Optimizing MPC for robust and scalable integer and floating-point arithmetic

Liisi Kerik[1], Peeter Laud[1], and Jaak Randmets[1,2]

[1] Cybernetica AS, Tartu, Estonia
[2] University of Tartu, Tartu, Estonia
{liisi.kerik, peeter.laud, jaak.randmets}@cyber.ee

Abstract. Secure multiparty computation (SMC) is a rapidly maturing field, but its number of practical applications so far has been small. Most existing applications have been run on small data volumes with the exception of a recent study processing tens of millions of education and tax records. For practical usability, SMC frameworks must be able to work with large collections of data and perform reliably under such conditions. In this work we demonstrate that with the help of our recently developed tools and some optimizations, the Sharemind secure computation framework is capable of executing tens of millions integer operations or hundreds of thousands floating-point operations per second. We also demonstrate robustness in handling a billion integer inputs and a million floating-point inputs in parallel. Such capabilities are absolutely necessary for real world deployments.

Keywords: Secure Multiparty Computation, Floating-point operations, Protocol design

1 Introduction

Secure multiparty computation (SMC) [19] allows a group of mutually distrusting entities to perform computations on data private to various members of the group, without others learning anything about that data or about the intermediate values in the computation. Theory-wise, the field is quite mature; there exist several techniques to achieve privacy and correctness of any computation [19, 28, 15, 16, 21], and the asymptotic overheads of these techniques are known. In practical terms, the search for best implementations and deployment strategies for performing computations on real-world scale is still ongoing. There exist several SMC platforms [4, 17, 9, 13, 20, 31, 35, 29] and independent implementations of SMC protocols for complex computational tasks [11, 25] looking for the right trade-offs.

Sharemind [9, 10] is one of the most mature SMC platforms and the base of some of the largest SMC deployments until now. With the help of Sharemind, we have performed statistical analyses over tens of millions of records [22, Chap. 6], and searched for anomalies in a set of 100 million records [5]. Sharemind achieves the versatility and scalability through a simple security model

(enabling efficient protocols) and a large set of composable protocols for primitive operations, which can be used as building blocks for large applications. The total number of implemented primitive protocols for integer, fixed- and floating-point operations for arguments of various sizes is significantly over 100. While historically the protocols have been implemented in C++, with more complex protocols invoking simpler ones in hierarchic manner, recently we have introduced a domain-specific language (the Protocol DSL) for specifying them [27]. The Protocol DSL brings at least two benefits. First, it allows tighter composition of protocols, enabling subprotocols with data dependencies to run in parallel without any additional effort from the developer of the protocol set. Second, it allows the developer to try out different implementation options for complex protocols with an effort that is orders of magnitude smaller compared to using C++.

In this paper, we report on our optimizations for the protocols in Sharemind's protocol set, enabled by the Protocol DSL. Many of the improved protocols are used for operations on private floating-point numbers. Our reported optimizations may be useful for other SMC platforms and protocol sets providing private floating-point numbers, as several of our optimizations are not that dependent on particular details of Sharemind. In addition to optimizations of private floating-point operations, we also show how the protocol construction toolchain, central to which is the Protocol DSL, allowed us to implement a major architectural change of all protocols with relatively little effort. This provides additional validation of the choices made in [27].

This paper has the following structure. In Sec. 2 we give an overview of Sharemind and the protocols it uses, as well as the related work on privacy-preserving floating-point operations. In Sec. 3 we describe our improvements to various floating-point protocols, both generic changes and modifications of specific protocols, as well as the constructions of protocols for new operations. In Sec. 4 we describe another optimization that applies to all protocols in the main protocol set of Sharemind. We show that the optimizations in this and previous section improve the performance of protocols for various operations. In Sec. 5 we give a more thorough description on how we have measured the performance of the protocols of Sharemind. We provide precise running times of certain protocols, thereby making clear the current state of the art. Finally, we conclude in Sec. 6.

