Rev. Roum. Sci. Techn.– Électrotechn. et Énerg.
Vol. **64**, *4*, pp. 417–422, Bucarest, 2019

COMPARATIVE ANALYSIS OF SECURE POLYNOMIAL EVALUATION PROTOCOLS

OCTAVIAN CATRINA[1]

Key words: Cryptographic protocols, Secure multiparty computation, Secure floating-point arithmetic, Polynomial evaluation.

Secure computation provides cryptographic protocols that support collaborative applications with private inputs and outputs. We examine in this paper a collection of protocols that evaluate polynomials with private inputs and outputs using secure floating-point arithmetic. A first goal is to provides a comparative analysis of their tradeoffs and performance. Moreover, the analysis is an opportunity to discuss several general problems: the tradeoffs between complexity metrics, their effects on protocol performance, and the importance of the execution environment in the experimental evaluation of secure computation protocols.

1. INTRODUCTION

Secure computation is a branch of cryptography that enables groups of parties to run collaborative applications without having to reveal their private inputs (*e.g.*, for legal or business reasons). The parties run a distributed computation that protects data privacy using cryptographic techniques (*e.g.,* linear secret sharing or homomorphic encryption). A fundamental challenge of secure computation is to find solutions that meet the application requirements, despite the inherent computation and communication overhead of the cryptographic protocols.

The applications include statistical analysis, data mining, and optimization of business processes, which require secure computation with real numbers [1–5]. In this paper, we focus on the evaluation of polynomials with private inputs (variables and coefficients) and private outputs, using floating-point arithmetic. An important application of these protocols is the secure evaluation of elementary functions by polynomial approximation.

We start from a family of secure floating-point arithmetic protocols based on secret sharing introduced in [6]. Follow-up work presented optimized protocols for multi-operand multiplication [7] and addition [8], with examples of their use in polynomial evaluation.

Our main goal is to provide a joint comparative analysis of the performance and tradeoffs of this collection of polynomial evaluation protocols. On the other hand, this analysis is an opportunity to examine several general problems: the tradeoffs between complexity metrics, their effects on protocol performance, and the importance of the execution environment in the experimental evaluation.

The paper is structured as follows. Section 2 is an overview of the secure computation framework, data encoding, and floating-point arithmetic protocols. In Section 3, we examine the design, complexity, and tradeoffs of the polynomial evaluation protocols and in Section 4 we analyze their performance. The main results are summarized in Section 5.

2. SECURE ARITHMETIC

2.1 SECURE COMPUTATION FRAMEWORK

The protocols discussed in this paper are part of the secure computation framework described in [6–8], that relies on standard primitives for secure computation using Shamir secret sharing [9] and related techniques [10,11]. We provide in the following a brief overview of this approach to secure computation.

We assume $n > 2$ parties, P_i, $i\in[1, n]$, that communicate on secure channels and want to run a joint computation where party P_i has private input x_i and expects output y_i. The parties use a linear secret-sharing scheme to create a distributed state of the computation, where each party has a random share of each secret variable, $[x_i]$, $i\in[1, n]$. Then, they run secure computation protocols that compute the shared outputs $[y_i]$, $i\in[1, n]$, preserving data privacy throughout the computation. Finally, they deliver to every party P_i its output y_i using the secret reconstruction protocol of the secret sharing scheme.

The core primitives use Shamir secret sharing over a finite field F and provide secure arithmetic in F with perfect privacy in the semi-honest model (assuming ideal secure channels). In this model, the parties do not deviate from the protocol and any $t + 1$ parties can reconstruct a secret, while t or less parties cannot distinguish it from random values in F. In this paper, we focus on protocols that use the field of integers modulo a prime q, denoted Z_q. The parties locally compute addition/subtraction of shared field elements by adding/subtracting their own shares. Tasks that involve multiplication require interaction and are computed by dedicated protocols. These protocols offer either perfect privacy, or statistical privacy with security parameter κ (for ideal secure channels) [7].

Throughout the paper, the logarithms are taken to base 2. The notation $x = (condition)? \; a : b$ means that x is assigned the value a if *condition* = *true* and b otherwise.

2.2. DATA ENCODING

Secure computation with Boolean, integer, fixed-point, and floating-point data types is achieved by encoding the data in F and using secure arithmetic in F. The data types are defined as follows:

- Binary values are naturally encoded as 0_F and 1_F.
- Integer numbers $Z_{\langle k\rangle} = \{x \in Z \mid x \in [-2^{k-1}, 2^{k-1}-1]\}$, are encoded in Z_q by fld : $Z_{\langle k\rangle} \rightarrow Z_q$, $\mathrm{fld}(x) = x \bmod q$, for $q > 2^{k+\kappa}$, where κ is the security parameter ($x \in [-2^{k-1}, -1]$ are mapped to $x \in [q-2^{k-1}, q-1]$).
- Fixed-point numbers $Q'_{\langle k,f\rangle} = \{x \in Q \mid x = v2^{-f}, v \in Z_{\langle k\rangle}, f \le k\}$ are first mapped to $Z_{\langle k\rangle}$ by int : $Q_{\langle k,f\rangle} \rightarrow Z_{\langle k\rangle}$, $v = \mathrm{int}(x) = x2^f$, and then encoded in Z_q.
- The set of floating-point numbers, $Q_{\langle l,g\rangle}$, is defined by tuples $\langle v, p, s, z\rangle$, where $v \in [2^{l-1}, 2^l-1] \cup \{0\}$, is the unsigned, normalized l-bit significand, $p \in Z_{\langle g\rangle}$ is

