Computer Security Symposium 2017
23 - 25 October 2017

# 劣線形通信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

濱田 浩気[1]

**概要**：本稿では，秘密計算で対象位置を隠したまま配列アクセスを効率的に行うアルゴリズムを提案する．通常の計算機で使われる多くのアルゴリズムは，秘密計算でそのまま使おうとすると効率が大きく低下してしまう．その大きな要因の 1 つは，通常の計算機では定数時間とみなされる大きさ $n$ の配列上の指定した位置の要素へのアクセスが，秘密計算では位置を秘匿するために単純な方法では $\Theta(n)$ 時間を要してしまうことにある．そのため，例えば，ソート，文字列検索，単一始点最短経路，安定結婚問題など一部のアルゴリズムでは，専用の効率の良い秘密計算向けのアルゴリズムが設計されている．もし高速な配列アクセスが実現できれば，このような秘密計算向けアルゴリズムを再設計する必要がなくなり，秘密計算の適用範囲が格段に広がる．このような重要性から，秘密計算配列アクセスアルゴリズムの研究は近年盛んに行われてきているが，1 回の読み書きを $o(n)$ の通信量で実現する既存の方式は，いずれも直列の通信回数 (通信ラウンド) が $\Omega(\log n)$ と大きかった．提案アルゴリズムは $O(\sqrt{n}\log^2 n)$ の通信量，かつ $O(1)$ の通信ラウンドで秘密計算配列アクセスを実現する．さらに実装評価を行い，提案アルゴリズムは大きさ $n = 2^{20}$ の配列へのアクセスを既存の最速の結果に比べて約 5.8 倍速い 0.0340 秒で実現することを確認した．

**キーワード**：秘密計算，RAM

# A Sublinear-Communications Constant-Rounds Array Access Algorithm for Secure Multi-party Computation

Koki Hamada[1]

***Abstract:*** In this paper, we propose an algorithm that efficiently performs memory access while keeping target address secret by secure multi-party computation (MPC). Recently, many studies on memory access algorithms for MPC have been actively conducted. As long as our knowledge, our algorithm is the first one that exhibits both sublinear communications and constant rounds simultaneously. We also demonstrated the practical efficiency of our algorithms by experiments.

***Keywords:*** Secure multi-party computation, RAM

## 1. はじめに

近年，個人に関する様々な情報を容易に取得できる環境が整い，データ分析技術の進歩と相まって，個人に関する情報の活用への期待が高まっている．その一方で，個人情報保護やプライバシの観点から個人に関する情報は極めて慎重な取扱いが必要とされ，データの保護と活用をどう両立させるかが問題となっている．

このようなデータの保護と活用の両立を目指して，プライバシ保護データ分析 (Privacy-Preserving Data Analysis: PPDA) 技術の研究が盛んになってきている．PPDA の有力なアプローチの一つに，秘密計算や秘匿関数計算，マルチパーティプロトコルと呼ばれる，暗号学に基づいた手法がある．秘密計算は Yao による基本的なアイディア [12] を端緒とする技術であり，暗号化などの方法でデータを秘匿化したまま一度も元のデータに戻すことなく任意の計算

[1] NTT セキュアプラットフォーム研究所
NTT Secure Platform Laboratories

**表 1** 秘密計算上での大きさ $n$ の配列に対するアクセスの漸近的な通信コストの比較．*は償却通信量であることを示す．$\tilde{O}(\cdot)$ は $\log\log n$ 倍を省略した $O(\cdot)$ を表す．

| 手法 | 通信量 | 通信ラウンド |
|---|---|---|
| 全要素アクセス (2.4.8 節) | $O(n\log^2 n)$ | $O(1)$ |
| SCSL ORAM [6] | $O(\log^4)$ | $\Omega(\log n)$ |
| Path ORAM [6] | $\tilde{O}(\log^3)$ | $\Omega(\log n)$ |
| Circuit ORAM [11] | $O(\log^3 n)$ | $\Omega(\log n)$ |
| Square-Root ORAM [13] | $O(\sqrt{n\log^3 n})$* | $\Omega(\log n)$ |
| 提案手法 | $O(\sqrt{n}\log^2 n)$* | $O(1)$ |

を行う．秘密計算を用いることで通常のデータ分析と同等の結果を高いプライバシ保護の下で得ることができるが，秘密計算では通常の計算機上の計算に比べて処理速度が低下してしまうことが実用上の課題となっている．その原因は大きく二つある．

一つは，基本演算のオーバーヘッドである．秘密計算ではデータの秘匿性を保つために，乗算のような通常の計算機では一命令で実行可能な基本演算にも複雑な処理を必要とする．その結果，処理時間も大きくなってしまう．これに対しては近年改良が進んでおり，Ben-David らによる FairplayMP [1] など，効率のよい基本演算を備えたフレームワークの提案，実装が行われている．

もう一つは，回路に基づいた構成による計算量の悪化である．秘密計算は Goldreich ら [3] や Ben-Or ら [2] により提案された回路に基づく構成方法を用いることで，任意の計算を実現することができる．この構成方法では，所望の計算を論理回路で表現し，秘密計算上の基本的な演算の組み合わせで論理回路を模倣することにより任意の計算をデータを秘匿したまま行う．

しかしながら，この回路に基づいた一般的な構成方法を用いると，実用上重要な多くのアルゴリズムで通常の計算機上での計算に比べて計算量が大きくなってしまう．基本演算の効率化は定数倍の改善に過ぎず，多量のデータを扱うためには，計算量の悪化の解決が不可欠である．このため，特定の処理ごとに秘密計算上の効率的なアルゴリズムを設計する研究が進んでおり，Nishide と Ohta による比較 [9]，Ning と Xu による剰余計算 [8]，Goodrich によるソート [5] などが提案されている．

### 1.1 RAM 計算

これらのアルゴリズムは秘密計算向けに設計されたもので，通常の計算機上で使われるアルゴリズムとは異なる．通常の計算機上のアルゴリズムが秘密計算で使われない大きな理由の一つは，配列のどの要素にアクセスしたのかを秘匿するために通常は定数時間とみなされる大きさ $n$ の配列へのアクセスに単純な方法では $\Omega(n)$ 時間を要してしまうことである．そのため，秘密計算上で単純に適用しようとしても計算量が大きくなってしまう．

この問題に対して，配列へのアクセスパターンを秘匿する ORAM [4] と呼ばれる手法が利用されてきている．ORAM はサーバとクライアントの二者間でクライアントが配列のどこにアクセスしたかを知られないままデータにアクセスする手法である．アクセスパターンを秘匿できるという性質を利用して，ORAM を秘密計算上で実現し，アクセスパターンを秘匿した配列アクセスを効率よく実現しようという研究が行われてきている [6], [11], [13]．ORAM を利用することで劣線形通信量でメモリアクセスを実現でき，漸近的には効率的な手法が実現されているが，多くの手法は実時間では線形通信量の自明な方法 (2.4.8 節) による配列アクセスの方が高速であった．近年 Zahur らによって実時間でも効率的な秘密計算での配列アクセスが実現されている [13]．

### 1.2 本研究の貢献

本研究では，秘密計算上で安全に配列アクセスを実現する手法を提案する．表 1 に示すように，提案するアルゴリズムは著者らの知る限り初めて劣線形の通信量かつ定数の通信ラウンドで秘密計算上で配列アクセスを実現する．