2 Background

In a Sharemind deployment, the involved parties are divided into three classes which may overlap: the *input parties* provide inputs to the private computation, the *computation parties* execute the SMC protocols for performing operations with private data, and *result parties* learn the result(s) of the computation [6]. While the architecture of Sharemind supports the use of several SMC protocol sets [8], the main set in use is based on additively sharing the private values among three computing parties [10]. The sharing can be over any finite ring

and there are protocols to convert between different rings. Hence the input parties secret share their inputs among computation parties, and the result parties recombine the shares of outputs they receive from computation parties. The computation parties follow the description of the private functionality specified in the SecreC language [8], invoking the SMC protocols in specified order.

Sharemind's protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [7]. The development of secure protocols is also greatly assisted by a protocol privacy checker [32] for the Protocol DSL [27].

Typically, rings $\mathbb{Z}_{2^n}$ are used in Sharemind applications and supported by the Protocol DSL. In the following, we let $[\![x]\!]$ denote the value x which has been secret-shared among the computing parties, and $[\![x]\!]_i$ denotes the i-th party's share.

For private numeric computations (e.g. for the satellite collision analysis [23]), Sharemind features a set of protocols for working with secret-shared fixed-point and floating-point numbers [23, 26]. In this protocol set, a floating-point number x is represented as $x = (-1)^s \cdot f \cdot 2^e$, where $s \in \{0, 1\}$ is the sign bit, $f \in \mathbb{Z}_{2^m}$ the significand, and $e \in \mathbb{Z}_{2^n}$ the exponent. The representation with $(m, n) = (32, 8)$ [resp. $(m, n) = (64, 11)$] is called *single precision* [resp. *double precision*]. For a private value, each part is separately secret-shared among the computing parties. The same representation (plus an indication whether the number is 0) is used also by Aliasgari et. al [3] who have built a private floating-point protocol set implementing arithmetic operations and a number of elementary functions on top of Shamir's threshold secret sharing [34]. In a different line of work, protocols for private floating-point operations have been built atop garbled circuits or the GMW protocol set [30, 18] with various optimizations.

Internally, many of our floating-point protocols call protocols for computations on private *fixed-point* numbers. In our protocols, a fixed-point number x is represented as an integer $x \cdot 2^M$ for a suitable M. Several sets of SMC protocols for fixed-point computations (including both arithmetic operations and elementary functions) have been proposed [14, 26]. Our Protocol DSL has allowed us to experiment with the details of these protocols and propose more efficient implementations.

3 Improvements in protocol design

In our floating-point protocols, we use the following operations as primitive building blocks:

- Zero-extension of secret shared integers denoted with `Extend(`$[\![u]\!]$`, n)` where $[\![u]\!] \in \mathbb{Z}_{2^m}$. This operation converts a private integer from $\mathbb{Z}_{2^m}$ to $\mathbb{Z}_{2^{n+m}}$ without changing its value.
- Dropping some least-significant bits of a secret shared integer, denoted with `Cut(`$[\![u]\!]$`, n)` where $[\![u]\!] \in \mathbb{Z}_{2^m}$ and $n \leq m$. The cut operation removes n

Algorithm 1: Protocol `PowArr` for integer powers of a fixed-point number.

Data: $[\![\widehat{x}]\!], k, n, n'$

Result: Computes the powers of a secret fixed-point number. Takes in a secret fixed-point number $[\![\widehat{x}]\!]$ with 0 bits before and n bits after the radix point. Outputs a secret fixed-point array $\{[\![\widetilde{x^i}]\!]\}_{i=1}^{k}$ with $n' + n$ bits before and n bits after the radix point.

```
if k = 0 then
    return {}
else
    l ← ⌈log_2 k⌉
    [[x̃^1]] ← Extend([[x̂]], n' + (l + 1)n)
    for i ← 0 to l − 1 do
        {[[x̃^j]]}_{j=2^i+1}^{2^{i+1}} ← MultArr([[x̃^{2^i}]], {[[x̃^j]]}_{j=1}^{2^i})
        for j ← 1 to 2^{i+1} do in parallel
            [[x̃^j]] ← Cut([[x̃^j]], n)
        end
    end
    return {[[x̃^i]]}_{i=1}^{k}
end
```

least significant bits of $[\![u]\!]$ and results in an $(m-n)$-bit integer. It computes $\lfloor u/2^n \rfloor$ more efficiently than division or shift-right operation.