[1] University Politehnica of Bucharest, Bucharest, Romania, E-mail: octavian.catrina@elcom.pub.ro.

the signed g-bit exponent, $s = (v < 0)?\ 1:\ 0$, and $z = (v = 0)?\ 1:\ 0$. These values are encoded in Z_q as described above. The number's value is $x = (1–2s)v2^p$. If $x = 0$ then $v = 0$, $z = 1$, and $p = –2^{g–1}$.

Secure arithmetic requires a modulus $q > 2^{2k+\kappa}$ for fixed-point numbers and $q > 2^{2l+\kappa}$ for floating-point numbers. However, some optimizations require a larger modulus [7].

2.3. COMPLEXITY AND PERFORMANCE

The parties run a synchronized distributed computation that alternates local computation and interactions. They run the same algorithm on their local set of variables and communicate when they reach an interactive operation: share a secret variable, multiply two shared variables, or reconstruct a shared variable.

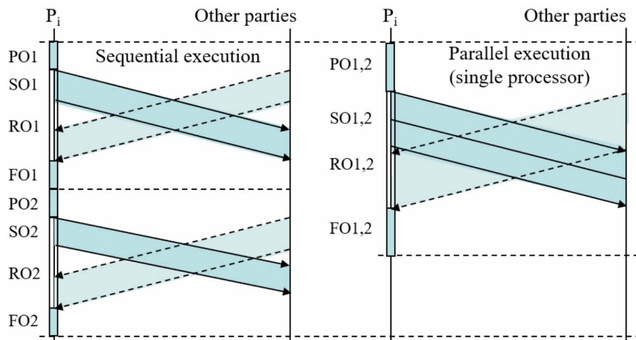

Fig. 1 – Interaction rounds during secure computation.

Figure 1 shows a simple task with 2 interactive primitives, executed either sequentially, or in parallel. Sequential execution proceeds as follows:

- PO1: Run locally until the computation reaches an interactive operation; prepare the data to be sent during this operation.
- SO1: Send data to the other parties (same amount).
- RO1: Receive data from the other parties.
- FO1: Process received data to finish the operation.
- PO2-FO2: Repeat the same steps for the next interactive operation.

The exchanged data depends only on the sender's local state, so sending and receiving can be done in parallel. The computation is synchronized at every interactive operation: all parties must receive all the expected data from the others before continuing the execution of the protocol.

We run a batch of interactive primitives in "parallel" by executing each of the steps listed above for the entire batch, so the computation is achieved with a single interaction (Fig. 1). The messages exchanged by the interactive primitives contain a very small payload: a single field element. This means 16–20 octets for secret integers encoded in Z_q. By parallel execution, we aggregate these tiny payloads in a single large message, to reduce packet encapsulation overhead and take advantage of efficient bulk data transfer methods. For large batches, local computation can be allocated to multiple threads, to benefit from multiple processors and/or cores. Thus, parallel algorithms are essential for fast and efficient secure computation.

The protocols' design assumes the following general optimizations. Interactive operations that do not depend on each other are executed in parallel, in a single round. Thus, all shared random values are precomputed in parallel. Some of them require interaction, while others are locally generated using techniques presented in [10,11].

We evaluate the protocols using complexity metrics that focus on interaction:

- Communication complexity measures the amount of data sent by each party. We use an abstract metric that counts the invocations of 3 primitives that send a share from each party to the others: input sharing, multiplication, and secret reconstruction.
- Round complexity is the number of sequential invocations of (batches of) interactive primitives and emphasizes the effects of network delay components that are independent of the amount of data.

2.4. FLOATING-POINT ARITHMETIC PROTOCOLS

The floating-point protocols used for polynomial evaluation are listed in Table 1 [7,8]. Let a_1, a_2, $a \in Q_{\langle l,g \rangle}$, $a_1 = (1–2s_1)v_12^{p_1}$, $a_2 = (1–2s_2)v_22^{p_2}$, $a = (1–2s)v2^p$.

The protocol AddFL computes $a = a_1+a_2$ (or $a = a_1–a_2$) with secret inputs and output. Let $v'_1 = (1–2s_1)v_1$ and $v'_2 = (1–2s_2)v_2$. Assume $p_1 \geq p_2$ and let $\Delta = p_1–p_2$ (if $p_1 < p_2$ the inputs are swapped). We want v' and p' so that $v'\,2^{p'} \approx v'_12^{p_1} + v'_22^{p_2}$. The protocol consists of 2 main sub-tasks:

- Align the radix point and add the significands: set $p' = p_1$ and $v' = v'_1 + \lfloor v'_2 / 2^{\Delta} \rfloor \in [-2^l+1, 2^{l+1}–2]$.
- Normalize v' and p' to obtain $a \in Q_{\langle l,g \rangle}$.

