Secure Linear Programming Using Privacy-Preserving Simplex

Amitabh Saxena[1], Octavian Catrina[1], and Sebastiaan J Hoogh[2]

[1] International University, Bruchsal 76646, Germany
[2] Technische Universiteit Eindhoven, Netherlands

Abstract. The SecureSCM project (www.securescm.org) aims to develop cryptographic solutions to the problem of data sharing in Supply Chain Optimization (SCO). The SCO problem has a precise mathematical structure. It is an instance of the general Linear Programming (LP) problem. However, standard techniques for LP problems are not suitable for this purpose because they require participants to reveal private data needed as input to the algorithm. The risk of revealing this information far exceeds the benefits gained. Therefore, the aim of the project is to develop efficient techniques for securely solving LP problems. In this paper we give a summary of work done in the cryptographic aspects of the project. We describe the state-of-the art building blocks for secure linear programming along with an analysis of their complexity.

1 Introduction

Consider the dining cryptographers' problem [2]: three cryptographers are sitting down to dinner at their favorite restaurant when their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they would like to be sure of which side is paying. They decide to solve their problem as follows: each cryptographer mentally decides a secret bit - 0 if (s)he paid and 1 if (s)he did not. Then they follow a protocol to compute the AND of these bits without revealing their secret bits and reveal the result (from which they solve their problem). The above strategy is secure assuming they use a protocol that computes the (public) AND of secret bits but reveals absolutely no information about the secret bits. Such a protocol is said to be *privacy-preserving* or *secure*. Secure multi-party computation (SMC) is a distributed computation using a protocol constructed from such privacy-preserving building blocks.

Problem Definition: Our target SMC problem involves some n parties with $n \geq 3$. An LP problem in *standard* or *canonical* form (depending on the algorithm used) is given. The various values defining the constraints and the objective function are the data to be protected. A mathematical description of an LP problem is given in Section 3.

2 Preliminaries

We follow the general model of a *threshold* adversary structure, where the adversary can corrupt up to t parties. The value t is called the *corruption threshold* (or simply the threshold). For simplicity we only consider passive corruptions. We refer the reader to [9] for these concepts.
Complexity Metrics Complexity is measured for three quantities. These are:

- **Data Transmitted:** To measure this, we use a metric called an *invocation*. An *invocation* is the amount of data transmitted when every party transmits to every other party an element of a finite field $\mathbb{Z}_q$.
- **Communication Time:** To measure this, we call this unit a *round*.
A *round* is the time needed for one invocation. If a protocol step requires two or more invocations that can be done in parallel, then that step still counts as only 1 round of time. On the other hand, if two or more invocations in a step cannot be done in parallel, then each such invocation adds one more round to the protocol.
- **Local Computation:** This defines the amount of computation done by each party locally in order to prepare and process data of invocations. In practice, addition and multiplication are fast enough to be discarded from analysis. The only expensive operation is field exponentiation. Therefore, our basic unit of measuring local computation is an *exponentiation*.

2.1 Secret Sharing

Notation: The symbol $a \rightarrow i$ denotes that value a is sent to party (or set of parties) represented by i over a secure channel. The symbol $(a_1, a_2, \ldots a_m) \rightarrow (i_1, i_2, \ldots, i_m)$ is shorthand for $(a_j \rightarrow i_j)$ for $1 \leq j \leq m$.

Shamir's Secret Sharing The basic framework we use is that of Shamir secret sharing [11,1]. In this framework, each party holds one share of the secret and a threshold number of parties must pool their shares in order to obtain the secret. The protocol is defined using the following parameters: an integer $n \geq 2$ (denoting the number of parties), an integer $t < n$ denoting a threshold (maximum number of parties the adversary can corrupt) and a finite field $\mathbb{Z}_q$.

1. *Share generation:* The secret s is an element of some finite field $\mathbb{Z}_q$. Uniformly select t field elements $a_1, a_2, \ldots, a_t \in \mathbb{Z}_q$. Then construct a degree-t polynomial $p(x) = s + a_1 x + a_2 x^2 + \ldots a_t x^t \in \mathbb{Z}_q[x]$ and compute $s_i = p(i)$ for $1 \leq i \leq n$. The n-vector $(s_1, s_2, \ldots, s_n)$ is called a *sharing* of s. Denote by $\mathsf{RandShare}(s, n, t)$ a function computing a sharing of s.
2. *Share distribution:* $s_i \rightarrow i$ for $1 \leq i \leq n$.
3. *Secret reconstruction:* Choose $J \subset [1..n]$ with $|J| = t + 1$ and reconstruct secret s as: $s = \sum_{j \in J} \left(s_j \prod_{i \in J, i \neq j} \frac{-i}{j-i} \right)$

In the remainder of this paper, the symbol $[s]$ denotes a Shamir sharing of s.

Replicated Secret Sharing (RSS): See [3] for details. Using previous notation, let $A = \{X | X \subset [1..n] \wedge |X| = t\}$ be the set of all maximal unqualified subsets of parties (note: $|A| = \binom{n}{t} = \frac{n!}{(n-t)!t!}$).

1. *Share Generation:* Secret is $s \in \mathbb{Z}_q$. Generate $r_i \xleftarrow{R} \mathbb{Z}_q$ for $1 \leq i \leq |A| - 1$ and set $r_{|A|} = s - \sum_{i=1}^{|A|-1} r_i$. Then $[s]^{RQ} = (r_1, r_2, \ldots, r_{|A|})$ is a sharing of s.
2. *Share Distribution:* Assign an arbitrary labeling to A as $\{X_1, X_2, \ldots, X_{|A|}\}$. Then distribute the shares as: $r_i \rightarrow [1..n] \backslash X_i$ $(1 \leq i \leq |A|)$
3. *Secret Reconstruction:* Let $B \in [1..n]$ such that $|B| = t+1$. Then members of B jointly share the entire vector $(r_1, r_2, \ldots, r_{|A|})$. They compute $s = \sum_{i=1}^{|A|} r_i$.

2.2 SMC Framework

We describe some building blocks based on Shamir's secret sharing scheme. Let $q \equiv 3 \pmod 4$ be a prime. We consider secrets to be elements of $\mathbb{Z}_q$ that are (t, n)-Shamir shared for $t, n \in \mathbb{Z}^+$. We further require $n \geq 3$ and $t < n/2$. These protocols are standard in the literature and provide perfect security [12,1].

(A) Addition of secrets: Party $i \in [1..n]$ holds a_i, b_i, the shares of $a, b \in \mathbb{Z}_q$. It computes $c_i = a_i + b_i$. Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[a + b]$.

(B) Addition of a secret with a public field element: Party $i \in [1..n]$ holds a_i, the share of $a \in \mathbb{Z}_q$. Let $\alpha \in \mathbb{Z}_q$ be a public value. Party i computes $c_i = \alpha + a_i$. Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[\alpha + a]$.