– Multiplication of integer with an array of integers `MultArr`$([\![u]\!], \{[\![v_i]\!]\}_{i=1}^{k})$, where $[\![u]\!] \in \mathbb{Z}_{2^n}$ and $[\![v_i]\!] \in \mathbb{Z}_{2^n}$ for every $i \in \{1, \ldots, k\}$. The operation results in an array $\{[\![w_i]\!]\}_{i=1}^{k} \in \mathbb{Z}_{2^n}^k$ where $w_i = u \cdot v_i$. The implementation is straightforward based on regular integer multiplication protocol. Efficiency is improved by sending the shares of u only once instead of k times.

We do not describe the implementations of those operations here. However, all of them are relatively straightforward to implement using the tools provided in [10].

3.1 Efficient polynomial evaluation

Most of our floating-point functions are implemented using polynomial approximation. For example, when computing the square root of $2^e \cdot f$ we approximate the square root of fixed-point f with a polynomial and return $2^{e/2} \cdot \sqrt{f}$ [26, Alg. 5]. Fast and precise fixed-point polynomial evaluation is important to ensure the speed and accuracy of floating-point operations. Recall that fixed-point addition is just regular integer addition. Multiplication requires extending both inputs to larger integers, integer multiplication and dropping the lowest bits.

We have significantly improved upon the fixed-point polynomial evaluation presented in [26, Alg. 1]. Improved protocol for polynomial evaluation is presented in Alg. 2 and a helper function for evaluating integer powers of a fixed-

Algorithm 2: Fixed-point polynomial evaluation protocol.

Data: $[\![\widetilde{x}]\!], \{\widetilde{c_i}\}_{i=0}^{k}, n, n'$

Result: Computes a public polynomial on a secret fixed-point number. Takes in a secret fixed-point number $[\![\widetilde{x}]\!]$ with 0 bits before and n bits after the radix point and public fixed-point coefficients $\{\widetilde{c_i}\}_{i=0}^{k}$ with $n' + n$ bits before and n bits after the radix point (the highest n bits are empty). Outputs a secret fixed-point number $[\![\widetilde{y}]\!]$ with 0 bits before and n bits after the radix point that is the value of the polynomial at x.

$\{[\![\widetilde{x^i}]\!]\}_{i=1}^{k} \leftarrow$ `PowArr(`$[\![\widetilde{x}]\!], k, n, n'$`)`
$[\![\widetilde{z_0}]\!] \leftarrow$ `Share(`$\widetilde{c_0}$`)`
for $i \leftarrow 1$ **to** k **do in parallel**
  $[\![\widetilde{z_i}]\!] \leftarrow \widetilde{c_i} \cdot [\![\widetilde{x^i}]\!]$
end
for $i \leftarrow 0$ **to** k **do in parallel**
  $[\![\widetilde{z'_i}]\!] \leftarrow$ `Trunc(`$[\![\widetilde{z_i}]\!], n'$`)`
end
$[\![\widetilde{y}]\!] \leftarrow$ `Cut(Sum(`$\{[\![\widetilde{z'_i}]\!]\}_{i=0}^{k}$`), `$n$`)`
return $[\![\widetilde{y}]\!]$

