Secure Multiparty Linear Programming Using Fixed-Point Arithmetic

Octavian Catrina[1] and Sebastiaan de Hoogh[2]

[1] Dept. of Computer Science, University of Mannheim, Germany
[2] Dept. of Mathematics and Computer Science, TU Eindhoven, The Netherlands

Abstract. Collaborative optimization problems can often be modeled as a linear program whose objective function and constraints combine data from several parties. However, important applications of this model (e.g., supply chain planning) involve private data that the parties cannot reveal to each other. Traditional linear programming methods cannot be used in this case. The problem can be solved using cryptographic protocols that compute with private data and preserve data privacy. We present a practical solution using multiparty computation based on secret sharing. The linear programming protocols use a variant of the simplex algorithm and secure computation with fixed-point rational numbers, optimized for this type of application. We present the main protocols as well as performance measurements for an implementation of our solution.

Keywords: Secure multiparty computation, linear programming, secure fixed-point arithmetic, secret sharing.

1 Introduction

The optimization of processes involving multiple parties can often be formulated as a collaborative linear programming problem: minimize (or maximize) a linear objective function subject to a set of linear constraints, where the function and the constraints are defined by combining data from all parties. This linear program may include confidential data that the parties cannot reveal to each other. For example, a linear program for supply chain planning uses business data whose disclosure has negative effects on the participant's negotiation position and competition with other suppliers (e.g., production costs and available capacity) [11]. The supply chain partners cannot use traditional methods to solve the linear program, since this would reveal their confidential data.

Secure computation preserves input privacy using cryptographic protocols. Roughly speaking, the protocols ensure that the output is correct and the computation is carried out without revealing anything else besides the agreed upon output. However, the high communication and computation overhead of cryptographic protocols makes secure computation slower than usual computation with public data. Moreover, finding efficient protocols for complex applications like linear programming is a particularly challenging task. The solutions proposed so far rely on variants of the simplex algorithm that use integer arithmetic [13,19].

For non-trivial linear programs, these algorithms require computation with very large integers (thousands of bits) and the protocols become impractical. Our goal is to obtain more efficient protocols, suitable for practical applications.

Our contributions. We take a different approach to secure multiparty simplex, by using computation with rational numbers in fixed-point representation, and we provide a complete solution (all building blocks) as well as performance measurements with a prototype implementation. The protocols are structured into three main layers. The core layer consists of protocols for secure arithmetic in a field and generation of secret random values. This layer could be instantiated using different secure computation methods (e.g., secret sharing or homomorphic encryption). We use multiparty computation based on secret sharing (semi-honest model), which offers the most efficient protocols. The arithmetic layer offers protocols for computation with boolean, integer, and rational (fixed-point) data types. Finally, the protocols in the application layer carry out an oblivious computation of the simplex algorithm with secret-shared input and output (a linear program and its solution). The simplex protocol leaks only the number of iterations and the termination condition (optimal solution or unbounded problem).

The complexity of a secure simplex iteration is dominated by several steps that consist of many secure comparisons or multiplications executed in parallel. Our approach to improving the performance of the protocol focuses on reducing the communication complexity of these steps. The protocol is based on a simplex variant that uses fixed-point arithmetic and needs a minimum number of comparisons and fixed-point multiplications. The design of the lower layer protocols, the data encoding, the use of secure fixed-point arithmetic, and the design of the simplex protocol contribute to achieving this goal.

Related Work. Secure linear programming protocols were proposed by Li and Atallah [13] for the two-party case and by Toft [19] for the multiparty case. Heuristics that apply simplex to "disguised" linear programs have both correctness flaws and security problems [1], so we do not discuss them in the following. We review the relevant features of the solutions in [13,19].

Both protocols use secure integer computation and the simplex algorithm. Let ℓ denote the bit-length of the integers in the initial tableau of the linear program. The first protocol [13] is based on a simplified variant of simplex, without divisions. An iteration of this algorithm can double the bit-length of the values in the tableau ($k = 2^\theta \ell$ bits after θ iterations, worst case). Therefore, the protocol can solve only small linear programs that terminate in few iterations.

Toft's protocol [19] uses a simplex variant [15] with the property that the divisions computed in every iteration yield integer results. This algorithm has important advantages: the computation can be carried out using secure integer arithmetic; the values in the tableau are exact (no rounding errors) and do not grow as fast as in the previous variant; secure division can be efficiently computed in this particular case. However, the values still grow during the initial iterations (up to $k = \theta\ell$ bits after θ iterations, worst case) and the growth levels off at large bit-lengths, reaching thousands of bits for practical problems. This

severely degrades the performance and limits the practical applications, since the protocol has to use a data encoding that avoids overflow and hides the bit-length variation throughout the computation. In particular, secure comparison becomes impractical for inputs of thousands of bits.

We avoid this drawback by using a simplex variant with small tableau and fixed-point arithmetic. The goal is to reduce the bit-length of the secret shares by a factor of 10 and the input bit-length of all comparisons to 100 bits. Due to the structure of the simplex algorithm the gains exceed by far the effects of more complex fixed-point arithmetic. We use our general framework for secure fixed-point computation introduced in [4], extending and adapting the protocols for this type of application. In particular we use a different division protocol that allows to efficiently compute many division operations with common divisor, so that a simplex iteration computes a single reciprocal. Rounding errors for an iteration are close to the resolution of the fixed-point representation.

Oblivious computation of simplex iterations is achieved in [13,19] using two different methods. Li and Atallah use secret permutations of the rows and columns of the tableau in each iteration. Toft introduces a secret indexing method that allows to read or write entries in the tableau without revealing the index. Our protocol uses secret indexing, which is simpler and more versatile. We give more efficient solutions for secret reading and pivot selection.

We use standard techniques for multiparty computation based on secret sharing, similar to [6,7,14,18]. However, the protocols in [7,18] aim at providing integer computation with perfect privacy and constant round complexity, while our goal is fixed-point computation and lower communication complexity, for more efficient parallel computation. We obtain important performance gains using a combination of techniques that includes additive hiding with statistical privacy (instead of perfect privacy), protocols with logarithmic round complexity (instead of constant round complexity), optimized data encoding (especially for binary values), and non-interactive generation of shared random values [5].

2 Preliminaries

2.1 Linear Programming and the Simplex Algorithm

The simplex algorithm is the most popular method for solving linear programs [2]. Its simple structure and the possibility to parallelize a large part of the computation also makes it the best choice for secure linear programming.

We consider the task of solving the linear program shown in Eq. 1, for $b_1, \ldots, b_m \geq 0$. We start by adding the slack variables $x_{n+1}, \ldots, x_{n+m}$, to transform Eq. 1 to the standard form with equality constraints shown in Eq. 2. A feasible solution is a vector $x_1, \ldots, x_{n+m} \geq 0$ that satisfies the constraints. The goal is to find an optimal solution that also maximizes the objective function. A basis is a set of m indexes corresponding to variables whose coefficients in the constraints are linearly independent vectors. A solution with null values for non-basis variables is called basic solution. Observe that $x_j = 0$ for $j = 1, \ldots, n$ and $x_{n+i} = b_i$ for $i = 1, \ldots, m$ is a basic feasible solution of Eq. 2.

$$\begin{array}{lll} \max & \sum_{j=1}^{n} f_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i & i = 1, \ldots, m \\ & x_j \geq 0 & j = 1, \ldots, n \end{array} \tag{1}$$

$$\begin{array}{lll} \max & \sum_{j=1}^{n} f_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j + x_{n+i} = b_i & i = 1, \ldots, m \\ & x_j \geq 0 & j = 1, \ldots, n+m \end{array} \tag{2}$$