(C) Multiplication of a secret with a public integer: Party $i \in [1..n]$ holds a_i, the share of $a \in \mathbb{Z}_q$. Let $\alpha \in \mathbb{Z}$ be a public value. Party i computes $c_i = \alpha a_i$. Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[\alpha a]$.

(D) Linear Combination of secrets: Party $i \in [1..n]$ holds a_i, b_i, the shares of $a, b \in \mathbb{Z}_q$. Let $\alpha, \beta \in \mathbb{Z}$ be public values. Party i computes $c_i = \alpha a_i + \beta b_i$. Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[\alpha a + \beta b]$.

(E) Multiplication of secrets: Party $i \in [1..n]$ holds a_i, b_i, the shares of $a, b \in \mathbb{Z}_q$. They follow Protocol 2.1. Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[ab]$.

(F) Inner Product of Secrets: Let $\mathbf{a} = (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m) \in \mathbb{Z}_q^m$ be an m-vector of field elements. For $i \in [1, m]$, let $[\mathbf{a}_i] = (a_{(i,1)}, a_{(i,2)}, \ldots, a_{(i,n)})$ be a sharing of $\mathbf{a}_i$. For $j \in [1, n]$, define $\mathbf{a}_j^* \overset{\text{def}}{=} (a_{(1,j)}, a_{(2,j)}, \ldots, a_{(m,j)})$. By $[\mathbf{a}]$, we denote the n-vector $(\mathbf{a}_1^*, \mathbf{a}_2^*, \ldots, \mathbf{a}_n^*)$. Let $\mathbf{a}, \mathbf{b}$ be two m-vectors. Party $i \in [1..n]$ holds two m-vectors $\mathbf{a}_i^*, \mathbf{b}_i^*$ as defined above. To compute the sharing $[\mathbf{a} \cdot \mathbf{b}]$, they use Protocol 2.2. In the protocol, $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[\mathbf{a} \cdot \mathbf{b}]$.

(G) Reveal: Protocol 2.3 is used to reveal a secret.

Protocol 2.1: $[c] \leftarrow \mathsf{Mul}([a],[b])$

```
foreach party $i \in [1..2t+1]$ do parallel
    $d_i \leftarrow a_i b_i$;
    $(d_{(i,1)}, d_{(i,2)}, \ldots, d_{(i,n)}) \leftarrow \mathsf{RandShare}(d_i, n, t)$;
    $(d_{(i,1)}, d_{(i,2)}, \ldots, d_{(i,n)}) \rightarrow (1, 2, \ldots, n)$;          // 1 rnd, ≈1 inv
foreach party $j \in [1..n]$ do
    $c_j \leftarrow \sum_{i=1}^{2t+1} \left( d_{(i,j)} \prod_{\ell=1,\ell\neq i}^{2t+1} \frac{-\ell}{i-\ell} \right)$;
    return $c_j$;
```

Protocol 2.2: $[c] \leftarrow \mathsf{Inner}([\mathbf{a}],[\mathbf{b}])$

```
foreach party $i \in [1..2t+1]$ do parallel
    $d_i \leftarrow \mathbf{a}_i^* \cdot \mathbf{b}_i^*$;                       // $d_i =$ inner product of $\mathbf{a}_i^*$ and $\mathbf{b}_i^*$
    $(d_{(i,1)}, d_{(i,2)}, \ldots, d_{(i,n)}) \leftarrow \mathsf{RandShare}(d_i, n, t)$;
    $(d_{(i,1)}, d_{(i,2)}, \ldots, d_{(i,n)}) \rightarrow (1, 2, \ldots, n)$;          // 1 rnd, ≈1 inv
foreach party $j \in [1..n]$ do
    $c_j \leftarrow \sum_{i=1}^{2t+1} \left( d_{(i,j)} \prod_{\ell=1,\ell\neq i}^{2t+1} \frac{-\ell}{i-\ell} \right)$;
    return $c_j$;
```

Protocol 2.3: $a \leftarrow \mathsf{Reveal}([a])$

```
foreach party $i \in [1..t+1]$ do parallel
    $a_i \rightarrow [1..n]$;                                              // 1 rnd, ≈1 inv
foreach party $j \in [1..n]$ do
    $a \leftarrow \sum_{i=1}^{t+1} \left( a_i \prod_{\ell=1,\ell\neq i}^{t+1} \frac{-\ell}{i-\ell} \right)$ ;                 // local computation
    return $a$;
```

Data Representation: Every field $\mathbb{Z}_q$ consists of the additive and multiplicative identities, 0 and 1 respectively. These can be used to encode the boolean variables $True$ and $False$ respectively. Integers in the range $[-2^\ell..2^\ell - 1]$ will be represented as elements of a prime field $\mathbb{Z}_q$ such that $q > 2^{\ell+1}$. For some integer α in the range, the corresponding field element representation is $\alpha \bmod q$.

Notation: We use infix notation to indicate addition, subtraction, multiplication and linear combination protocols. For instance, $[c] \leftarrow [a][b]$ indicates that $[c]$ is the (secret) output of the protocol for multiplication with $[a], [b]$ as inputs. Let $\mathbf{Fld}_q : [-2^\ell..2^\ell - 1] \mapsto \mathbb{Z}_q$ be the function that maps integers to their field element representation. In other words, $\mathbf{Fld}_q(x) = x \bmod q$.

Arithmetic: For integers $a, b \in [-2^\ell..2^\ell - 1]$, the operation $a \odot b$ for $\odot \in \{+, -, \times\}$ can be done as: $a \odot b = \mathbf{Fld}_q^{-1}(\mathbf{Fld}_q(a) \odot \mathbf{Fld}_q(b))$. Furthermore, if $b|a$ then the division a/b can be done as $a/b = \mathbf{Fld}_q^{-1}(\mathbf{Fld}_q(a) \times \mathbf{Fld}_q(b)^{-1})$.

Complexity: Appendix B gives the complexity of the above protocols.

2.3 Other Building Blocks

Let $[s]^{RQ} = (r_1, r_2, \dots, r_{|A|})$ be an RSS sharing of some $s \in \mathbb{Z}_q$. Then the share vector of each party $i \in [1..n]$ is $(r_j)_{i \notin X_j}$ with $\binom{n-1}{t}$ elements.

(A) Conversion of RSS Shares to Shamir Shares: To convert the $[s]^{RQ}$ to a Shamir sharing $[s]$ for the same access structure, use Protocol 2.4 [3].

Protocol 2.4: $[s] \leftarrow \mathsf{RSStoShamir}([s]^{RQ})$

```
foreach party $i \in [1..n]$ do
    $s_i \leftarrow \sum_{j=1, i \notin X_j}^{|A|} \left( r_j \prod_{\ell \in X_j} \frac{(\ell - i)}{\ell} \right)$;  // All arithmetic is done in $\mathbb{Z}_q$
    return $s_i$;
```

Then $(s_1, s_2, \dots, s_n)$ is the Shamir sharing $[s]$.

