保存し，また S^T を最上層へ保存する．次に $U_{i+1,j}$ を作成するため，$\kappa^U_{i+1,j} \leftarrow \mathcal{H}_{i+1}.\mathbf{Gen}(N), (U_{i+1,j}, S^U) \leftarrow \mathcal{H}_{i+1}.\mathbf{Build}(\kappa^U_{i+1,j}, Y)$ を実行する (Y はタグ付けされた配列)．テーブル作成に失敗した場合は鍵の再生成を行う．
13: $\mathcal{S}_z$ は $T_{i+1,j}$ と同様にしてクライアントへ $\kappa^U_{i+1,j}, \sigma, U_{i+1,j}, S^U$ を送信し，クライアントは必要に応じて変更を加えつつこれを $\mathcal{S}_{1-z}$ へ送る．
14: $\mathcal{S}_{1-z}$ は $i+1$ 段目・j 番目のテーブルとして保存し，また S^U を最上層へ保存する．

ごとに分けて複数作成することである．上記の方式では，$U_{i,j}$ は写像 $v_2 \mapsto v_1$ としての役割を果たしていた．同様に全ての $p = 2, \ldots, m$ に対して $v_p \mapsto v_1$ という写像を作成することで，m 種類の索引に対応可能となる．

各 v_p に対応する $U^p_{i,j}$ は，テーブル $T_{i,j}$ を所持していないサーバ (前述の例では $\mathcal{S}_{i-1 \mod 2}$) がまとめて持つものとする．

4.5 方式の効率

上記の提案方式の効率について，(償却) オーバーヘッドコストとブロックサイズを考察する．

まずデータアクセスの際に発生する通信について，第 2〜第 m 索引に関する通信量と，第 1 索引に関する通信量を分けて考える．各索引が高々 $\log N$ ビットであることを踏まえれば，第 2 索引の検索コストは $2\log N \times (k+s+\sum_{i=1}^{L}(d-1)C_{\mathbf{Lookup}}(\mathcal{H}_i))$ となる．ただし，s はスタッシュのサイズ (格納ブロック数)，$L = \log_d N$ は ORAM の階層数，$C_{\mathbf{Lookup}}$ は 1 回のテーブル参照で通信されるブロック数である．$\alpha = \max_i C_{\mathbf{Lookup}}(\mathcal{H}_i)$ とおくと，$\sum_{i=1}^{L}(d-1)C_{\mathbf{Lookup}}(\mathcal{H}_i) \leq \alpha(d-1)L$ より，通信量は $\leq 2\log N \times (s + \alpha(d-1)L)$ となる．

同様に，$m-1$ 個の索引の検索にかかる通信量は $\leq 2(m-1)\log N \times (k+s+\alpha(d-1)L)$ と押さえることができる．

一方で第 1 索引の検索コストは，データ $(v_1, \ldots, v_m, x)$ のビット長を Λ とすると，$\Lambda \times (k+s+\sum_{i=1}^{L}(d-1)C_{\mathbf{Lookup}}(\mathcal{H}_i))$ と表せる．したがって第 2 索引と同様の議論により，通信量を $\leq \Lambda \times (k+s+\alpha(d-1)L)$ と押さえられる．

次に，再シャッフルで発生する償却通信量を考える．ハッシュテーブル $T_{i,j}$ および $U_{i,j}$ に対する再シャッフルは，クエリ kd^{i-1} 回ごとに発生する．ハッシュ $\mathcal{H}_i$ によって生成されるテーブルのサイズを μ_i とすると，再シャッフルによる償却通信量は $O((\Lambda + (m-1)\times 2\log N)\sum_{i=1}^{L}\frac{\mu_i}{kd^{i-1}})$ となる．