Both sub-tasks are expensive, so secure floating-point addition and subtraction are much more complex than multiplication and division. The normalization needs 9 rounds (out of 16) and $5l$ online interactive operations (out of $6l+3g+29$). The precomputation is also expensive.

MulFL computes $a = a_1a_2$ with secret inputs and output. It starts by computing $v_3 = v_1v_2$, $p_3 = p_1 + p_2$, $s = s_1 \oplus s_2$, and $z = z_1 \vee z_2$, and then normalizes the result. Since $v_3 \in [2^{2l–2}, 2^{2l}–2^{l+1}+1] \cup \{0\}$, the normalization is much simpler than for addition: if $v_3 < 2^{2l–1}$ then $v = \lfloor v_3 / 2^{l–1} \rfloor$ and $p = p_3 + l –1$, otherwise $v = \lfloor v_3 / 2^l \rfloor$ and $p = p_3 + l$. Nevertheless, the complexity is dominated by normalization: 4 rounds (out of 5) and l+4 online interactive operations (out of l+7).

The applications also need multi-operand arithmetic: compute $a = \sum_{i=0}^{m-1} a_i$ and $a = \prod_{i=0}^{m-1} a_i$ for secret $a_i \in Q_{\langle l,g \rangle}$, $a_i = (1–2s_i)v_i2^{p_i}$, $i \in [0, m–1]$, and $a \in Q_{\langle l,g \rangle}$, $a = (1–2s)v2^p$. Since sequential computation is inefficient, we need algorithms that execute many operations in parallel.

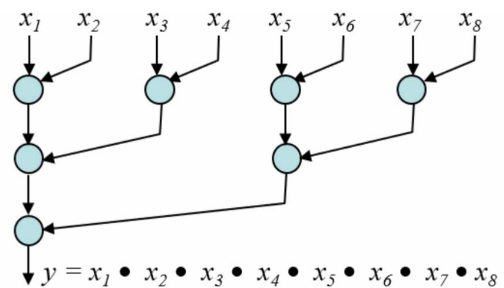

Fig. 2 – Generic construction for multi-operand arithmetic.

The traditional approach uses AddFL and MulFL in the generic algorithm shown in Fig. 2. The algorithm computes $m–1$ operations (optimal) in $\lceil \log(m) \rceil$ iterations. Each iteration takes as input a vector, splits it into pairs of elements, adds or multiplies them in parallel, and returns the results in a vector of half length.

The protocols SumGFL and ProdGFL compute sums and products using this generic parallel algorithm. However, both are relatively expensive (Table 1). We can obtain more efficient solutions based on the complexity analysis of AddFL and MulFL, outlined above.

Table 1

Complexity of arithmetic protocols for floating-point numbers with l-bit significand and g-bit exponent (selection).

Protocol	Task	Rounds	Interactive operations (online)	Modulus
AddFL	$a_1 + a_2$	16	$6l + 3g + 29$ (precomputation: $13l + 9g$)	$q > 2^{l+\kappa}$
MulFL	$a_1 \cdot a_2$	5	$l + 7$ (precomputation: $4l + 3$)	$q > 2^{2l+\kappa}$
SumGFL	$\sum_{i=0}^{m-1} a_i$	$16\lceil\log(m)\rceil$	$6(m-1)l + (m-1)(3g+26)$	$q > 2^{l+\kappa}$
SumFL	$\sum_{i=0}^{m-1} a_i$	$4\lceil\log(m)\rceil + 15$	$(m+5)l + m(2g+8)$	$q > 2^{l+\kappa+\lceil\log(m)\rceil}$
ProdGFL	$\prod_{i=0}^{m-1} a_i$	$5\lceil\log(m)\rceil$	$(m-1)l + 6(m-1)$	$q > 2^{2l+\kappa}$
ProdFL	$\prod_{i=0}^{m-1} a_i$	$\lceil\log(m)\rceil + 6$	$2l + 7(m-1)$	$q > 2^{2l+\kappa+m-2}$
PowAllFL	x^i, $i \in [1, m]$	$5\lceil\log(m)\rceil$	$(m-1)l + 5(m-1)$	$q > 2^{2l+\kappa}$

Table 2

Complexity of polynomial evaluation protocols for floating-point numbers with l-bit significand and g-bit exponent.