Simplex starts from an initial basic feasible solution and improves it by performing a sequence of iterations until it finds the optimal solution or detects that the linear program is unbounded. In each iteration, a basis variable is replaced by another variable and the linear program is re-written accordingly, using a procedure called pivoting. Simplex uses a tableau representation of the linear program. Two variants of tableau are shown in Fig. 1. The vectors S and U contain the indexes of the current basis and non-basis variables, respectively.

	x_1	$\ldots$	x_n	x_{n+1}	$\ldots$	x_{n+m}	
$x_{S(1)}$	a_{11}	$\ldots$	a_{1n}	1	$\ldots$	0	b_1
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$x_{S(m)}$	a_{m1}	$\ldots$	a_{mn}	0	$\ldots$	1	b_m
F	$-f_1$	$\ldots$	$-f_n$	0	$\ldots$	0	0

	$x_{U(1)}$	$\ldots$	$x_{U(n)}$	
$x_{S(1)}$	a_{11}	$\ldots$	a_{1n}	b_1
$\vdots$	$\vdots$		$\vdots$	$\vdots$
$x_{S(m)}$	a_{m1}	$\ldots$	a_{mn}	b_m
F	$-f_1$	$\ldots$	$-f_n$	0

Fig. 1. Initial simplex tableau (left) and condensed tableau variant (right).

The computation can be carried out in many different ways. For secure simplex, the choice of the algorithm depends on the complexity of the building blocks and has a strong impact on performance. We considered different variants of plain and revised simplex. Variants for integer arithmetic work with very large numbers, making secure comparison impractical and increasing the communication overhead. The protocol presented in this paper uses an algorithm for fixed point arithmetic and the condensed tableau in Fig. 1. This variant needs a minimum number of secure comparisons and fixed-point multiplications. The algorithm is described below. We denote $V(i)$ the element of vector V at index i and $M(i,j)$ the element of matrix M at row index i and column index j.

1. *Initialization:* For $i \in [1..m]$, $j \in [1..n]$, set $A(i,j) \leftarrow a_{ij}$, $F(j) \leftarrow f_j$, $B(i) \leftarrow b_i$, $U(j) \leftarrow j$, $S(i) \leftarrow n+i$. Set $T \leftarrow \begin{pmatrix} A & B \\ -F & 0 \end{pmatrix}$.
2. *Iterations:*

a) *Get Pivot Column:* Select $c \in [1..n]$ such that $T(m+1, c) < 0$. If no such c, report "Optimal Solution" and exit. If more options, choose at random or using Bland's rule (minimum $U(c)$).

b) *Get Pivot Row:* Select $r \in [1..m]$, such that $T(r,c) > 0$ and $T(n+1, r)/T(r,c)$ is minimal. If no such r, report "Unbounded Problem" and exit. If more options, choose at random or using Bland's rule (minimum $S(r)$).

c) *Update the tableau (pivoting):*

$T(i,j) \leftarrow T(i,j) - T(i,c)T(r,j)/T(r,c)$ $i \in [1..m+1]\backslash\{r\}, j \in [1..n+1]\backslash\{c\}$
$T(i,c) \leftarrow -T(i,c)/T(r,c)$ $i \in [1..m+1]\backslash\{r\}$
$T(r,j) \leftarrow T(r,j)/T(r,c)$ $j \in [1..n+1]\backslash\{c\}$
$T(r,c) \leftarrow 1/T(r,c)$
$U(c) \quad \leftrightarrow S(r)$ (swap)

3. *Final solution:* For $i \in [1..m]$ set $x_{S(i)} \leftarrow T(i, n+1)$. All other variables take the value 0. The objective function takes the value $T(m+1, n+1)$.

2.2 Core Protocols

Basic framework. Assume a group of $n > 2$ parties, $P_1, \ldots, P_n$, that communicate on secure channels. Party P_i has private input x_i and output y_i, function of all inputs. Multiparty computation using secret sharing proceeds as follows. The parties use a linear secret sharing scheme to deliver shares of their private inputs to the group. Thus, they create a distributed state of the computation where each party has a share of each secret variable. Certain subsets of parties can reconstruct a secret by pooling together their shares, while any other subset cannot learn anything about it. The secret sharing scheme allows to compute with shared variables. The protocols used for this purpose take on input shared data and return shared data, and thus enable secure protocol composition.

The protocols offer perfect or statistical privacy, in the sense that the views of protocol execution (all values learned by an adversary) can be simulated such that the distributions of real and simulated views are perfectly or statistically indistinguishable, respectively. Let X and Y be distributions with finite sample spaces V and W and $\Delta(X,Y) = \frac{1}{2}\sum_{v \in V \bigcup W} |Pr(X = v) - Pr(Y = v)|$ the statistical distance between them. We say that the distributions are perfectly indistinguishable if $\Delta(X,Y) = 0$ and statistically indistinguishable if $\Delta(X,Y)$ is negligible in some security parameter.

The basic framework uses Shamir secret sharing over a finite field $\mathbb{F}$ and allows secure arithmetic in $\mathbb{F}$ with perfect privacy against a passive threshold adversary able to corrupt t out of n parties. Essentially, in this model, the parties do not deviate from the specified protocol and any $t+1$ parties can reconstruct a secret, while t or less parties cannot distinguish it from random uniform values in $\mathbb{F}$. We assume $|\mathbb{F}| > n$, to enable Shamir sharing, and $n > 2t$, for multiplication of secret-shared values. We denote $[x]$ a Shamir sharing of x and $[x]^{\mathbb{F}}$ a sharing in a particular field $\mathbb{F}$. We refer the reader to [6] for a more formal and general presentation of this approach to secure computation.

Complexity metrics. The running time of the protocols is (usually) dominated by the communication between parties. We evaluate the complexity of the protocols using two metrics that reflect different aspects of the interaction. Communication complexity measures the amount of data sent by each party. For our protocols, a suitable abstract metric is the number of invocations of a primitive during which every party sends a share (field element) to the others. Round complexity

Table 1. Secure arithmetic in a finite field $\mathbb{F}$.

Operation	Purpose	Rounds	Invocations
$[c]^{\mathbb{F}} \leftarrow [a]^{\mathbb{F}} + [b]^{\mathbb{F}}$	Add secrets	0	0
$[c]^{\mathbb{F}} \leftarrow [a]^{\mathbb{F}} + b$	Add secret and public	0	0
$[c]^{\mathbb{F}} \leftarrow [a]^{\mathbb{F}} b$	Multiply secret and public	0	0
$[c]^{\mathbb{F}} \leftarrow [a]^{\mathbb{F}} [b]^{\mathbb{F}}$	Multiply secrets	1	1
$a \leftarrow \mathsf{Output}([a]^{\mathbb{F}})$	Reveal a secret	1	1
$[z] \leftarrow \mathsf{Inner}([X]^{\mathbb{F}}, [Y]^{\mathbb{F}})$	$[\sum_{i=1}^{m} X(i)Y(i)]^{\mathbb{F}}$	1	1

measures the number of sequential invocations and is relevant for the inherent network delay, independent of the amount of data. Invocations that can be executed in parallel count as a single round.

Shared random values. Secure computation often combines secret sharing with additive or multiplicative hiding. For example, given a shared variable $[x]$ the parties jointly generate a shared random value $[r]$, compute $[y] = [x] + [r]$, and reveal $y = x + r$. This is similar to one-time pad encryption of x with key r.

