最悪オーバーヘッドが$O(\log N)$の分散型ORAM

市川 敦謙[1,a)]

概要：Oblivious RAM(ORAM) は暗号化データへの検索や操作 (読み書き) を可能とし，さらにその際のアクセスパターンを秘匿する性質を持った暗号学的プリミティブである．既存手法として，クライアント-サーバ型および分散型のそれぞれで理論最適な効率 ($O(\log N)$, N はデータの総数) を達成した方式が知られているが，いずれの方式も償却時間としてこれを達成しており，最悪ケースでは $O(N)$ のコストを要する課題がある．これに対し，クライアント-サーバ型では最悪ケースのオーバーヘッドを $O(\log N)$ に改善した方式も存在するが，分散型モデルではこうした手法は知られていない．本研究ではこの問題を解決し，分散型モデルにおいて最悪ケースのオーバーヘッドが $O(\log N)$ となる手法を提案する．

キーワード：Oblivious RAM，De-amortization，マルチパーティ計算

Distributed ORAM with Worst-Case Logarithmic Overhead

Atsunori Ichikawa[1,a)]

Abstract: Oblivious RAM (ORAM) is a cryptographic primitive that enables reading and writing of encrypted data, while also keeping the access patterns secret. Known schemes include client-server and distributed models, each of which has achieved asymptotically optimal overhead ($O(\log N)$, where N is the number of data). However, both schemes achieve this overhead in terms of amortized time, and in the worst case, they incur a cost of $O(N)$. In the client-server model, there is a scheme that improves the worst-case overhead to $O(\log N)$, but no such scheme is known for the distributed model. We propose a novel scheme that achieves a worst-case overhead of $O(\log N)$ in the distributed model.

Keywords: Oblivious RAM, De-amortization, Secure Multiparty Computation

1. はじめに

Oblivious RAM (ORAM) は暗号化されたデータに対してアクセスパターンを秘匿したまま効率的に読み書きを行える技術であり，高安全な暗号化ストレージや秘密計算の改良，TEE の拡張など，幅広い応用が期待できる重要な暗号学的プリミティブである．ORAM は初めに Goldreich と Ostrovsky [6] によって提案され，近年では Asharov らによって理論上最適な効率である $O(\log N)$ オーバーヘッド[*1]を達成した方式 [2] が実現された．これらの方式は ORAM の最も標準的なモデル (クライアント-サーバ型とも呼ばれる) に基づくもので，trusted なクライアントが semi-honest なサーバに対してデータを暗号化して預け，また両者による二者間プロトコルによりデータの検索・操作を可能としている．

これに対し，クライアントが存在せず複数台のサーバのみで同等の機能を実現する，分散型 ORAM と呼ばれるモデルも存在する．分散型はクライアント-サーバ型と異なり，trusted party がシステムモデル上に存在しないことから，特に秘密計算へ導入・適用しやすいメリットがある．分散型 ORAM の例として，Ichikawa らは理論上最適な $O(\log N)$ オーバーヘッドを達成する分散型 ORAM を

[1] NTT 社会情報研究所
NTT Social Informatics Laboratories
[a)] atsunori.ichikawa@ntt.com

[*1] N はデータの総数とする．「オーバーヘッド」とはデータ 1 つにアクセスする際に操作が必要な (余剰な) データの個数を指しており，ORAM においては通信量・計算量ともに関わる効率の指標として用いられる．

提案している [7].

1.1 課題：非償却化

先に上げた Asharov らのクライアント-サーバ型 ORAM [2]，および Ichikawa らの分散型 ORAM [7] はいずれも「階層型」と呼ばれる ORAM の構成法に則っている．階層型の構成法は既存の ORAM の中でも特に理論的な効率に優れることから最もポピュラーな手法であり，実際に上記両方式は 1 アクセスあたりの償却 (“ならし”) 計算量において理論最適な効率を達成している．

しかしながら，同時に階層型構成法に共通する課題として最悪計算量が大きいことも知られており，上記の両方式はともに最悪ケースで $O(N)$ のオーバーヘッドがかかることも知られている．これに対し，Asharov らは [2] の手法を**非償却化** (*de-amortize*) し，最悪ケースでも $O(\log N)$ オーバーヘッドとなるクライアント-サーバ型 ORAM を提案している [1].

1.2 本研究の貢献

本研究では，分散型モデルにおいて最悪オーバーヘッドが $O(\log N)$ となる ORAM を初めて実現する．具体的には，Ichikawa らの分散型 ORAM に以下の改良を行う．

- データ構造へのバッファ領域の追加とライフサイクルの改善 (Section 4.2)：非償却化のためには，データ数に対して線形時間がかかる「データ構造の再構築 (再ランダム化)」をデータアクセス毎に分割して実施する必要があるが，その際に一時的にアクセス不可能なデータが生じないよう，時間に伴うデータの流れを工夫する必要がある．
- 重複データの削除 (Section 4.3)：階層型 ORAM は静的データ構造により構成されるため，その再構築を複数回のアクセスクエリに跨って行う場合，再構築中に更新されるデータが発生しうる．この時，ORAM 上には「再構築の入力となっていた旧データ」と「更新された新データ」の 2 つが重複して存在してしまうため，旧データを削除できる機能を追加する必要がある．

本研究では以上の 2 点により，Ichikawa らの方式の非償却化を達成している．

2. 準備

2.1 Oblivious RAM

Oblivious RAM (ORAM) はデータの持ち主たるクライアントと，データを預かる (信頼できない) サーバとの間で実行されるデータアクセスプロトコルであり，主にはデータアクセス時にサーバ側で観測されるメモリ参照位置の列 (これをアクセスパターンと呼ぶ) の秘匿を目的とする．ORAM の機能は次の $\mathcal{F}_{\mathrm{RAM}}$ で定義される．

$\underline{\mathcal{F}_{\mathrm{RAM}}(\mathsf{op}, k, v')}$: 内部状態として容量 N の配列 X を持つ．

(1) $\mathsf{op} = \mathsf{read}$ ならば X からキー・バリューペア (k, v) を探し，出力する．X に (k, v) が存在しない場合はダミー $(\perp, \perp)$ を出力する．