Notation: Let $[s_m]^{RQ} = (r_1, r_2, \dots, r_{|A|})$ be an RSS master sharing over $\mathbb{Z}_q$. Let $H : \mathbb{Z}_q \times \mathbb{Z}^+ \times \mapsto \mathbb{Z}_q$ be a PRF with keys and outputs in $\mathbb{Z}_q$, and inputs in $\mathbb{Z}^+$. For any $i \in \mathbb{Z}^+$, let $[H_{[s_m]}(i)]^{RQ}$ be the RSS sharing $(H_{r_1}(i), H_{r_2}(i), \dots, H_{r_{|A|}}(i))$. Let F be the set of polynomials over $\mathbb{Z}_q$ of degree $\leq 2t$. Then F is a vector space of dimension $2t + 1$. Let $F_i = \{f | (f \in F) \wedge (f(0) = 0) \wedge (\forall j \in X_i : f(j) = 0)\}$ for $i \in [1..|A|]$. Then F_i is a subspace of F of dimension $(2t - (1 + t)) + 1 = t$. Let $\{f_{i,1}, f_{i,2}, \dots, f_{i,t}\}$ be any basis of F_i, which is public information. Let $H' : \{0,1\}^\alpha \times \mathbb{Z}^+ \mapsto \mathbb{Z}_q$ be a PRF with keys in $\{0,1\}^\alpha$, and let $H'' : \mathbb{Z}_q \times \mathbb{Z}^+ \mapsto \{0,1\}^\alpha$ be a PRF with keys in $\mathbb{Z}_q$. Denote by $[s_m]^{RQ} \leftarrow \mathsf{MasterRSS}(\mathbb{Z}_q)$ a protocol generating an initial RSS sharing [3].

Selecting a basis: For each $i \in [1..|A|]$ and each $\ell \in [1..t]$, consider the polynomial $f_{i,\ell} = x^\ell \prod_{j \in X_i} (x - j)$. By construction $f_i = \{f_{i,1}, f_{i,2}, \dots, f_{i,t}\} \subset F_i$; each element of f_i is linearly independent of the others (because they are of different degree); and $|f_i| = t$. Thus, f_i is a basis of F_i.

(B) Random RSS sharing: Protocol 2.5 generates a random RSS sharing [3].

Protocol 2.5: $[s]^{RQ} \leftarrow \mathsf{RandRSS}()$

```
static $ctr \leftarrow 0$;                              // static used as in C language
static $[s_m]^{RQ} \leftarrow \mathsf{MasterRSS}(\mathbb{Z}_q)$;
$[s]^{RQ} \leftarrow [H_{[s_m]}(ctr)]^{RQ}$;
$ctr{+}{+}$;
return $[s]^{RQ}$;
```

(C) Random Field Element: Protocol 2.6 generates a random field element [3].

Protocol 2.6: $[r] \leftarrow \mathsf{RandFld}()$

```
$[r]^{RQ} \leftarrow \mathsf{RandRSS}()$;
$[r] \leftarrow \mathsf{RSStoShamir}([r]^{RQ})$;
return $[r]$;
```

(D) Non interactive random zero sharing: Protocol 2.7 was proposed in [3]. It generates a random Shamir-sharing [0] using a polynomial of degree $2t < n$.

Protocol 2.7: $[z] \leftarrow \mathsf{RandZero}()$

```
static $ctr \leftarrow 0$;
static $[s]^{RQ} \leftarrow \mathsf{RandRSS}()$;                    // Let $[s]^{RQ} = (r_1, r_2, \ldots, r_{|A|})$
foreach $i \in [1..|A|]$ do
    foreach party $j \in X_i$ do
        foreach $\ell \in [1..t]$ do
            static $r_{i,\ell} \leftarrow H''_{r_i}(\ell)$;          // $r_{i,\ell}$ act like keys to a PRF
foreach party $j \in [1..n]$ do
    $z_j \leftarrow \sum_{i=1, j \notin X_i}^{|A|} \left( \sum_{\ell=1}^{t} H'_{r_{i,\ell}}(ctr) \cdot f_{i,\ell}(j) \right)$;
    return $z_j$;
$ctr{+}{+}$;
```

(E) Multiplication of secrets with public output: Party $i \in [1..n]$ holds a_i, b_i, the shares of $a, b \in \mathbb{Z}_q$. They follow Protocol 2.8:

Protocol 2.8: $c \leftarrow \mathsf{MulPub}([a], [b])$

```
$[z] \leftarrow \mathsf{RandZero}()$;        // Let $[z] = (z_1, z_2, \ldots, z_n)$, the Shamir shares
foreach party $i \in [1..2t+1]$ do parallel
    $d_i \leftarrow a_i b_i + z_i$;
    $d_i \rightarrow [1..n]$;                                    // 1 rnd, $\approx$1 inv
foreach party $j \in [1..n]$ do
    $c \leftarrow \sum_{i=1}^{2t+1} \left( d_i \prod_{\ell=1, \ell \neq i}^{2t+1} \frac{-\ell}{i-\ell} \right)$;
    return $c$;
```

(F) Computing inverse of secret non-zero field element: Protocol 2.9 [1] computes the sharing $[a^{-1}]$ given the sharing $[a]$ for some $a \in \mathbb{Z}_q^*$.

Protocol 2.9: $[b] \leftarrow \mathsf{Inv}([a])$

```
$[r] \leftarrow \mathsf{RandFld}()$;
$x \leftarrow \mathsf{MulPub}([r][a])$;                               // 1 rnd, 1 inv
$y \leftarrow x^{-1}$;
$[b] \leftarrow y[r]$;
return $[b]$;
```

(G) Shared random bits: Protocol 2.10 generates a secret random bit.

Protocol 2.10: $[b] \leftarrow \mathsf{RandBit}()$

```
$[r] \leftarrow \mathsf{RandFld}()$;                              // Requires $q \equiv 3 \pmod 4$
$u \leftarrow \mathsf{MulPub}([r],[r])$;                                    // 1 rnd, 1 inv
$v \leftarrow u^{-\frac{q+1}{4}} \bmod q$;                                         // 1 exp
$[b] \leftarrow (v[r]+1)(2^{-1} \bmod q)$;
return $[b]$;
```

(H) Normally Distributed Integers: Protocol 2.11 outputs a Shamir sharing of a normally distributed integer in $\in [0..\binom{n}{t}(2^\alpha - 1)]$ for some positive integer α. The protocol requires that $q > \binom{n}{t}2^\alpha$ to prevent wraparound modulo q.