また，提案アルゴリズムは以下の特徴を持ち，構造が単純である．

- 提案アルゴリズムは (外部から見る限り) 決定的で，確率的な振る舞いがない．そのため，SCSL ORAM [6] や Path ORAM [6] などこれまで提案された多くの手法のように確率的に失敗 (オーバーフローなど) することがなく，失敗確率を下げるためのオーバーヘッドが不要である．
- 提案アルゴリズムは初期化のための特別な計算が不要である．これまで提案されてきた手法の多くは構造が複雑で，初期化に大きな時間を要するものが多かった．例えば Zahur らの Square-Root ORAM の実装 [13] は $n = 2^{20}$ の場合約 1000 秒を初期化に要している．提案手法は，秘匿化された値の配列をそのまま使うため，初期化で特別な計算が不要である．
- 提案アルゴリズムは著者らの知る限り，再帰構造を要しない初めての劣線形通信量の手法である．これまで提案されてきた手法はいずれも大きさ $n$ の配列のアクセスのためにポジションマップと呼ばれる $\Theta(n)$ の配列へのアクセスを必要とする構造となっており，自信よりも少し小さい配列へ再帰的にアクセスするため通信ラウンドが $\Omega(\log n)$ 必要となっていた．

このような構造の単純さから，表 1 のように漸近的には通信量が先行研究の手法に劣るものの，実用上の効率の良さが期待される．実際，3 パーティの秘密計算上での実装評価の結果，大きさ $n = 2^{20}$ のメモリアクセスが 0.0340 秒で実現できることがわかった．これは著者らが知る限りこれまで最速だった Zahur ら [13] の結果の約 5.8 倍の配列アクセス速度である．

## 2. 準備

本稿で提案するアルゴリズムは，秘密計算上の演算の組み合わせにより構築される．本節では，本稿で用いる記法，定義と提案アルゴリズムがサブルーチンとして用いる既存の秘密計算上の各演算の説明を行う．

### 2.1 計算効率の尺度

本稿では秘密計算がマルチパーティプロトコルとして構成されるものとして計算効率の評価を行う．マルチパーティプロトコルは複数のパーティ間で通信を行いながら協調計算を行う方式である．一般的な構成では，各パーティが単独で行うローカルの計算に比べて通信に要する時間が著しく大きい．このため，ローカルの計算は無視できるものとみなし，通信したデータの量 (通信量) と一度に送ることのできるデータ量に制限がない場合に要する通信回数 (通信ラウンド) の 2 つの尺度で計算効率の評価を行う．本稿では，パーティ数を定数とみなして評価する．

#### 2.1.1 償却計算量

多くのデータ構造は内部状態を持ち，内部状態に応じて計算量が異なる場合がある．償却計算量はこのような状態に応じた計算量の違いを考慮に入れて一連の十分な回数の実行全体での計算量を考える．本稿では書き込みアルゴリズムを償却通信量で評価する．

### 2.2 記法，定義

$p$ を素数とし，本稿で扱う値はすべて素体 $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ 上の値とする．$\mathbb{F}_p$ の大きさを $\ell = \lceil \log_2 p \rceil$ とする．ベクトルや行列の添字は 0 始まりとし，ベクトル $\boldsymbol{a}$ の第 $i$ 要素を $\boldsymbol{a}[i]$ と書く．すなわち，ベクトル $\boldsymbol{a}$ の大きさが $n$ のとき，$\boldsymbol{a} = (\boldsymbol{a}[0], \boldsymbol{a}[1], \ldots, \boldsymbol{a}[n-1])$ である．ベクトル $\boldsymbol{a}$ とベクトル $\boldsymbol{b}$ を連結したベクトルを $\boldsymbol{a} \,||\, \boldsymbol{b}$ と書く．すなわち，$\boldsymbol{a} = (\boldsymbol{a}[0], \ldots, \boldsymbol{a}[m-1])$，$\boldsymbol{b} = (\boldsymbol{b}[0], \ldots, \boldsymbol{b}[n-1])$ のとき，$\boldsymbol{a} \,||\, \boldsymbol{b} = (\boldsymbol{a}[0], \ldots, \boldsymbol{a}[m-1], \boldsymbol{b}[0], \ldots, \boldsymbol{b}[n-1])$ である．

### 2.3 秘密計算

本稿では素体 $\mathbb{F}_p$ 上の秘密計算向けのアルゴリズムを提案する．提案するアルゴリズムは，本節で延べる各基本演算を提供する秘密計算であれば，手法に依存しない．

本稿では特に，線形演算に通信を伴わず，かつ，内積が乗算と同じ通信量で実現できる秘密計算を仮定する．このような方式としては，$(2,3)$-複製秘密分散や $(k, 2k+1)$-Shamir の秘密分散 [10] などが知られている．

### 2.4 既存の秘密計算上の演算

本稿で提案する配列アクセスアルゴリズムは，既存の秘密計算上の演算の組み合わせにより構成する．本稿で用いる各演算は，semi-honest な攻撃者に対して情報理論的に安全に以下で述べる機能を実現する．

#### 2.4.1 秘匿化，復元

$a \in \mathbb{F}_p$ を暗号化や秘密分散などの手段で秘匿化した値を $a$ の秘匿文と呼び，$[\![a]\!]$ と表記する．また，$a$ を $[\![a]\!]$ の平文と呼ぶ．ベクトル $\boldsymbol{v} = (\boldsymbol{v}[0], \ldots, \boldsymbol{v}[n-1])$ の各要素を秘匿化したベクトル $([\![\boldsymbol{v}[0]]\!], \ldots, [\![\boldsymbol{v}[n-1]]\!])$ を $[\![\boldsymbol{v}]\!]$ と表記する．秘匿化と復元は通信量 $O(\ell)$，通信ラウンド $O(1)$ であるものとする．

#### 2.4.2 算術演算

加算，減算，乗算の各演算は 2 つの値 $a, b \in \mathbb{F}_p$ の秘匿文 $[\![a]\!], [\![b]\!]$ を入力とし，それぞれ $a+b$，$a-b$，$ab$ の計算結果 $c_1$，$c_2$，$c_3$ の秘匿文 $[\![c_1]\!]$，$[\![c_2]\!]$，$[\![c_3]\!]$ を計算する．これらの演算の実行をそれぞれ，

$$[\![c]\!] \leftarrow \mathsf{Add}([\![a]\!], [\![b]\!]),$$
$$[\![c]\!] \leftarrow \mathsf{Sub}([\![a]\!], [\![b]\!]),$$
$$[\![c]\!] \leftarrow \mathsf{Mul}([\![a]\!], [\![b]\!])$$

と記述する．誤解を招く恐れのない場合は，$\mathsf{Add}([\![a]\!], [\![b]\!])$，$\mathsf{Sub}([\![a]\!], [\![b]\!])$，$\mathsf{Mul}([\![a]\!], [\![b]\!])$ をそれぞれ $[\![a]\!] + [\![b]\!]$，$[\![a]\!] - [\![b]\!]$，$[\![a]\!] \times [\![b]\!]$ と略記する．加算，減算は通信量およびラウンド数はともに 0，乗算は通信量 $O(\ell)$，通信ラウンド $O(1)$ であるものとする．

#### 2.4.3 内積

内積の演算は，2 つの長さ $n$ のベクトル $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}_p^n$ の秘匿文 $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$ を入力とし，$c = \boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=0}^{n-1} \boldsymbol{a}[i]\boldsymbol{b}[i]$ を満たす $c$ の秘匿文 $[\![c]\!]$ を計算する．この演算の実行を

$$[\![c]\!] \leftarrow \mathsf{InnerProd}([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$$