(2) $\mathsf{op} = \mathsf{write}$ ならば X からキー・バリューペア (k, v) を削除し (k, v') で置き換える．X に (k, v) が存在しない場合は単に (k, v') を追加する．

本稿で想定する ORAM は複数のサーバ上で分散されたデータ構造を構築し，trusted な参加者を交えずにデータとアクセスパターンを秘匿してデータ探索を行う，分散型 *(distributed) ORAM* と呼ばれるモデルである．この手法では，各データは秘密分散などで秘匿化されサーバ間に共有されており，サーバ同士のマルチパーティ計算によってデータ構造の操作とアクセスが実行される．攻撃者は一部のサーバと結託する (同様に多くは semi-honest な) アルゴリズムである．Ichikawa らの分散型 ORAM [7] によれば，$O(\log N)$ オーバーヘッドによりデータとアクセスパターンを秘匿したアクセスが可能である．

ORAM においてクエリ列 $Q = (q_1, \ldots, q_m)$ のもとアクセスアルゴリズム $\textsc{Access}(q_i);\ i = 1, \ldots, m$ を実行した際，サーバ上で観測できるデータおよびアクセスパターンを $\mathsf{View}(Q)$ で表すとする．このとき ORAM の安全性は次のように定義される．

定義 1 (Oblivious RAM (informal))**.** λ をセキュリティパラメータとし，$m = \mathsf{poly}(\lambda)$ とする．任意のアクセスクエリ列 $X = (x_1, \ldots, x_m)$ および $Y = (y_1, \ldots, y_m)$ に対し，攻撃者と結託するサーバが観測できる $\mathsf{View}(X)$ と $\mathsf{View}(Y)$ が識別不可能ならば，これを *Oblivious RAM* と呼ぶ．

2.2 マルチパーティ計算

マルチパーティ計算 (multi-party computation, MPC) は，複数の参加者間で情報を秘匿したまま計算を行う技術である．本稿では特に秘密分散法に基づく手法を指す．

以下では，ある値 (平文)a を秘密分散法によって秘匿化した状態を $[\![a]\!]$ で表すとし，これを以後「a のシェア」もしくは単に「シェア」と呼ぶ．また，a から $[\![a]\!]$ へ変換する処理;$[\![a]\!] \leftarrow \textsc{Share}(a)$ を「分散」，$[\![a]\!]$ から a へ変換する処理;$a \leftarrow \textsc{Reveal}([\![a]\!])$ を「復元」と呼ぶ．

以下では特に $(1,3)$-複製秘密分散 [8] および $(1,2)$-加法秘密分散 [5] を用いることを想定する．以下では $(1,3)$-複製秘密分散のシェアを $[\![\cdot]\!]$,$(1,2)$-加法秘密分散のシェアを $\langle\cdot\rangle$ で表す．また本稿では簡単のため，配列 $A = (a_1, \ldots, a_n)$ のシェア $([\![a_1]\!], \ldots, [\![a_n]\!])$ を単に $[\![A]\!]$ と記述する．

2.2.1 加算・乗算

本稿では，秘密の加算および乗算を単に $[\![c]\!] \leftarrow [\![a]\!] + [\![b]\!]$ および $[\![d]\!] \leftarrow [\![a]\!] \times [\![b]\!]$ と表記する $(c = a + b, d = ab)$．秘

密の加算はシェア同士の和によって達成できる他，各シェアのサイズが ℓ ビットの時，秘密の乗算は通信量 $O(\ell)$ ビットで実現できる [3, 5].

2.2.2 条件付き選択

条件付き選択を以下のように記述する．

$$[\![z]\!] \leftarrow \textsc{IfElse}([\![e]\!], [\![x]\!], [\![y]\!]).$$

入力 $[\![x]\!], [\![y]\!]$ は u ビットのシェア，$[\![e]\!]$ は 1 ビットのシェアとする．出力は $e = 1$ ならば $z = x$，さもなくば $z = y$ を満たす．この処理は定数回の秘密乗算および加算により実現でき[*2]，通信量 $O(\ell)$ ビットで実行できる．

なお以下では簡単のため，アルゴリズム中の条件分岐の記法として上記の式を「$[\![e]\!]$ ならば $[\![z]\!] \leftarrow [\![x]\!]$，さもなくば $[\![z]\!] \leftarrow [\![y]\!]$」といった自然言語でも表現する．

2.2.3 比較，等号判定

シェアの大小比較および等号判定を以下のように記述する．

$$[\![w]\!] \leftarrow [\![x \le y]\!], [\![z]\!] \leftarrow [\![x =_? y]\!]. \quad (w, z \in \{0, 1\})$$

[4] によれば，シェアサイズが ℓ ビットの時に等号判定を $O(\ell)$ ビットの通信量で実現できる．

3. 階層型 ORAM

階層型 (Hierarchical)ORAM [2, 6, 7] は，ORAM の構成法で最もポピュラーなものの 1 つである．方式ごとにパラメータ等は様々であるが，多くは共通して以下のようなフレームワークに沿って構成される．

- データ構造は Level 0 から Level $L \sim \log N$ までの階層で構成される．
- Level 0 はデータを T 個まで保持できる配列で構成される．
- Level $i > 0$ はデータを $2^{i-1}T$ 個まで保持し，また相異なる $2^{i-1}T$ 回のデータアクセスに関してアクセスパターンを秘匿できる，ランダムにシャッフルされた配列で構成される．

Level $i > 0$ を構成する際は，例えば *Oblivious Hashing* という手法を用いてデータを秘匿した**ハッシュテーブル**を作成し，各層で効率的にデータアクセスを実現するといった方法が知られている．

3.1 Oblivious Hashing

データを配列中にランダムに配置しつつ，アクセスの際には効率的に探索ができるようなデータ構造の一例として，ハッシュ関数によりデータの格納先を決定するハッシュテーブルと呼ばれるデータ構造が挙げられる．特にデータを秘匿したまま構築・参照が可能なハッシュテーブルを実現する手法として Oblivious Hashing と呼ばれるプロトコルが存在する．例えば Ichikawa らが提案した MPC 上で実現される Oblivious Hashing [7] は，以下の 4 つのアルゴリズムからなる．