Protocol	Task	Rounds	Interactive operations (online)	Modulus
PolyEFL	$\sum_{i=0}^{m} a_i x^i$	$18\lceil\log(m+1)\rceil$	$(6m + \lceil\log(m+1)\rceil)l + m(3g+39) + 4\lceil\log(m+1)\rceil$	$q > 2^{2l+\kappa+2}$
PolySGFL	$\sum_{i=0}^{m} a_i x^i$	$21\lceil\log(m)\rceil + 5$	$8ml + m(3g+41)$	$q > 2^{2l+\kappa}$
PolySFL	$\sum_{i=0}^{m} a_i x^i$	$9\lceil\log(m)\rceil + 16$	$(2m+4)l + m(2g+16)$	$q > 2^{2l+\kappa+\lceil\log(m)\rceil}$
PolySOFL	$\sum_{i=0}^{m} a_i x^i$	$4\lceil\log(m)\rceil + 17$	$(m+5)l + m^2 + m(2g+18)$	$q > 2^{2l+\kappa+m}$
PolyPFL	$\prod_{i=1}^{m} (x + \alpha_i)$	$\lceil\log(m)\rceil + 22$	$(6m+2)l + m(3g+36)$	$q > 2^{2l+\kappa+m-2}$

The protocol SumFL uses a more efficient algorithm for multi-operand addition. We need v, p, and s so that $(1-2s)2^p = \sum_{i=0}^{m-1}(1-2s_i)v_i 2^{p_i}$. Let $v'_i = (1-2s_i)v_i$, $p' = \max(\{p_i\})$, and $\Delta_i = p' - p_i$. SumFL uses the following algorithm [8]:

- Align the radix point and add the significands: Compute p' using the protocol MaxInt and then $v' = \sum_{i=0}^{m-1}\lfloor v'_i / 2^{\Delta_i}\rfloor$ by generalizing the method in AddFL. The maximum bit-length of v' is $l + \lceil\log(m)\rceil$.
- Normalize v' and p' to obtain $a \in Q_{\langle l,g\rangle}$, as in the protocol AddFL.

MaxInt computes $m-1$ secure comparisons in $\lceil\log(m)\rceil$ iterations, using the generic algorithm in Fig. 2. Each iteration takes as input a vector, splits it into pairs of elements, selects in parallel the larger value in each pair, and returns the results in a vector of half length.

SumFL reduces round complexity and communication complexity by simplifying the radix alignment and normalizing only once (Table 1). Since $q > 2^{2l+\kappa}$ (for multiplication), the growth of the maximum integer bitlength by $\lceil\log(m)\rceil$ bits does not require larger q.

The protocol ProdFL computes products more efficiently than ProdGFL [7]. It is based on the same algorithm as ProdGFL (Fig. 2), but simplifies the multiplication of the significands and normalizes only once. Instead of $a_j = a_{2j-1}a_{2j}$, the iterations compute $v_j = \lfloor v_{2j-1}v_{2j} / 2^{l-1}\rfloor$, $v_j \in [2^{l-1}, 2^{l+1} - 2^2] \cup \{0\}$ and $p_j = p_{2j-1} + p_{2j} - (l-1)$. The simplified multiplication is computed in a single round (instead of 5), but increases the bitlength of v' by $l' \in [0, m-1]$ bits. Larger integer bitlength means more complex normalization and larger modulus. The normalization takes on input $v' \in [2^{l-1}, 2^{l+l'}-1] \cup \{0\}$, so it is more complex than in MulFL (larger range), but simpler than in SumFL (smaller range). The maximum integer bitlength is $l'' = 2l + m - 2$ (in the last iteration, before truncation), so the protocol needs $q > 2^{2l+\kappa+m-2}$. However, for practical values of m, ProdFL is more efficient than ProdGFL (Table 1).

We conclude this section with the protocol PowAllFL, which plays an important role in secure polynomial evaluation [7]. PowAllFL computes all powers x^i, $i \in [1, m]$, with secret input and outputs, using MulFL and the generic algorithm shown in Fig. 3. The algorithm computes $m-1$ floating-point multiplications, which is optimal, in $\lceil\log(m)\rceil$ iterations. A variant with simplified multiplications, similar to ProdFL, is inefficient, since it needs $m-1$ normalizations in the end, with input bitlengths $l + l'$, $l' \in [0, m-1]$.

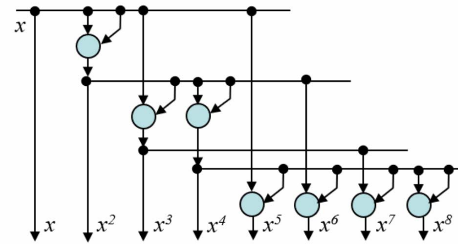

Fig. 3 – Generic construction for computing all powers.

3. SECURE POLYNOMIAL EVALUATION

Our final goal is to evaluate a polynomial with secret inputs (variables and coefficients) and secret outputs: $y = P(x) = \sum_{i=0}^{m} a_i x^i$, $a_i \in Q_{\langle l,g\rangle}$, $a_i = (1-2s_i)v_i 2^{p_i}$, $i \in [0, m-1]$, and $x, y \in Q_{\langle l,g\rangle}$, $x = (1-2s)v\,2^p$, $y = (1-2s')v'\,2^{p'}$.

The design of polynomial evaluation protocols must take into account the specific features of secure floating-point arithmetic: addition is much slower than multiplication (for $m \geq 2$ operands); normalization is expensive; performance is substantially improved by parallel execution of interactive operations. Note that traditional algorithms are optimized according to other assumptions. For example,

Horner's algorithm is sequential [13], so it is not suitable for secure computation. We examine in the following several protocols that offer different tradeoffs.