と記述する．通信量 $O(\ell)$，通信ラウンド $O(1)$ であるものとする．

#### 2.4.4 等号判定

等号判定の演算は 2 つの値 $a, b \in \mathbb{F}_p$ の秘匿文 $[\![a]\!], [\![b]\!]$ を入力とし，$a = b$ の真偽値 $c$ の秘匿文 $[\![c]\!]$ を計算する．真偽値は真のとき 1，偽のとき 0 とする．この演算の実行を

$$[\![c]\!] \leftarrow ([\![a]\!] \stackrel{?}{=} [\![b]\!])$$

と記述する．Nishide と Ohta [9] により，$\mathbb{F}_p$ の大きさを $\ell$ とすると，通信量 $O(\ell^2)$，通信ラウンド $O(1)$ の手法が提案されている．

#### 2.4.5 Demux

Demux 演算は，1 つの値 $a \in \mathbb{F}_p$ の秘匿文 $[\![a]\!]$ と出力の長さ $n$ を入力とし，$a$ 番目の要素だけが 1 で他が 0 であるような長さ $n$ のベクトル $\boldsymbol{b}$ の秘匿文 $[\![\boldsymbol{b}]\!]$ を計算する．この演算の実行を

$$[\![\boldsymbol{b}]\!] \leftarrow \mathsf{Demux}([\![a]\!], n)$$

と記述する．Launchbury ら [7] により通信量 $O(\ell^2 n)$，通信ラウンド $O(1)$ の手法が提案されている．

**Algorithm 1** 辞書読み込み

**Notation:** $[\![y]\!] \leftarrow \mathsf{DictionaryRead}([\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!], [\![x]\!])$.
**Input:** キーを表す大きさ $n$ のベクトルの秘匿文 $[\![\boldsymbol{k}]\!]$，値を表す大きさ $n$ のベクトルの秘匿文 $[\![\boldsymbol{v}]\!]$，読み込み位置の秘匿文 $[\![x]\!]$.
**Output:** 秘匿文 $[\![y]\!]$．ただし，$y = \sum_{i s.t. \boldsymbol{k}[i]=x} \boldsymbol{v}[i]$.

1: **for** $i \in [0, n)$ **do in parallel**
2: $\quad [\![\boldsymbol{e}[i]]\!] \leftarrow (\boldsymbol{k}[i] \stackrel{?}{=} [\![x]\!])$.
3: **return** $\mathsf{InnerProd}([\![\boldsymbol{v}]\!], [\![\boldsymbol{e}]\!])$.

#### 2.4.6 公開値剰余，公開値除算

公開値剰余，公開値除算の演算は，$a \in \mathbb{F}_p$ の秘匿文 $[\![a]\!]$ と公開値 $b \in \mathbb{F}_p$ を入力とし，$c_1 = a \mod b, c_2 = \lfloor \frac{a}{b} \rfloor$ を満たす $c_1, c_2$ の秘匿文を計算する．この演算の実行を

$$[\![c_1]\!] \leftarrow \mathsf{PubMod}([\![a]\!], b),$$
$$[\![c_2]\!] \leftarrow \mathsf{PubDiv}([\![a]\!], b)$$

と記述する．Ning と Xu [8] により通信量 $O(\ell^2)$，通信ラウンド $O(1)$ の公開値剰余の計算手法が提案されている．公開値除算は $([\![a]\!] - (\mathsf{PubMod}([\![a]\!], b))) \times b^{-1}$ により計算できる．

#### 2.4.7 辞書読み込み

辞書読み込みの演算は，大きさ $n$ の連想配列からの読み込みを行う処理である．入力は $[\![\boldsymbol{k}]\!]$ と $[\![\boldsymbol{v}]\!]$，$[\![x]\!]$ で，各 $(\boldsymbol{k}[i], \boldsymbol{v}[i])$ は要素 $\boldsymbol{k}[i]$ に紐付けられた値が $\boldsymbol{v}[i]$ であることを表す．出力は $x$ に紐付けられた値の秘匿文 $[\![y]\!]$ である．もし $\boldsymbol{k}$ に $x$ が複数回含まれる場合には，紐付けられた値の総和が出力される．アルゴリズムを Algorithm 1 に示す．通信量は $O(n\ell^2)$，通信ラウンドは $O(1)$ である．

#### 2.4.8 全要素アクセスの配列読み書き

配列読み書きの演算は，大きさ $n$ の配列からの読み込み，書き込みを行う処理である．読み込みの入力は配列の秘匿文 $[\![\boldsymbol{a}]\!]$ と読み込み位置の秘匿文 $[\![x]\!]$，書き込みの入力は配列の秘匿文 $[\![\boldsymbol{a}]\!]$ と書き込み位置の秘匿文 $[\![x]\!]$，書き込む値 $[\![y]\!]$ である．

配列読み込みは，Algorithm 1 で $[\![\boldsymbol{k}]\!]$ の代わりに $(0, 1, \ldots, n-1)$ を使うことで実現できる．このアルゴリズムは通信量が $O(n\ell^2)$，通信ラウンドが $O(1)$ である．

配列書き込みは，$\boldsymbol{a}$ の位置 $x$ に値 $y$ を書き込んだ配列の秘匿文 $[\![\boldsymbol{b}]\!]$ を出力し，

(1) 各 $i \in [0, n)$ について $[\![\boldsymbol{e}[i]]\!] \leftarrow (i \stackrel{?}{=} [\![x]\!])$.
(2) 各 $i \in [0, n)$ について $[\![\boldsymbol{b}[i]]\!] \leftarrow [\![\boldsymbol{e}[i]]\!] \times [\![y]\!] + (1 = [\![\boldsymbol{e}[i]]\!]) \times [\![\boldsymbol{a}[i]]\!]$.

により実現できる．このアルゴリズムは通信量が $O(n\ell^2)$，通信ラウンドが $O(1)$ である．

## 3. 提案手法

本稿では，配列の読み書きを，読み書きした位置と配列を秘匿したまま効率よく秘密計算上で実現する方法を提案する．まず，3.1 節で大きさ $n$ の配列から $O(\ell^2\sqrt{n})$ の通信量で読み込みを行う方法を，3.2 節で配列に $k$ 個の要素を

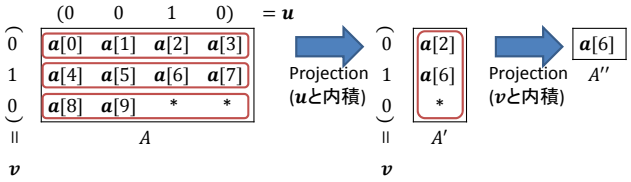


**図 1** 配列読み込みアルゴリズムの実行例．$n = 10$，$m_0 = 4$，$m_1 = 3$，$x = 6$ の場合の例である．*は任意の値を表す．

**Algorithm 2** 配列読み込み

**Notation:** $[\![y]\!] \leftarrow \mathsf{ArrayRead}([\![\boldsymbol{a}]\!], [\![x]\!])$.
**Input:** 読み込み対象の配列を表す大きさ $n$ のベクトルの秘匿文 $[\![\boldsymbol{a}]\!]$，読み込み位置の秘匿文 $[\![x]\!]$.
**Output:** 秘匿文 $[\![y]\!]$．ただし，$y = [\![\boldsymbol{a}[x]]\!]$.

1: $m := \lceil \sqrt{n} \rceil$ とする．
2: $[\![x_0]\!] \leftarrow \mathsf{PubMod}([\![x]\!], m)$,
$\quad [\![x_1]\!] \leftarrow \mathsf{PubDiv}([\![x]\!], m)$.
3: $[\![\boldsymbol{u}]\!] \leftarrow \mathsf{Demux}([\![x_0]\!], m)$,
$\quad [\![\boldsymbol{v}]\!] \leftarrow \mathsf{Demux}([\![x_1]\!], \lceil \frac{n}{m} \rceil)$.
4: **return** $\mathsf{Projection}(\mathsf{Projection}([\![\boldsymbol{a}]\!], [\![\boldsymbol{u}]\!]), [\![\boldsymbol{v}]\!])$.