- $(\langle \mathsf{T} \rangle, [\![\mathsf{S}]\!]) \leftarrow \textsc{Build}([\![D]\!])$: データセット D を元にハッシュテーブルを構築するアルゴリズム．T はテーブル本体，S はテーブルから溢れたデータを保持するスタッシュ．
- $[\![\mathsf{d}]\!] \leftarrow \textsc{Lookup}([\![k]\!], \langle \mathsf{T} \rangle)$: キー値 k によりテーブル T を探索するアルゴリズム．T の中にペア (k, v) が存在するならば $\mathsf{d} = (k, v)$，さもなくば $\mathsf{d} = (0, 0)$.
- $[\![\mathsf{d}]\!] \leftarrow \textsc{StashSearch}([\![k]\!], [\![\mathsf{S}]\!])$: キー値 k によりスタッシュ S を探索するアルゴリズム．S の中にペア (k, v) が存在するならば $\mathsf{d} = (k, v)$，さもなくば $\mathsf{d} = (0, 0)$.
- $[\![D']\!] \leftarrow \textsc{Extract}(\langle \mathsf{T} \rangle)$: テーブルに格納されているデータを全て取り出すアルゴリズム．

Ichikawa らは Cuckoo ハッシュテーブルと呼ばれる手法 [10] を MPC 上で実現しており，入力データセットのサイズ $|D| = n$，セキュリティパラメータを λ とすると，$|\mathsf{T}| = O(n), |\mathsf{S}| = O(\log \lambda)$，Build および Extract は計算量 $O(n)$，Lookup は計算量 $O(1)$，StashSearch は計算量 $O(\log \lambda)$ で実現できる．

3.2 階層型 ORAM のアクセスアルゴリズムとその効率

階層型 ORAM は前述のような Oblivious Hashing によってアクセスパターンを秘匿したままデータの読み書きが可能である．具体的には次のような手順でデータアクセスと，必要に応じて階層の再ランダム化を実行する．

(1) 最初に Level 0 に存在する全てのデータについて，クエリされたキー値 k と一致するデータを探索する．

(2) 次に Level $i = 1, \ldots, L$ で，階層を構成するテーブル $\langle \mathsf{T}_i \rangle$ に対して $\textsc{Lookup}([\![k]\!], \langle \mathsf{T}_i \rangle)$ を実行する．ただし，ある Level $j, 0 \le j < L$ でキー値 k に一致するデータが発見されている場合は，以降の Level $i > j$ では k ではなくダミー値を入力として Lookup を実行する．

(3) 最後に，取り出したデータ $[\![\mathsf{d}]\!]$ を，必要に応じて書き換えた上で Level 0 の配列へ格納する．このとき，Level 0 の配列の容量 T が全て埋まったならば，次の手順 (再構築) を実行する．

[*2] 式としては $[\![y]\!] + [\![e]\!]([\![x]\!] - [\![y]\!])$ により表せ，ビット毎の乗算・加算によっても等価な結果を得られる．

(4) Level 0 の配列が埋まり，また Level $1,\ldots,u-1$ でハッシュテーブルが構築済みであり，かつ Level u のハッシュテーブルが未構築であるとする．このとき，全ての Level $i<u$ で $\textsc{Extract}(\langle \mathsf{T}_i\rangle)$ を実行し，取り出した全てのデータと Level 0 のデータを合わせた配列 $[\![D_u]\!]$ をもって新たなハッシュテーブル $\langle \mathsf{T}_u\rangle, [\![\mathsf{S}_u]\!] \leftarrow \textsc{Build}([\![D_u]\!])$ を構築する．その後，Level $0,\ldots,u-1$ の全てのデータを削除し，$[\![\mathsf{S}_u]\!]$ を新たな Level 0 配列・$\langle \mathsf{T}_u\rangle$ を Level u のテーブルとする．

上記のアクセスアルゴリズムにおける手順 (1) から (3) までは $O(T+L)$ で実行可能．一方手順 (4) については，Level u の再構築は $2^{u-1}T$ 回のアクセス毎に発生しコストが $O(2^{u-1}T)$ であるため，$u=1,\ldots,L$ でそれぞれ**償却**オーバーヘッドが $O(1)$ となる．よってアクセスアルゴリズム全体で $O(T+L)$ オーバーヘッドとなり，特に $T=L=O(\log N)$ ならば $O(\log N)$ となる．

4. 提案手法：最悪ケースで $O(\log N)$ オーバーヘッドの非償却化分散型 ORAM

前述の通り階層型 ORAM は償却オーバーヘッドにおいては効率的なアルゴリズムを構成できる反面，これらの既存手法は最悪ケースでは $\Omega(N)$ オーバーヘッドとなり，クエリのタイミング次第で大きく効率を悪化させてしまうことが知られている．こうした課題に対し，各階層のハッシュテーブルの参照および再構築に関するスケジュール (以下，ハッシュテーブルの「ライフサイクル」と呼ぶ) を緻密に調整し，ORAM へのアクセス 1 回毎に各階層の特に再構築処理を少しずつ実施，負荷分散することで最悪計算量を償却計算量と同等に抑える手法が知られており [1, 9, 11]，こうした手法は非償却化 (De-amortized)ORAM と呼ばれる．以下に述べる本研究の提案手法も同様に，Ichikawa らの分散型 ORAM [7] を非償却化したアルゴリズムである．

4.1 非償却化へのステップ

Ichikawa らの分散型 ORAM の非償却化に向けて，本研究が解決するべき課題は以下の 2 点である．

- ハッシュテーブルのライフサイクルの調整
- ハッシュテーブル再構築に際する複製データ削除 (de-duplication)

前者について，Ichikawa らの Oblivious Hashing は n 個のデータを格納する際，参照に $O(1)$，分解・構築にそれぞれ $O(n)$ のオーバーヘッドがかかる．元となる階層型 ORAM のハッシュテーブルのライフサイクルでは入力サイズ n であるハッシュテーブルが $O(n)$ アクセス毎に分解・再構築が実施されるため，基本的にはこの分解・再構築の計算を $O(n)$ 回のアクセスが発生する間，定数量の計算として徐々に進めることで処理量の平均化が達成できるように見える．