PolySGFL is a generic polynomial evaluation protocol that uses the generic arithmetic protocols discussed in Section 2: it computes $x_i = x^i = \langle v'_i, p'_i, s'_i, z'_i\rangle$ using PowAllFL and $y_i = a_i x_i = \langle v''_i, p''_i, s''_i, z''_i\rangle$ using MulFL, for $i\in[1, m]$, then $y = \sum_{i=0}^{m} y_i$, with $y_0 = a_0$. This solution is simple, but expensive (Table 2).

Estrin's parallel algorithm [13] computes $m-1$ multiply-and-add operations in $\lceil\log(m+1)\rceil$ iterations, structured as in Fig. 2, and an additional square operation per iteration. A protocol based on this algorithm, PolyEFL, is similar to SumGFL, but the iterations run a multiply-and-add protocol instead of AddFL [7]. However, its complexity is dominated by additions (it is slightly more complex than SumGFL) and the improvement is rather modest (Table 2).

Alternatively, we can optimize PolySGFL as shown in Protocol P1, PolySFL. Step 1 computes the significands and exponents of $x_i = x^i$, $i \in [1, m]$ with Protocol P2, PowAllFLS. The protocols PowAllFLS and MulFLS are variants that skip the computation of s and z; they are computed in parallel in steps 2–4, in a single round. Steps 5-7 compute $y_i = a_i x_i$, $i\in[1, m]$, without normalization: $v''_i = v_i v'_i / 2^{l-1} \in [2^{l-1}, 2^{l+1}-4] \cup \{0\}$; if $z''_i = 0$ then $p''_i = p'_i + p_i + l - 1$ else $p''_i = -2^{g-1}$. Steps 8 and 9 compute $y = \sum_{i=0}^{m} y_i$, using a slightly modified variant of SumFL, with input significands of $l+1$ bits and output significands of l bits.

P1: PolySFL([v], [p], [s], [z], $\{[v_i], [p_i], [s_i], [z_i]\}_{i=0}^{m}$, l, g)

1 $\{[v'_i], [p'_i]\}_{i=1}^{m}$ ← PowAllFLS([v], [p], m, l, g);
2 foreach $i\in[1, m]$ do in parallel
3 $[s''_i] \leftarrow (i \bmod 2 = 1)?\ [s_i] : [s] \oplus [s_i]$;
4 $[z''_i] \leftarrow [z] \vee [z_i]$;
5 foreach $i\in[1, m]$ do in parallel
6 $[v''_i]$ ← Div2mPD($[v_i]*[v'_i]$, $2l$, $l-1$);
7 $[p''_i] \leftarrow ([p_i] + [p'_i] + l-1)(1- [z''_i]) - [z''_i]2^{g-1}$;
8 $([v''_0], [p''_0], [s''_0]) \leftarrow ([v_0], [p_0], [s_0])$;
9 $([v'], [p'], [s'], [z']) \leftarrow$ SumFL($\{[v''_i], [p''_i], [s''_i]\}_{i=0}^{m}$, $l+1$, l, g);
10 return $([v'], [p'], [s'], [z'])$;

P2: PowAllFLS ([v], [p], m, l, g)

1 $([v_1], [p_1]) \leftarrow ([v], [p])$; $\alpha \leftarrow \lceil\log(m)\rceil$;
2 foreach $i\in[1, \alpha]$ do
3 $\gamma \leftarrow 2^{i-1}$; $\beta \leftarrow (i < \alpha)?\ \gamma : m - \gamma$;
4 foreach $j\in[1, \beta]$ do
6 $([v_{\gamma+j}], [p_{\gamma+j}])$ ← MulFLS($[v_\gamma], [p_\gamma], [v_j], [p_j]$, l, g);
7 return $\{[v_i], [p_i]\}_{i=1}^{m}$;

PolySFL achieves an important reduction of the round and communication complexity with respect to both PolySGFL and PolyEFL (Table 2). We can try to further simplify it by separating the computation of v''_i and p''_i:

- Compute p''_i, $i\in[1, m]$: if $z''_i = 0$ then $p''_i = p_i + ip + l - 1$, else $p''_i = -2^{g-1}$. Set $p''_0 = p_0$.
- Compute in parallel $p' = \max(\{p_i\})$ using MaxInt (for SumFL) and v'_i, $i\in[1, m]$, without normalization, using a simpler variant of PowAllFLS.
- Finalize the polynomial evaluation as in PolySFL.

For this variant, called PolySOFL, the tradeoff is more complicated (Table 2): it further reduces the number of rounds, but the maximum integer bitlength becomes $2l + m$. Larger integers means larger modulus and higher communication complexity for the computation of the sum. For $m > l$, PolySOFL needs more interactive operations than PolySFL, with a larger modulus. This may cancel the benefits of the lower number of rounds.

Assume now that the roots of $P(x)$ are the real numbers α_i, $i\in[1, m]$, and the application can precompute them (a similar approach works for complex roots). Our task can be reformulated as the evaluation of $P(x) = \prod_{i=1}^{m}(x + \alpha_i)$. This is shown in Protocol P3, PolyPFL [7]. By computing the factors in parallel and then using ProdFL, this variant offers the lowest round complexity (Table 2). However, the communication complexity, dominated by AddFL, is quite large, and ProdFL requires a larger modulus.

