スケーラブルな二者間秘匿計算のサーバー構成と効率的な通信方法の実装

石田 祐介[1,2,a)]　桶谷 純一[1,2]　國井 淳[1,2]　大畑 幸矢[2]　花岡 悟一郎[2]

概要：秘密分散に基づく秘匿計算 (MPC) は計算量，通信ラウンド数，データ量において性能改善が進んでおり実用化が近いことが伺える．一方，MPC は複数のサーバーを要する構成であるため，実際のサービス化を考えると，純粋な性能以外にサーバーの管理と運用を考慮しなくてはならない．本論文では，このようなサービス化を考慮した場合の二者間秘匿計算の利点に着目しサーバー構成を検討した．特に，各計算パーティーにサービス提供サーバーと汎用依頼計算サーバーという役割を持たせることで，スケーラブルな構成にできることを説明する．加えて，この構成において，総データ量を変化させずに通信ラウンド数を半分程度にできる通信方法を示す．この方法はパーティー間の計算の並列性を若干犠牲にするものの，通信がボトルネックとなる WAN 環境においては有効である．最後に提案する構成と通信方法を実装しその効果を検証する．

キーワード：秘匿計算，MPC，通信，サーバー構成

A Scalable Server Configuration and Round-Efficient Implementation in Secure Two-Party Computation

Yusuke Ishida[1,2,a)]　Junichi Oketani[1,2]　Atsushi Kunii[1,2]　Satsuya Ohata[2]　Goichiro Hanaoka[2]

Abstract: The efficiency of protocols in secure two-party computation (2PC) has been improved. When we make 2PC practical, we also need to consider the management and operation of the servers. In this paper, we explain a (1) scalable server configuration (2) round-efficient implementation method in 2PC. Moreover, we implement our methods and show their performances via experiments.

Keywords: secure computation, MPC, communication, server configuration

1. はじめに

秘匿計算は古くから研究されていた [1], [2] が，その計算量の多さから長く理論研究レベルに留まってきた．しかし，近年のコンピューティング能力の進化と共にプロトコルの改善も進み，実用化が近いことが伺える．さらにプライバシー保護の観点から秘匿計算に対する社会的需要も高まりつつあり，いくつかの実装（例えば [3], [4], [5] ）も既に公開されている．秘匿計算を実現する技術として代表的なものは，秘密分散に基づく多者間秘匿計算 (multi party computation, MPC)，準同型暗号に基づく手法，ガーブルド回路に基づく手法などがある．本稿ではこれらの内，プロトコルの工夫と実装の最適化によって高いスループットが出せることが分かってきた秘密分散に基づく MPC を扱う．

[1] 株式会社 ZenmuTech
ZenmuTech Inc.
[2] 産業技術総合研究所 サイバーフィジカルセキュリティ研究センター
CPSEC (Cyber Physical Security Research Center), AIST
[a)] yusuke.ishida@zenmutech.com

1.1 既存研究における課題

近年研究されている高速な秘密分散に基づく MPC は 2-out-of-3 秘密分散を利用するものが多い [3], [6], [7]．これはシェアを 3 分散するため，必然的に三者間秘匿計算となる．二者間秘匿計算では乗算に必要な Beaver triple (multiplication triple とも呼ぶ) の生成がボトルネックであったが，クライアント補助型 (client aided model) を導入することでその課題を部分的に解決したものがある [8]．

ここで，実際に秘匿計算をサービスとして提供することを考えてみよう．三者間秘匿計算の場合， 3 サーバーを準備し相互に接続する構成としなければならない．さらに，これら 3 サーバーはいずれも結託することが許されない[*1]ので，各サーバーをどのように運用・管理すべきかが実用化における課題である．二者間秘匿計算の場合は必要なサーバーが 1 つ少ないので結託を考慮しなければならない対象が減る．これは，三者間秘匿計算に比べて考慮すべき運用課題が少なくなるが，実用化にあたってどのようなサーバー構成とすべきかは依然として課題である．さらに，秘匿計算対象の関数を各サーバーへどのように与えるかも検討する必要がある．これらは，理論研究の範ちゅう外の為，今までほとんど検討されてこなかった．しかし，近年秘匿計算の実用化が近いということを鑑みると，このような課題がより顕在化してくることが想定されるので，現実問題として考慮すべき課題である．

また，秘匿計算サービスとしての適用範囲を広げ，一般に普及させることを考えると，インターネット等の広域ネットワーク (WAN) 上で動作できることが望ましい．一方で WAN 環境では一般的に通信遅延が大きいので，その影響を顕著に受ける．この影響を小さくするためには通信ラウンド数を削減することが望ましい．プロトコル改善による通信ラウンド数の削減の研究が進められている [9], [10], [11] が，それでも例えば 32 bit 整数値の除算は 75 ラウンド必要 [12] であり，プロトコルの改善にも限界がある．

1.2 貢献

実用化を前提とした場合に顕在化する運用・管理の課題を解決するサーバー構成について検討する．特に本稿では実用的なサーバー構成として二者間秘匿計算の利点に着目したサーバー構成を提案する．2 つのサーバーそれぞれに，サービス提供サーバー (App サーバー) と汎用依頼計算サーバー (CoSC サーバー) という役割を持たせることで，スケーラブルなサーバー構成にできる．これは秘匿計算をサービスとして提供していく一形態として，より現実的な構成と考えている．

さらに，この構成において，総データ量を変化させずに通信ラウンド数を半分程度にできる通信方法を示す．従来のプロトコルでは N 回の通信ラウンドを必要としていたものに対して，提案手法では $\lfloor \frac{N+1}{2} \rfloor$ 回の通信ラウンドで計算完了できる．この手法はパーティー間の計算の並列性を若干犠牲にするものの，通信遅延がボトルネックとなる WAN 環境においては有効な手法である．