For a secret $x \in \mathbb{Z}_q$ and random uniform $r \in \mathbb{Z}_q$ we obtain $\Delta(x+r \bmod q, r) = 0$, hence perfect privacy. Alternatively, for $x \in [0..2^k - 1]$, random uniform $r \in [0..2^{k+\kappa}-1]$, and $q > 2^{k+\kappa+1}$ we obtain $\Delta(x+r \bmod q, r) < 2^{-\kappa}$, hence statistical privacy with security parameter κ. This property holds for other distributions of r that can be generated more efficiently. The variant with statistical privacy can substantially simplify the protocols by avoiding wraparound modulo q, although it requires larger q (hence larger shares) for a given data range.

We use Pseudo-random Replicated Secret Sharing (PRSS) [5] to generate without interaction shared random values in $\mathbb{F}$ with uniform distribution and random sharings of 0. Also, we use the integer variant RISS [8] to generate shared random integers in a given interval and the ideas in [9] for share conversions. To enable these techniques, we assume that numbers are encoded in $\mathbb{Z}_q$ and $q > 2^{k+\kappa+\nu+1}$, where k is the required integer bit-length, κ is the security parameter, $\nu = \lceil \log(\binom{n}{t}) \rceil$, n is the number of parties, and t is the corruption threshold.

Efficient inner product. Consider the following common task: given two shared vectors $[X] = ([X(1)], \ldots, [X(m)])$ and $[Y] = ([Y(1)], \ldots, [Y(m)])$, $X, Y \in \mathbb{F}^m$, compute their inner product $[z] = [\sum_{i=1}^{m} X(i)Y(i)]$. A naive solution is to use the multiplication protocol and compute $[z] = \sum_{i=1}^{m} [X(i)][Y(i)]$, with complexity 1 round and m invocations. We present an efficient protocol with perfect privacy and complexity reduced to 1 invocation. Assume Shamir sharing for n parties with threshold $t < n/2$. Denote $[X(i)]_j$, $[Y(i)]_j$, $i \in [1..m]$, the input shares and $[z]_j$ the output share of party P_j. The protocol, called Inner, proceeds as follows:

1. Party P_j, $j \in [1..n]$, computes $d_j = \sum_{i=1}^{m} ([X(i)]_j [Y(i)]_j)$ and then shares d_j sending $[d_j]_k$ to party P_k, $k \in [1..n]$.
2. Party P_k, $k \in [1..n]$, computes the share $[z]_k = \sum_{j \in J} \lambda_j [d_j]_k$, where $J \subseteq [1..n]$, $|J| = 2t + 1$, and $\{\lambda_j\}_{j \in J}$ is the reconstruction vector for J.

Protocol Inner is a generalization of the classical protocol for secure multiplication in a field [12]. The proofs of correctness and security are similar.

Secret indexing. The purpose of secret indexing is to read/write a value from/to an array without revealing the value and its index. Efficient secret indexing can be achieved by encoding an index $x \in [1..m]$ as a secret bitmask $[V] = ([V(1)], \ldots, [V(m)])$ such that $V(i) = 1$ for $i = x$ and $V(i) = 0$ for $i \neq x$ [19]. We use this technique to obtain oblivious computation of simplex iterations.

Protocols 2.1 and 2.2 allow secret reading and writing from/to a vector. The secure multiplications re-randomize the shares, providing perfect privacy. Extension to a matrix is obvious. We call the protocols that read/write a column/row SecReadCol, SecReadRow, SecWriteCol, and SecWriteRow. Protocol Inner reduces the complexity of secret reading to 1 invocation (instead of m) for a vector of length m, and to m (or n) invocations (instead of mn) for an $m \times n$ matrix. This has a significant impact on the complexity of the simplex protocol.

Protocol 2.1: $[s] \leftarrow \mathsf{SecRead}([A], [V])$

```
1 [s] ← Inner([A], [V]) ;                          // 1 rnd, 1 inv
2 return [s];
```

Protocol 2.2: $[A] \leftarrow \mathsf{SecWrite}([A], [V], [s])$

```
1 foreach i ∈ [1..m] do parallel
2     [A(i)] ← [A(i)] + [V(i)] ([s] − [A(i)]);     // 1 rnd, m inv
3 return [A];
```

3 Arithmetic Protocols

Fixed-point representation. Fixed-point numbers are rational numbers represented as a sequence of digits split into integer and fractional parts by a virtual radix point: $\tilde{x} = s \cdot (d_{e-2} \ldots d_0 . d_{-1} \ldots d_{-f})$. For binary digits the value is $\tilde{x} = s \cdot \sum_{i=-f}^{e-2} d_i 2^i$, where $s \in \{-1, 1\}$, e is the length of the integer part (including the sign bit), and f is the length of the fractional part. Denote $\bar{x} = s \cdot \sum_{i=0}^{e+f-2} d_i 2^i$ and observe that $\tilde{x} = \bar{x} \cdot 2^{-f}$, hence $\tilde{x}$ is encoded as an integer $\bar{x}$ scaled by the factor 2^{-f}.

We define a fixed-point data type as follows. Let k, e, and f be integers such that $k > 0$, $f \geq 0$, and $e = k - f \geq 0$. Denote $\mathbb{Z}_{\langle k \rangle} = \{x \in \mathbb{Z} \mid -2^{k-1} + 1 \leq x \leq 2^{k-1} - 1\}$. The fixed-point data type with resolution 2^{-f} and range 2^e is the set $\mathbb{Q}_{\langle k,f \rangle} = \{\tilde{x} \in \mathbb{Q} \mid \tilde{x} = \bar{x} \cdot 2^{-f}, \bar{x} \in \mathbb{Z}_{\langle k \rangle}\}$. Intuitively, $\mathbb{Q}_{\langle k,f \rangle}$ is obtained by sampling the range of real values $[-2^{e-1} + 2^{-f}, 2^{e-1} - 2^{-f}]$ at 2^{-f} intervals.

Data encoding in a field. Any secret value in a secure computation has a data type which is public information. Data types are encoded in a field $\mathbb{F}$ as follows.

Logical values $false, true$ and bit values $0, 1$ are encoded as 0_F and 1_F, respectively. $\mathbb{F}$ can be a small binary field $\mathbb{F}_{2^m}$ or prime field $\mathbb{Z}_q$. This encoding

Table 2. Secure fixed-point arithmetic: addition and multiplication.