point number is presented in Alg. 1. First, polynomial coefficients are now represented in two's complement form as opposed to using sign bits. This means we do not need to pick different multiplication results depending on the sign bits. Second, we have improved the efficiency of fixed-point multiplications which are used to evaluate the polynomial. The algorithm in [26] uses ordinary fixed-point multiplications throughout. Fixed-point multiplication requires extending the operands beforehand, multiplying, and then cutting off the lowest bits. This approach is costly, and we would like to avoid extending the operands before each multiplication. So, we extend the argument of the polynomial only once, in the beginning, by a sufficient number of bits to allow for all subsequent cuts. This approach is analogous to the one used in [27, Alg. 8] for computing the product of several fixed-point numbers. Third, we have made the last round of polynomial evaluation more efficient; while in [26, Alg. 1] the powers of the argument are multiplied by the corresponding coefficients, the lowest bits of the results are cut off, and then they are added up to find the value of the polynomial, we first perform the summation and *then* cut off the lowest bits of the sum, thus replacing k cut operations with 1. In addition to efficiency this shortcut slightly improves precision as it results in smaller rounding error of the end result.

Our polynomial evaluation algorithm is in a way less general than [26, Alg. 1] as both the argument and the result have to be in range $[0, 1)$. However, this approach is sufficient for all the floating-point functions that we have implemented. In fact, this striction offers an advantage as it ensures that the powers of x do not overflow. Note that we do not place any restrictions on the size of the *coefficients*, while [26, Alg. 1] requires the coefficients to fit into the same fixed-point format as the argument and the result. In [26, Alg. 5], when com-

puting the square root of a fixed-point number in range $[0.5, 1)$, the argument has to be shifted right in order to achieve a fixed-point format with enough bits before radix point to fit in the coefficients; our approach allows for coefficients that are larger than the argument, and therefore, no precision is lost through shifting out the lowest bits of the argument.

We, similarly to [26], approximate functions by interpolating through Chebyshev nodes [12, p. 521]. We have implemented two adjustments which result in better approximations.

First, sometimes we want the result to be in a certain range. For example, we assume that the result of 2^{x-1} where $x \in [0, 1)$ ought to be in range $[0.5, 1)$. However, approximation errors might cause results outside the range and overflows. In [26] this problem was solved by the so-called *correction* protocol which normalizes the result into the correct range. We get a suitable result directly, with no need for the correction step. If we interpolate function $f(x)$ in range (a, b) and we need $f(a)$ to be rounded upwards and $f(b)$ to be rounded downwards we pick a small positive constant ϵ and interpolate function $f(x) + \epsilon \cdot (a + b - 2x)/(b - a)$ instead. The small linear term ensures that approximation errors are in the right direction. If we want to round $f(a)$ downwards and $f(b)$ upwards then ϵ has to be negative. Should need arise to achieve errors in the same direction on both ends a small quadratic term added to the function can achieve this result.

Second, large coefficients pose a problem: due to the particularities of fixed-point polynomial evaluation they can result in large approximation errors and make the algorithm too imprecise for practical use in some cases. For example, interpolating $\mathrm{erf}(8x)$ in range $[0.125, 0.25)$ with 17 nodes results in coefficients that are larger than 2^{30} and therefore need 31 bits before radix point; when evaluating this polynomial, the rounding errors inherent to fixed-point computations result in an extremely imprecise approximation. We can improve the situation by noting that the first three bits of the input are always the same (001) and shifting the input 3 bits to the left, which amounts to multiplying it by 8 and subtracting 1. The initial range $[0.125, 0.25)$ is mapped into $[0, 1)$ and the new function that has to be interpolated is $\mathrm{erf}(x + 1)$. Interpolation with 17 nodes yields coefficients which are less than 1 and therefore require 0 bits before radix point and thus, precision is improved, and in this example the length of most variables in polynomial computation is reduced by almost 4 bytes. This approach of shifting out the known highest bit(s) of the argument and modifying the function for interpolation has improved the efficiency and precision of square root, logarithm, and error function.