しかしながら，階層型 ORAM で用いられる Oblivious Hashing が**静的**ハッシュテーブルを提供するものであることを踏まえると，ライフサイクルはより注意して構成する必要が生じる．すなわち，ハッシュテーブルの再構築を開始するには格納すべき全てのデータが揃っている必要があり，かつ一度再構築を開始したならば，これが終了するまでは入力データに対するアクセスが不可能となってしまう．端的に述べるならば，単純な負荷分散では ORAM の中に「一時的にアクセス不可能なデータ」を発生させてしまうこととなる．これを防ぐため，本研究では元のハッシュテーブルを**転写**するための新たなデータ領域を導入し，常に全てのデータがアクセス可能な状態を保つようなアルゴリズムを提案する (Section 4.2 を参照されたし)．

一方，後者の複製データ削除は上記のライフサイクル調整に伴う不具合を解消するために必要な処理となっている．例えば本研究で提案するようなライフサイクルでは，各データが常に「ハッシュテーブルの再構築の入力」であると同時に「(別の) テーブル上のアクセス可能なデータ」として存在している．このような状況でいずれか既存データの書き換え等の処理が発生した場合，「再構築中の (古い) データ」と「更新されたデータ」が**同一のキー値を持つ別のデータ**として ORAM 上に存在してしまうこととなる．

こうした状況が発生した場合[*3]，階層型 ORAM の構造上直ちに問題とはならないが，より下層のハッシュテーブルの再構築の際には対処が必要となる．階層型 ORAM において，原則として**新しいデータはより上層**にあるため，アクセスの際には更新済みデータが優先して読み出されることでアクセスの正当性は (暫くの間) 担保される．しかし，いずれ更新済みデータが古いデータと同じ階層 (ないしハッシュテーブル) に格納されるタイミングが訪れた際には，キー値が重複する古いデータを削除する必要が生じる．本研究ではデータに新たな補助情報として「更新回数」を付加し，また ORAM の構築とは別途に Oblivious Hashing を利用して重複を削除する方法を提案する (Section 4.3 を参照されたし)．

4.2 提案ライフサイクルとデータ構造

本節では，Ichikawa らの ORAM の非償却化に向けたハッシュテーブルのライフサイクルと新たなデータ構造を提案する．非償却化に際しては，各階層のハッシュテーブルの再構築プロセスを分割し，各 ORAM アクセス毎に定数量の計算を行うだけで適切なスパンでのテーブル更新が実施される必要がある．一方で，テーブル再構築中にアクセス不可能なデータが生じないよう，適切なバッファ領域も用意する必要がある．

[*3] ORAM 上に十分なデータが格納されている場合，これは実際には高頻度で発生しうる．

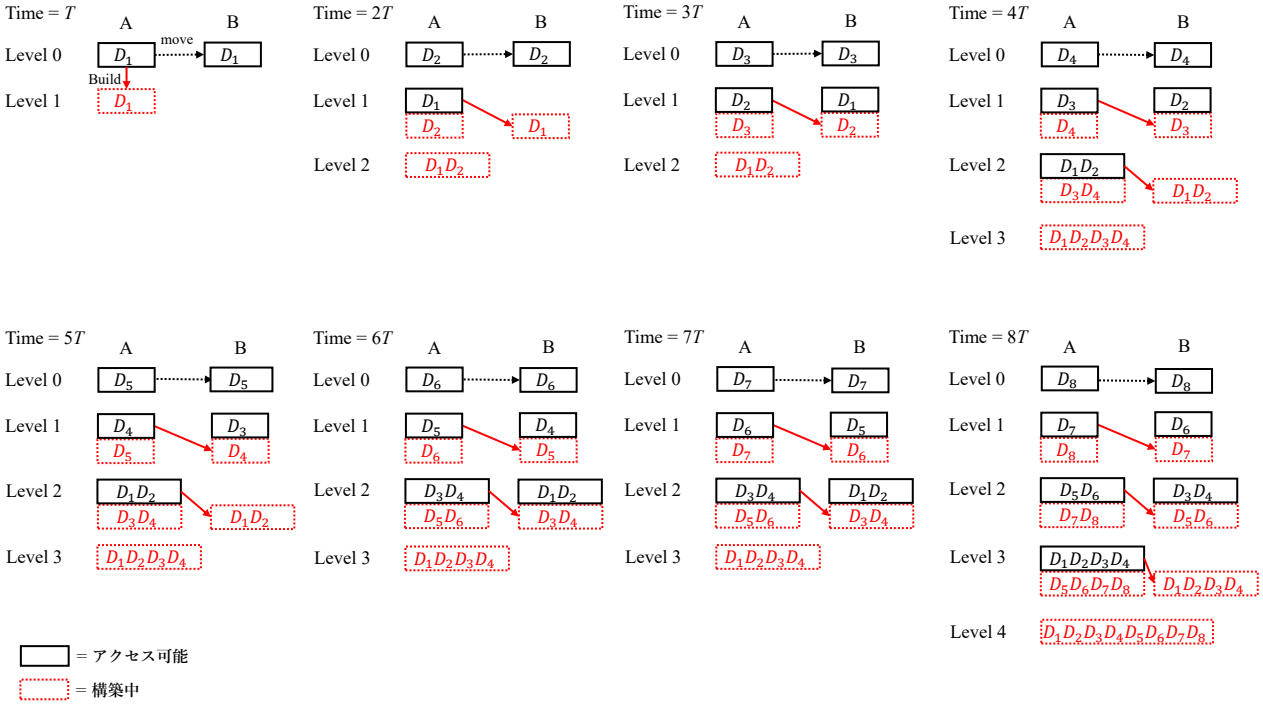

図 1 提案方式のライフサイクルとデータ構造において，$8T$ 個までのデータが格納されていく様子．ただし T は Level 0 配列の容量とする．

こうした問題は既存手法でも同様にアプローチがなされており，例えば Ichikawa らと同じく最適効率を持つ ORAM を非償却化した Asharov らの方法 [1] では，各階層のハッシュテーブルが「満たされた」状態と「半分満たされた」状態を行き来するようなライフサイクルを構成し，さらに階層構造を複製することでアクセス不可能なデータが生じないようなアルゴリズムを提案している．残念ながら Asharov らの ORAM はクライアント-サーバ型であり，分散型である Ichikawa らの方式に同じフレームワークを適用することは不可能である．例えば，Ichikawa らのハッシュテーブルの構築には MPC による配列置換を主たるサブルーチンとして用いるが，このとき「満たされた」状態を作る場合にも「半分」の状態を作る場合にも同じ長さの配列を置換する必要があり[*4]，全く同じ計算量および通信量がかかることからこのアプローチでは適切に処理を細分化できない．