	Fixed-point Op.	Integer Op./ Secure Op.	Abs. Error
Add (Subtract)	$\tilde{c} = \tilde{a} + \tilde{b} \in \mathbb{Q}_{\langle k,f \rangle}$ $\tilde{a}, \tilde{b} \in \mathbb{Q}_{\langle k,f \rangle}$	$\bar{c} = \bar{a} + \bar{b}$ $[c] \leftarrow [a] + [b]$	$\delta = 0$
Multiply w/o scaling	$\tilde{c} = \tilde{a}\tilde{b} \in \mathbb{Q}_{\langle k+f,2f \rangle}$ $\tilde{a}, \tilde{b} \in \mathbb{Q}_{\langle k,f \rangle}$	$\bar{c} = \bar{a}\bar{b}$ $[c] \leftarrow [a][b]$	$\delta = 0$
Multiply w/ scaling	$\tilde{c} = \tilde{a}\tilde{b} \in \mathbb{Q}_{\langle k,f \rangle}$ $\tilde{a}, \tilde{b} \in \mathbb{Q}_{\langle k,f \rangle}$	$\bar{c} = \mathsf{trunc}(\bar{a}\bar{b}, f)$ $[c] \leftarrow \mathsf{TruncPr}([a][b], k+f, f)$	$\delta = \delta_t 2^{-f}$ $\lvert\delta_t\rvert < 1$
Inner product $A = (a_1, \ldots, a_m)$ $B = (b_1, \ldots, b_m)$	$\tilde{c} = \sum_{i=1}^{m} \tilde{a}_i \tilde{b}_i$ $\tilde{a}_i, \tilde{b}_i, \tilde{c} \in \mathbb{Q}_{\langle k,f \rangle}$	$\bar{c} = \mathsf{trunc}(\sum_{i=1}^{m} \bar{a}_i \bar{b}_i, f)$ $[x] \leftarrow \mathsf{Inner}([A], [B])$ $[c] \leftarrow \mathsf{TruncPr}([x], k+f, f)$	$\delta = \delta_t 2^{-f}$ $\lvert\delta_t\rvert < 1$
Multiply double optim.	$\tilde{d} = \tilde{a}\tilde{b}\tilde{c} \in \mathbb{Q}_{\langle k,f \rangle}$ $\tilde{a}, \tilde{b}, \tilde{c}, \tilde{d} \in \mathbb{Q}_{\langle k,f \rangle}$	$\bar{d} = \mathsf{trunc}(\bar{a}\bar{b}\bar{c}, 2f)$ $[c] \leftarrow \mathsf{TruncPr}([a][b][c], k+2f, 2f)$	$\delta = \delta_t 2^{-f}$ $\lvert\delta_t\rvert < 1$

allows secure evaluation of boolean functions using secure arithmetic in $\mathbb{F}$. Efficient encoding of binary values is essential for reducing the complexity of shared random bit generation, comparison, and other secure simplex building blocks.

Signed integers are encoded in $\mathbb{Z}_q$ using $\mathsf{fld} : \mathbb{Z}_{\langle k \rangle} \mapsto \mathbb{Z}_q$, $\mathsf{fld}(\bar{x}) = \bar{x} \bmod q$, $q > 2^k$. For any $\bar{a}, \bar{b} \in \mathbb{Z}_{\langle k \rangle}$ and $\odot \in \{+, -, \cdot\}$ we have $\bar{a} \odot \bar{b} = \mathsf{fld}^{-1}(\mathsf{fld}(\bar{a}) \odot \mathsf{fld}(\bar{b}))$. Moreover, if $\bar{b} | \bar{a}$ then $\bar{a}/\bar{b} = \mathsf{fld}^{-1}(\mathsf{fld}(\bar{a}) \cdot \mathsf{fld}(\bar{b})^{-1})$. Secure arithmetic with signed integers can thus be computed using secure arithmetic in $\mathbb{Z}_q$.

A fixed-point number $\tilde{x} \in \mathbb{Q}_{\langle k,f \rangle}$ is represented as a secret integer $\bar{x} = \tilde{x} 2^f$ encoded in $\mathbb{Z}_q$ and public parameters that specify the resolution and the range, f and e (or $k = e + f$). We define the map $\mathsf{int}_f : \mathbb{Q}_{\langle k,f \rangle} \mapsto \mathbb{Z}_{\langle k \rangle}$, $\mathsf{int}_f(\tilde{x}) = \tilde{x} 2^f$.

We distinguish different representations of a number using the following simplified notation: we denote $\tilde{x}$ a rational number of some fixed-point type $\mathbb{Q}_{\langle k,f \rangle}$ and $\bar{x} = \tilde{x} 2^f \in \mathbb{Z}_{\langle k \rangle}$ the integer value of its fixed-point representation; for secure computation using secret-sharing we denote $x = \bar{x} \bmod q \in \mathbb{Z}_q$ the field element that encodes $\bar{x}$ (and hence $\tilde{x}$) and $[x]$ a sharing of x. The notation $x = (\texttt{condition})?\ a : b$ means that the variable x is assigned the value a when `condition`=*true* and b otherwise.

Fixed-point arithmetic. Tables 2 and 3 contain a summary of the main arithmetic protocols used in simplex. Secure addition, subtraction, and comparison of fixed-point numbers are immediate extensions of the integer operations. We need additional protocols for multiplication and division.

Multiplication. Table 2 shows several cases of secure fixed-point multiplication used in simplex. Let $\tilde{a}, \tilde{b} \in \mathbb{Q}_{\langle k,f \rangle}$ and $\tilde{c} = \tilde{a}\tilde{b}$. We obtain the representation of $\tilde{c}$ with resolution 2^{-2f} by integer multiplication, $\bar{c} = \bar{a}\bar{b} = \tilde{a}\tilde{b}2^{2f}$, and we can scale down $\tilde{c}$ to resolution 2^{-f} by truncation (when necessary). The truncation protocol $\mathsf{TruncPr}$ computes $\bar{c}/2^f$ and rounds to the nearest integer with probability $1 - \alpha$, where α is the distance to that integer [4]. The absolute error is with high

Table 3. Complexity of arithmetic protocols ($\log(q_1) \approx \kappa$).

Operation	Protocol	Rounds	Invocations	Field
$\lfloor \bar{a}/2^f \rfloor + u$	$[d] \leftarrow \mathsf{TruncPr}([a], k, f)$	1	1	$\mathbb{Z}_q$
$\bar{a} \in \mathbb{Z}_{\langle k \rangle}$, $u \in_R \{0,1\}$		2	$2f$	$\mathbb{Z}_{q_1}$
	$\mathsf{TruncPr}$ after precomp.	1	1	$\mathbb{Z}_q$
	$[d] \leftarrow \mathsf{TruncPrN}([a], k, f)$	1	1	$\mathbb{Z}_q$
$(\bar{a} < 0)?1 : 0$, $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$	$[s] \leftarrow \mathsf{LTZ}([a], k)$	1	1	$\mathbb{Z}_q$
$(\bar{a} > 0)?1 : 0$, $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$	$\mathsf{GTZ}([a], k) = \mathsf{LTZ}(-[a], k)$	2	$2k$	$\mathbb{Z}_{q_1}$
		$\log(k) + 1$	$2k - 3$	$\mathbb{F}_{2^8}$
	LTZ after precomp.	1	1	$\mathbb{Z}_q$
		$\log(k) + 1$	$2k - 3$	$\mathbb{F}_{2^8}$
$\tilde{x} \approx 1/\tilde{c} \in \mathbb{Q}_{\langle p+1,p \rangle}$	$[x] \leftarrow \mathsf{RecItNR}([c], p)$	3θ	3θ	$\mathbb{Z}_q$
$\tilde{c} \in \mathbb{Q}_{\langle p+1,p \rangle} \bigcap (0.5, 1)$	(using $\mathsf{TruncPr}$)	2	$2p\theta$	$\mathbb{Z}_{q_1}$
	$\mathsf{RecItNR}$ after precomp.	3θ	3θ	$\mathbb{Z}_q$
Normalization: $\bar{c} = \bar{v}\bar{x}$,	$([c], [v]) \leftarrow \mathsf{Norm}([x], k)$	2	2	$\mathbb{Z}_q$
$\tilde{x} \in \mathbb{Q}_{\langle k+1,f \rangle}$,		2	$4k$	$\mathbb{Z}_{q_1}$
$\tilde{c} \in \mathbb{Q}_{\langle k+1,k \rangle} \bigcap (0.5, 1)$		$2\log(k) + 1$	$k + 1.5k\log(k)$	$\mathbb{F}_{2^8}$
	Norm after precomp.	2	2	$\mathbb{Z}_q$
		$2\log(k) + 1$	$k + 1.5k\log(k)$	$\mathbb{F}_{2^8}$

probability $|\delta_t| \leq 0.5$, and always $|\delta_t| < 1$. $\mathsf{TruncPr}$ provides statistical privacy, while $\mathsf{TruncPrN}$ performs the same operation more efficiently but with weaker protection of the discarded part (additive hiding with non-uniform random). $\mathsf{TruncPrN}$ is sufficient for multiplications in simplex, since values less than 2^{-f} are negligible and the computation is carried out with extended precision (large f). Note that the optimizations for inner product and double multiplication shown in Table 2 are also important for improving the accuracy.