最後に提案するサーバー構成と通信方法を実装し，その効果を検証する．前述したように本手法は計算の並列性を犠牲にしている為，全体の計算時間に対して，通信遅延よりも計算時間の割合が大きくなると，従来手法に比べて本手法は不利となる．計算時間は主に処理対象の要素数 (バッチ数) に依存する為，提案手法は小規模バッチに対して効果を発揮する．実装検証では，実際どの程度のバッチ数までは提案手法の方が有利であるかも検証した．WAN 環境を想定した状況において 32 bit 値で 1,000 バッチ，64 bit 値で 200 バッチほどであれば提案手法のほうがトータル処理時間の面で有利であることを示した．

2. 準備

2.1 表記

秘密 x を秘密分散したシェアは $[\![x]\!]$ で表す．2 分散した場合は $[\![x]\!] = ([\![x]\!]_1, [\![x]\!]_2)$ であり，3 分散した場合は $[\![x]\!] = ([\![x]\!]_1, [\![x]\!]_2, [\![x]\!]_3)$ である．$\mathcal{CP}_i$ を i 番目の計算パーティーとすると， $[\![x]\!]_i$ は $\mathcal{CP}_i$ が持つ秘密 x のシェアを表す．シェアには算術値を扱う算術 (arithmetic) シェアと，真理値を扱うブーリアン (boolean) シェアがある．これらを区別する必要がある場合はそれぞれ $[\![x]\!]^{\mathsf{A}}$ と $[\![x]\!]^{\mathsf{B}}$ というように表記する．

$\mathbb{Z}_{2^k}$ 上の 2-out-of-2 秘密分散は 2 つのアルゴリズム Share と $\mathsf{Reconst}$ からなる．分散アルゴリズム Share は $x \in \mathbb{Z}_{2^k}$ を入力に取り， 2 つのシェア $([\![x]\!]_1, [\![x]\!]_2) \in \mathbb{Z}_{2^k}^2$ を出力する．再構成アルゴリズム $\mathsf{Reconst}$ は $[\![x]\!]$ を入力に取り，x を出力する．

2.2 加法秘密分散と二者間秘匿計算

本稿で用いる加法秘密分散とそれを用いた semi-honest 安全な二者間秘匿計算を説明する．分散・再構成アルゴリズムを以下のように構成する．

- $\mathsf{Share}(x)$: ランダムな $r \in \mathbb{Z}_{2^k}$ を選び，$[\![x]\!]^{\mathsf{A}}_1 = r$ と $[\![x]\!]^{\mathsf{A}}_2 = x - r$ を出力する
- $\mathsf{Reconst}([\![x]\!]^{\mathsf{A}}_1, [\![x]\!]^{\mathsf{A}}_2)$: $[\![x]\!]^{\mathsf{A}}_1 + [\![x]\!]^{\mathsf{A}}_2$ を出力する

これらのシェアの上での加算 $\mathsf{Add}(x, y) \stackrel{\text{def}}{=} x + y$ は，単にシェア同士を足すだけで実現できる．乗算 $\mathsf{Mult}(x, y) \stackrel{\text{def}}{=} xy$ は様々な実現方法があるが，本稿では Beaver triple に基づく手法 [13] を採用する．Beaver triple とは $a_1 + a_2 = a$, $b_1 + b_2 = b$, $c_1 + c_2 = c$, $ab = c$ という 4 つの条件を満たす 3 つ組乱数 (a_1, b_1, c_1) および (a_2, b_2, c_2) のことで，乗算プロトコルへの入力と関係なく事前に計算しておける．

[*1] いずれか二者が結託すると元の秘密情報を復元できる

パーティー $\mathcal{CP}_1$ は $[\![x]\!]_1^{\mathsf{A}} - a_1$ と $[\![y]\!]_1^{\mathsf{A}} - b_1$ を，パーティー $\mathcal{CP}_2$ は $[\![x]\!]_2^{\mathsf{A}} - a_2$ と $[\![y]\!]_2^{\mathsf{A}} - b_2$ をそれぞれ計算して送りあった後，互いのローカルで $\mathsf{Reconst}([\![x]\!]_1^{\mathsf{A}} - a_1, [\![x]\!]_2^{\mathsf{A}} - a_2)$ 及び $\mathsf{Reconst}([\![y]\!]_1^{\mathsf{A}} - b_1, [\![y]\!]_2^{\mathsf{A}} - b_2)$ を実行して $x' = x - a$ 及び $y' = y - b$ を再構成する．その後，$\mathcal{CP}_1$ は $[\![z]\!]_1^{\mathsf{A}} = x'y' + x'b_1 + y'a_1 + c_1$ を，$\mathcal{CP}_2$ は $[\![z]\!]_2^{\mathsf{A}} = x'b_2 + y'a_2 + c_2$ を計算すると，$\mathsf{Reconst}([\![z]\!]_1^{\mathsf{A}}, [\![z]\!]_2^{\mathsf{A}}) = xy$ となる．乗じる値が公開値 c の場合は，それぞれがローカルで自身のシェアを c 倍すればよい．このプロトコルは semi-honest 安全であることが証明されており，これらの組み合わせで構成するプロトコルの semi-honest 安全性は Composition Theorem [14] より成立するため，本稿ではこれ以降プロトコル自体の安全性に関して議論しない．また，加算を排他的論理和に置き換えることで，論理ゲート (XOR および AND) も計算可能である．論理否定 NOT は $\mathsf{NOT}([\![x]\!]^{\mathsf{B}}) = (\neg [\![x]\!]_1^{\mathsf{B}}, [\![x]\!]_2^{\mathsf{B}})$ で，論理和 OR は $\mathsf{OR}([\![x]\!]^{\mathsf{B}}, [\![y]\!]^{\mathsf{B}}) = \neg\mathsf{AND}(\neg [\![x]\!]^{\mathsf{B}}, \neg [\![y]\!]^{\mathsf{B}})$ で実現できる．