そこで本研究では，各階層でもう一つのハッシュテーブルをバッファ領域として準備し，メインのハッシュテーブルに対して**周回遅れ**でデータを格納することで，メイン階層では再構築途中のデータがバッファ領域にてアクセス可能となるようなデータ構造およびライフサイクルを提案する．具体的には，提案データ構造は以下の要素からなる．

- データ構造は Level 0 から Level L までの層によって構成され，Level 0 には容量 T の配列が 2 つ，Level i には容量 $2^{i-1}T$ のハッシュテーブルが 2 つ，メイン・サブとしてそれぞれ用意される．
- Level 0 のメイン配列はアクセス毎に取り出されたデータが再格納されていき，T アクセス毎に全要素がサブ配列へ移動 (上書き) される．
- データ構造が初期化された時点を時間 Time $= 0$ とすると，各 Level $i > 0$ では Time $= 2^{i-1}T$ でメインのハッシュテーブルの構築が開始され，また Time $= 2^iT$ でサブハッシュテーブルの構築が開始される．
- Level i のメインハッシュテーブル構築における入力データは，構築が開始された時点での Level $0, \ldots, i-1$ のメインハッシュテーブルに格納されたデータである．またサブハッシュテーブル構築における入力データは，同階層のメインハッシュテーブルの格納データである．
- Level i の各ハッシュテーブルは構築が完了した直後から次の構築が開始され，時間 $2^{i-1}T$ 毎に更新される．

なお，本研究では Ichikawa らの Oblivious Hashing を用いてハッシュテーブルの構築・参照を行うとする．

本研究で提案するデータ構造およびライフサイクルの一部を図 1 に例示する．各 Level i のハッシュテーブルは $2^{i-1}T$ 個のデータを格納でき，従って構築してから分解するまでに最大 $2^{i-1}T$ 回までのデータ参照が可能である．一方で再構築には $O(2^{i-1}T)$ の計算量が必要なので，構築済

[*4] 置換する配列のサイズは入力サイズではなく構築対象のテーブルサイズに依存するため．

みのテーブルの裏で逐次再構築アルゴリズムを実施[*5]していくことにより，それぞれ $2^{i-1}T$ アクセス毎の更新が実現できる．

一方で前述の通り，テーブル再構築中には当該テーブルへ格納されるべきデータがアクセス不可となってしまうため，これを回避するため各階層において**周回遅れでデータを保持**するサブテーブルを用意する．図 1 では 'B' とラベル付けしたこれらのサブテーブルは，本体の階層構造 (図中の 'A') とは独立した乱数によってデータを格納するハッシュテーブルであり，'A' のテーブルの構築完了直後から同格納データによる構築を開始し，'A' のテーブルの使用期限と同時に構築が完了するようなライフサイクルを持つ．これによって，'A' のデータがより下層のハッシュテーブルに格納されるまでの間を取り持ち，一時的にアクセス不可能なデータが生じないようなデータ構造を実現できる．

4.3 複製データ削除 (De-duplication) アルゴリズム

次に，複数回の ORAM アクセスの過程で複製されたデータを適切に削除するための方法を示す．データの複製は階層型 ORAM のアルゴリズムで必ず発生しうる現象であり，例えば図 1 において，Time $= T+1$ でデータ群 D_1 のいずれかの要素 (k, v) が別の値で上書きされ，(k, v') として D_2 の要素として再格納された場合に，既に再構築が開始されている Level 1 メインテーブルに含まれる (k, v) と同じキー値のデータとして同時に存在してしまうこととなる．このとき D_2 は常に D_1 より上位にあるため，例えば Time $= T+2$ で再度 k をクエリした際には更新済みの (k, v') が問題なく読み出される．しかし，後に Time $= 2T$ で Level 2 メインテーブルの構築が開始される際には，当該のテーブルに $(k, v'), (k, v)$ の 2 つが同時に存在してしまうためハッシュテーブルの正当性が崩れることとなる．

こうした問題を回避するために，Level $i \geq 2$ でメインハッシュテーブルを再構築する際には必ず重複を削除して配列のマージが必要である．先の例に則れば，Level 2 メインテーブルの再構築における入力データは $D_1 \parallel D_2$ ではなく $D_1 \cup D_2$ とする必要がある．これを実現するための方法として，Asharov ら [1] は以下のような (non-oblivious な) 方法を示している．

(1) 2 つのデータセット D_1, D_2 を独立に 2 つの Cuckoo ハッシュテーブルへ格納し，それぞれ $\mathsf{H}_1 = (\mathsf{T}_1, \mathsf{S}_1), \mathsf{H}_2 = (\mathsf{T}_2, \mathsf{S}_2)$ とする (T はテーブル本体，S はスタッシュ)．

(2) S_1 にある全てのデータに関して，同じキー値で H_2 を探索し重複データを削除する．

(3) H_2 に残る全てのデータに関して，同じキー値で T_1 を探索し重複データを削除する．

(4) 残った D_1, D_2 のデータをマージして 1 つのデータセットとする．

Cuckoo ハッシュテーブルは T の参照コストが $O(1)$，S のサイズが $O(\log \lambda)$ であるため，$|D_1| = |D_2| = n$ とすると上記の手順 (2) は $O(\log^2 \lambda)$，手順 (3) は $O(n)$ の計算量となり，大きな n においては線形 ($O(n)$) 計算量で重複データの削除が可能である．