Reciprocal and division. Simplex needs an accurate and efficient protocol for multiple division operations with the same positive divisor, $\tilde{a}_1/\tilde{b}, \ldots, \tilde{a}_m/\tilde{b}$. This can be achieved by computing $\tilde{y} = 1/\tilde{b}$ followed by m parallel multiplications $\tilde{z}_i = \tilde{a}_i\tilde{y}$. Protocols 3.1, $\mathsf{RecItNR}$, and 3.2, DivNR, follow this approach (the division protocol in [4] is not suitable for this type of application).

Let $\tilde{c} \in \mathbb{Q}_{\langle p+1,p \rangle} \bigcap (0.5, 1)$. $\mathsf{RecItNR}$ computes $\tilde{x} \approx 1/\tilde{c}$, $\tilde{x} \in \mathbb{Q}_{\langle p+1,p \rangle}$, for secret-shared input and output. The protocol uses the Newton-Raphson method and starts by computing the initial approximation $\tilde{x}_0 \approx 1/\tilde{c}$, $\tilde{x}_0 = 2.9142 - 2\tilde{c}$, with relative error $\epsilon_0 < 0.08578$ (at least $\log_2(\epsilon_0) = 3.5$ exact bits) [10]. Each iteration computes an improved approximation $\tilde{x}_{i+1} = \tilde{x}_i(2 - \tilde{x}_i\tilde{c})$. For exact arithmetic (without truncation) the relative error after iteration i is $\epsilon_i = \epsilon_{i-1}^2 = \epsilon_0^{2^i}$. Intuitively, the number of exact bits doubles at each iteration, so $p + 1$ bits ($\delta < 2^{-p}$) are obtained after $\theta = \lceil \log \frac{p+1}{3.5} \rceil$ iterations. The error due to computation of an iteration with limited precision is $|\delta_T| < 2^{-p}$. Since the error introduced by an iteration decreases quadratically during next iterations, we conclude that the output error of $\mathsf{RecItNR}$ is approximately bounded by 2^{-p}.

Let $\tilde{a}, \tilde{b} \in \mathbb{Q}_{\langle k+1,f \rangle}$ and $\tilde{b} > 0$. Protocol DivNR computes the quotient $\tilde{z} \approx \tilde{a}/\tilde{b}$, $\tilde{z} \in \mathbb{Q}_{\langle k+1,f \rangle}$, with secret-shared inputs and output. The protocol consists of the following main steps: compute the normalized divisor $\tilde{c} \in \mathbb{Q}_{\langle p+1,p \rangle} \bigcap (0.5, 1)$ using the protocol Norm; compute the reciprocal $\tilde{x} \approx 1/\tilde{c}$ using the protocol RecItNR; then compute the quotient $\tilde{z} \approx \tilde{a}/\tilde{b}$ and scale it to obtain $\tilde{z} \in \mathbb{Q}_{\langle k+1,f \rangle}$. A variant for multiple divisors repeats steps 4 and 5 for each divisor (parallel computation).

Protocol 3.1: $[x] \leftarrow \mathsf{RecItNR}([c], p)$

1 $(\theta, \alpha, \beta) \leftarrow (\lceil \log \frac{p+1}{3.5} \rceil, \mathsf{fld}(\mathsf{int}_p(2.9142)), \mathsf{fld}(\mathsf{int}_{2p}(2.0)))$;
2 $[x] \leftarrow \alpha - 2[c]$;
3 **foreach** $i \in [1..\theta]$ **do**
4 $\quad [x] \leftarrow [x](\beta - [x][c])$;
5 $\quad [x] \leftarrow \mathsf{TruncPr}([y], 3p, 2p)$;
6 **return** $[x]$;

Protocol 3.2: $[z] \leftarrow \mathsf{DivNR}([a], [b], k, f)$

1 $([c], [v]) \leftarrow \mathsf{Norm}([b], k)$;
2 $[x] \leftarrow \mathsf{RecItNR}([c], k)$;
3 $[y] \leftarrow [v][x]$;
4 $[z] \leftarrow [a][y]$;
5 $[z] \leftarrow \mathsf{TruncPr}([z], 3k - f, 2k - f)$;
6 **return** $[z]$;

Protocol Norm computes $\bar{c}$ and $\bar{v}$ such that $2^{k-1} \leq \bar{c} < 2^k$ and $\bar{c} = \bar{b}\bar{v}$, with secret-shared input and outputs [4,17]. Let $\tilde{c} = \bar{c}2^{-k}$ and let $0 < m \leq k$ such that $2^{m-1} \leq \bar{b} < 2^m$. Observe that $\bar{v} = 2^{k-m}$, $\tilde{c} \in \mathbb{Q}_{\langle k+1,k \rangle} \bigcap (0.5, 1)$, and $\tilde{c} = \tilde{b}2^{f-m}$; $\tilde{c}$ is the normalized input for RecItNR and $\bar{v}$ the normalization factor. Steps 3-4 of DivNR compute $\tilde{z} = \tilde{a}\tilde{x}2^{f-m} \approx \tilde{a}/\tilde{b}$ without loss of accuracy and then step 4 scales this value to obtain $\tilde{z} \in \mathbb{Q}_{\langle k+1,f \rangle}$. Observe that $\tilde{a}\tilde{x}2^{f-m}2^f = \bar{a}\bar{x}\bar{v}2^{-(2k-f)}$, so the output $\bar{z} = \mathsf{trunc}(\bar{a}\bar{x}\bar{v}, 2k - f)$ is the representation of $\tilde{z}$ with resolution 2^{-f}. The output error of DivNR is upper bounded by 2^{-f}.

RecItNR and DivNR do not open any secret-shared value and their building blocks provide perfect or statistical privacy. The number of iterations depends only on public configuration parameters, hence it can be revealed. We conclude that the two protocols offer statistical privacy.

4 Secure Simplex Protocol

Protocol 4.1, Simplex, solves linear programs using the algorithm and the building blocks presented in the previous sections. The inputs are the secret-shared values of the linear program: the matrix $[A]$ and the vectors $[B]$ and $[F]$. The output consists of a public value indicating the termination state, optimal or unbounded, a secret-shared array $[X]$ containing the solution, and the optimum $[z]$ of the objective function. The protocol reveals only the number of iterations and the termination condition. The tableau, pivot indexes, and related variables are

protected throughout the computation using the techniques discussed in Section 2.2. For a passive adversary that corrupts $t < n/2$ parties, the building blocks provide perfect or statical privacy. By the composition theorem in Chapter 4 of [3] we conclude that the simplex protocol provides statistical privacy.

Complexity is shown in the protocol specifications by annotating the relevant steps; for better clarity, we assume a generic comparison protocol with complexity ρ rounds and μ invocations. For a vector V and matrix M we denote: $V(i..j) = (V(i), \ldots, V(j))$; $M(i, \cdot)$ the row i; $M(\cdot, j)$ the column j. Angle brackets are used to specify the number of elements, e.g., $V\langle m\rangle$, $M\langle m, n\rangle$.