P3: PolyPFL([v], [p], [s], [z], $\{[v_i], [p_i], [s_i], [z_i]\}_{i=0}^{m}$, l, g)

1 foreach $i\in[1, m]$ do in parallel
2 $([v'_i], [p'_i], [s'_i], [z'_i]) \leftarrow$ AddFL([v], [p], [s], [z], $[v_i], [p_i], [s_i], [z_i]$, l, g);
3 $([v'], [p'], [s'], [z']) \leftarrow$ ProdFL($\{[v_i], [p_i], [s_i], [z_i]\}_{i=1}^{m}$, l, g);
4 return $([v'], [p'], [s'], [z'])$;

4. EXPERIMENTAL RESULTS

The protocols run a distributed, synchronized computation, with batches of interactive operations executed in parallel. Therefore, their running time depends on available processing and communication resources, as well as implementation optimizations aimed at using these resources more efficiently (especially for large batches).

The complexity listed in Table 2 shows that PolySFL, PolySOFL, and PolyPFL are more efficient than PolySGFL and PolyEFL, so they should be much faster. However, the optimizations offer different tradeoffs between complexity metrics, so it is not obvious which variant is faster in different execution environments. Moreover, some optimizations move part of the workload from online interaction to local computation and precomputation. This approach is justified by several observations:

- The protocols combine local computation and interaction over the Internet. The parties can allocate endpoint resources based on application requirements and budget constraints, but have limited control over network performance. As technology evolves, the cost of improving endpoint performance declines faster.
- Precomputation (local and interactive operations) can be efficiently executed in parallel for all the sub-tasks of a complex task, in a single interaction round.

The effects of local computation and precomputation on the total running time are significant and have to be taken into account in the evaluation of the protocols, as well as in the design of their implementations.

We studied the behavior of the polynomial evaluation protocols using our Java implementation of the secure computation framework. The aim of this implementation is to provide early feedback during protocol design, including correctness and accuracy tests, verification of complexity metrics (rounds and amount of data), and comparative analysis of protocol variants.

To facilitate protocol development and analysis, the implementation is single-threaded (except that data sending and receiving run in parallel on different threads).

We evaluated PolySGFL, PolySFL, PolySOFL, and PolyPFL (abbreviated SG, S, SO, and P, respectively) for 3 parties, polynomials of degree $m \in \{4, 8, 16, 32\}$, and floating-point numbers with l = 32 bits and g = 10 bits, encoded in a field with $\lceil \log(q) \rceil$ = 128 bits. For degree m = 32, PolySOFL and PolyPFL need $\lceil \log(q) \rceil$ = 160 bits.

Table 3 shows the complexity of the implementations for the configuration parameters used in the tests: the number of online rounds and the amount of data sent by party P_i to party P_j during the online and precomputation phases. The number of rounds is slightly larger than the minimum allowed by protocol design due to implementation tradeoffs between modularity and round optimization. The minimum round complexity is specified between brackets; note also that the formulas in Table 2 are slightly simplified.

Table 3

Complexity of the implementation of polynomial evaluation protocols.

m	Protocol	Online Rounds	Online Octets	Precomputation Octets
4	PolySGFL	69 (63)	19998	41298
	PolySFL	41 (38)	9449	22935
	PolySOFL	33 (29)	8176	19662
	PolyPFL	28 (24)	17108	32167
8	PolySGFL	92 (86)	40587	84375
	PolySFL	50 (47)	15889	40875
	PolySOFL	38 (33)	13452	34600
	PolyPFL	30 (25)	33286	63104
16	PolySGFL	115 (105)	81783	170500
	PolySFL	59 (56)	28698	76607
	PolySOFL	43 (37)	25546	68685
	PolyPFL	32 (26)	65658	124952
32	PolySGFL	138 (126)	164143	342724
	PolySFL	68 (65)	54227	147948
	PolySOFL	48 (41)	69842	177384
	PolyPFL	34 (27)	162999	290607

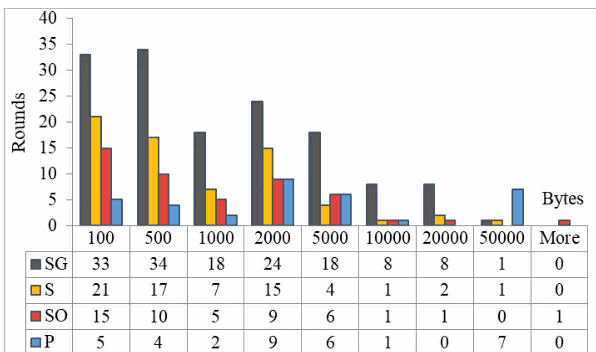

Fig. 4 – Frequency distribution of the amount of data sent in a round.