**Algorithm 3** 射影

**Notation:** $[\![\boldsymbol{b}]\!] \leftarrow \mathsf{Projection}([\![\boldsymbol{a}]\!], [\![\boldsymbol{u}]\!])$.
**Input:** 大きさ $n$ のベクトルの秘匿文 $[\![\boldsymbol{a}]\!]$，大きさ $m$ のベクトルの秘匿文 $[\![\boldsymbol{u}]\!]$
**Output:** 大きさ $\lceil \frac{n}{m} \rceil$ のベクトルの秘匿文 $[\![\boldsymbol{b}]\!]$．ただし，各 $i \in [0, \lceil \frac{n}{m} \rceil)$ について $\boldsymbol{b}[i] = \sum_{j=0}^{\min(m, n-mi)-1} \boldsymbol{a}[mi+j]\boldsymbol{u}[j]$.

1: $[\![\boldsymbol{a}]\!]$ を $[\![\boldsymbol{a}]\!] = [\![\boldsymbol{a}_0]\!] \,||\, [\![\boldsymbol{a}_1]\!] \,||\, \cdots$ となるように大きさ $m$ の部分ベクトル $[\![\boldsymbol{a}_0]\!], [\![\boldsymbol{a}_1]\!], \ldots$ に分割．ただし最後の部分ベクトルは大きさが $m$ になるように末尾に $[\![0]\!]$ を追加．
2: **for** $i \in [0, \lceil \frac{n}{m} \rceil)$ **do in parallel**
3: $\quad [\![\boldsymbol{b}[i]]\!] \leftarrow \mathsf{InnerProd}([\![\boldsymbol{a}_i]\!], [\![\boldsymbol{u}]\!])$.
4: **return** $[\![\boldsymbol{b}]\!]$.

$O(\ell(n + \ell k\sqrt{n\ell}))$ の通信量で加算する方法を，それぞれ提案する．3.3 節ではこれらの方法を部品として用い，読み込みを常に通信量 $O(\ell^2\sqrt{n})$，通信ラウンド $O(1)$ で，書き込みを償却通信量 $O(\ell^2\sqrt{n})$，通信ラウンド $O(1)$ で，同時に実現するデータ構造である秘密計算 RAM を提案する．

なお，本稿で提案するアルゴリズムは $n$ が $\mathbb{F}_p$ の要素，すなわち $n < p$ である必要がある．このため，素体のビット長 $\ell$ は $\ell = \Theta(\log n)$ であるとする．

### 3.1 配列読み込みアルゴリズム

配列読み込みアルゴリズムは，大きさ $n$ の配列の秘匿文 $[\![\boldsymbol{a}]\!]$ と読み込み位置の秘匿文 $[\![x]\!]$ が与えられたとき，配列 $\boldsymbol{a}$ の $x$ 番目の要素 $\boldsymbol{a}[x]$ の秘匿文 $[\![\boldsymbol{a}[x]]\!]$ を $o(n)$ の通信量かつ $O(1)$ の通信ラウンドで計算する．

#### 3.1.1 アルゴリズム

配列の大きさ $n$ がある自然数 $m_0, m_1$ に対して $n \le m_1 m_0$ を満たしているとする．提案アルゴリズムは配列の各要素を $m_1 \times m_0$ 行列上に配置し，(1) 行方向で射影して $m_1 \times 1$ 行列に圧縮，(2) 列方向で射影して $1 \times 1$ 行列に圧縮，の 2 段階の計算により読み込みを実現する．

最初に，配列 $\boldsymbol{a}$ の $i$ 番目の要素 $\boldsymbol{a}[i]$ が行列の $\lfloor \frac{i}{m_0} \rfloor$ 行 $i \bmod m_0$ 列の要素となるように $\boldsymbol{a}$ の要素を $m_1 \times m_0$ 行列上に配置し，この行列を $A$ とする．$n < m_1 m_0$ の場合は $\boldsymbol{a}$ の要素を配置しなかった $A$ の要素は任意の値とする．図 1 に $n = 10$，$m_0 = 4$ の場合の例を示す．配列 $\boldsymbol{a}$ の $x$ 番目の要素は $x_0 := x \bmod m_0$，$x_1 := \lfloor \frac{x}{m_0} \rfloor$ であるとき，$A$ の第 $x_1$ 行第 $x_0$ 列の要素に該当する．

次に，$A$ の第 $x_0$ 列を取り出した $m_1 \times 1$ 行列 $A'$ を計算する．これは，$x_0$ 番目だけが 1 で他が 0 の大きさ $m_0$ のベクトル $\boldsymbol{u}$ を作成し，$A$ の各行と $\boldsymbol{u}$ の内積を縦に $m_1$ 個並べることで実現する．例えば，$n = 10$，$m_0 = 4$，$x = 6$ の場合は $x_0 = 2$，$x_1 = 1$ で $\boldsymbol{u} = (0, 0, 1, 0)$ である (図 1)．

続いて，$A'$ の第 $x_1$ 行を取り出した $1 \times 1$ 行列 $A''$ を計算する．これも先程と同様に，$x_1$ 番目だけが 1 で他が 0 の大きさ $m_1$ のベクトル $\boldsymbol{v}$ を作成し，$A'$ の各行と $\boldsymbol{v}$ の内積を唯一の要素に持つ行列を作ることで実現する．例えば，$n = 10$，$m_0 = 4$，$x = 6$ の場合は $\boldsymbol{v} = (0, 1, 0)$ である (図 1)．

得られた $1 \times 1$ 行列 $A''$ の唯一の要素は $A$ の第 $x_1$ 行第 $x_0$ 列の要素であり，すなわち配列 $\boldsymbol{a}$ の $x$ 番目の要素である．アルゴリズムを Algorithm 2 に示す．このアルゴリズムでは，漸近的な通信量を小さくするため $m_0 = m = \lceil \sqrt{n} \rceil$，$m_1 = \lceil \frac{n}{m} \rceil$ としている．$\boldsymbol{u}$，$\boldsymbol{v}$ の秘匿文の作成には Demux$(\cdot)$ を使う．また，射影の計算は Algorithm 3 に示す Projection$(\cdot)$ により実現する．

#### 3.1.2 安全性

このアルゴリズムは途中で一切の値を復元しない．従って，2.4 節の各演算が安全であれば，このアルゴリズムもまた安全である．

#### 3.1.3 効率

公開値除算と公開値剰余を並列に 1 回ずつ，大きさ $m_0$，$m_1$ の Demux$(\cdot)$ を並列に 1 回ずつ，大きさ $m_0$ の内積を並列に $m_1$ 回，大きさ $m_1$ の内積を 1 回実行する．$m_0 = O(\sqrt{n})$，$m_1 = O(\sqrt{n})$ であるので，通信量は $O(\ell^2 \sqrt{n})$，通信ラウンドは $O(1)$ である．

#### 3.1.4 一般化

Algorithm 2 は 1 回の内積が乗算と同じ効率でできる秘密計算で $O(\ell^2 \sqrt{n})$ の通信量を実現したが，一般に $k-1$ 回の直列の内積が乗算と同じ効率で実現可能な秘密計算 (例えば $(2, 1+k)$-Shamir 秘密計算) では一辺 $\Theta(n^{\frac{1}{k}})$ の $k$ 次元立方体内に配列の要素を配置し，Algorithm 2 と同様に $k$ 本のベクトルで順に射影の計算を行うことで通信量 $O(\ell^2 n^{\frac{1}{k}})$，通信ラウンド $O(1)$ で配列の読み込みを実現することができる．