Notation: Let $[s_m]^{RQ} = (r_1, r_2, \ldots, r_{|A|})$ be an RSS master sharing. Let $H^\alpha : \mathbb{Z}_q \times \mathbb{Z}^+ \mapsto [0..2^\alpha - 1]$ be a PRF with keys in $\mathbb{Z}_q$ for $\alpha \in \mathbb{Z}^+$. For any $i \in \mathbb{Z}$, $[H^\alpha_{[s_m]}(i)]^{RZ}$ denotes the RSS sharing $(H^\alpha_{r_1}(i), H^\alpha_{r_2}(i), \ldots, H^\alpha_{r_{|A|}}(i))$.

Protocol 2.11: $[r] \leftarrow \mathsf{RandInt}(\alpha)$

```
static $ctr \leftarrow 0$;                         // static used as in C language
static $[s_m]^{RQ} \leftarrow \mathsf{MasterRSS}(\mathbb{Z}_q)$;
$[r]^{RZ} \leftarrow [H^\alpha_{[s_m]}(ctr)]^{RZ}$;
foreach party $i \in [1..n]$ do
    $s_i \leftarrow \sum_{j=1, i \notin X_j}^{|A|} \left( r_j \prod_{\ell \in X_j} \frac{(\ell - i)}{\ell} \right)$; // All arithmetic is done in $\mathbb{Z}_q$
    return $s_i$;
$ctr{+}{+}$;
```

Then $[s] = (s_1, s_2, \ldots, s_n)$ is a Shamir sharing of $s \in [0..\binom{n}{t}(2^\alpha - 1)]$.

(I) k-ary, Prefix and Binary Operations Let $\mathcal{A}$ be a set and $\odot : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ an associative binary operator. We denote $[a] \odot [b]$ a secure evaluation of $a \odot b$ with secret inputs and secret output. Define the following operations:

k-ary: $[p] = [a_1] \odot \ldots \odot [a_k] = \bigodot_{i=1}^{k} [a_i]$.

Prefix: $([p_1], \ldots, [p_k]) = \mathsf{pre}_{\bigodot}([a_1], \ldots, [a_k])$, $[p_j] = \bigodot_{i=1}^{j} [a_i]$, $1 \le j \le k$.

Assuming that one evaluation of $\odot$ takes α rounds and β invocations, Protocol 2.12 computes $[p] = \bigodot_{i=1}^{k} [a_i]$, while Protocol 2.13 computes the prefix in $O(\log(k))\alpha$ rounds. For simplicity k is assumed to be a power of 2.

Protocol 2.12: $[p] \leftarrow \mathsf{kOp}(\odot, [a_1], \ldots, [a_k])$

```
if $k > 1$ then
    foreach $i \in [1..k/2]$ do parallel
        $[u_i] \leftarrow [a_{2i}] \odot [a_{2i-1}]$;                 // $\alpha$ rnd, $\beta(\frac{k}{2} - 1)$ inv
    $[p] \leftarrow \mathsf{kOp}(\odot, [u_1], \ldots, [u_{k/2}])$;          // $\alpha \log_2(\frac{k}{2})$ rnd, $\beta\frac{k}{2}$ inv
else
    $[p] \leftarrow [a_1]$;
return $[p]$;
```

Protocol 2.13: $([p_1],[p_2],\ldots,[p_k]) \leftarrow \mathsf{PreOp}(\odot,[a_1],\ldots,[a_k])$

```
foreach $i \in [1..\log_2(k)]$ do
    foreach $j \in [1..k/2^i]$ do parallel
        $y \leftarrow 2^{i-1} + j \cdot 2^i$;
        foreach $z \in [1..2^{i-1}]$ do parallel
            $[a_{y+z}] \leftarrow [a_y] \odot [a_{y+z}]$;        // $\alpha\log_2(k)$ rnd, $\beta(\frac{k}{2}\log_2(k))$ inv
return $([a_1],\ldots,[a_k])$ ;
```

The above protocols can be used for any associative operation. They provide perfect privacy if the operation is evaluated with perfect privacy. The protocols are based on well known techniques from computer arithmetic [4] and parallel algorithms [5]. The protocols can easily be adapted to any k. The complexity is given in Appendix B. We next need a protocol for carry-out in binary addition.

Carry propagation in binary addition: Let $a = (a_k,\ldots,a_1)$ and $b = (b_k,\ldots,b_1)$ the two integer inputs, $p_i = a_i \oplus b_i$ the carry propagation bit and $g_i = a_i \wedge b_i$ the carry generation bit, for $1 \le i \le k$. The carry bits can be computed as: $c_1 = g_1$ and $c_i = g_i \vee (p_i \wedge c_{i-1})$ for $i \in [2..k]$. Define an operator: $\circ : \{0,1\}^4 \ni (p_1,g_1,p_2,g_2) \mapsto (p_1 \wedge p_2, g_2 \vee (p_2 \wedge g_1)) \in \{0,1\}^2$. This operator can be rewritten as $\circ(p_1,g_1,p_2,g_2) = (p_1p_2, g_2 + p_2g_1)$ and can be computed in 1 round and 2 invocations.

Computation of carry-out: Protocol 2.14 takes as input two bitwise-shared integers, $(a_k,\ldots,a_1)$ and $([b_k],\ldots,[b_1])$. It outputs the carry-out.

Protocol 2.14: $[g] \leftarrow \mathsf{Carry}((a_k,\ldots,a_1),([b_k],\ldots,[b_1]),c)$

```
foreach $i \in [1..k]$ do parallel
    $[d_i] \leftarrow (a_i + [b_i] - 2a_i[b_i], a_i[b_i])$;                // 1 rnd, $k$ inv
$[g_1] \leftarrow [g_1] + c[p_1]$;
$[d]_B \leftarrow \mathsf{kOp}(\circ,[d_k],\ldots,[d_1])$;                 // $\log k$ rnd, $2k-2$ inv
$([p],[g]) \leftarrow [d]_B$;
return $[g]$;
```

(J) Inequality test for bitwise-shared integers. Given two bitwise-shared k-bit unsigned integers $[a]_B$ and $[b]_B$, Protocol 2.15 computes $[a < b]$.

Protocol 2.15: $[s] \leftarrow \mathsf{BitLT}(a,[b]_B)$

```
foreach $i \in [0..k-1]$ do
    $[b'_i] \leftarrow 1 - [b_i]$;
$[s] \leftarrow \mathsf{Carry}((a_{k-1},\ldots,a_0),([b'_{k-1}],\ldots,[b'_0]),1)$;    // $\log k$ rnd, $2k-2$ inv
return $1 - [s]$;
```

(K) Protocol LTZ (Less Than Zero). Protocol 2.16 computes $s = (x < 0)$ for any $x \in [-2^{k-1}..2^{k-1}-1]$. The protocol takes as inputs $[a] = [\mathbf{Fld}_q(x)]$ and the public integer k, and returns $[s] = [\mathbf{Fld}_q(x < 0)]$.