3. サーバー構成

秘密分散に基づく秘匿計算は MPC – Multi Party Computation – にて実現される．ここでは，MPC におけるサーバー構成について検討する．

3.1 クライアントサーバーモデルの MPC

本稿では，クライアント-サーバーモデルの MPC を扱う．このモデルでは n 個のサーバーと t 個のクライアントで構成さる．i ($i \in \{1, \ldots, t\}$) 番目のクライアントが秘密情報 x_i を持っており，クライアントは互いに自分自身の秘密情報を明かさずに，ある関数 f の計算結果 $f(x_1, \ldots, x_t)$ を得たい．このモデルでは，これを次のような流れで実施する．

(1) 各クライアントは秘密情報 x_i を秘密分散したシェア $[\![x_i]\!]_j$ ($j \in \{1, \ldots, n\}$) を 各サーバーへ送信する．
(2) 各サーバーはクライアントから得たシェアに対して互いに協調して計算を行い $[\![f(x_1, \ldots, x_t)]\!]_j$ を得る．
(3) クライアントは全てのサーバーからこの計算結果を集めて (受信して) 復元することで $f(x_1, \ldots, x_t)$ を得る．

3.2 三者間秘匿計算 (3PC) の実装

クライアント-サーバーモデル MPC の実装としては 3 パーティーで実施するものがある．例えば [3] では 図 1 のように，秘匿計算機能を提供する 3 つのサーバー群を，命令 (Instructions) と入力 (Input data) を受け，結果 (Result) を出力するブラックボックスと考える．このブラックボックスは仮想計算機 (Virtual machine) または仮想プロセッサ (Virtual processor) とみなすことができる．この実装においては，命令をコントローラー (Controller) が外部から

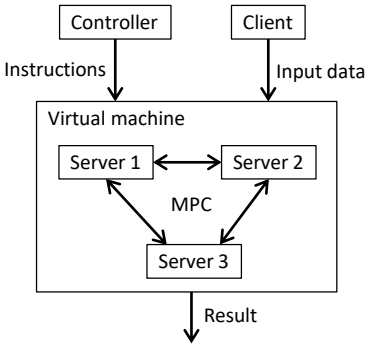

図 1 外部から命令を与える 3PC の概念図

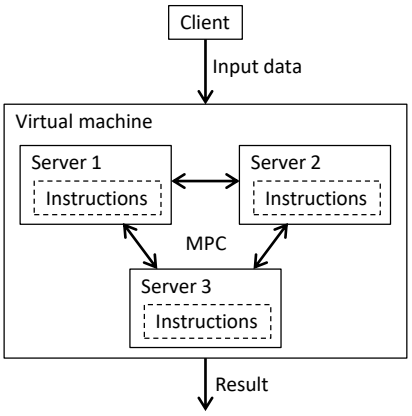

図 2 命令をサーバーに内包する 3PC の概念図

入力する必要がある．

このコントローラーに相当する部分は 図 2 のように，サーバーに内包することができる．例えば [4], [5] では，コンパイルされたコードを各サーバーに配置するこのモデルが採用されている．

3.3 二者間秘匿計算 (2PC) の実装

2 パーティーで秘匿計算を実現するものとして [15] 等が提案されている．今まで二者間秘匿計算においては，乗算に必要な Beaver triple の生成がボトルネックとなっていた．三者間秘匿計算では，乗算に必要な correlated randomness (Beaver triple に相当するもの) の生成を各パーティーのローカル演算で実施できるので，非常に効率的である．一方，二者間秘匿計算では Beaver triple の生成に通信と秘匿処理が必要であり，これが原因で三者間秘匿計算と比較して非効率であった．この課題を解決すべく [8] ではクライアント補助型 (client aided model) を導入し，クライアントから計算サーバーへ Beaver triple を供給する構成としている．本稿で扱う二者間秘匿計算はこのクライアント補助型とする．

3.4 サービス化を見据えた時の課題

前述したような 3PC 及び 2PC による実装は高速な秘匿計算の実装として現実的である一方で，これらを実際に

サービス化しようとすると次のような点が課題となる可能性がある．

サーバーの管理と運用面

3PC の場合，いずれか 2 つのサーバーが結託すると安全性が崩れる．よって，サーバー同士が結託しないような運用及び管理を実施する必要がある．お互いが利益を共有しない 3 者が個別にサーバーを運用することが望ましいと考えられるが，そのような 3 者が集まって同じサービスを提供するようなサービスモデルを成り立たせるのは難しい．

2PC の場合も 2 つのサーバーが結託すると安全性が崩れるが，参加するサーバーが 2 つであるので，互いに結託しないプレーヤーがサーバーを管理すれば良い．その点では， 3 者が参加しなければならない 3PC よりも 2PC の方が運用及び管理のハードルは低く， 3PC よりも 2PC の方がサービスモデルとして成り立つ可能性が高いと言える．