ここで $\Lambda = \Omega(m\log N)$ であることを踏まえると，本提案方式のオーバーヘッドはブロックサイズを $\Omega(m\log N)$ として，

$$O\left(k+s+\alpha(d-1)L+\sum_{i=1}^{L}\frac{\mu_i}{kd^{i-1}}\right)$$

となる．

$\mathcal{H}$ を cuckoo hashing とすると，[5], [8], [9] から，$\alpha = 2, \mu_i = \Omega(\log^7 kd^{i-1}), s = \Theta(\omega(1)\cdot \log N/\log\log N)$ となるため，オーバーヘッドは $O((d+\omega(1))\log_d N)$ となる．特に，$d = \log^\delta N, \delta \in (0,1)$ とすれば，オーバーヘッドは

$$O\left((\omega(1)+\log^\delta N)\cdot\frac{\log N}{\log\log N}\right)$$

となる．

また，$\mathcal{H}$ を two-tier hashing とすると，[1], [3], [9] から，$\alpha = O(\log^\epsilon N)$(ただし $\epsilon \in (0.5, 1)$), $\mu_i = O(kd^{i-1}/\alpha), s = 0$ となるため，オーバーヘッドは $O(d\log^\epsilon N \log_d N)$ となる．特に，$d = \log^\delta N, \delta \in (0,1)$ とすれば，オーバーヘッドは

$$O\left(\frac{\log^{1+\delta+\epsilon} N}{\log\log N}\right)$$

となる．

5. 複数の索引で検索可能な ORAM 〜PIR を用いる場合〜

前章の方式は，[9] のように PIR を用いた構成へと変換することができる．本節にその概略を示す．

前提として，distributed point function に基づく PIR[2] には，2 つのサーバが複製された同一の配列を所持していることが必要である．したがって 2 サーバ構成である本提案方式においては，両サーバが互い違いに所持する $T_{i,j}, U_{i,j}$ を全て両者が所持するように変更するといった方法が考えられる．

ところが単純にテーブルの複製を試みた場合，アクセスパターンが識別可能となってしまう問題がある．シャッフル役のサーバにとっては，各データのタグ (すなわち仮想アドレス) とテーブル上の (実体) アドレスの関係性が既知なためである．そこで本研究では，[9] と同様に oblivious hashing を利用してこの問題の解決を図る．

なお，PIR を用いる点，通常のハッシュ方式ではなく oblivious hashing を用いている点を除けば，前章のアルゴリズムとの差異はほとんどないことから，本章の方式については擬似コードの記述や安全性の議論を省略する．[*7]

5.1 データアクセス

まず初めにデータアクセスアルゴリズムについて述べる．前述の方式との変更点は，スタッシュやテーブルを探索する際に，探索対象となるバケツを線形探索するのではなく，代わりに「ヘッダ」と呼ぶ情報の列を一括で受信し，必要なデータのみを PIR によって取得する点である．

具体的には，$U_{i,j}$ や S^U における索引 v_2，$T_{i,j}$ や S^T における索引 v_1 がヘッダにあたる．各 i 段目において，クライアントは全ての $d-1$ 個のテーブルについて，読み出されたバケツ内のヘッダを受信し，クエリ v^q_p と一致したヘッダの位置 w を探索する．この時，取得するべきデータが見つからなければ，無作為に w_{rnd} を選択する．

そして位置 w(または w_{rand}) を入力として，全てのバケツを連結した配列に対して PIR を実行し，データを取

[*7] 安全性については，前章にて議論した内容に PIR と oblivious hashing のアクセス秘匿性および正当性が加味されるのみである．

得する．その後サーバにヘッダを返送するが，その際に適切な w を得られていた場合には，位置 w のヘッダを “$Hashdummy$” $\circ t$ によって上書きする．

5.2 再シャッフル

次に再シャッフルアルゴリズムについて，oblivious hashing $\mathcal{H}^o$ を用いて行う方法を述べる．前章の方法と違い，両サーバは全く同じテーブルを複製し所持しているため，階層 i ごとにそれぞれの役割を交換する必要がない．ここでは仮に，$\mathcal{S}_0$ がアルゴリズムの起点を担うとして記述する．