Figure 4 shows the frequency distribution of the amount of data sent in a round for a polynomial of degree m = 32. The amount of data varies between tens of octets and tens of thousands of octets (depending on the number of interactive primitives run in parallel and the size of the field elements). Observe that many rounds send less than a typical full packet payload (1460 octets) and only few rounds need tens of packets. Combined, the two abstract complexity metrics provide good performance predictions, but the workload per round varies a lot. Therefore, we need experimental analysis to understand the effects of different tradeoffs.

The protocols' running time was measured in a testbed consisting of 3 computers with 3.6 GHz processors (a computer for each party) connected to an Ethernet local area network (LAN). The computers run the operating system Linux 18.04. Longer Internet delays were emulated using the Linux network emulation tool NetEm, which increased the one-way transfer delay by 10 milliseconds.

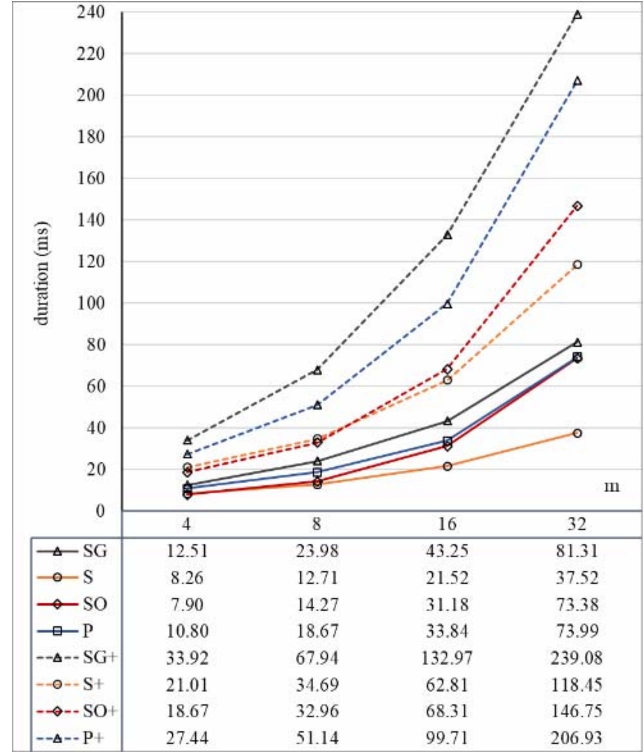

Fig. 5 – Online and total running-time for 1 Gbps Ethernet LAN.

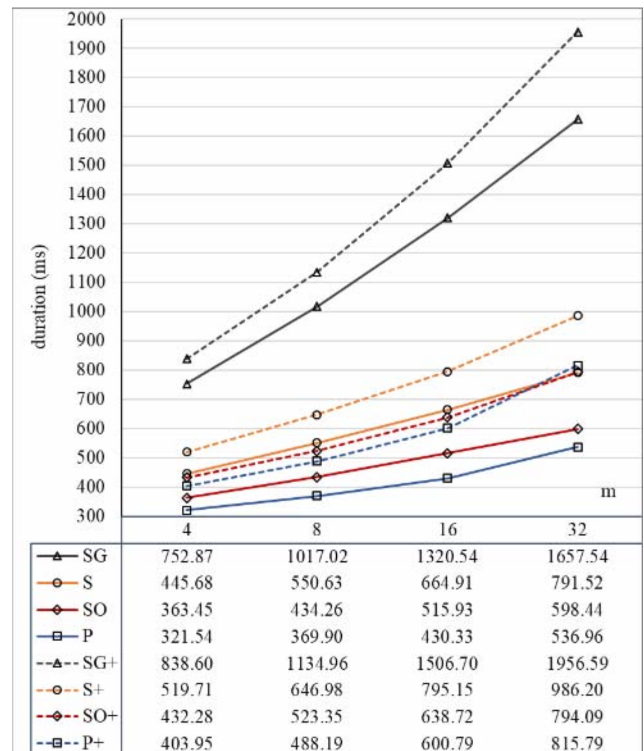

Fig. 6 – Online and total running-time for simulated Internet.

Figure 5 shows the protocols' running time for parties connected to an Ethernet LAN with 1 Gbps data rate and 0.35 milliseconds round-trip time. Figure 6 shows the running time for emulated Internet paths, with 100 Mbps end-to-end data rate and 20.8 milliseconds round-trip time. Solid lines show the online running time. Dotted lines show the total time, including the precomputation.

The results show that all the optimized variants are clearly faster than PolySGFL, for both network types. Moreover, their performance gain grows significantly for slower networks. This shows that the optimizations are effective and confirms the predictions of the complexity metrics (combined effects of reducing both metrics).

The running time and ranking of the optimized variants depend on network performance and polynomial degree:

- In LAN tests (Fig. 5), the near-zero network delay strongly attenuates the benefits of low round complexity. The results suggest that PolySFL offers the best tradeoff, with good performance over a larger range of m.
- For simulated Internet (Fig. 6), the results of the experiments emphasize the importance of lower round complexity: PolySOFL and PolyPFL are faster than PolySFL, even for larger m.
- For $m = 32$, the running time of PolySOFL and PolyPFL grows faster due to the larger integer and modulus bitlength needed by ProdFL.