Protocol 2.16: $[s] \leftarrow \mathsf{LTZ}([a], k)$

```
$[b] \leftarrow 2^{k-1} + [a]$;
foreach $i \in [0..k-2]$ do parallel
    $[r_i] \leftarrow \mathsf{RandBit}(q)$;                        // 1 rnd, $k-1$ inv, $k-1$ exp
$[r']_B \leftarrow ([r_{k-2}], \ldots, [r_0])$;
$[r'] \leftarrow \sum_{i=0}^{k-2} 2^i \cdot [r_i]$;
$[r''] \leftarrow \mathsf{RandInt}(\kappa + 1)$;
$[r] \leftarrow 2^{k-1} \cdot [r''] + [r']$;
$c \leftarrow \mathsf{Reveal}([b] + [r])$;                                    // 1 rnd, 1 inv
$c' \leftarrow c \bmod 2^{k-1}$;
$[u] \leftarrow \mathsf{BitLT}(c', [r']_B)$;                       // $\log(k-1)$ rnd, $2k-4$ inv
$[b'] \leftarrow c' - [r'] + [u] \cdot 2^{k-1}$;
$[s] \leftarrow ([a] - [b'])(2^{1-k} \bmod q)$;
return $[s]$;
```

(L) Protocol GTZ (Greater Than Zero). $\mathsf{GTZ}(a) = \mathsf{LTZ}(-a)$.

(M) Secret Indexing: Let $[A]$ be an array of secret-shared values, using Shamir sharing. Given a secret index $[v]$, we want to read (or write) $[A(v)]$ without revealing the value $A(v)$ and the index v.

Notation: Elements of array (say A) with n elements start from index 1 and are written successively as $A(1), A(2), \ldots, A(n)$. W e denote by $[A]$ the array $([A(1)], [A(2)], \ldots, [A(n)])$. Multidimensional arrays are handled similarly. Bitmask vectors will be represented using boldface letters $(\mathbf{A}, \mathbf{B}, \ldots)$.

The following protocols are from [12]. The element at secret index $[v]$ is selected using an array of secret binary values $[\mathbf{V}]$ (bitmask), so that $[\mathbf{V}(i)] = [0]$ for $i \neq v$ and $[\mathbf{V}(v)] = [1]$. Protocols 2.17 and 2.18 allow secret reading and writing from/to a vector. Protocols 2.19, 2.20, 2.21, and 2.22 are used for secretly reading/writing rows/cols of matrix.

Protocol 2.17: $[s] \leftarrow \mathsf{SecRead}([A], [\mathbf{V}])$

```
Input: $m$-vector $[A]$, $m$-vector $[\mathbf{V}]$.
Output: value $[s]$.
$[s] \leftarrow \mathsf{Inner}([A], [\mathbf{V}])$;                                // 1 rnd, 1 inv
return $[s]$;
```

Protocol 2.18: $[A] \leftarrow \mathsf{SecWrite}([A], [\mathbf{V}], [s])$

```
Input: $m$-vector $[A]$, $m$-vector $[\mathbf{V}]$, and a value $[s]$
Output: $m$-vector $[A]$ (overwritten)
foreach $i \in [1..m]$ do parallel
    $[A(i)] \leftarrow [A(i)] + [\mathbf{V}(i)]\,([s] - [A(i)])$;             // 1 rnd, $m$ inv
return $[A]$;
```

Protocol 2.19: $[R] \leftarrow \mathsf{SecReadRow}([T], [\mathbf{V}])$

```
Input: $(m+1) \times (n+1)$-matrix $[T]$, $m$-vector $[\mathbf{V}]$.
Output: $(n+1)$-vector $[R]$.
foreach $i \in [1..n+1]$ do parallel
    foreach $j \in [1..m]$ do  $[X(j)] \leftarrow [T(j,i)]$;
    $[R(i)] \leftarrow \mathsf{SecRead}([X], [\mathbf{V}])$;                    // 1 rnd, $n+1$ inv
return $[R]$;
```

Protocol 2.20: $[C] \leftarrow \mathsf{SecReadCol}([T], [\mathbf{W}])$

```
Input: $(m+1) \times (n+1)$-matrix $[T]$, $n$-vector $[\mathbf{W}]$.
Output: $(m+1)$-vector $[C]$.
foreach $i \in [1..m+1]$ do parallel
    foreach $j \in [1..n]$ do  $[X(j)] \leftarrow [T(i,j)]$;
    $[C(i)] \leftarrow \mathsf{SecRead}([X], [\mathbf{W}])$;                    // 1 rnd, $m+1$ inv
return $[C]$;
```

Protocol 2.21: $[T] \leftarrow \mathsf{SecWriteRow}([T], [\mathbf{V}], [R])$

```
Input: $(m+1) \times (n+1)$-matrix $[T]$, $m$-vector $[\mathbf{V}]$, $(n+1)$-vector $[R]$
Output: $(m+1) \times (n+1)$-matrix $[T]$ (overwritten)
foreach $i \in [1..n+1]$ do parallel
    $[T(*,i)] \leftarrow \mathsf{SecWrite}(T(*,i), [\mathbf{V}], [R(i)])$;         // 1 rnd, $m(n+1)$ inv
return $[T]$;
```

Protocol 2.22: $[T] \leftarrow \mathsf{SecWriteCol}([T], [\mathbf{W}], [C])$

```
Input: $(m+1) \times (n+1)$-matrix $[T]$, $n$-vector $[\mathbf{W}]$, $(m+1)$-vector $[C]$
Output: $(m+1) \times (n+1)$-matrix $[T]$ (overwritten)
foreach $i \in [1..m+1]$ do parallel
    $[T(i,*)] \leftarrow \mathsf{SecWrite}([T(i,*)], [\mathbf{W}], [C(i)])$;       // 1 rnd, $n(m+1)$ inv
return $[T]$;
```

Complexity: Appendix B gives the complexity of above protocols.
Correctness: We refer to [10,12,1,3] for the correctness of above protocols.

3 Secure Linear Programming

A linear programming (LP) *maximization* problem in *standard form* [6] consists of n non-negative variables x_j $(1 \leq j \leq n)$ and m constraints $\sum_{j=1}^{n} a_{i,j} \cdot x_j \leq b_i$ for $(1 \leq i \leq m)$. The goal is to maximize an objective function $\sum_{j=1}^{n} f_j \cdot x_j$ where $f_j, a_{i,j}, b_i, x_j \in \mathbb{R}$ and $b_i, x_j \geq 0$. (note that n is **not** the number of parties). The most suitable algorithm for our purpose is *Simplex* [6,7]. The algorithm is given in Appendix A. Compared to the method of [12], we use a variant with $(m+1) \times (n+1)$ matrix T instead of $(m+1) \times (m+n+1)$. Secondly, we use a more efficient method for selecting the pivot row.

Secure Simplex: Protocol 3.1 implements the iteration of Simplex. The input to the protocol, $[T]$ is a $(m+1) \times (n+1)$ matrix representing the matrices A, B, F, Z as defined above. The matrix $[T]$ is assumed to exist in the system.