ハッシュテーブル $T_{i+1,j}, U_{i+1,j}$ を作成するにあたり，まず前述の方法と同様に i 段目以下の全ての $U_{g,h}$ および S^U を削除する．次に，$\mathcal{S}_0$ がクライアントを経由して $\mathcal{S}_1$ へ，$T_{g,h}, S^T$ および最上層配列の全てのデータ $(v_1, (v_2, x))$ を受け渡す．その際，空のデータは全てクライアントにて削除する．また，この時に $U_{i+1,j}$ の要素となる (v_2, v_1) もクライアントが抽出し，$\mathcal{S}_1$ へ送る．

続いて，$\mathcal{S}_1$ はデータのヘッダ v_1 の列，および (v_2, v_1) のヘッダ v_2 の列をそれぞれ，クライアントを介して $\mathcal{S}_0$ へ受け渡す．この 2 つのヘッダ列に対して，クライアントはハッシュ鍵 $\kappa^T_{i+1,j}, \kappa^U_{i+1,j}$ を生成し，$\mathcal{S}_0$ と共に $\mathcal{H}^o_{i+1}$.**Build** を実行してヘッダのテーブル $(\hat{T}_{i+1,j}, \hat{S}^T), (\hat{U}_{i+1,j}, \hat{S}^U)$ を作成する．

その後，クライアントは $(\hat{T}_{i+1,j}, \hat{S}^T), (\hat{U}_{i+1,j}, \hat{S}^U)$ の要素を順番に受け取りながら，全ての要素にタグ $\mathbf{PRF}_s(e^{\{T,U\}}_{i+1,j}, v_p)$ を付けてゆく．空のレコードに対しては数 y をインクリメントさせながら $\mathbf{PRF}_s(e^{\{T,U\}}_{i+1,j}, \text{“}empty\text{”} \circ y)$ を付与する．y の最大値 Y は i から一意に算出可能である．

同時に $\mathcal{S}_1$ は，自身の持つ $(v_1, (v_2, x))$ の列および (v_2, v_1) の列に対して，Y 個の空データ $(\text{“}empty\text{”} \circ y, (\text{“}empty\text{”} \circ y, \text{“}empty\text{”}))$ および $(\text{“}empty\text{”} \circ y, \text{“}empty\text{”})$ を挿入し，ランダムな置換を行う．

最後に，$\mathcal{S}_1$ は各データの列をクライアントを介して $\mathcal{S}_0$ へ受け渡す．その際，クライアントは各データにタグを作成し付与することで，$\mathcal{S}_0$ の持つ各テーブルへのデータ格納が可能となる．その後，$\mathcal{S}_0$ はタグを削除し，全てのデータが格納された $(T_{i+1,j}, S^T), (U_{i+1,j}, S^U)$ を $\mathcal{S}_1$ と共有する．

5.3 方式の効率

上記の PIR を用いる構成についても，オーバーヘッドとブロックサイズを考察する．

この方式における，PIR を用いない構成との大きな相違点は，データアクセスの際に通信されるブロック数が $1/d \cdot C_{\mathbf{Lookup}}(\mathcal{H}_i)$ 倍になっている点と，再シャッフルの際に oblivious hashing を用いる点の 2 点である．

まずデータアクセスについて，PIR を用いない構成においてはビット長 $2\log N$ ないし $\Lambda = \Omega(m \log N)$ のデータを，それぞれ $k + s + \sum_{i=1}^{L}(d-1)C_{\mathbf{Lookup}}(\mathcal{H}_i)$ 回参照していた．しかし今回，データはそれぞれ $k+1+L$ 回のみ参照されており，代わりに高々 $\log N$ ビットのヘッダが $s + \sum_{i=1}^{L}(d-1)C_{\mathbf{Lookup}}(\mathcal{H}_i)$ 個読み出されている．したがって，この方式における第 2〜第 m 索引の検索コストは

$$\leq (m-1) \times (\log N \times (s + \alpha(d-1)L) + 2\log N \times (k+1+L))$$