### 3.2 一括配列書き込みアルゴリズム

一括配列書き込みアルゴリズムは，大きさ $n$ の配列 $\boldsymbol{a}$ への $k$ 回分の値の加算を効率良く行う．入力は書き込みを行う大きさ $n$ の配列の秘匿文 $[\![\boldsymbol{a}]\!]$ と，$\boldsymbol{a}[\boldsymbol{x}[i]]$ に値 $\boldsymbol{d}[i]$ を加算することを表現する 2 つの大きさ $k$ の配列の秘匿文 $[\![\boldsymbol{x}]\!]$，$[\![\boldsymbol{d}]\!]$ である．出力は大きさ $n$ の配列の秘匿文 $[\![\boldsymbol{b}]\!]$ で，$\boldsymbol{b}[i] = \boldsymbol{a}[i] + \sum_{j \text{ s.t. } \boldsymbol{x}[j]=i} \boldsymbol{d}[j]$ を満たす．単純な方法 (各 $j$ について，Demux$([\![\boldsymbol{x}[j]]\!])$ の各要素に $[\![\boldsymbol{d}[j]]\!]$ を乗じたベクトルを $[\![\boldsymbol{a}]\!]$ に加算) では $\Omega(kn\ell^2)$ の通信量が必要となるが，提案アルゴリズムは $O((n + k\sqrt{n}\ell)\ell)$ で実現する．

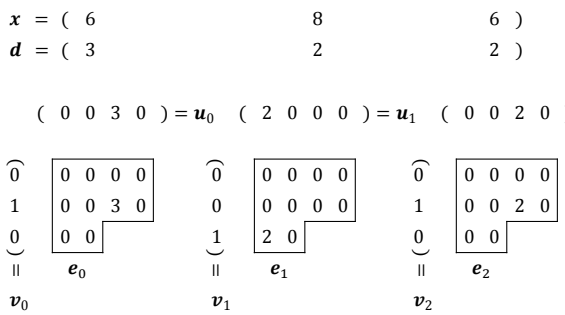


**図 2** 一括書き込みアルゴリズムの実行例 1．$n = 10$，$m_0 = 4$，$m_1 = 3$ の場合の例．

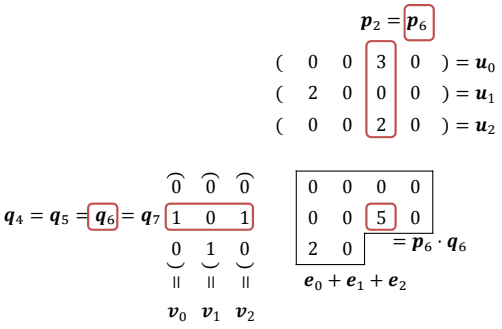


**図 3** 一括書き込みアルゴリズムの実行例 2．図 2 の続き．

**Algorithm 4** 一括配列書き込み

**Notation:** $[\![\boldsymbol{b}]\!] \leftarrow \mathsf{BatchWrite}([\![\boldsymbol{a}]\!], [\![\boldsymbol{x}]\!], [\![\boldsymbol{d}]\!])$.
**Input:** 配列を表す大きさ $n$ のベクトルの秘匿文 $[\![\boldsymbol{a}]\!]$，大きさ $k$ のベクトルの秘匿文 $[\![\boldsymbol{x}]\!]$，$[\![\boldsymbol{d}]\!]$.
**Output:** 配列を表す大きさ $n$ のベクトルの秘匿文 $[\![\boldsymbol{b}]\!]$．ただし，各 $i \in [0, n)$ に対して $\boldsymbol{b}[i] = \boldsymbol{a}[i] + \sum_{j \text{ s.t. } \boldsymbol{x}[j]=i} \boldsymbol{d}[j]$.

1: $m := \lceil \sqrt{n} \rceil$ とする．
2: **for** $j \in [0, k)$ **do in parallel**
3: $\quad [\![\boldsymbol{x}_0[j]]\!] \leftarrow \mathsf{PubMod}([\![\boldsymbol{x}[j]]\!], m)$,
$\quad [\![\boldsymbol{x}_1[j]]\!] \leftarrow \mathsf{PubDiv}([\![\boldsymbol{x}[j]]\!], m)$.
4: **for** $j \in [0, k)$ **do in parallel**
5: $\quad [\![\boldsymbol{u}'_j]\!] \leftarrow \mathsf{Demux}([\![\boldsymbol{x}_0[j]]\!], m)$,
$\quad [\![\boldsymbol{v}_j]\!] \leftarrow \mathsf{Demux}([\![\boldsymbol{x}_1[j]]\!], \lceil \frac{n}{m} \rceil)$.
6: **for** $j \in [0, k), i \in [0, m)$ **do in parallel**
7: $\quad [\![\boldsymbol{u}_j[i]]\!] \leftarrow [\![\boldsymbol{u}'_j[i]]\!] \times [\![\boldsymbol{d}[j]]\!]$.
8: 各 $j \in [0, k), i \in [0, n)$ に対して，$[\![\boldsymbol{p}_i[j]]\!] := [\![\boldsymbol{u}_j[i \bmod m]]\!]$，$[\![\boldsymbol{q}_i[j]]\!] := [\![\boldsymbol{v}_j[\lfloor \frac{i}{m} \rfloor]]\!]$ とする．
9: **for** $i \in [0, n)$ **do in parallel**
10: $\quad [\![\boldsymbol{b}[i]]\!] \leftarrow [\![\boldsymbol{a}[i]]\!] + \mathsf{InnerProd}([\![\boldsymbol{p}_i]\!], [\![\boldsymbol{q}_i]\!])$.
11: **return** $[\![\boldsymbol{b}]\!]$.

#### 3.2.1 アルゴリズム

提案アルゴリズムは，書き込む位置と値を表す各組 $(\boldsymbol{x}[j], \boldsymbol{d}[j])$ について，大きさ $n$ で $\boldsymbol{x}[j]$ 番目の要素が $\boldsymbol{d}[j]$ で他の要素がすべて 0 であるベクトル $\boldsymbol{e}_j$ を考え，

$\boldsymbol{b}[i] = \boldsymbol{a}[i] + \sum_{j=0}^{k-1} \boldsymbol{e}_j[i]$ となるベクトル $\boldsymbol{b}$ を計算する．$\boldsymbol{b}$ の計算を効率良く行うため，$\boldsymbol{e}_j$ が 2 つのベクトル $\boldsymbol{u}_j$ と $\boldsymbol{v}_j$ のテンソル積で表現できることを利用する．すなわち，ある $m$ について $\boldsymbol{e}_j[i] = \boldsymbol{u}_j[i \mod m]\boldsymbol{v}_j[\lfloor \frac{i}{m} \rfloor]$ と表されることを利用し，$\boldsymbol{b}[i] = \boldsymbol{a}[i] + \sum_{j=0}^{k-1} \boldsymbol{u}_j[i \mod m]\boldsymbol{v}_j[\lfloor \frac{i}{m} \rfloor]$ により計算する．