Protocol 3.1: $([T], [U], [S], [p], \mathsf{Result}) \leftarrow \mathsf{Simplex}([T], [U], [S], [p'])$

```
Input: $(m+1) \times (n+1)$-matrix $[T]$, $m$-vector $[S]$, $n$-vector $[U]$, $[p']$
Output: $([T], [S], [U])$, (updated) , $[p]$, Result $\in \{$Opt, Unb, None$\}$
$[\mathbf{V}] \leftarrow \mathsf{GetPivCol}([T])$;                          // see Table 1
if Null($[\mathbf{V}]$) then return $([T], [S], [U], [p']$, Opt);        // 1 rnd, 1 inv
$[C] \leftarrow \mathsf{SecReadCol}([T], [\mathbf{V}])$;                     // 1 rnd, $m+1$ inv
$([\mathbf{W}], [\mathbf{D}]) \leftarrow \mathsf{GetPivRow}([B], [C])$;              // see Table 1
if Null($[\mathbf{D}]$) then return $([T], [S], [U], [p']$, Unb);        // 1 rnd, 1 inv
$[R] \leftarrow \mathsf{SecReadRow}([T], [\mathbf{W}])$;                     // 1 rnd, $n+1$ inv
$[p] \leftarrow \mathsf{SecRead}([R], [\mathbf{V}])$;                          // 1 rnd, $n$ inv
$[T] \leftarrow \mathsf{UpdTab}([T], [C], [R], [\mathbf{V}], [\mathbf{W}], [p], [p'])$;     // see Table 1
$([S], [U]) \leftarrow \mathsf{UpdVar}([S], [U], [\mathbf{V}], [\mathbf{W}])$;            // 2 rnd, $2m+2$ inv
return $([T], [S], [U], [p]$, None);
```

The above protocol implements one iteration of Simplex. In the next iteration, $[p']$ is set to $[p]$ and the process is repeated. The remaining protocols are used as building blocks in Protocol 3.1. Protocol 3.2 returns a bitmask vector $[\mathbf{V}]$ indicating the pivot column c if it exists (i.e., $\mathbf{V}(c) = 1$ and $\mathbf{V}(i) = 0$ for all $i \neq c$ if pivot column exists).

Protocol 3.2: $[\mathbf{V}] \leftarrow \mathsf{GetPivCol}([T])$

```
Input: $(m+1) \times (n+1)$-matrix $[T]$.
Output: $n$-vector $[\mathbf{V}]$.
foreach $i \in [1..n]$ do parallel
    $[\mathbf{D}(i)] \leftarrow \mathsf{LTZ}([T(m+1, i)], k)$;               // $n$ parallel LTZs
$[\mathbf{D}'] \leftarrow \mathsf{PreOp}(\mathsf{OR}, [\mathbf{D}])$;                 // $\log n$ rnd, $\frac{n}{2}\log n$ inv
$[\mathbf{V}(1)] \leftarrow [\mathbf{D}'(1)]$;
foreach $i \in [2..n]$ do
    $[\mathbf{V}(i)] \leftarrow [\mathbf{D}'(i)] - [\mathbf{D}'(i-1)]$;
return $[\mathbf{V}]$;
```

Protocol 3.3 outputs 0 if the bitmask $\mathbf{V}$ contains all zeros.

Protocol 3.3: $s \leftarrow \mathsf{Null}([\mathbf{V}])$

```
$[v] \leftarrow [\mathbf{V}(1)]$;
foreach $i \in [2..\ell]$ do
    $[v] \leftarrow [v] + [\mathbf{V}(i)]$;
$s \leftarrow \mathsf{Reveal}([v])$;                                       // 1 rnd, 1 inv
return $s$;
```

Protocol 3.4 returns a bitmask vector $[\mathbf{W}]$ indicating the pivot row r if it exists (i.e., $\mathbf{W}(r) = 1$ and $\mathbf{W}(i) = 0$ for all $i \neq r$ if pivot row exists). $\mathbf{D}$ is another bitmask indicating if a pivot row exists or not. The protocol 3.5 is the log round comparison protocol from [12]. The constraint comparison protocol, CompCons is adapted from [12] with optimizations in Steps 3 and 4.

Protocol 3.4: $([\mathbf{W}], [\mathbf{D}]) \leftarrow \mathsf{GetPivRow}([B], [C])$

```
  Input: $m$-vectors $[B]$, $[C]$.
  Output: $m$-vector $[\mathbf{W}]$.
1 foreach $i \in [1..m]$ do parallel
2     $[\mathbf{D}(i)] \leftarrow \mathsf{GTZ}([C(i)])$;                          // $m$ parallel GTZs
3     $[C^*(i)] \leftarrow [C(i)][\mathbf{D}(i)]$;                            // 1 rnd, $m$ inv
4     $[B^*(i)] \leftarrow [B(i)] + 1 - [\mathbf{D}(i)]$;
5                 // $C^* =$ copy of $C$ with non-app entries set to zero
           // $B^* =$ copy of $B$ with non-app entries set to non-zero
6 $[\mathbf{W}] \leftarrow \mathsf{MinCons}([B^*], [C^*], m)$;                // $(5 + \log(k-1)) \log m$ rnd,
                                                  // $m(3k+5)$ inv, $m(k-1)$ exp
7 return $([\mathbf{W}], [\mathbf{D}])$;
```

The reader is referred to [12] for the rationale behind Protocols 3.5 and 3.6.

Protocol 3.5: $[\mathbf{W}] \leftarrow \mathsf{MinCons}([B^*], [C^*], \alpha)$ (assume α is a power of 2)

```
1  if $\alpha = 1$ then return $[1]$;
2  else
3      foreach $i \in [1..\alpha/2]$ do parallel
4          $[A_0] \leftarrow ([B^*(2i-1)], [C^*(2i-1)])$;
5          $[A_1] \leftarrow ([B^*(2i)], [C^*(2i)])$;
6          $[\mathbf{Z}(i)] \leftarrow \mathsf{CompCons}([A_0], [A_1])$;
                          // $3 + \log(k-1)$ rnd, $\frac{\alpha}{2}(3k+2)$ inv, $\frac{\alpha}{2}(k-1)$ exp
7          do parallel                                          // 1 rnd, $\alpha$ inv
8              $[B^*_{\text{new}}(i)] \leftarrow [\mathbf{Z}(i)]\big([B^*(2i-1)] - [B^*(2i)]\big) + [B^*(2i)]$;
9              $[C^*_{\text{new}}(i)] \leftarrow [\mathbf{Z}(i)]\big([C^*(2i-1)] - [C^*(2i)]\big) + [C^*(2i)]$;
10     $[\mathbf{W}_{\text{new}}] \leftarrow \mathsf{MinCons}([B^*_{\text{new}}], [C^*_{\text{new}}], \alpha/2)$;   // $(5 + \log(k-1)) \log \frac{\alpha}{2}$ rnd,
                                 // $(3k+5)(\alpha-1)$ inv, $(k-1)(\alpha-1)$ exp
11     foreach $i \in [1..\alpha/2]$ do parallel
12         $[\mathbf{W}(2i)] \leftarrow [\mathbf{Z}(i)][\mathbf{W}_{\text{new}}(i)]$;                        // 1 rnd, $\frac{\alpha}{2}$ inv
13         $[\mathbf{W}(2i-1)] \leftarrow [\mathbf{W}_{\text{new}}(i)] - [\mathbf{W}(2i)]$;
14     return $[\mathbf{W}]$;
```