サーバー同士の結合度

既存の 3PC 及び 2PC による実装では，各サーバーがお互いを知っている (情報を送受信可能な) 状態にしておく必要がある．加えて， 図 2 のケースでは，サーバー内に命令が内包されており，それらは全てのサーバーで共有されている必要がある．このような実装は，各サーバーが密結合されている状態であり，いずれか 1 サーバーの変更が，他の全てのサーバーに影響してしまう．秘匿計算において多種多様なサービスを展開することを考えると，このような密結合された構成よりも，より疎結合な構成の方が望ましい．

3.5 スケーラブルなサーバー構成

これらの課題を鑑みて，本稿ではスケーラブルなサーバー構成を新たに提案する．まず， 3PC か 2PC かであるが，これは前述したように，サービスモデルとして成り立つ可能性が高い 2PC を前提とする．さらに，この 2 パーティーそれぞれに，サービス提供 (App) サーバーと汎用依頼計算 (Cooperative Secure Computation, CoSC) サーバーという役割を持たせる．CoSC サーバーには汎用的な秘匿計算命令セットがあらかじめ定義してある．App サーバーはその秘匿計算命令セットを任意に組み合わせて，サービス固有の処理 (instructions) を組み立てる．この模式図を図 3 に示す．App サーバーが実行したい秘匿計算命令は都度 CoSC サーバーへ送付する．App サーバーが持つサービス固有の処理は変更される可能性があるが，その場合，影響があるのは App サーバーのみであり， CoSC サーバーは何ら影響を受けない．さらに，これらサーバー間の通信は，必ず App サーバー から CoSC サーバーへ通信を開始する，リクエスト-レスポンス型とする．これによって， CoSC サーバーは App サーバーの状態やアドレス等を知らなくても，コネクションを確立し通信することができる．これら特徴により，前述の課題であったサーバー同士の結合度を，より疎結合にしている．

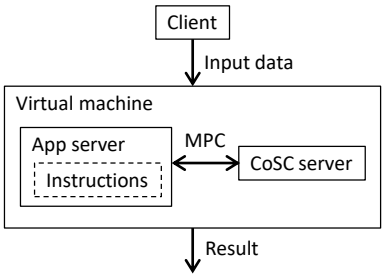

図 3 App サーバーと CoSC サーバーによる 2PC

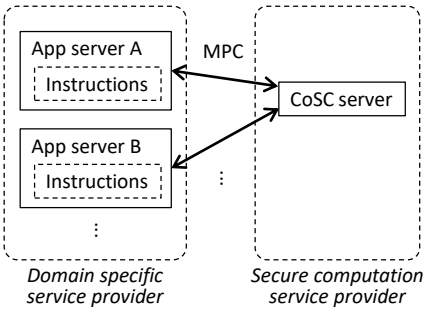

図 4 複数 App サーバーと単一 CoSC サーバーによる構成

さらに，App サーバーと CoSC サーバーが疎結合になったことで，秘匿計算システムとしてスケールしやすい構成となっている．CoSC サーバーは個々のサービスについて気にする必要が無い汎用的な秘匿計算サーバーという位置づけとなり，複数の App サーバーと連携して秘匿計算サービスを提供できる．この様子を図 4 に示す．App サーバーはドメイン固有のサービスにおいて自由に秘匿計算処理を設計できる．これは App サーバーを自由に拡張できるという点で，スケーラブルなサーバー構成となっている．

4. 通信ラウンド削減方法

前述の提案サーバー構成における通信方法と，その効率化について考える．

4.1 プロトコルの分割

通信を検討するための準備として，秘匿計算プロトコルを複数の Phase に分割する．Phase は他パーティーとパラメータ交換が必要となる直前までをまとめて構成したメソッドである．Phase_n は相手パーティーの Phase_{n-1} の出力 (交換パラメータ) を入力とし，相手パーティーの Phase_{n+1} で必要となる交換パラメータを出力する．ただし， Phase_0 は少なくとも計算対象オペランドを入力とし，Phase_N は計算結果シェアを出力とする．各計算パーティーの同じ秘匿計算プロトコルにおいては，同じ数の Phase を持ち，それぞれの計算パーティーが同期的に Phase の処理を進めることになる．

4.2 通信の効率化

一般に秘匿計算における通信ラウンド数は，計算パーティー間でパラメータ交換が必要な回数 N である．しかし，前述したような二者間秘匿計算において App サーバーと CoSC サーバーに役割分担し，App サーバーから CoSC サーバーへリクエストを送出する形で構成すると，通信ラウンド数を $\lfloor \frac{N+1}{2} \rfloor$ とすることができる．ここではその手順を説明する．

図 5 に提案通信手法の模式図を示す．本サーバー構成では App サーバーが実行すべき関数を知っている前提である．そこで，まず App サーバーは実行すべき秘匿計算プロトコルを選択し，その Phase_0 を実行する．Phase_0 の入力は計算対象オペランド (シェア) で，出力は CoSC サーバーの Phase_1 で必要とされる交換パラメータである．ここで， App サーバーは CoSC サーバーへリクエストを送信する．リクエストの内容としては，実行すべき秘匿計算プロトコル (instruction)，計算対象オペランド (operand)，Phase_0 の出力として得た交換パラメータ の 3 つである．なお，計算対象オペランドについてはシェアの実体ではなく， CoSC サーバーが所有しているシェアを特定する ID 等の情報である．App サーバーが CoSC サーバーで使用するシェアの実体を所有している訳ではないことに注意されたい．