The single-thread implementation does a poor job handling large batches of operations. This affects the online computation and, especially, the precomputation, for both network types. The inefficiency is better observed in LAN tests: most of the precomputation time was spent on local computation (the total load of the quad-core processor was about 20 % and the average data transfer rate was less than 50 Mbps). The running time can be substantially reduced by optimizing batch processing, so that the implementation distributes the workload on multiple processor cores and transfers the data more efficiently on network paths with high bandwidth-delay product.

5. CONCLUSIONS

We examined a family of secure computation protocols that evaluate polynomials with secret floating-point inputs and output [7,8], to provide a comparative analysis of their construction, complexity, and running time.

These protocols rely on the same secure computation model, based on Shamir secret sharing, and the same collection of building blocks, but offer different tradeoffs between round and communication complexity.

We observed that the abstract complexity metrics offer useful performance predictions, but we need experimental analysis to understand the effects of the tradeoffs.

The performance of secure computation protocols is usually reported for lab environments with near-zero network delay and highest data rate (*e.g.*, 1 or 10 Gbps).

However, these ideal conditions are not available (or affordable) in practical deployments of secure computation, when mutually distrustful organizations communicate via the Internet. Experiments in fast LANs provide useful information about implementation optimizations and the protocols' best possible running time. However, in this environment, the communication delay almost vanishes, so the experiments offer rather poor guidance for protocol optimizations, comparison of different solutions or tradeoffs, and practical protocol performance.

For the polynomial evaluation protocols analyzed in this paper, LAN tests show a clear advantage of the optimized variants over PolySGFL, but attenuate the advantage of PolySOFL and PolyPFL, suggesting that PolySFL is better. Experiments with a more realistic communication delay show higher performance gains for the optimized variants and reverse their ranking, emphasizing the importance of round-efficient solutions.

The optimized variants have their own advantages and disadvantages and choosing the most suitable variant depends on specific application requirements (*e.g.*, PolySFL scales up better with m, PolyPFL requires the roots of the polynomial) and execution environment (e.g., PolyPFL is faster for Internet communications). On the other hand, optimizing the implementation of large batches is essential for improving the overall performance.

Received on October 15, 2019

REFERENCES

1. D. Bogdanov, L. Kamm, S. Laur, V. Sokk, *Rmind: A Tool for Cryptographically Secure Statistical Analysis*, IEEE Transactions on Dependable and Secure Computing, **15**, *3*, pp. 481–495 (2018).
2. M. Aliasgari, M. Blanton, F. Bayatbabolghani, *Secure Computation of Hidden Markov Models and Secure Floating-point Arithmetic in the Malicious Model*, International Journal of Information Security, **16**, *6*, pp. 577–601 (2017).
3. L. Kamm, J. Willemson, *Secure Floating Point Arithmetic and Private Satellite Collision Analysis*, International Journal of Information Security, **14**, *6*, pp. 531–548 (2015).
4. D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson, *High-performance Secure Multi-party Computation for Data Mining Applications*, International Journal of Information Security, **11**, *6*, pp. 403–418 (2012).
5. F. Kerschbaum, A. Schröpfer, A. Zilli, R. Pibernik, O. Catrina, S. de Hoogh, B. Schoenmakers, S. Cimato, E. Damiani, *Secure Collaborative Supply-Chain Management*, IEEE Computer, **44**, *9*, pp. 38–43 (2011).
6. O. Catrina, *Efficient Secure Floating-point Arithmetic Using Shamir Secret Sharing*, In: 16th International Joint Conference on e-Business and Telecommunications - **2**: SECRYPT (Security and Cryptography), SciTePress, 2019, pp. 49–60.
7. O. Catrina, *Optimization and Tradeoffs in Secure Floating-Point Computation: Products, Powers, and Polynomials*, In: Proc. of the 6th Conference on the Engineering of Computer-Based Systems (ECBS'19), ACM, 2019, pp. 7:1–7:10.
8. O. Catrina, *Optimizing Secure Floating-point Arithmetic: Sums, Dot Products, and Polynomials*, In: Proc. of the 5th Romanian Cryptology Days Conference (RCD-2019), 2019.
9. R. Cramer, I. Damgård, J. B. Nielsen, *Secure Multiparty Computation and Secret Sharing*, Cambridge University Press, 2015.
10. R. Cramer, I. Damgård, Y. Ishai, *Share Conversion, Pseudorandom Secret-sharing and Applications to Secure Computation*, In: Theory of Cryptography (TCC'05), Lecture Notes in Computer Science, **3378**, Springer, 2005, pp. 342–362.
11. I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, T. Toft, U*nconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation*, In: Theory of Cryptography (TCC 2006), Lecture Notes in Computer Science, **3876**, Springer, 2006, pp. 285–304.
12. I. Damgård, R. Thorbek, *Non-interactive Proofs for Integer Multiplication*, In: Advances in Cryptology - EUROCRYPT 2007, Lecture Notes in Computer Science, **4515**. Springer, 2007, pp. 412–429.
13. D. E. Knuth, *The Art of Computer Programming*, **2** (3rd Ed.): Seminumerical Algorithms, Addison-Wesley, 1997.