最初に，配列読み込みアルゴリズムと同様に，配列の要素を $n \leq m_1 \times m_0$ を満たす $m_1 \times m_0$ 行列の要素と対応付け，書き込む位置 $\boldsymbol{x}[j]$ の行列上の行番号 $\boldsymbol{x}_1[j]$ と列番号 $\boldsymbol{x}_0[j]$ を求める．例えば，図 2 のように $n = 10$，$x = (6, 8, 6)$，$\boldsymbol{d} = (3, 2, 2)$, $m_0 = 4$ のとき，$\boldsymbol{x}[0] = 6$ であるので $\boldsymbol{x}_1[0] = 6 \mod 4 = 2$，$\boldsymbol{x}_1[0] = \lfloor \frac{6}{4} \rfloor = 1$ である．

次に，テンソル積が $m_1 \times m_0$ 行列で第 $\boldsymbol{x}_1[j]$ 行第 $\boldsymbol{x}_0[j]$ 列の要素だけが $\boldsymbol{d}[j]$ で他がすべて 0 となるような 2 つのベクトル $\boldsymbol{u}_j$ と $\boldsymbol{v}_j$ を作成する．$\boldsymbol{u}_j$ は大きさ $m_0$ で $\boldsymbol{x}_0[j]$ 番目の要素だけが $\boldsymbol{d}[j]$ で他がすべて 0 のベクトルである．$\boldsymbol{v}_j$ は大きさ $m_1$ で $\boldsymbol{x}_1[j]$ 番目の要素だけが 1 で他がすべて 0 のベクトルである．例えば，図 2 のように $n = 10$，$\boldsymbol{x} = (6, 8, 6)$，$\boldsymbol{d} = (3, 2, 2)$，$m_0 = 4$ のときは $\boldsymbol{v}_0 = (0, 1, 0)$ $\boldsymbol{u}_0 = (0, 0, 3, 0)$ であり，これらのテンソル積は $\boldsymbol{e}_0 = (0, 0, 0, 0, 0, 0, 3, 0, 0, 0)$ を $3 \times 4$ 行列に上から順に左から並べて残りを 0 で埋めた行列

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

と一致する．

最後に，各 $\boldsymbol{u}_j$ から $i \mod m_0$ 番目の要素 $\boldsymbol{u}_j[i \mod m_0]$ を取り出して並べたベクトル $\boldsymbol{p}_i$ と，各 $\boldsymbol{v}_j$ から $\lfloor \frac{i}{m_0} \rfloor$ 番目の要素 $\boldsymbol{v}_j[\lfloor \frac{i}{m_0} \rfloor]$ を取り出して並べたベクトル $\boldsymbol{q}_i$ を作成し，$\boldsymbol{b}[i] = \boldsymbol{a}[i] + \boldsymbol{p}_i \cdot \boldsymbol{q}_i$ により $\boldsymbol{b}$ を計算する．例えば，図 3 のように上述の例では $6 \mod 4 = 2$，$\lfloor \frac{6}{4} \rfloor = 1$ であるので $\boldsymbol{p}_6 = (\boldsymbol{u}_0[2], \boldsymbol{u}_1[2], \boldsymbol{u}_2[2]) = (3, 0, 2)$，$\boldsymbol{q}_6 = (\boldsymbol{v}_0[1], \boldsymbol{v}_1[1], \boldsymbol{v}_2[1]) = (1, 0, 1)$ となり，$\sum_{j=0}^{2} \boldsymbol{e}_j[6] = 5 = \boldsymbol{p}_6 \cdot \boldsymbol{q}_6$ が成立している．

アルゴリズムを Algorithm 4 に示す．このアルゴリズムでは，配列読み込みアルゴリズムと同様に漸近的な通信量を小さくするため $m_0 = m = \lceil \sqrt{n} \rceil$，$m_1 = \lceil \frac{n}{m} \rceil$ としている．

#### 3.2.2 安全性

このアルゴリズムは途中で一切の値を復元しない．従って，2.4 節の各演算が安全であれば，このアルゴリズムもまた安全である．

#### 3.2.3 効率

アルゴリズムは公開値除算と公開値剰余を並列にそれぞれ $k$ 回，大きさ $m_0$，$m_1$ の $\mathsf{Demux}(\cdot)$ を並列にそれぞれ $k$ 回，乗算を並列に $km_0$ 回，内積を並列に $n$ 回実行する．$m_0 = O(\sqrt{n})$，$m_1 = O(\sqrt{n})$ であるので，通信量は $O((n + k\sqrt{n}\ell)\ell)$ で通信ラウンドは $O(1)$ である．

**Algorithm 5** 秘密計算 RAM の読み込み

**Notation:** $[\![y]\!] \leftarrow \mathsf{RAM\text{-}Read}(([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!]), [\![x]\!])$.
**Input:** $\mathrm{RAM}([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!])$，読み込み位置 $[\![x]\!]$.
**Output:** $[\![y]\!]$．ただし，$y$ は RAM の位置 $x$ の値.

1: $[\![y]\!] \leftarrow \mathsf{ArrayRead}([\![\boldsymbol{a}]\!], [\![x]\!]) + \mathsf{DictionaryRead}([\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!], [\![x]\!])$.
2: **return** $[\![y]\!]$.

### 3.3 秘密計算 RAM

秘密計算 RAM は大きさ $n$ の仮想的な配列に対する**読み込み**と**書き込み**の 2 つの操作を配列の値と対象位置を秘匿したまま実現するデータ構造である．読み込みの操作は，仮想的な配列の指定された位置の値を出力する処理で，通信量 $O(\ell^2\sqrt{n})$，通信ラウンド $O(1)$ で実現される．書き込みの操作は，仮想的な配列の指定された位置の値を上書きする処理で，償却通信量 $O(\ell^2\sqrt{n})$，通信ラウンド $O(1)$ で実現される．

#### 3.3.1 アルゴリズム

配列読み込みアルゴリズムを利用することで読み込みは $o(n)$ の通信量で実現できる．一方，2.4.8 節の配列書き込みアルゴリズムや 3.2 節の一括配列書き込みアルゴリズムは一つの値を書き込むのに $\Omega(n)$ の通信量を必要とし，コストが大きい．そこで，書き込みをその都度ではなくまとめて一括配列書き込みアルゴリズムを使って実行することにより，書き込みの償却通信量を $o(n)$ にすることを目指す．具体的には，$T$ を配列の大きさ $n$ から決めるある値として，書き込む値が $T$ 個蓄積するまでは配列への反映 (フラッシュと呼ぶ) を保留したまま読み込みを実現し，書き込む値が $T$ 個となった時点でまとめて $T$ 個分の書き込みを行うこととする．

秘密計算 RAM は配列を表すベクトル $\boldsymbol{a}$ の秘匿文と $\boldsymbol{a}$ への反映を保留された書き込みの位置と値を表す 2 つのベクトル $\boldsymbol{k}$，$\boldsymbol{v}$ の秘匿文による 3 つ組 $([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!])$ により構成される．読み込みと書き込みのいずれの操作後も以下の関係が保たれるように処理を行う．

**関係 1.** 大きさ $n$ の秘密計算 $RAM$ $([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!])$ の各位置 $x \in [0, n)$ の値が $\boldsymbol{a}[x] + \sum_{\boldsymbol{k}[i]=x} \boldsymbol{v}[i]$ と一致する．

##### 3.3.1.1 初期化

最初に一度だけ行う初期化の処理では，配列の初期値を表す大きさ $n$ の配列の秘匿文 $[\![\boldsymbol{a}_0]\!]$ から秘密計算 $\mathrm{RAM}([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!])$ を作成する．具体的には，$([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!]) := ([\![\boldsymbol{a}_0]\!], (), ())$ とする．ここで $()$ は大きさ 0 のベクトルを表す．初期化後は明らかに関係 1 が成り立つ．