Protocol 4.1 initializes the tableau $[T]$ and the basis and non-basis index vectors $[S]$ and $[U]$, performs the simplex iterations, and then extracts from the final tableau the optimal solution (if it exists). The iterations are computed by Protocol 4.2. The computation is structured into several sub-protocols that select the pivot's column and row (GetPivCol and GetPivRow) and then update the tableau $[T]$ (UpdTab) and the vectors $[S]$ and $[U]$ (UpdVar).

Protocol 4.1: $(result, [X], [z]) \leftarrow \mathsf{Simplex}([A], [B], [F])$

Input: $[A\langle m, n\rangle]$, $[B\langle m\rangle]$, $[F\langle m\rangle]$.
Output: $result \in \{\mathsf{Opt}, \mathsf{Unb}\}$; $[X\langle n\rangle]$ and $[z]$ if $result = \mathsf{Opt}$.

1 $[T] \leftarrow \begin{pmatrix} [A] & [B] \\ -[F] & [0] \end{pmatrix}$;
2 $([S], [U]) \leftarrow \mathsf{InitVar}(m, n)$;
3 $([T], [S], result) \leftarrow \mathsf{Iteration}([T], [S], [U])$;
4 **if** $result = \mathsf{Unb}$ **then return** Unb;
5 $[X] \leftarrow \mathsf{GetSolution}([T(\cdot, n+1)], [S])$;
6 **return** $(\mathsf{Opt}, [X], [T(m+1, n+1)])$;

Protocol 4.2: $([T], [S], result) \leftarrow \mathsf{Iteration}([T], [S], [U])$

Input: $[T\langle m+1, n+1\rangle]$, $[S\langle m\rangle]$, $[U\langle n\rangle]$;
Output: $[T\langle m+1, n+1\rangle]$, $[S\langle m\rangle]$; $result \in \{\mathsf{Opt}, \mathsf{Unb}\}$;

1 **repeat forever**
2 $([V], s) \leftarrow \mathsf{GetPivCol}([T(m+1, 1..n)])$; // protocol 4.5
3 **if** $s = 0$ **then return** $([T], [S], \mathsf{Opt})$;
4 $[C] \leftarrow \mathsf{SecReadCol}([T], [V])$; // 1 rnd, $m+1$ inv
5 $([W], s) \leftarrow \mathsf{GetPivRow}([T(1..m, n+1)], [C])$; // protocol 4.6
6 **if** $s = 0$ **then return** $([T], [S], \mathsf{Unb})$;
7 $[R] \leftarrow \mathsf{SecReadRow}([T], [W])$; // 1 rnd, $n+1$ inv
8 $[p] \leftarrow \mathsf{SecRead}([R], [V])$; // 1 rnd, 1 inv
9 $[T] \leftarrow \mathsf{UpdTab}([T], [C], [R], [V], [W], [p])$; // protocol 4.8
10 $([S], [U]) \leftarrow \mathsf{UpdVar}([S], [U], [V], [W])$; // 2 rnd, $m+n+2$ inv
11 **end**

The iterations terminate when the algorithm finds the optimal solution or determines that the linear program is unbounded. Termination is detected by the pivot selection protocols, which report that no pivot exists ($s = 0$). Protocol 4.3, GetSolution, extracts the solution from the tableau by assigning $X(S(i)) \leftarrow B(i)$

for $i \in [1..m]$ and 0 to the other elements; it uses secret indexing and the protocol Int2BitMask [17] that converts secret integers to secret bitmasks.

Protocol 4.3: $[X] \leftarrow \mathsf{GetSolution}([B], [S])$

Input: $[B\langle m\rangle]$, $[S\langle m\rangle]$;
Output: $[X\langle n+m\rangle]$;
1 **foreach** $i \in [1..n+m]$ **do** $[X(i)] \leftarrow 0$;
2 **foreach** $i \in [1..m]$ **do parallel**
3 $[V] \leftarrow \mathsf{Int2BitMask}([S(i)], m+n)$; // 3 rnd, $\approx 3m(m+n)$ inv
4 $\mathsf{SecWrite}([X], [V], [B(i)])$; // 1 rnd, $m(m+n)$ inv
5 **return** $[X]$; // decision variables: $[X(1)], \ldots, [X(n)]$

The vectors $[S]$ and $[U]$ are initialized by InitVar with the indexes of the initial basis and non-basis variables. At each iteration, the basis variable with index $[S([W])]$ is replaced by the non-basis variable with index $[U([V])]$. Protocol 4.4 updates $[S]$ and $[U]$ by swapping the corresponding entries.

Protocol 4.4: $([S], [U]) \leftarrow \mathsf{UpdVar}([S], [U], [V], [W])$

Input: $[S\langle m\rangle]$, $[U\langle n\rangle]$, $[V\langle n\rangle]$, $[W\langle m\rangle]$
Output: $[S\langle m\rangle]$, $[U\langle n\rangle]$ (updated)
1 $[s] \leftarrow \mathsf{SecRead}([S], [W])$; // 1 rnd, 1 inv
2 $[u] \leftarrow \mathsf{SecRead}([U], [V])$; // + 1 inv
3 $[S] \leftarrow \mathsf{SecWrite}([S], [W], [u])$; // 1 rnd, m inv
4 $[U] \leftarrow \mathsf{SecWrite}([U], [V], [s])$; // + n inv
5 **return** $([S], [U])$;

Pivot selection. Protocol 4.5, GetPivCol, finds the index of the pivot's column by selecting a negative entry in the cost vector F; it returns a public bit s indicating if the pivot column was found or not and a secret bitmask $[V]$ that encodes the column's index. If none of the F values is negative then $s = 0$; simplex has found the optimal solution and terminates. Otherwise, $s = 1$ and $[V]$ encodes the index of the first negative entry[3].

Protocol 4.5: $([V], s) \leftarrow \mathsf{GetPivCol}([F])$

Input: $[F\langle n\rangle]$;
Output: $[V\langle n\rangle]$, $s \in \{0, 1\}$;
1 **foreach** $i \in [1..n]$ **do parallel**
2 $[D(i)] \leftarrow \mathsf{LTZ}([F(i)], k)$; // ρ rnd, $n\mu$ inv
3 $s \leftarrow 1 - \mathsf{EQZPub}(\sum_{i=0}^{n} [D(i)])$; // 1 rnd, 1 inv
4 **if** $s = 0$ **then return** $([D], s)$;
5 $[V] \leftarrow \mathsf{SelectFirst}([D])$; // $\log(n)$ rnd, $n\log(n)/2$ inv
6 **return** $([V], s)$;

[3] An implementation of the pivot selection protocols has to take into account the roundoff errors, e.g., by evaluating $\mathsf{LTZ}([a] + \delta, k)$ instead of $\mathsf{LTZ}([a], k)$ where $\delta > 0$ is an estimate of the maximum error.

c_0	c_1	$c_0 b'_1$	$c_1 b'_0$	Output	Constraints	Selection
≤ 0	≤ 0	≤ 0	≤ 0	1	None applicable	b_0/c_0
				0	None applicable	b_1/c_1
> 0	≤ 0	> 0	≤ 0	1	b_1/c_1 not applicable	b_0/c_0
≤ 0	> 0	≤ 0	> 0	0	b_0/c_0 not applicable	b_1/c_1
> 0	> 0	≥ 0	≥ 0	1	$b_0/c_0 < b_1/c_1$	b_0/c_0
				0	$b_1/c_1 \leq b_0/c_0$	b_1/c_1

Fig. 2. Constraint comparison using CompCons.