As a result of aforementioned changes, evaluating a polynomial of degree 16 on a 64-bit fixed-point number takes 57 rounds and 7.5 KB of communication, while with the old algorithm, it takes 89 rounds and 27 KB of communication.

3.2 Additional improvements to floating-point protocols

In addition to improvements made to polynomial evaluation that benefit most floating-point functions, we have also modified other protocols from [26], namely inverse, square root, exponent function, and error function.

The new inverse protocol has been presented in [27, Alg. 8]. We have found that correction of fixed-point inverse approximation results is not necessary as with this method 0.5^{-1} is always rounded down and 1^{-1} is always rounded up.

Computation of exponent function begins by separating the input x into whole part and fractional part. In [26, Alg. 6] the whole part $\lfloor x \rfloor$ is computed in integer format and converted to floating-point format. The fractional part $\{x\}$ is computed through floating-point subtraction: $\{x\} = x - \lfloor x \rfloor$. Then $\{x\}$ has to be converted to fixed-point format in order to approximate 2^x. Instead of combining costly integer to floating-point conversion and floating-point subtraction, we have designed a special separation protocol which efficiently separates a floating-point number into whole and fractional part (in integer and fixed-point format, respectively) by obliviously choosing between all possible results.

Another optimization we have devised for exponent function is an improvement to the computation of polynomials on $\{x\}$ and $1-\{x\}$. Instead of computing the powers of $1-\{x\}$ in ordinary manner we use the powers of $\{x\}$ and binomial coefficients. This employs only fast, local operations - multiplication by a public integer and addition. (For why we need to compute the value of a polynomial on both $\{x\}$ and $1-\{x\}$ see [26, Alg. 6].)

When $2^{\{x\}}$ has been found and converted to floating-point format, the end result is computed as $2^{\lfloor x \rfloor} \cdot 2^{\{x\}}$. In [26, Alg. 6] this is achieved through floating-point multiplication. We have found a more efficient approach: since $2^{\{x\}}$ is a floating-point number we can just add $\lfloor x \rfloor$ to the exponent (which allows us to avoid an integer to floating-point conversion and a floating-point multiplication).

Finally, we have added a new feature to exponent function. When 2^x becomes so small it cannot be represented accurately, we round the result down to zero.

In [26, Alg. 7] $\mathrm{erf}(x)$ is approximated by $2x/\sqrt{\pi}$ if $x < \epsilon$ and 1 if $x \geqslant 4$. The interval $[\epsilon, 4)$ is divided into 4 pieces and in each one the function is approximated with a different polynomial. In our implementation, double-precision $\mathrm{erf}(x)$ is approximated by 1 if $x \geqslant 8$. The interval $[\epsilon, 8)$ is divided into 8 pieces; in first six the function is approximated with polynomials and in last two with constants. We compute several different polynomials (4 in single-precision case and 6 in double-precision case) on the same number and perform oblivious choices in the end. We can optimise this calculation by computing the powers of the argument only once as they are the same for all polynomials. But the main improvement in performance comes from restructuring the algorithm to compute only the correct value of $\mathrm{erf}(x)$ instead of computing several different values and obliviously choosing between them in the end. In [26, Alg. 7] several possible shift rights of the significand are computed (essentially giving us several possible results of the floating-point to fixed-point conversion). On all of them, error function is computed, and finally, the correct result is picked obliviously. We have reversed the order of the last two steps: first, we obliviously pick the correct shift right of the significand (essentially performing a floating-point to fixed-point conversion) and *then* we compute the error function on the single correct value.