となり，また第 1 索引の検索コストは

$$\leq \log N \times (s + \alpha(d-1)L) + \Lambda \times (k+1+L)$$

となる．

次に再シャッフルについて，$\mathcal{H}^o_i$.**Build** にかかるコストを $C_{\mathbf{Build}}(\mathcal{H}^o_i)$ とし，$\beta = \max_i \frac{C_{\mathbf{Build}}(\mathcal{H}^o_i)}{kd^{i-1}}$ とおく．この時，全ての階層 i において $C_{\mathbf{Build}}(\mathcal{H}^o_i) \leq \beta k d^{i-1}$ と押さえることができ，またここで構築されるテーブルの要素は全て高々 $\log N$ ビットのヘッダのみである．よって計 m 個のテーブル構築における通信量は

$$\leq \beta k d^{i-1} \times m \times \log N$$

となる．また，ヘッダのテーブルとデータ本体の紐付けにかかる通信量は各階層 i で $\leq \mu_i \Lambda + (m-1)\mu_i \log N$ となる．したがって，再シャッフルにかかる償却通信量は

$$\leq \sum_{i=1}^{L} (\beta k d^{i-1} m \log N + \mu_i \Lambda + (m-1)\mu_i \log N)/kd^{i-1}$$
$$\leq \beta m \log N \times L + 2\Lambda \sum_{i=1}^{L} \frac{\mu_i}{kd^{i-1}}$$

となる．

以上がこの方式における各アルゴリズムのコストとなるが，オーバーヘッドの評価にあたってはブロックサイズの考察も必要となる．例えばデータアクセスの際に，この方式では [9] と同様，ブロック 1 つを読み出すために複数のヘッダを送受信している．この場合，ブロックサイズの評価には実際のデータブロックのサイズに総ヘッダサイズを加えることが望ましい．そこで，各アルゴリズムで通信されるヘッダ・データのサイズを以下のように整理する．

- データアクセスで通信されるヘッダサイズは，第 2〜第 m 索引で $O((m-1) \times \alpha d \log N)(\times L$ 階層) ビット，第 1 索引で $O(\alpha d \log N)$ $(\times L$ 階層) ビット．
- データアクセスで通信されるデータサイズは，第 2〜第 m 索引で $O((m-1) \times 2\log N)$ $(\times L$ 階層) ビット，第 1 索引で $\Omega(m \log N)$ $(\times L$ 階層) ビット．
- 再シャッフルで送受信される (償却) ヘッダサイズは $O(m \times \beta \log N)$ $(\times L$ 階層) ビット．

- 再シャッフルで送受信される (償却) データサイズは $\Omega(m \log N)$ ($\times \sum_{i=1}^{L} \frac{\mu_i}{kd^{i-1}}$ 個) ビット．

これを踏まえ，[9] に倣いブロックサイズを $\Omega(m(\alpha d + \beta) \log N)$ とすれば，この方式のオーバーヘッドは

$$O\left(k + L + \sum_{i=1}^{L} \frac{\mu_i}{kd^{i-1}}\right)$$

となる．

$\mathcal{H}^o$ が two-tier hashing ならば [3] より $\beta = O(\log N)$ となる．この時，$d = \log^{\delta} N$ とすれば，$\alpha d = \log^{\delta+\epsilon} N = \Omega(\log^{0.5} N)$ より[*8]，ブロックサイズ $\Omega(m \log^2 N)$，オーバーヘッド $O(\frac{\log N}{\log \log N})$ と評価できる．

6. 索引の重複を許す ORAM

既存方式と提案方式との最も大きな違いは，第 2 以下の副索引と主索引 (第 1 索引) を紐付ける写像，ないしポインタの役割を果たす $U_{i,j}$ の存在である．この $U_{i,j}$ を利用し，逆に第 1 索引を副次的 (あるいは仮想的) な索引とすることで，索引の重複が可能な ORAM を構成することができる．