Protocol 3.6: $[\mathbf{z}] \leftarrow \mathsf{CompCons}([A_0], [A_1])$

```
1 parse $[A_0]$ as $([b^*_0], [c^*_0])$;
2 parse $[A_1]$ as $([b^*_1], [c^*_1])$;
3 $[x] \leftarrow [b^*_0][c^*_1] - [b^*_1][c^*_0]$;                                 // 1 rnd, 2 inv
4 return $\mathsf{LTZ}([x], k)$;                  // $2 + \log(k-1)$ rnd, $3k$ inv, $k-1$ exp
```

Protocol 3.7 updates the tableau T after the pivot has been selected.

Protocol 3.7: $[T] \leftarrow \mathsf{UpdTab}([T], [C], [R], [\mathbf{V}], [\mathbf{W}], [p], [p'])$

```
Input: $(m+1) \times (n+1)$-matrix $[T]$, $(m+1)$-vector $[C]$, $(n+1)$-vector $[R]$,
       $n$-vector $[\mathbf{V}]$, $m$-vector $[\mathbf{W}]$, value $[p]$.
Output: $(m+1) \times (n+1)$-matrix $[T]$ (overwritten).
$[y] \leftarrow \mathsf{Inv}([p'])$;                                    // 1 rnd, 1 inv
foreach $i \in [1..m+1]$ do
    $([C'(i)], [C''(i)], [P(i)], [Y(i)]) \leftarrow (-[C(i)], [C'(i)], [p], [y])$;
do parallel                                                            // 1 rnd
    $\mathsf{SecWrite}([C'], [\mathbf{W}], 0)$;                          // $m$ inv
    $\mathsf{SecWrite}([C''], [\mathbf{W}], [p'])$;                      // $m$ inv
    $\mathsf{SecWrite}([P], [\mathbf{W}], 1)$;                           // $m$ inv
    $\mathsf{SecWrite}([Y], [\mathbf{W}], 1)$;                           // $m$ inv
foreach $i \in [1..m+1]$ do parallel
    foreach $j \in [1..n+1]$ do parallel
        $[x] \leftarrow [T(i,j)][P(i)] + [R(j)][C'(i)]$; // 1 rnd, $2(m+1)(n+1)$ inv
        $[T(i,j)] \leftarrow [x]$;
foreach $i \in [1..m+1]$ do parallel
    foreach $j \in [1..n+1]$ do parallel
        $[T(i,j)] \leftarrow [T(i,j)][Y(i)]$;            // 1 rnd, $(m+1)(n+1)$ inv
$\mathsf{SecWriteCol}([T], [\mathbf{V}], [C''])$;             // 1 rnd, $n(m+1)$ inv
return $[T]$;
```

Finally, Protocol 3.8 updates the vectors S, U (representing the basis and co-basis variables respectively) to reflect the new basis/co-basis.

Protocol 3.8: $([S], [U]) \leftarrow \mathsf{UpdVar}([S], [U], [\mathbf{V}], [\mathbf{W}])$

```
Input: $m$-vectors $[S]$ and $[\mathbf{V}]$, $n$-vectors $[U]$ and $[\mathbf{W}]$
Output: $m$-vector $[S]$, $n$-vector $[U]$ (overwritten)
do parallel                                                            // 1 rnd
    $[s] \leftarrow \mathsf{SecRead}([S], [\mathbf{V}])$;                // 1 inv
    $[u] \leftarrow \mathsf{SecRead}([U], [\mathbf{W}])$;                // 1 inv
do parallel                                                            // 1 rnd
    $[S] \leftarrow \mathsf{SecWrite}([S], [\mathbf{V}], [u])$;          // $m$ inv
    $[U] \leftarrow \mathsf{SecWrite}([U], [\mathbf{W}], [s])$;          // $m$ inv
return $[S]$, $[U]$;
```

Correctness: All the building blocks we use are build using standard protocols in the literature. The correctness of most of them is trivial to verify. Due to lack of space, we do not prove correctness of the remaining protocols. Instead we refer the reader to the literature [9,8,10,12,1,3]. The missing details will be provided in a full version of this paper.

Security: Protocols LTZ and GTZ provide statistical security in some parameter κ, with an overhead of κ bits. For typical situations $\kappa \geq 32$ should be sufficient, which roughly amounts to $\leq 2^{-\kappa}$ probability of information leakage. The remaining protocols provide perfect security.

Complexity: Table 1 gives the complexity of the secure Simplex protocols of this section. The work of [12] is based on similar techniques. However, only asymptotic complexity bounds are presented there. Consequently, it is not possible to make a rigorous comparison between the two protocols. However, due to various improvements over the methods of [12], the actual complexity of our protocol is much better.

Protocol	**Rounds**	**Invocations**	**Exp.**
Null	1	1	0
GetPivCol	$2 + \log(k-1) + \log n$	$2kn + \frac{n}{2}\log n$	$n(k-1)$
GetPivRow	$\log(8m^5) + \log(k-1)(\log 2m)$	$6m(k+1)$	$2m(k-1)$
UpdTab	5	$4mn + 7m + 4n + 4$	0
UpdVar	2	$2m + 2$	0

Table 1. Complexity of operations in secure Simplex protocol.

4 Conclusion

We presented a summary of state-of-the-art in privacy-preserving building blocks that can be used to construct a secure Simplex protocol for solving linear programming problems in a restricted form (i.e., maximization problem in standard form). The protocols presented in this paper are adapted from the literature with several optimizations. As of now the only other secure protocol for simplex with a provable guarantee of security is proposed in [12]. The protocol presented here is more efficient than that of [12] because of the following optimizations: (1) use of a smaller tableau (i.e., $(m+1)\times(n+1)$ matrix instead of $(m+1)\times(m+n+1)$), (2) avoidance of Bland's rule, (3) use of a faster CompCons protocol, and (4) use of a faster comparison protocol (LTZ). A detailed description of this work will appear in a full version of this paper.