##### 3.3.1.2 RAM 読み込み

秘密計算 RAM $([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!])$ からの読み込みは，書き込み反映済みの配列 $[\![\boldsymbol{a}]\!]$ からの読み込みを配列読み込みアルゴリズム (Algorithm 2) により，書き込み前の値を表す $[\![\boldsymbol{k}]\!]$，$[\![\boldsymbol{v}]\!]$ からの読み込みを辞書読み込みアルゴリズム (Algorithm 1) によりそれぞれ行い，これらの出力の和を出

**Algorithm 6** 秘密計算 RAM の書き込み

**Notation:**
$([\![\boldsymbol{a}']\!], [\![\boldsymbol{k}']\!], [\![\boldsymbol{v}']\!]) \leftarrow \mathsf{RAM\text{-}Write}(([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!]), [\![x]\!], [\![y]\!])$.
**Input:** 秘密計算 RAM$([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!])$，書き込み位置の秘匿文 $[\![x]\!]$，書きこむ値の秘匿文 $[\![y]\!]$.
**Output:** 秘密計算 RAM$([\![\boldsymbol{a}']\!], [\![\boldsymbol{k}']\!], [\![\boldsymbol{v}']\!])$．ただし，秘密計算 RAM$([\![\boldsymbol{a}']\!], [\![\boldsymbol{k}']\!], [\![\boldsymbol{v}']\!])$ の番地 $i$ の値は $i = x$ の場合は $\boldsymbol{x}$，その他の場合は $([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!])$ の番地 $i$ の値と等しい．

1: $[\![y']\!] \leftarrow \mathsf{RAM\text{-}Read}(([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!]), [\![x]\!])$.
2: $[\![d]\!] \leftarrow [\![y]\!] - [\![y']\!]$.
3: $[\![\boldsymbol{k}'']\!] \leftarrow [\![\boldsymbol{k}]\!] \mid\mid ([\![x]\!])$.
4: $[\![\boldsymbol{v}'']\!] \leftarrow [\![\boldsymbol{v}]\!] \mid\mid ([\![d]\!])$.
5: **if** $[\![\boldsymbol{k}'']\!]$ の大きさが $T$ 未満 **then**
6: $\quad([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}'']\!], [\![\boldsymbol{v}'']\!])$ を出力して終了.
7: $[\![\boldsymbol{a}'']\!] \leftarrow \mathsf{BatchWrite}([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}'']\!], [\![\boldsymbol{v}'']\!])$.
8: **return** $([\![\boldsymbol{a}'']\!], (), ())$.

力とする．アルゴリズムを Algorithm 5 に示す．読み込み前に関係 1 が成り立っていたとすると，配列読み込みの出力は $\boldsymbol{a}[x]$，辞書読み込みの出力は $\sum_{\boldsymbol{k}[i]=x} \boldsymbol{v}[i]$ であるので，これらの和である出力 $y$ は秘密計算 RAM $([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!])$ の位置 $x$ の値と一致する．また，読み込みは秘密計算 RAM に変更を加えないので，読み込み後も関係 1 が成立する．

#### 3.3.1.3 RAM 書き込み

秘密計算 RAM $([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}]\!], [\![\boldsymbol{v}]\!])$ への書き込みでは，まず最初に RAM からの読み込みアルゴリズムを使って書き込み位置 $x$ の現在の値 $y'$ を取得し，書き込む値 $y$ と $y'$ の差 $d$ と書き込み位置 $x$ の秘匿文 $[\![d]\!]$，$[\![x]\!]$ をそれぞれ $[\![\boldsymbol{v}]\!]$，$[\![\boldsymbol{k}]\!]$ に追加して $[\![\boldsymbol{v}'']\!]$，$[\![\boldsymbol{k}'']\!]$ とする．

$[\![\boldsymbol{k}'']\!]$ の大きさが $T$ 未満の場合は $([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}'']\!], [\![\boldsymbol{v}'']\!])$ が出力される．

$[\![\boldsymbol{k}]\!]$ の大きさが $T$ 以上の場合にはフラッシュを行う．すなわち，書き込みを保留していた値を $[\![\boldsymbol{a}]\!]$ に反映し，$[\![\boldsymbol{k}]\!]$ と $[\![\boldsymbol{v}]\!]$ を空にする．値の反映は一括書き込みアルゴリズムにより実現される．アルゴリズムを Algorithm 6 に示す．Algorithm 6 のステップ 7 とステップ 8 の処理をフラッシュと呼ぶ．

書き込み前に関係 1 が成り立っていたとすると，(i) $[\![\boldsymbol{k}'']\!]$ の大きさが $T$ 未満の場合は出力 $([\![\boldsymbol{a}]\!], [\![\boldsymbol{k}'']\!], [\![\boldsymbol{v}'']\!])$ の位置 $x$ の値は $\boldsymbol{a}[x] + \sum_{\boldsymbol{k}''[i]=x} \boldsymbol{v}''[i] = d + \boldsymbol{a}[x] + \sum_{\boldsymbol{k}[i]=x} \boldsymbol{v}[i] = d + y' = y$ であるので，書き込み後も関係 1 が成り立つ．(ii) $[\![\boldsymbol{k}'']\!]$ の大きさが $T$ 以上の場合は一括配列書き込みアルゴリズムの性質から $\boldsymbol{a}''[x] = \boldsymbol{a}[x] + \sum_{\boldsymbol{k}''[i]=x} \boldsymbol{v}''[i] = y$ であり，出力 $([\![\boldsymbol{a}'']\!], (), ())$ は関係 1 を満たす．

### 3.3.2 安全性

初期化，読み込み，書き込みのいずれの操作も途中で一切の値を復元しない．従って，2.4 節の各演算が安全であれば，これらのアルゴリズムもまた安全である．

### 3.3.3 効率

フラッシュの周期を $T = \lceil\sqrt{n}\rceil$ とすると，読み込みは常に通信量 $O(\ell^2\sqrt{n})$ で通信ラウンド $O(1)$，書き込みはフラッシュを除いて常に通信量 $O(\ell^2\sqrt{n})$ で通信ラウンド $O(1)$ である．また，フラッシュは通信量 $O(\ell^2 n)$ で $O(1)$ ラウンドである．フラッシュは $T$ 回の書き込みに一回しか行わないので，$m$ を整数として $mT$ 回の書き込み全体の通信量は $O(m\ell^2 n)$ となる．従って，書き込みの償却通信量は $O(\ell^2\sqrt{n})$ である．

$\ell = \Theta(\log n)$ であるため，読み込みは常に通信量 $O(\sqrt{n}\log^2 n)$ で通信ラウンド $O(1)$，書き込みは償却通信量 $O(\sqrt{n}\log^2 n)$ で通信ラウンド $O(1)$ である．