リクエストを受信した CoSC サーバーでは instruction を元に実行すべき Phase_0 の選択と， operand を元にシェアの実体を取得する．そのシェアを Phase_0 へ入力し，出力として App サーバーの Phase_1 で必要とされる交換パラメータを得る．さらに，CoSC サーバーは Phase_1 で必要とされる交換パラメータも既に App サーバーから得ているので， Phase_1 も実行する．その結果，App サーバーの Phase_2 で必要とされる交換パラメータを得る．ここで，CoSC サーバーは App サーバーへレスポンスを返す．レスポンスの内容としては， Phase_0 の出力として得た交換パラメータと， Phase_1 の出力として得た交換パラメータの 2 つである．

レスポンスを受信した App サーバーは Phase_1 と Phase_2 の実行に必要な交換パラメータを得られたので，これら 2 つの Phase を実行する．そして，次のリクエストとしてこれらの出力を CoSC サーバーへ送信する．すると CoSC サーバーでは Phase_2 と Phase_3 を実行し，その結果をレスポンスとして返すことができる．つまり以降は，App サーバーが Phase_i と Phase_{i+1} を実行し，その出力を CoSC サーバーへリクエスト送信．CoSC サーバーでは Phase_{i+1} と Phase_{i+2} を実行しその結果を App サーバーへレスポンスとして返す，ということを繰り返す．このように， App サーバー， CoSC サーバーそれぞれが， 2 つの Phase を 1 回の通信 (リクエスト - レスポンス) で実行できるという点が本手法のポイントである．

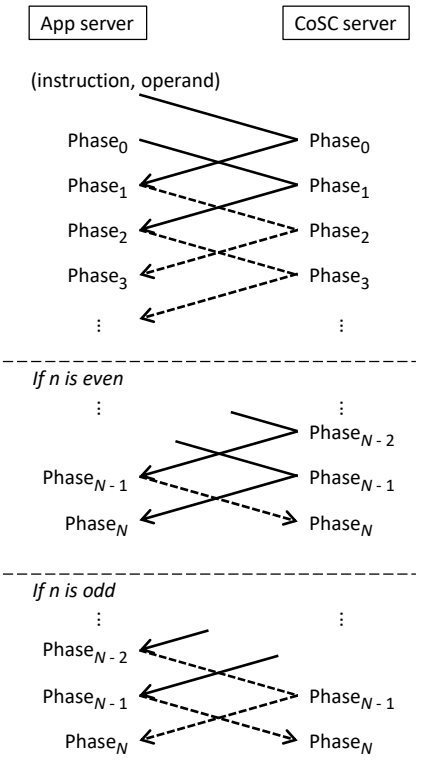

図 5 提案通信方式の模式図

最後の Phase_N は計算結果シェアを出力するのでパラメータ交換 (通信) は不要である．よって， N が偶数か奇数かによって最後の通信内容が変化する．N が偶数の場合は，最後に App サーバーから Phase_{N-1} の出力のみを送信し， CoSC サーバーで Phase_N を実行して終了する．N が奇数の場合は，最後に App サーバーから Phase_{N-2} と Phase_{N-1} の出力を送信し，CoSC サーバーで Phase_{N-1} と Phase_N を実行．レスポンスとして Phase_{N-1} の出力を返し， App サーバーは Phase_N を実行して終了する．

このように構成することで，通信ラウンド数を $\lfloor \frac{N+1}{2} \rfloor$ とすることができる．本手法では App サーバーまたは CoSC サーバーが，今知っている交換パラメータ全てを使い切るまで計算した後に，相手サーバーへ次の交換パラメータを送出する構成である．これは，相手サーバーが計算を終えないと次の計算を進められないことを意味しており，各サーバーで Phase を並行して実行することができない．この点が通信ラウンド数の削減と並列性のトレードオフとなっているので，実際に適用する際には全体の速度を鑑みて検討することが望ましい．なお，本提案の通信手法と従来の通信手法において，通信データ量は変わらない．

4.3 等価評価プロトコルでの例

具体的に等価評価プロトコル Equal の例で考えてみる．アルゴリズム 1 に 加法秘密分散に基づく 32 bit 整数値の二者間秘匿等価評価プロトコル [16] を示す．ここで $\mathcal{CP}_1$ と $\mathcal{CP}_2$ は計算パーティーであり，それぞれ App サーバー

Algorithm 1 32bit 算術値の等価評価プロトコル

Functionality: $[\![b]\!]^{\mathsf{B}} \leftarrow \mathsf{Equal}([\![x]\!]^{\mathsf{A}}, [\![y]\!]^{\mathsf{A}})$ where $b = 1$ iff $x = y$

Setup: $\mathcal{CP}_1, \mathcal{CP}_2$ は 必要数の Beaver triple を共有しておく

1: $\mathcal{CP}_1$ は次を計算 $[\![d]\!]_1^{\mathsf{B}} = [\![x]\!]_1^{\mathsf{A}} - [\![y]\!]_1^{\mathsf{A}}$
2: $\mathcal{CP}_2$ は次を計算 $[\![d]\!]_2^{\mathsf{B}} = [\![y]\!]_2^{\mathsf{A}} - [\![x]\!]_2^{\mathsf{A}}$
3: $i = 2j, j \in \{0, \cdots, 15\}$ に対して $[\![d_i]\!]^{\mathsf{B}} \leftarrow \mathsf{OR}([\![d_i]\!]^{\mathsf{B}}, [\![d_{i+1}]\!]^{\mathsf{B}})$
4: $i = 4j, j \in \{0, \cdots, 7\}$ に対して $[\![d_i]\!]^{\mathsf{B}} \leftarrow \mathsf{OR}([\![d_i]\!]^{\mathsf{B}}, [\![d_{i+2}]\!]^{\mathsf{B}})$
5: $i = 8j, j \in \{0, \cdots, 3\}$ に対して $[\![d_i]\!]^{\mathsf{B}} \leftarrow \mathsf{OR}([\![d_i]\!]^{\mathsf{B}}, [\![d_{i+4}]\!]^{\mathsf{B}})$
6: $i = 16j, j \in \{0, 1\}$ に対して $[\![d_i]\!]^{\mathsf{B}} \leftarrow \mathsf{OR}([\![d_i]\!]^{\mathsf{B}}, [\![d_{i+8}]\!]^{\mathsf{B}})$
7: $[\![b]\!]^{\mathsf{B}} \leftarrow \mathsf{OR}([\![d_0]\!]^{\mathsf{B}}, [\![d_{16}]\!]^{\mathsf{B}})$
8: $[\![b]\!]^{\mathsf{B}} \leftarrow \mathsf{NOT}([\![b]\!]^{\mathsf{B}})$
9: **return** $[\![b]\!]^{\mathsf{B}}$