Similar to [12], we use the *integer-pivoting* (IP) variant of Simplex [7] which has the advantage of providing an exact solution. Furthermore, privacy-preserving building-blocks for secure IP simplex already exist in the literature. On the other hand, the problem with IP Simplex is that the values in the tableau increase very rapidly in size. Due to this, we must use quite a large value of q (e.g., 1024 bits for typical problems) which becomes a communication bottleneck. A theoretical upper-bound on the values is given in [12]. However, this bound is not tight. As further work, we would like to explore the use of fixed-point Simplex (instead of IP Simplex) in order to avoid this rapid increase. Another open problem is to find faster variants of protocols for generating secret random bits and comparison.

References

1. J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds of interaction. In *Proc. 8th annual ACM Symposium on Principles of distributed computing*, pages 201–209. ACM Press, 1989.
2. D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.
3. 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.
4. M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.
5. J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
6. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992.
7. 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, Department of Mathematics, 2005. www.cdam.lse.ac.uk/Reports/Files/cdam-2005-18.pdf.
8. SecureSCM. Protocol Description V1. Public Deliverable D3.1, SecureSCM project, Jan 2009.
9. SecureSCM. Secure Computation Models and Frameworks. Public Deliverable D9.1, SecureSCM project, July 2008.
10. SecureSCM. Analysis. Public Deliverable D9.2, SecureSCM project, July 2009.
11. A. Shamir. How to share a secret. In *Communications of the ACM, 22(11)*, pages 612–613, 1979.
12. T. Toft. *Primitives and Applications for Multi-party Computation*. PhD dissertation, University of Aarhus, Denmark, BRICS, Department of Computer Science, 2007.

A Simplex Algorithm

The Simplex method [6,7,12] to solve such a problem starts by introducing m dummy (or "slack") variables $\{x_{n+1}, x_{n+2}, \ldots, x_{n+m}\}$ and starting from an initial feasible solution over a set of m basic variables (initially the slack variables), iteratively improves it by swapping a basic and a co-basic variable until an optimum is reached (if it exists) or the problem is found to be unbounded. The solution is the set of m basic variables at the end of iterations. Any slack variables contained in the basis are ignored. For details see [6].

There are the following data structures: a $m \times n$ matrix A, a m vector B, a n vector F, a variable Z, a n vector U and a m vector S. See Fig. 1.

1. *Initialization:* For $1 \leq i \leq m$ and $1 \leq j \leq n$, set $A(i,j) \leftarrow a_{i,j}$, $F(j) \leftarrow -f_j$, $B(i) \leftarrow b_i$, $Z \leftarrow 0$, $U(j) \leftarrow j$, $S(i) \leftarrow n+i$.
2. *Repeat Forever:*
   (a) *Get Pivot Column:* Select $c \in [1..n]$ such that $F(c) < 0$. If no such c, report "Optimal Solution" and exit. If more options, choose at random or using Bland's rule (minimum $U(c)$).

$$\begin{array}{ccc}
\begin{pmatrix} A(1,1) & \dots & A(1,n) \\ \vdots & \ddots & \vdots \\ A(m,1) & \dots & A(m,n) \end{pmatrix} & \begin{pmatrix} B(1) \\ \vdots \\ B(m) \end{pmatrix} & \begin{pmatrix} S(1) \\ \vdots \\ S(m) \end{pmatrix} \\
\begin{pmatrix} F(1) & \dots & F(n) \end{pmatrix} & Z & \\
\begin{pmatrix} U(1) & \dots & U(n) \end{pmatrix} & &
\end{array}$$

Fig. 1. Global data structures in Simplex.

(b) *Get Pivot Row:* Select $r \in [1..m]$, such that $A(r,c) > 0$ and $|B(r)/A(r,c)|$ 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):* Set $T \leftarrow \begin{pmatrix} A & B \\ F & Z \end{pmatrix}$ then do the following:

$$\begin{array}{lll}
C(i) & \leftarrow T(i,c) & 1 \le i \le m+1 \\
R(j) & \leftarrow T(r,j) & 1 \le j \le n+1 \\
p & \leftarrow R(c) & \\
T(i,j) & \leftarrow T(i,j) \cdot p - C(i) \cdot R(j) & 1 \le i \le m+1, i \ne r,\ 1 \le j \le n+1 \\
T(i,j) & \leftarrow T(i,j)/p' & 1 \le i \le m+1, i \ne r,\ 1 \le j \le n+1 \\
C(i) & \leftarrow -C(i) & 1 \le i \le m+1, i \ne r \\
C(r) & \leftarrow p' & \\
T(i,c) & \leftarrow C(i) & 1 \le i \le m+1 \\
p' & \leftarrow p & \\
U(c) & \leftrightarrow S(r) & \text{(swap)}
\end{array}$$

Decompose T back into A, B, F, Z.

3. *Final solution:* For $1 \le i \le m$ variable $x_{S(i)}$ takes the value $B(i)/p'$. All other variables take the value 0. The objective function takes the value Z.

The above variant of Simplex is a combination of the variants described in [7,12] (integer-pivoting) and [6] (small-tableau). Due to the structure of the algorithm, the division by p' in Step 2(c) is guaranteed to produce integer values.

B Complexity of the Operations in SMC Framework

Tables 2 and 3 give the complexity of all building blocks.

Operation	Protocol	Rounds	Invocations	Exp.
$[c] \leftarrow [a] + [b]$		0	0	0
$[c] \leftarrow a + [b]$		0	0	0
$[c] \leftarrow a[b]$		0	0	0
$[c] \leftarrow \alpha[a] + \beta[b]$		0	0	0
$[c] \leftarrow [a][b]$	Mul	1	1	0
$[c] \leftarrow [\mathbf{a}] \cdot [\mathbf{b}]$	Inner	1	1	0
$a \leftarrow \mathsf{Reveal}([a])$	Reveal	1	1	0
$[c] \leftarrow [a] + [b] - [a][b]$	OR	1	1	0

Table 2. Complexity of basic SMC framework.

Protocol	Rounds	Invocations	Exp.
$\mathsf{RandFld}(\mathbb{F})$	0	0	0
$\mathsf{RandBit}(q)$	1	1	1
$\mathsf{RandInt}(q, m)$	0	0	0
$\mathsf{MulPub}([a], [b])$	1	1	0
$\mathsf{Inv}([a])$	1	1	0
kOp	$\alpha \log k$	$\beta(k-1)$	0
PreOp	$\alpha \log k$	$\beta(\frac{k}{2}\log(k))$	0
$\mathsf{BitLT}(a, [b]_B)$	$\log k$	$2k-2$	0
$\mathsf{LTZ}([a], k)$	$2 + \log(k-1)$	$3k$	$k-1$
$\mathsf{GTZ}([a], k)$	$2 + \log(k-1)$	$3k$	$k-1$
SecRead	1	1	0
SecWrite	1	m	0
SecReadRow	1	$n+1$	0
SecReadCol	1	$m+1$	0
SecWriteRow	1	$m(n+1)$	0
SecWriteCol	1	$n(m+1)$	0

Table 3. Complexity of building blocks.