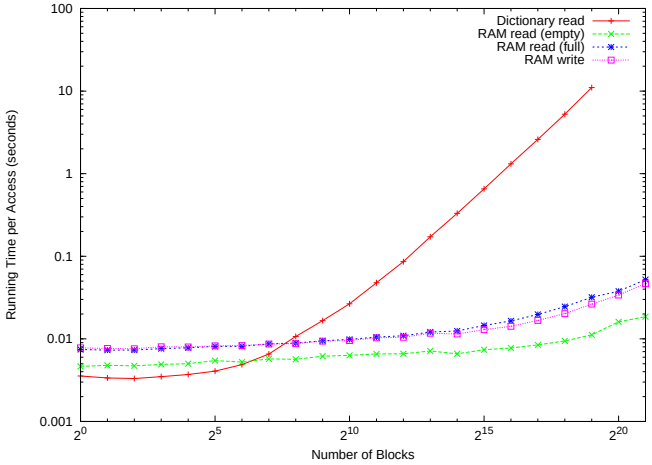


**図 4** 測定結果

**表 2** ベンチマークの結果．単位は秒

| ベンチマーク | パラメータ | 提案手法 | Zahur ら [13] |
|---|---|---|---|
| 2 分探索 | 1 回 | 0.295 | 10.41 |
| | $2^5$ 回 | 7.216 | 26.25 |
| | $2^{10}$ 回 | 230.501 | 824.81 |
| 2 分探索 (並列) | 1 回 | 0.294 | - |
| | $2^5$ 回 | 0.672 | - |
| | $2^{10}$ 回 | 10.244 | - |
| 幅優先探索 | $n = 2^2$ | 0.165 | 0.34 |
| | $n = 2^5$ | 1.147 | 4.08 |
| | $n = 2^{10}$ | 52.535 | 679.63 |

## 4. 評価

提案手法の実用上の有効性を確認するため，提案手法を実装して処理時間を測定した．

### 4.1 実装

$p = 2^{32} - 5$ とし，$\mathbb{F}_p$ 上の $(2,3)$-複製秘密分散で実装を行った．実用上の性能を見るために，いくつかの漸近的に効率的な演算を実用上効率的だが漸近的には効率が悪い演算に置き換えて実装を行った．具体的には，以下の変更を行った．

- $m = 2^{\lceil \frac{\lceil \log_2 n \rceil}{2} \rceil}$ とし，公開値除算と公開値剰余の代わりにビット分解プロトコル [9] の結果の下位 $\lceil \frac{\lceil \log_2 n \rceil}{2} \rceil$ ビット分を $x_0$ に，残りを $x_1$ に割り当てた．
- Demux をビット表現された値に対して実行可能なを通信量 $\Theta(\ell n)$，通信ラウンド $\Theta(\log n)$ の Demux [7] に

置き換えた．

### 4.2 実験環境

3 台のラップトップ PC を 1 台のハブを介して 1Gbps の有線 LAN で互いに接続し，実験を行った．

3 台の PC は，CPU が Intel Core i7-2860QM 2.50GHz，メモリが 8 GB，SSD が 120 GB である．すべての PC の OS は Linux (Ubuntu 16.04.3) であり，プログラミング言語は C++ を，コンパイラは g++ 5.4.0 を用いて提案手法を実装した．

### 4.3 実験結果

#### 4.3.1 配列の読み書き

提案した秘密計算 RAM のフラッシュ直後の状態の読み込み (RAM read (empty))，フラッシュ直前の状態の読み込み (RAM read (full))，書き込み (RAM write) と，比較用に Algorithm 1 の辞書読み込み (Dictionary read) について配列の大きさ $n$ を変えながら一回のアクセス時間の測定を行った．結果を図 4 に示す．各測定結果は，提案手法は必ずフラッシュが起こるように $n \in [2^0, 2^{12}]$ のとき 100 回，$n \in [2^{13}, 2^{18}]$ のとき 1000 回，$n \in [2^{19}, 2^{21}]$ のとき 2000 回をそれぞれ連続して実行した結果の平均値である．辞書読み込みは $n \in [2^0, 2^{19}]$ のとき 100 回の測定結果の平均値である．提案手法は $n = 2^{20}$ のとき書き込み 1 回あたり 0.0340 秒であり，先行研究 [13] の約 0.2 秒と比較して 5.8 倍の速度を達成している．

#### 4.3.2 ベンチマーク

また，ベンチマークとして，2 分探索と幅優先探索を実装した．2 分探索は [13] と同様，$n = 2^{15}$ の配列に対する 2 分探索を直列に規定回数実行した．幅優先探索は [13] と同様，節点数 $n$，各節点の次数 $8(n = 4$ のときのみ 3) のグラフに対する幅優先探索である．結果を表 2 に示す．提案手法の結果はすべて 3 回の測定結果の平均値である．提案手法では Zahur ら [13] に比べて数倍から 10 倍程度高速な結果が得られた．また，提案手法は並列のアクセスも可能であるため，2 分探索を直列ではなく並列にも実行した．表 2 のように，並列に実行した場合は Zahur ら [13] の直列の場合に比べて約 80 倍の性能を達成した．

## 5. おわりに

本稿では，秘密計算で対象位置を隠したまま配列アクセスを効率的に行うアルゴリズムを提案した．提案アルゴリズムは配列の大きさ $n$ に対して劣線形の $O(\sqrt{n}\log^2)$ 通信量かつ定数の通信ラウンドで秘密計算配列アクセスを実現した．また，提案アルゴリズムは既存の方式に比べて構成が単純で，実装評価により実用上も既存の最速の結果に比べても高速であることが確認できた．$n = 2^{20}$ の場合にはこれまで最速の結果の約 5.8 倍の 0.0340 秒で一回のアクセスを実現した．

### 参考文献


[1] Ben-David, A., Nisan, N. and Pinkas, B.: FairplayMP: a system for secure multi-party computation, *ACM Conference on Computer and Communications Security* (Ning, P., Syverson, P. F. and Jha, S., eds.), ACM, pp. 257–266 (2008).

[2] Ben-Or, M., Goldwasser, S. and Wigderson, A.: Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract), *STOC* (Simon, J., ed.), ACM, pp. 1–10 (1988).

[3] Goldreich, O., Micali, S. and Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority, *STOC*, ACM, pp. 218–229 (1987).

[4] Goldreich, O. and Ostrovsky, R.: Software Protection and Simulation on Oblivious RAMs, *J. ACM*, Vol. 43, No. 3, pp. 431–473 (online), DOI: 10.1145/233551.233553 (1996).

[5] Goodrich, M. T.: Randomized Shellsort: A Simple Oblivious Sorting Algorithm, *SODA*, pp. 1262–1277 (2010).

[6] Keller, M. and Scholl, P.: Efficient, Oblivious Data Structures for MPC, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II* (Sarkar, P. and Iwata, T., eds.), Lecture Notes in Computer Science, Vol. 8874, Springer, pp. 506–525 (online), DOI: 10.1007/978-3-662-45608-8 (2014).

[7] Launchbury, J., Diatchki, I. S., DuBuisson, T. and Adams-Moran, A.: Efficient lookup-table protocol in secure multiparty computation, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012* (Thiemann, P. and Findler, R. B., eds.), ACM, pp. 189–200 (online), DOI: 10.1145/2364527.2364556 (2012).

[8] Ning, C. and Xu, Q.: Multiparty Computation for Modulo Reduction without Bit-Decomposition and a Generalization to Bit-Decomposition, *ASIACRYPT*, pp. 483–500 (2010).

[9] Nishide, T. and Ohta, K.: Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol, *PKC*, pp. 343–360 (2007).

[10] Shamir, A.: How to Share a Secret, *Commun. ACM*, Vol. 22, No. 11, pp. 612–613 (1979).

[11] Wang, X., Chan, T. H. and Shi, E.: Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (Ray, I., Li, N. and Kruegel, C., eds.), ACM, pp. 850–861 (online), DOI: 10.1145/2810103.2813634 (2015).

[12] Yao, A. C.: How to Generate and Exchange Secrets (Extended Abstract), *FOCS*, pp. 162–167 (1986).

[13] Zahur, S., Wang, X. S., Raykova, M., Gascón, A., Doerner, J., Evans, D. and Katz, J.: Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation, *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, IEEE Computer Society, pp. 218–234 (online), DOI: 10.1109/SP.2016.21 (2016).