Asharov らは上記の方法をベースラインとしつつも，(クライアント-サーバモデルでは)Cuckoo ハッシュテーブルの構築コストが $O(n \log n)$ と非効率的であるため，データを秘匿する (oblivious な) 重複削除アルゴリズムを構成する際には別の手法を用いていた．しかし，分散型 ORAM においては Ichikawa らの線形効率の Oblivious Cuckoo Hashing を利用できるため，上記の方法を直接 oblivious なアルゴリズムに変換できる．

ただし Asharov らのクライアント-サーバモデルとは異なり，分散型は信頼できるパーティ (すなわちクライアント) が存在しないため，そのままではデータの新旧をサーバが判断できる方法がない問題も残る．よって本研究では各データに対してカウンタ (タイムスタンプ) を付与することで更新されたデータの新旧を識別可能とした．

Algorithm 1 $[\![\widetilde{D}]\!] \leftarrow \textsc{DeDuplicate}([\![D_1]\!], [\![D_2]\!])$

入力: $D_i = \big((k_{i,0}, v_{i,0}, t_{i,0}), \ldots, (k_{i,n_i}, v_{i,n_i}, t_{i,n_i})\big)$，ただし $k_{i,j}, v_{i,j}$ はキー・バリューペア，$t_{i,j}$ はタイムスタンプ．

1: $\langle \mathsf{T}_1 \rangle, [\![\mathsf{S}_1]\!] \leftarrow \mathsf{HT}_1.\textsc{Build}([\![D_1]\!])$.
2: $\langle \mathsf{T}_2 \rangle, [\![\mathsf{S}_2]\!] \leftarrow \mathsf{HT}_2.\textsc{Build}([\![D_2]\!])$.
3: $[\![\mathsf{S}_1]\!]$ の全てのデータ $([\![k_{1,i}]\!], [\![v_{1,i}]\!], [\![t_{1,i}]\!])$ について以下を実施：
4: $\quad([\![k'_{2,j}]\!], [\![v'_{2,j}]\!], [\![t'_{2,j}]\!]) \leftarrow \mathsf{HT}_2.\textsc{Lookup}([\![k_{1,i}]\!], \langle \mathsf{T}_2 \rangle)$.
5: $\quad[\![k_{1,i} =_? k'_{2,j}]\!] \wedge [\![t_{1,i} >_? t'_{2,j}]\!] = 1$ ならば，$\langle \mathsf{T}_2 \rangle$ 上のデータ $([\![k'_{2,j}]\!], [\![v'_{2,j}]\!], [\![t'_{2,j}]\!])$ をダミー $([\![\bot]\!], [\![\bot]\!], [\![\bot]\!])$ で上書き．
6: $\quad[\![k_{1,i} =_? k'_{2,j}]\!] \wedge [\![t_{1,i} \leq_? t'_{2,j}]\!] = 1$ ならば，$[\![\mathsf{S}_1]\!]$ 上のデータ $([\![k_{1,i}]\!], [\![v_{1,i}]\!], [\![t_{1,i}]\!])$ をダミー $([\![\bot]\!], [\![\bot]\!], [\![\bot]\!])$ で上書き．
7: $\quad[\![\mathsf{S}_2]\!]$ 上の全てのデータ $([\![k''_{2,j}]\!], [\![v''_{2,j}]\!], [\![t''_{2,j}]\!])$ に以下を実施：
8: $\quad\quad[\![k_{1,i} =_? k''_{2,j}]\!] \wedge [\![t_{1,i} >_? t''_{2,j}]\!] = 1$ ならば，$[\![\mathsf{S}_2]\!]$ 上の $([\![k''_{2,j}]\!], [\![v''_{2,j}]\!], [\![t''_{2,j}]\!])$ をダミー $([\![\bot]\!], [\![\bot]\!], [\![\bot]\!])$ で上書き．
9: $\quad\quad[\![k_{1,i} =_? k''_{2,j}]\!] \wedge [\![t_{1,i} \leq_? t''_{2,j}]\!] = 1$ ならば，$[\![\mathsf{S}_1]\!]$ 上の $([\![k_{1,i}]\!], [\![v_{1,i}]\!], [\![t_{1,i}]\!])$ をダミー $([\![\bot]\!], [\![\bot]\!], [\![\bot]\!])$ で上書き．
10: $[\![D'_2]\!] \leftarrow \mathsf{HT}_2.\textsc{Extract}()$ の全てのデータ $([\![k'_{2,j}]\!], [\![v'_{2,j}]\!], [\![t_{1,j}]\!])$ について以下を実施：
11: $\quad([\![k_{1,i}]\!], [\![v_{1,i}]\!], [\![t_{1,i}]\!]) \leftarrow \mathsf{HT}_1.\textsc{Lookup}([\![k'_{2,j}]\!], \langle \mathsf{T}_1 \rangle)$.
12: $\quad[\![k_{1,i} =_? k'_{2,j}]\!] \wedge [\![t_{1,i} >_? t'_{2,j}]\!] = 1$ ならば，$[\![D'_2]\!]$ 上のデータ $([\![k'_{2,j}]\!], [\![v'_{2,j}]\!], [\![t'_{2,j}]\!])$ をダミー $([\![\bot]\!], [\![\bot]\!], [\![\bot]\!])$ で上書き．
13: $\quad[\![k_{1,i} =_? k'_{2,j}]\!] \wedge [\![t_{1,i} \leq_? t'_{2,j}]\!] = 1$ ならば，$\langle \mathsf{T}_1 \rangle$ 上のデータ $([\![k_{1,i}]\!], [\![v_{1,i}]\!], [\![t_{1,i}]\!])$ をダミー $([\![\bot]\!], [\![\bot]\!], [\![\bot]\!])$ で上書き．
14: $[\![D'_1]\!] \leftarrow \mathsf{HT}_1.\textsc{Extract}()$.
15: $[\![\widetilde{D}]\!] \leftarrow [\![D'_1]\!] \parallel [\![D'_2]\!]$

Algorithm 1 に，本研究で提案する重複データ削除アルゴリズムを示す．入力 $D_1, D_2; |D_1| = n_1, |D_2| = n_2$ をそれぞれデータセットとし，これに含まれる各データは

[*5] 計算量は $O(1)/access$.