具体的な方法を次に示す．

各データは (v_{main}, x) で表されるとし，各 v_{main} が最大 ρ 個まで重複すると仮定する．各索引 v_{main} に対し，ポインタ v をインクリメントさせながら次のようなデータブロックを作成する．

$$(v_{main}, (v, v+1, \ldots, v+\rho-1)).$$

次に，v_{main} と紐付いた各ポインタ $v+i$ に対して，v_{main} と対応するデータを 1 つずつ登録する．すなわち，データ (v_{main}, x_i) に対して $(v+i, x_i)$ を作成する．索引 v_{main} を持つデータが $\lambda < \rho$ 個しかない場合は，代わりにダミーデータ $(v+\lambda, \text{“}empty\text{”}), \ldots, (v+\rho-1, \text{“}empty\text{”})$ を作成する．

その後，$(v_{main}, (v, v+1, \ldots, v+\rho-1))$ がテーブル U へ，$(v+i, x_i)$ がテーブル T へ保存されるよう提案方式を構成し，以降クエリ v_{main} によるポインタ $v+i$ $(i = 1, \ldots, \rho)$ の読み出し・$v+i$ によるデータ x_i の探索という順でデータ検索が可能である．この際に，全てのクエリ v_{main} に対して階層 T への探索が ρ 回必要となるため，通信量は前章の提案方式の ρ 倍になる．

7. 結論

本論文では，複数の種類の索引でデータの検索が可能な階層的 Oblivious RAM を提案した．また，異なるデータの間で同じ索引を共有する，すなわち索引の重複があるようなデータセットに対して，重複した索引によるデータ検索が可能な階層的 Oblivious RAM を提案した．既存の階層的 ORAM は，いずれも索引が 1 種類かつ重複しないことを前提としていたが，本論文の提案方式によってこの束縛条件を緩和することができた．

今後の課題としては，複数の索引において重複を許可するような場合についての検討が必要であることなどが挙げられる．

[*8] 定義より $0.5 < \delta, 0 < \epsilon$.

参考文献

[1] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen. Parallel randomized load balancing. In Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing, STOC ’95, pages 238–247, New York, NY, USA, 1995. ACM.

[2] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In E. Oswald and M. Fischlin (eds) Advances in Cryptology – EUROCRYPT 2015, pages 337367, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[3] T-H. H. Chan, Y. Guo, W-K. Lin, and E. Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. Cryptology ePrint Archive, Report 2017/924, 2017.

[4] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In P. Q. Nguyen and E. Oswald (eds) Advances in Cryptology – EUROCRYPT 2014, pages 640–658, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[5] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In L. Aceto, M. Henzinger, and J. Sgall (eds) Automata, Languages and Programming, pages 576–587, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. J. ACM, 43(3):431–473, May 1996.

[7] A. Ichikawa, W. Ogata, K. Hamada, R. Kikuchi. Efficient secure multi-party protocols for decision tree classification. In Jang-Jaccard J., Guo F. (eds) Information Security and Privacy. ACISP 2019. Lecture Notes in Computer Science, vol 11547. Springer, Cham

[8] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete algorithms (SODA ’12). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2012. pages 143–156.

[9] E. Kushilevitz, T. Mour. Sub-logarithmic distributed oblivious RAM with small block size. In D. Lin, K. Sako (eds) Public-Key Cryptography – PKC 2019. Lecture Notes in Computer Science, vol 11442. Springer, Cham. Cryptography and Security arXiv:1802.05145, 2018.

[10] S. Lu, R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In A. Sahai (eds) Theory of Cryptography. TCC 2013. Lecture Notes in Computer Science, vol 7785. Springer, Berlin, Heidelberg.

[11] 濱田 浩気，市川 敦謙．劣線形ローカル計算量で定数ラウンドの秘密計算配列アクセスアルゴリズム．暗号とセキュリティシンポジウム 2019 予稿集．