または CoSC サーバーとする．このプロトコルにおいて，パラメータ交換が必要となるのは OR である．3 行目から 7 行目は d の各ビット毎に論理和をとっている．ここは，分割統治 (divide-and-conquer) により OR 演算を並列実行できるように構成し，通信ラウンド数を抑えている．このプロトコルの詳細については [16] を参照のこと．

アルゴリズム 1 を Phase に分解することを考えよう．前述したようにパラメータ交換が必要となるのは OR なので，その直前までを Phase としてひとまとめにしていく．$\mathcal{CP}_1$ と $\mathcal{CP}_2$ とで共通の処理を実行すれば良いものは共通の Phase_i として構成し，それぞれで異なる処理を実行する必要があるものは $\mathsf{Phase}_i^{\mathcal{CP}_1}$ と $\mathsf{Phase}_i^{CP_2}$ というように個別に構成する．共通の Phase_i として構成した場合でも，その実行時は $\mathcal{CP}_1$ と $\mathcal{CP}_2$ それぞれが独立して実行することに注意されたい．

まずは Phase_0 を構成する．Phase_0 では入力として計算対象のオペランドを受け取る．それは Equal プロトコルの入力である $[\![x]\!]$ と $[\![y]\!]$ を入力とすれば良い．次に，アルゴリズム 1 の 1 行目は $\mathcal{CP}_1$， 2 行目は $\mathcal{CP}_2$ の処理なので，これらを別メソッドとして構成しなければならない．Phase_0 として処理できるのは交換パラメータが必要となる 3 行目までである．OR はその実行に相手パーティーとのパラメータ交換が必要となる．そのため， Phase_0 では OR の交換パラメータ送出までとし，次の Phase_1 で相手パーティーの交換パラメータを受信し OR の実行を完了させる．Phase_0 の処理をアルゴリズム 2 にまとめた．

ここで $\mathbf{t}_i$ を Phase_i が出力する交換パラメータとし，特に $\mathcal{CP}_1$ と $\mathcal{CP}_2$ 向けの交換パラメータを区別して記載する

Algorithm 2 $\mathsf{Phase}_0^{\mathcal{CP}_1}$ と $\mathsf{Phase}_0^{\mathcal{CP}2}$

Functionality: $\mathbf{t}_0^{\mathcal{CP}_2} \leftarrow \mathsf{Phase}_0^{\mathcal{CP}_1}([\![x]\!]_1^{\mathsf{A}}, [\![y]\!]_1^{\mathsf{A}})$

1: $[\![d]\!]_1^{\mathsf{B}} = [\![x]\!]_1^{\mathsf{A}} - [\![y]\!]_1^{\mathsf{A}}$
2: $\mathbf{t}_0^{\mathcal{CP}_1} \leftarrow$ アルゴリズム 1 の 3 行目の OR を実行するための交換パラメータ
3: **return** $\mathbf{t}_0^{\mathcal{CP}_1}$

Functionality: $\mathbf{t}_0^{\mathcal{CP}_2} \leftarrow \mathsf{Phase}_0^{\mathcal{CP}_2}([\![x]\!]_2^{\mathsf{A}}, [\![y]\!]_2^{\mathsf{A}})$

4: $[\![d]\!]_2^{\mathsf{B}} = [\![y]\!]_2^{\mathsf{A}} - [\![x]\!]_2^{\mathsf{A}}$
5: $\mathbf{t}_0^{\mathcal{CP}_2} \leftarrow$ アルゴリズム 1 の 3 行目の OR を実行するための交換パラメータ
6: **return** $\mathbf{t}_0^{\mathcal{CP}_2}$

Algorithm 3 Phase_i, $i \in \{1, 2, 3, 4\}$

Functionality: $\mathbf{t}_i \leftarrow \mathsf{Phase}_i(\mathbf{t}_{i-1})$

1: $[\![d]\!]^{\mathsf{B}} \leftarrow \mathbf{t}_{i-1}$ を使用してアルゴリズム 1 の $3 + (i-1)$ 行目の OR 実行
2: $\mathbf{t}_i \leftarrow$ アルゴリズム 1 の $3 + i$ 行目の OR を実行するための交換パラメータ
3: **return** $\mathbf{t}_i$

場合にはそれぞれ $\mathbf{t}_i^{\mathcal{CP}_1}, \mathbf{t}_i^{\mathcal{CP}_2}$ と表記する．

次に Phase_1 を構成する．Phase_1 は入力として Phase_0 の出力である交換パラメータ $\mathbf{t}_0$ を受け取る．相手パーティーの交換パラメータを入手できたので，アルゴリズム 1 の 3 行目に相当する OR の実行を完了させることができる．アルゴリズム 1 の 4 行目では，再度 OR が必要となるので，Phase_0 と同様に OR を実行するための交換パラメータを出力する．アルゴリズム 1 の 3 行目から 7 行目では OR を連続的に実行しているので Phase_1 から Phase_4 の構成としては似たものになる．つまり， Phase_i で相手パーティの交換パラメータを受信し OR の実行を完了させた後，次の OR の交換パラメータを送出するという構成となる．この構成をアルゴリズム 3 に示す．