キー・バリューペア k, v にタイムスタンプ t を付与したものとする．提案アルゴリズムでは D_1, D_2 に対して独立に Oblivious Cuckoo Hashing を実施し，Asharov らが示した non-oblivious な重複削除方法をシミュレートしている．その際，データを削除する条件として「キー値が一致すること」に加えて「タイムスタンプが小さいこと」を設け，より古いデータをいダミーによる上書き (すなわちデータ削除) の対象としている．

Algorithm 1 では Step.1–2 で HT_i.Build を 2 回, Step.3–6 で HT_2.Lookup, 統合判定，大小比較，および条件分岐を $O(\log\lambda)$ 回，Step.7–9 で統合判定，大小比較，および条件分岐を $O(\log^2\lambda)$ 回，Step.10–13 で HT_2.Extract を 1 回に HT_1.Lookup, 統合判定，大小比較，および条件分岐を $O(n_2)$ 回，最後に Step.14 で HT_1.Extract を 1 回実施している．D_1, D_2 の各要素を $O(\ell)$-bit とし，またある n について $n_1 = n_2 = O(n)$ とすれば，HT.Build および HT.Extract$(i = 0, 1)$ の計算量は $O(n\ell)$，HT.Lookup および統合判定，大小比較，条件分岐の計算量は $O(\ell)$ となるため，Algorithm 1 全体の計算量は $O((n + \log^2\lambda)\ell)$ となる．

4.4 データアクセスアルゴリズム

Algorithm 2 $[\![\mathsf{d}]\!] \leftarrow$ Access$([\![\mathsf{op}]\!], [\![k]\!], [\![v']\!])$

入力: $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $[\![k]\!]$ は探索対象のキー，$[\![v']\!]$ は write クエリで書き換えるためのデータ．

1: $[\![\mathsf{d}]\!], [\![\mathsf{found}]\!] \leftarrow$ StashSearch$([\![k]\!], [\![\mathsf{S}^{\mathrm{A}}]\!] \,\|\, [\![\mathsf{S}^{\mathrm{B}}]\!])$.
2: **for** $i = 1$ to L **do**
3: 　$[\![\mathsf{found}]\!] =_? 1$ ならば $[\![\widetilde{k}]\!] := [\![\bot]\!]$，さもなくば $[\![\widetilde{k}]\!] := [\![k]\!]$ とする．
4: 　$[\![\mathsf{d}_i^{\mathrm{A}}]\!], [\![\mathsf{found}_i^{\mathrm{A}}]\!] \leftarrow \mathsf{HT}_i^{\mathrm{A}}$.Lookup$([\![\widetilde{k}]\!]\langle T^{\mathrm{A}}\rangle)$.
5: 　$[\![\mathsf{d}]\!] \leftarrow [\![\mathsf{d}]\!] + [\![\mathsf{d}_i^{\mathrm{A}}]\!], [\![\mathsf{found}]\!] \leftarrow [\![\mathsf{found}]\!] + [\![\mathsf{found}_i^{\mathrm{A}}]\!]$.
6: 　再度 $[\![\mathsf{found}]\!] =_? 1$ ならば $[\![\widetilde{k}]\!] := [\![\bot]\!]$，さもなくば $[\![\widetilde{k}]\!] := [\![k]\!]$ とする．
7: 　$[\![\mathsf{d}_i^{\mathrm{B}}]\!], [\![\mathsf{found}_i^{\mathrm{B}}]\!] \leftarrow \mathsf{HT}_i^{\mathrm{B}}$.Lookup$([\![\widetilde{k}]\!]\langle T^{\mathrm{B}}\rangle)$.
8: 　$[\![\mathsf{d}]\!] \leftarrow [\![\mathsf{d}]\!] + [\![\mathsf{d}_i^{\mathrm{B}}]\!], [\![\mathsf{found}]\!] \leftarrow [\![\mathsf{found}]\!] + [\![\mathsf{found}_i^{\mathrm{B}}]\!]$.
9: $[\![\mathsf{found} =_? 0]\!]$ ならば $[\![\mathsf{d}]\!] \leftarrow [\![\mathsf{d}^{\mathsf{dummy}}]\!]$.
10: $[\![\mathsf{found} =_? 1]\!] \wedge [\![\mathsf{op} =_? \mathsf{write}]\!]$ ならば，$\mathsf{d} = (k, v, t)$ として $[\![\mathsf{d}]\!] \leftarrow ([\![k]\!], [\![v']\!], [\![t+1]\!])$.
11: $[\![\mathsf{d}]\!]$ を配列 $[\![\mathsf{S}^{\mathrm{A}}]\!]$ へ格納．このとき $|\mathsf{S}^{\mathrm{A}}| = T$ ならば，$[\![\mathsf{S}^{\mathrm{A}}]\!]$ の全ての要素を $[\![\mathsf{S}^{\mathrm{B}}]\!]$ へ移動させる．

最後に，本研究で提案する非償却化 3-Party ORAM のアクセスアルゴリズムおよび再構築アルゴリズムについて述べる．まずは改めて提案手法のデータ構造および格納されるデータの形式について説明する．

- 提案 ORAM へ格納される各データはキー・バリューペアにタイムスタンプが付与された組 $\mathsf{d} = (k, v, t)$ で表現される．ただし t は初期値を 0 とし，簡単のため以下では $O(\log N)$-bit の値とする．
- 格納されるデータの最大数は $N = \mathsf{poly}(\lambda)$ とする．

Algorithm 3 Reshuffle(u)

入力: $u \in \{1, \ldots, L\}$

1: $u = 1$ ならば $\mathsf{HT}_1^{\mathrm{A}}$.Build$([\![\mathsf{S}^{\mathrm{A}}]\!])$ を実行し，また既に $\mathsf{HT}_1^{\mathrm{A}}$ が存在する場合には $[\![\mathsf{D}]\!] \leftarrow \mathsf{HT}_1^{\mathrm{A}}$.Extract() および $\mathsf{HT}_1^{\mathrm{B}}$.Build$([\![\mathsf{D}]\!])$ を実行して終了．
2: さもなくば，$[\![\mathsf{D}_0]\!] \leftarrow [\![\mathsf{S}^{\mathrm{A}}]\!]$ として以下を実行：
3: **for** $i = 1$ to $u - 1$ **do**
4: 　$[\![\mathsf{D}_i]\!] \leftarrow \mathsf{HT}_i^{\mathrm{A}}$.Extract().
5: 　$[\![\mathsf{D}_i]\!] \leftarrow$ DeDuplicate$([\![\mathsf{D}_{i-1}]\!], [\![\mathsf{D}_i]\!])$.
6: $\mathsf{HT}_u^{\mathrm{A}}$.Build$([\![\mathsf{D}_{u-1}]\!])$ を実行．既に $\mathsf{HT}_1^{\mathrm{A}}$ が存在する場合，$[\![\mathsf{D}]\!] \leftarrow \mathsf{HT}_u^{\mathrm{A}}$.Extract() の後 $\mathsf{HT}_u^{\mathrm{B}}$.Build$([\![\mathsf{D}]\!])$ を実行．