EQZPub($[v]$) is a simple equality test with public output [16] and returns $(v = 0)?\ 1 : 0$. SelectFirst($[D]$) computes the secret bitmask of the minimum index i such that $D(i) = 1$ [17].

The index of the pivot's row is determined by Protocol 4.6, GetPivRow. The protocol computes $\text{argmin}_i\{\frac{B(i)}{C(i)} \mid C(i) > 0\}$. If none of the C values is strictly positive, it returns $s = 0$; the simplex protocol terminates and reports that the linear program is unbounded. Otherwise, $s = 1$ and $[W]$ is a secret bitmask that encodes the index of the pivot's row.

Protocol 4.6: $[W], s \leftarrow \mathsf{GetPivRow}([B], [C])$

Input: $[B\langle m\rangle]$, $[C\langle m\rangle]$.
Output: $[W\langle m\rangle], s \in \{0, 1\}$.

1 **foreach** $i \in [1..m]$ **do parallel**
2 $\quad [D(i)] \leftarrow \mathsf{GTZ}([C(i)])$; // ρ rnd, $m\mu$ inv
3 $s \leftarrow 1 - \mathsf{EQZPub}(\sum_{i=0}^{m}[D(i)])$; // 1 rnd, 1 inv
4 **if** $s = 0$ **then return** $([D], s)$;
5 **foreach** $i \in [1..m]$ **do parallel**
6 $\quad [B'(i)] \leftarrow [B(i)] + (1 - [D(i)])2^f$;
7 $[W] \leftarrow \mathsf{MinCons}([B'], [C], m)$; // $\lceil \log(m) \rceil(\rho + 3)$ rnd, $(m-1)(\mu+5)$ inv
8 **return** $([W], s)$;

Protocol 4.7: $[s] \leftarrow \mathsf{CompCons}([b'_0], [c_0], [b'_1], [c_1])$

1 $[x] \leftarrow [b'_0][c_1] - [b'_1][c_0]$; // 1 rnd, 2 inv
2 $[s] \leftarrow \mathsf{LTZ}([x], k + f)$; // ρ rounds, μ inv
3 **return** $[s]$;

GetPivRow uses the following method. Steps 1-4 select the relevant constraints by computing the secret bitmask $[D]$, $D(i) = (C(i) > 0)?\ 1 : 0$, $i \in [1..m]$. If D is null the protocol terminates and reports that no pivot row was found. Steps 5-7 compute the secret bitmask $[W]$ that encodes $\text{argmin}_i\{\frac{B'(i)}{C(i)}\}$, where $B'(i) = B(i)$ if $C(i) > 0$ and $B'(i) > 0$ if $C(i) \leq 0$. Replacing $B(i)$ with $B'(i)$ avoids the combination $C(i) \leq 0$ (non-applicable constraint) and $B(i) = 0$ when the constraints are compared by Protocol 4.7, CompCons. The selection done by CompCons is shown in Figure 2. The complexity of CompCons can be reduced by modifying LTZ to scale down the input to resolution 2^{-f} before comparison,

without interaction [16]. The constraint comparison in [19] uses a similar method to avoid division, but for $C(i) \leq 0$ sets $B'(i) = \infty$, i.e., greater than any value, and $C'(i) = 1$. The method used in Protocol 4.6 is more efficient (it eliminates 1 round and $m + n$ invocations) and reduces the risk of overflow.

The protocol MinCons computes $\text{argmin}_i\{\frac{B'(i)}{C(i)}\}$ by combining CompCons and a generic protocol that finds the index of the minimum value in a vector of length m in $\lceil \log(m) \rceil$ steps using $m - 1$ comparisons [19].

Update of the tableau. Protocol 4.8 updates the secret-shared tableau without revealing the position of the pivot. The computation can be carried out for different trade-offs between accuracy and efficiency. The solution shown as Protocol 4.8 achieves low complexity with minimum effects on accuracy (i.e., close to the best accuracy for a given fixed-point representation, $\delta < 2^{-f}$). Division is computed by multiplication with the reciprocal of the pivot as in Protocol 3.2, and with a single final scaling, in order to minimize rounding errors.

The complexity of the protocol is reduced by adapting the algorithm in Section 2.1 as follows. Let r and c be the indexes of the pivot's row and column, respectively, and $p = T(r, c)$. The tableau is updated by computing:

$$R'(c) \leftarrow p + 1; \;\; R'(j) \leftarrow T(r, j), \;\; j \in [1..n + 1] \backslash \{c\};$$
$$C'(r) \leftarrow (p - 1)/p; \;\; C'(i) \leftarrow T(i, c)/p, \;\; i \in [1..m + 1] \backslash \{r\};$$
$$T(i, j) \leftarrow T(i, j) - C'(i)R'(j), \;\; i \in [1..m + 1], \; j \in [1..n + 1].$$

Protocol 4.8: $[T] \leftarrow \mathsf{UpdTab}([T], [C], [R], [V], [W], [p])$

Input: $[T\langle m + 1, n + 1\rangle]$, $[C\langle m + 1\rangle]$, $[R\langle n + 1\rangle]$, $[V\langle n\rangle]$, $[W\langle m\rangle]$; $[p]$.
Output: $[T\langle m + 1, n + 1\rangle]$ (updated).

1 $[y] \leftarrow \mathsf{Rec}([p], k)$; // protocol 4.9
2 $[R'] \leftarrow \mathsf{SecWrite}([R], [V], [p] + 2^f)$; // 1 rnd, n inv
3 $[C'] \leftarrow \mathsf{SecWrite}([C], [W], [p] - 2^f)$; // + m inv
4 **foreach** $i \in [1..m + 1]$ **do parallel**
5 $\quad [C'(i)] \leftarrow [C'(i)][y]$; // 1 rnd, $m + 1$ inv
6 **foreach** $i \in [1..m + 1]$, $j \in [1..n + 1]$ **do parallel**
7 $\quad [T'(i, j)] \leftarrow [C'(i)][R'(j)]$; // 1 rnd, $(m + 1)(n + 1)$ inv
8 $\quad [T'(i, j)] \leftarrow \mathsf{TruncPrN}([T'(i, j)], 3k, 2k)$; // 1 rnd $(m + 1)(n + 1)$ inv
9 $\quad [T(i, j)] \leftarrow [T(i, j)] - [T'(i, j)]$;
10 **return** $[T]$;

Protocol 4.9: $[y] \leftarrow \mathsf{Rec}([p], k)$

1 $([c], [v]) \leftarrow \mathsf{Norm}([p], k)$; // see Table 3
2 $[x] \leftarrow \mathsf{RecItNR}([c], k)$; // see Table 3
3 $[y] \leftarrow [v][x]$; // 1 rnd, 1 inv
4 **return** $[y]$;

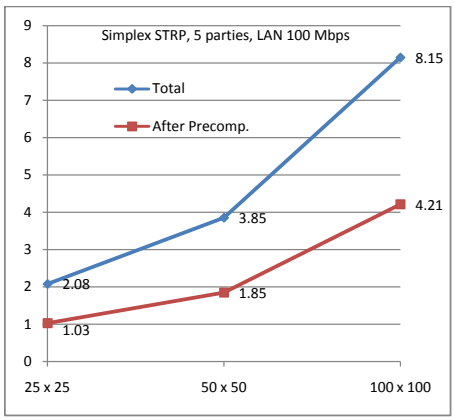
The protocol computes R' and C' in 2 rounds and $2m + n + 1$ invocations (steps 2-5), then $T'(i, j) \leftarrow C'(i)R'(j)$ in 2 rounds and $2(m+1)(n+1)$ invocations (steps 7-8, multiplication and scaling) and, finally, $T(i, j) \leftarrow T(i, j) - T'(i, j)$.