次に Phase_5 を構成する．まずは相手パーティから受け取った交換パラメータ $\mathbf{t}_4$ を使用して，アルゴリズム 1 の 7 行目の OR を完了させる．それ以降はパラメータ交換が必要な処理が無いので，Phase_5 が最後の Phase となる．アルゴリズム 1 の 8 行目 NOT は，$\mathcal{CP}_1$ と $\mathcal{CP}_2$ で異なる演算を実行する必要があるので，個別の Phase として構成する．Phase_5 をアルゴリズム 4 に示す．

このように，等価評価プロトコル Equal は Phase_0 から Phase_5 の 6 つの処理に分割できた．この時 $N = 5$ となり 5 回のパラメータ交換が必要であることが分かる．

ここで提案する通信手法でこのプロトコルを実行すると，$\lfloor \frac{5+1}{2} \rfloor = 3$ ラウンドで計算完了することができる．

Algorithm 4 $\mathsf{Phase}_5^{\mathcal{CP}_1}$ と $\mathsf{Phase}_5^{\mathcal{CP}2}$

Functionality: $[\![b]\!]_1^{\mathsf{B}} \leftarrow \mathsf{Phase}_5^{CP_1}(\mathbf{t}_4^{CP_2})$

1: $[\![b]\!]_1^{\mathsf{B}} \leftarrow \mathbf{t}_4^{CP_2}$ を使用してアルゴリズム 1 の 7 行目の OR 実行

2: $[\![b]\!]_1^{\mathsf{B}} \leftarrow \neg\, [\![b]\!]_1^{\mathsf{B}}$

3: **return** $[\![b]\!]_1^{\mathsf{B}}$

Functionality: $[\![b]\!]_2^{\mathsf{B}} \leftarrow \mathsf{Phase}_5^{CP_2}(\mathbf{t}_4^{CP_1})$

4: $[\![b]\!]_2^{\mathsf{B}} \leftarrow \mathbf{t}_4^{CP_1}$ を使用してアルゴリズム 1 の 7 行目の OR 実行

5: **return** $[\![b]\!]_2^{\mathsf{B}}$

5. 実装と実験評価

提案手法の有効性を実装し検証するため，[12] で示された除算プロトコルに基づき，提案したサーバー構成と通信方法を Python3.6(+Numpy v1.16.4) を用いて実装した．今回の実験では App サーバーと CoSC サーバーを仮想的に 1 台の PC 上で実行し，オンライン時間をその実験結果から計算で求める．実験には Core i5 4670 3.4GHz，32GB RAM のマシンを用いた．

まずは，1 要素同士の除算 1 回あたりの計算時間と通信データ量を実験から求めた．実験結果を表 1 に示す．計算時間については，App サーバーと CoSC サーバーが 1 回の除算プロトコルを完了するまでの時間を計測した．この時，通信時間は除外し，各サーバーで計算処理に要する時間のみを集計している．32 bit と 64 bit の整数値それぞれに対して 100 回ずつ計測し，その平均時間 (ms) を求めた．この平均時間は App サーバーと CoSC サーバー両方の計算時間が合算されているため，これを半分にして 1 ノードあたりの計算時間とした．通信データ量については，App サーバーから CoSC サーバーへのリクエストの通信量 (byte) を実測した．厳密にはリクエストのデータ量とレスポンスのデータ量は異なるが，交換パラメータの数としてはほとんど同じため，今回はリクエストの通信量のみを計測し，それを 1 ノードあたりの通信データ量として用いることとする．なお，今回の実験では簡単のため通信データをテキストデータ (JSON 形式) としているため，理論的なデータ量よりも大きくなっていることに注意されたい．

この実測値を踏まえて，提案通信方式の効果を検証してみる．我々が実装した除算プロトコルのラウンド数は，32 bit の平文空間で 75 回，64 bit の平文空間で 93 回である．本稿で提案する通信方法を用いれば，それぞれ 38 回および 47 回となる．広域ネットワークでの利用を前提として，通信のバンド幅 10MB/s，通信遅延 40ms という条件を設定する．オンライン時間は (データ転送時間) + (遅延時間) + (計算時間) で求めるものとする．

表 1 1 ノードあたりの性能評価結果

	計算時間 [ms]	通信量 [byte]
32bit	32.699	92,066
64bit	86.922	284,200

この条件下において，提案通信方法による 32 bit 除算プロトコル 1 回あたりに要するオンライン時間は，

$$(92066\cdot 2)/(10\cdot 1024\cdot 1024) + 40\cdot 38 + 32.699\cdot 2 \approx 1585.416$$

と計算できる．本提案方式の場合は各サーバーのデータ転送と計算を並列実行できないので，これらがシーケンシャルに実行されるものとし，データ転送時間 (データ量) と計算時間を 2 倍している．一方，従来通信方式の場合，同プロトコルを実行するのに要するオンライン時間は，

$$92066/(10\cdot 1024\cdot 1024) + 40\cdot 75 + 32.699 \approx 3032.708$$

と計算できる．従来通信方式の場合は，各サーバーのデータ転送と計算をお互い一切ロックすることなく理想的な形で並列実行できることとした．

このように 1 要素同士の 1 回の除算実行であれば，提案手法の方が高速であることが分かる．

一方，全体の処理時間に占める計算時間の割合が大きくなってくると，提案手法の方が不利になってくる．通信時間はそもそも小さいので無視し，従来手法のラウンド数 N に対して提案手法のラウンド数はおおよそ半分の $\frac{N}{2}$ として考えると，提案手法は従来手法に比べて (遅延時間) $\cdot \frac{N}{2}$ − (計算時間) だけ高速となる．この式が意味するところは，計算時間が大きくなると，この値が負となり，提案手法のほうが不利になるということである．