- $\mathsf{d}^{\mathsf{dummy}} = (\bot, \bot, \bot)$ をダミーデータとする．
- 提案 ORAM は Level 0 から Level $L = \lceil \log N - \log\log N \rceil$ までの階層で構成され，各層にはメイン・サブの 2 つのデータ領域が存在する (以下ではメイン・サブをそれぞれ A,B の記号で表す)．
- メインのデータ構造は Ichikawa らの階層型 ORAM [7] と同様であり，Level 0 には容量 $T = 2\lceil \log N \rceil$ の配列 S^{A}，Level $i > 0$ には容量 $2^{i-1}T$ のハッシュテーブル $\mathsf{HT}_i^{\mathrm{A}}$ を持つ．
- サブデータ構造もメインと同様に，Level 0 には容量 $T = 2\lceil \log N \rceil$ の配列 S^{B}，Level $i > 0$ には容量 $2^{i-1}T$ のハッシュテーブル $\mathsf{HT}_i^{\mathrm{B}}$ を持つ．

以上のデータ構造に基づき，Algorithm 2 および Algorithm 3 にそれぞれデータアクセスアルゴリズムおよび階層 (Level u) の再構築アルゴリズムを示す．

Algorithm 2 については，サブデータ構造へのアクセスを除けば Ichikawa らの 3-Party ORAM のアクセスアルゴリズムと同様である．サブデータ構造はメインのデータ構造と同じサイズであることから，Algorithm 2 のコストは Ichikawa らのアクセスアルゴリズムの高々 2 倍となり，$O(\ell \log N)$ の計算量で実現できる (ℓ はデータのビット長とする)．

一方，Algorithm 3 は Ichikawa らの再構築アルゴリズムに対して重複データの削除が追加されている (Step.5, DeDuplicate)．各 i での DeDuplicate の計算量は $O((2^{i-1}T + \log^2\lambda)\ell)$ であるため，Algorithm 3 全体の計算量は $O((2^u T + u\log^2\lambda)\ell)$ となる．

非償却化にあたっては，Algorithm 3 はアクセス $2^u T$ 回に渡って分割して実行されることとなるため，1 アクセスあたり必要な計算量は $O((1 + \frac{u\log^2\lambda}{2^{u-1}T})\ell)$ となる．$T = 2\log\lambda$ を踏まえればこれは $O((1 + \frac{u\log\lambda}{2^u})\ell)$ と表せることから，より大きな計算が必要となる下層 (大きな u) では定数時間に丸められる[*6]．よってデータ構造全体 $(u = 1, \ldots, L)$ でアクセス 1 回毎に必要な Reshuffle の計算量の総和は

[*6] 例えば $u = L$ では $O((1 + \frac{\log^2\lambda}{N})\ell)$，$N = \mathsf{poly}(\lambda)$ であることから式中の分数部分は無視できる．

$O(L\ell) = O(\ell \log N)$ と評価でき，Algorithm 2 と合わせて最悪ケースでも $O(\log N)$ オーバーヘッドの 3-Party 分散型 ORAM が構成できた．

5. 結論

本研究では Ichikawa らの分散型 ORAM [7] について，データ構造とそのライフサイクルに改良を行うことで最悪ケースのオーバーヘッドを改善した．Ichikawa らの方式は償却オーバーヘッドにおいて理論上最適効率である $O(\log N)$ を達成しているものの，最悪ケースでのオーバーヘッドが $O(N)$ であるという課題があった．本研究の提案方式はこれを解決し，最悪ケースでも $O(\log N)$ オーバーヘッドとなる効率的な分散型 ORAM を実現した．

参考文献

[1] Asharov, G., Komargodski, I., Lin, W.-K. and Shi, E. Oblivious RAM with Worst-Case Logarithmic Overhead, *J. Cryptol.*, **36**, 2 (2023).

[2] Asharov, G., Komargodski, I., Lin, W., Nayak, K., Peserico, E. and Shi, E. OptORAMa: Optimal Oblivious RAM, *J. ACM*, **70**, 1 (2023), 4:1–4:70.

[3] Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R. and Pinkas, B. High-Throughput Secure AES Computation, WAHC (2018).

[4] Cramer, R., Damgård, I. and Ishai, Y. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation, TCC (2005).

[5] Damgård, I., Pastro, V., Smart, N. and Zakarias, S. Multiparty Computation from Somewhat Homomorphic Encryption, CRYPTO (2012).

[6] Goldreich, O. and Ostrovsky, R. Software Protection and Simulation on Oblivious RAMs, *J. ACM*, **43**, 3 (1996), 431–473.

[7] Ichikawa, A., Komargodski, I., Hamada, K., Kikuchi, R. and Ikarashi, D. 3-Party Secure Computation for RAMs: Optimal and Concretely Efficient, *IACR Cryptol. ePrint Arch.* (2023), 516.

[8] Ito, M., Saito, A. and Nishizeki, T. Secret sharing scheme realizing general access structure, *GLOBECOM* (1987), 99–102.

[9] Kushilevitz, E., Lu, S. and Ostrovsky, R. On the (in)security of hash-based oblivious RAM and a new balancing scheme, SODA (2012).

[10] Kushilevitz, E. and Mour, T. Sub-logarithmic Distributed Oblivious RAM with Small Block Size, PKC (2019).

[11] Ostrovsky, R. and Shoup, V. Private information storage, ACM STOC (1997).