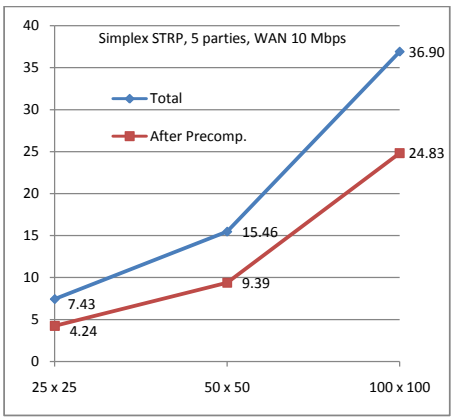
The cost of achieving best accuracy per iteration is a larger modulus, $\log(q) > 3k$. The modulus can be reduced to $\log(q) > 2k$ by scaling down C' before step 7 and/or by computing $1/p$ with precision $k' < k$.

5 Performance Evaluation and Conclusions

We implemented and tested the simplex protocol using our Java libraries for secure computation. We measured the running time of the protocol for five processes (parties) running on different PCs (Intel Core Duo, 1.8 GHz) with full mesh interconnection topology. The experiments were carried out in an isolated network for two settings: Ethernet LAN with 100 Mbps links and WAN with 10 Mbps links. The average round-trip time of the WAN was 40 ms. The LAN experiments show the protocol performance for low network delay, while the WAN experiments show the effects of higher network delay and lower bandwidth.

Figure 3 shows the running time of an iteration for $\log(q) = 288$ bits, $k = 2f = 80$ bits, and linear programs of several sizes: $m = n = 25$, $m = n = 50$, $m = n = 100$. To reduce the number of rounds, all the shared random bits needed by an iteration (for comparisons and reciprocal) are generated in parallel by an initial precomputation phase. Moreover, the running time can be reduced by executing the precomputation in parallel with the previous iteration.





	LAN			WAN		
	25×25	50×50	100×100	25×25	50×50	100×100
Precomputation	1.05	2.00	3.93	3.19	6.08	12.08
Select pivot column	0.22	0.37	0.65	0.67	1.44	2.52
Select pivot row	0.52	0.83	1.40	1.82	3.06	5.41
Update the tableau	0.29	0.65	2.17	1.75	4.88	16.90

Fig. 3. Running time (seconds) for secure simplex iterations.

The simplex algorithm in Section 2.1 can be modified to carry out the computation using integer pivoting [15,19]. Correctness and accuracy of our protocol were verified using an implementation (for public data) of the algorithm with integer pivoting, which performs the same pivot operations with exact arithmetic.

A simplex protocol for this variant of the algorithm can easily be obtained by adapting the protocols in Section 4. The main difference is the update of the tableau using secure integer arithmetic instead of fixed-point arithmetic. This protocol is more efficient than the variant in [19] (e.g., $2m + n$ comparisons instead of $3m+n$ and $2mn$ multiplications for the update of the tableau instead of $3m(n + m)$). However, they are both affected by the growth of the values in the tableau, that can reach thousands of bits for linear programs with tens of variables and constraints [19]. The experiments showed a large increase of the running time for pivot selection and precomputation (comparisons) and for the update of the tableau (large shares) even for $\log(q) = 1024$ bits.

The tests show that our approach offers an important performance gain and suitable accuracy for secure linear programming. The main performance bottleneck is the secure comparison. Our comparison protocol [16] provides statistical privacy and performs most of the computation in a small field, hence with low overhead (Table 3). By encoding binary values in small fields and using efficient share conversions, the amount of data exchanged is reduced from $O(k^2)$ bits (when integer and binary values are encoded in the same field) to $O(k)$. The simplex algorithm used by the protocol needs only $2m+n$ comparisons (instead of $3m+n$ when using the large tableau to select the pivot) and fixed-point arithmetic can reduce their input bit-length by a factor of 10 with respect to integer pivoting. Nevertheless, most of the precomputation time and pivot selection time shown in Fig. 3 is due to comparisons. Further performance improvement would require secure comparison with sublinear complexity; currently, in the multiparty setting, this can be achieved only by trading off privacy for efficiency.

The solutions presented in this paper can be applied to other simplex variants (e.g., revised simplex) and to protocols for general linear programs (finding an initial basic feasible solution). The building blocks can be used in other applications with similar requirements (accurate secure computation with rational numbers or computation with many parallel operations). These are topics of ongoing research, in parallel with the improvement of secure fixed-point arithmetic.

Acknowledgements. Part of this work was funded by the European Commission through the grant FP7-213531 to the SecureSCM project. We thank Claudiu Dragulin for his contribution to the implementation of the protocols.

References

1. A. Bednarz, N. Bean, and M. Roughan. Hiccups on the road to privacy-preserving linear programming. In *WPES '09: Proc. of the 8th ACM workshop on Privacy in the electronic society*, pages 117–120. ACM, 2009.
2. D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, 1997.

3. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
4. O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography and Data Security*, LNCS. Springer, 2010.
5. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proc. of 2nd Theory of Cryptography Conference (TCC'05)*, pages 342–362, 2005.
6. R. Cramer, I. Damgård, and U. Maurer. General Secure Multi-Party Computation from any Linear Secret-Sharing Scheme. In *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 316–334. Springer, 2000.
7. I. Damgård, M. Fitzi, E. Kiltz, J. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Proc. of 3rd Theory of Cryptography Conference (TCC'06)*, volume 3876 of *LNCS*, pages 285–304. Springer, 2006.
8. I. Damgård and R. Thorbek. Non-interactive Proofs for Integer Multiplication. In *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 412–429. Springer, 2007.
9. I. Damgard and R. Thorbek. Efficient conversion of secret-shared values between different fields. Cryptology ePrint Archive, Report 2008/221, 2008.
10. M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.
11. F. Frati, E. Damiani, P. Ceravolo, S. Cimato, C. Fugazza, G. Gianini, S. Marrara, and O. Scotti. Hazards in full-disclosure supply chains. In *Proc. 8th Conference on Advanced Information Technologies for Management (AITM'08)*, 2008.
12. R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multi-party computations with applications to threshold cryptography. In *Proc. of ACM Symposium on Principles of Distributed Computing (PODC'98)*, 1998.
13. J. Li and M. Atallah. Secure and Private Collaborative Linear Programming. In *Proc. 2nd Int. Conference on Collaborative Computing: Networking, Applications and Worksharing (ColaborateCom 2006)*, pages 19–26, Atlanta, USA, 2006.
14. T. Nishide and K. Ohta. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In *PKC 2007*, volume 4450 of *LNCS*, pages 343–360. Springer, 2007.
15. G. Rosenberg. Enumeration of All Extreme Equilibria of Bimatrix Games with Integer Pivoting and Improved Degeneracy Check. Research Report LSE-CDAM-2005-18, London School of Economics and Political Science, 2005.
16. SecureSCM. Security Analysis. Deliverable D9.2, EU FP7 Project Secure Supply Chain Management (SecureSCM), 2009.
17. SecureSCM. Protocol Description V2. Deliverable D3.2, EU FP7 Project Secure Supply Chain Management (SecureSCM), 2010.
18. T. Toft. *Primitives and Applications for Multi-party Computation*. PhD dissertation, Univ. of Aarhus, Denmark, BRICS, Dep. of Computer Science, 2007.
19. T. Toft. Solving Linear Programs Using Multiparty Computation. In *Financial Cryptography*, volume 5628 of *LNCS*, pages 90–107. Springer, 2009.