計算時間は計算対象の要素数 (バッチ数) によって非線形に変化する．そこで，要素数を変化させつつ，1 回の除算にかかる計算時間を計測した．その結果を表 2 に示す．今回の条件で考えると，32 bit 整数値の除算において，提案手法が不利になる境界は $\frac{40\cdot 75}{2} = 1,500$ ms なので，表 2 から約 1,200 要素未満においては提案手法が有利であるが，それ以降は全体速度として従来手法の方が有利である．同様に， 64 bit 整数値では，境界が $\frac{40\cdot 93}{2} = 1,860$ ms なので，表 2 から約 250 要素未満においては提案手法が有利で，それ以降は従来手法の方が有利である．

このように，提案手法は小規模バッチにおいては高速で有効な手法であることが確認されたのと同時に，実際には全体の速度を鑑みて通信方式を検討する必要があることが確認された．

6. まとめ

秘匿計算のサービス化を考慮した場合のサーバー構成を検討し，二者間秘匿計算によるスケーラブルなサーバー構

表 2　要素数に対する計算時間の評価結果

要素数	計算時間 [ms]	
	32 bit	64 bit
1	32.699	86.922
10	40.154	117.979
100	139.262	778.459
200	242.897	1580.148
250	287.758	1946.171
500	630.608	
1000	1229.365	
1200	1508.823	

成を新たに提案した．このサーバー構成では，各サーバーにサービス提供 (App) サーバーと汎用依頼計算 (CoSC) サーバーという役割を持たせることを特徴とする．これにより，サーバー間の結合度を疎結合にし，スケーラブルな構成にできることを示した．加えて，この構成において，サーバー間で N 回のパラメータ交換が必要である場合に，総データ量を変化させずに通信ラウンド数を半分程度 ($\lfloor \frac{N+1}{2} \rfloor$) にできる通信方法を示した．最後に，提案サーバー構成と通信方式を実装し，その効果を検証した．その結果，パーティー間の計算の並列性を若干犠牲にするものの，小規模なバッチ数で，通信がボトルネックとなる WAN 環境においては有効であることを示した．

謝辞　本研究の一部は JST CREST JPMJCR19F6 の支援を受けて行われた．

参考文献

[1] A. C. Yao: “How to generate and exchange secrets (extended abstract)”, 27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986, pp. 162–167 (1986).

[2] O. Goldreich, S. Micali and A. Wigderson: “How to play any mental game or A completeness theorem for protocols with honest majority”, Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA, pp. 218–229 (1987).

[3] D. Bogdanov, M. Niitsoo, T. Toft and J. Willemson: “High-performance secure multi-party computation for data mining applications”, Int. J. Inf. Sec., **11**, 6, pp. 403–418 (2012).

[4] M. Keller, P. Scholl and N. P. Smart: “An architecture for practical actively secure mpc with dishonest majority”, Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS ’13, New York, NY, USA, ACM, pp. 549–560 (2013).

[5] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara and H. Tsuchida: “Generalizing the spdz compiler for other protocols”, Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS ’18, New York, NY, USA, ACM, pp. 880–895 (2018).

[6] T. Araki, J. Furukawa, Y. Lindell, A. Nof and K. Ohara: “High-throughput semi-honest secure three-party computation with an honest majority”, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pp. 805–817 (2016).

[7] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell and A. Nof: “Fast large-scale honest-majority MPC for malicious adversaries”, Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III, pp. 34–64 (2018).

[8] P. Mohassel and Y. Zhang: “Secureml: A system for scalable privacy-preserving machine learning”, 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017, pp. 19–38 (2017).

[9] T. Schneider and M. Zohner: “GMW vs. yao? efficient secure two-party computation with low depth circuits”, Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers, pp. 275–292 (2013).

[10] H. Morita, N. Attrapadung, T. Teruya, S. Ohata, K. Nuida and G. Hanaoka: “Constant-round client-aided secure comparison protocol”, Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II, pp. 395–415 (2018).

[11] S. Ohata and K. Nuida: “Towards high-throughput secure MPC over the internet: Communication-efficient two-party protocols and its application”, CoRR, **abs/1907.03415**, (2019).

[12] H. Morita, N. Attrapadung, S. Ohata, K. Nuida, S. Yamada, K. Shimizu, G. Hanaoka and K. Asai: “Secure division protocol and applications to privacy-preserving chi-squared tests”, International Symposium on Information Theory and Its Applications, ISITA 2018, Singapore, October 28-31, 2018, pp. 530–534 (2018).

[13] D. Beaver: “Efficient multiparty protocols using circuit randomization”, Advances in Cryptology - CRYPTO ’91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings, pp. 420–432 (1991).

[14] O. Goldreich: “The Foundations of Cryptography - Volume 2: Basic Applications”, Cambridge University Press (2004).

[15] D. Demmler, T. Schneider and M. Zohner: “ABY - A framework for efficient mixed-protocol secure two-party computation”, 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015 (2015).

[16] S. Siim: “A comprehensive protocol suite for secure two-party computation”, Master’s thesis, University of Tartu (2016).