Our improvements have increased precision compared to [26]. The maximum relative error of inverse is $2.69 \cdot 10^{-9}$ for single precision and $7.10 \cdot 10^{-19}$ for

double precision (compared to $1.3 \cdot 10^{-4}$ and $1.3 \cdot 10^{-8}$ in [26]). For square root our errors are respectively $4.92 \cdot 10^{-9}$ and $1.30 \cdot 10^{-15}$ (compared to $5.1 \cdot 10^{-6}$ and $4.1 \cdot 10^{-11}$ in [26]). In a few cases we have achieved better accuracy guarantees than what IEEE 754 single- and double-precision floating-point numbers allow. This is possible because we are using slightly longer fractional parts.

3.3 New floating-point protocols

In addition to improving the floating-point protocols published in [23, 26, 27] we have also designed a few new ones, namely logarithm, sine, floor and ceiling. Here we shall present a short explanation of logarithm and sine.

In order to compute the binary logarithm of a floating-point number we note that $\log_2(2^e \cdot f) = e + \log_2 f$. As f is in range $[0.5, 1)$ its binary logarithm is in range $[-1, 0)$. However, in order to easily convert it to a floating-point number, we would like to get a result in range $[0.5, 1)$. Therefore, we transform the expression above as follows: $e + \log_2 f = (e-2) + 2(\log_4 f + 1)$. If f is in range $[0.5, 1)$ then the value of $\log_4 f + 1$ is in range $[0.5, 1)$. This is the function that we approximate with a fixed-point polynomial. For double precision, we split the interval into two equal parts and use two different polynomials. Finally, $e - 2$ is converted to floating-point format and the end result is computed through floating-point addition. Near 1 we use second degree Taylor polynomial $\log_2 x \approx \log_4 e \cdot (x-1)(3-x)$ to achieve better precision. In order to convert binary logarithm to natural logarithm we use the conversion $\ln x = \ln 2 \cdot \log_2 x$.

The algorithm for computing the sine is relatively straightforward as we can use to our advantage all kinds of symmetry inherent to the function. First, we divide the argument by 2π and find the fractional part in fixed-point format, thus reducing the computation to two full turns (from -2π to 2π). We note that $\sin(-x) = -\sin x$, $\sin(x + \pi) = -\sin x$, and $\sin(\pi/2 - x) = \sin(\pi/2 + x)$. Thus, we have reduced the computation to one quarter-turn (from 0 to $\pi/2$). Then we use fixed-point polynomial approximation and convert the end result to floating-point format. When the argument is near zero we use the approximation $\sin x \approx x$ to achieve better precision.

4 Optimization techniques

The Protocol DSL has allowed us to easily apply certain optimizations across the entire suite of protocols employed in Sharemind. They are described in the following. The optimizations are specific to the "main" protocol set [10] of Sharemind based on additive secret sharing over finite rings, using three computing parties.

4.1 Shared random number generators

To ensure that a party's view in a protocol could be generated from only its inputs, we commonly use the *resharing* protocol, to ensure independence from

Table 1. Speedups of shared RNG (SRNG) and symmetric multiplication protocols over the regular multiplication. The speedups have been measured from 1 element inputs to 10^8 element input vectors.

Bit-width	SRNG					Symmetric					SRNG & Symmetric				
	10^0	10^2	10^4	10^6	10^8	10^0	10^2	10^4	10^6	10^8	10^0	10^2	10^4	10^6	10^8
64	1.03	1.03	1.48	1.44	1.61	1.08	1.09	1.13	1.08	1.04	1.10	1.12	1.67	1.55	1.68
32	0.95	0.98	1.34	1.45	1.30	1.09	1.08	1.14	1.02	1.08	1.04	1.06	1.53	1.48	1.41
16	0.85	0.90	1.14	1.36	1.41	1.18	1.12	1.17	1.02	1.02	1.00	1.01	1.34	1.39	1.43
8	0.96	0.96	1.03	1.11	1.01	1.04	1.03	0.91	1.03	1.07	0.99	0.98	0.95	1.14	1.08

other parties' inputs and outputs. For example, usually every input of a protocol is explicitly reshared. The resharing protocol takes a private value $[\![u]\!] \in R$ and returns a $[\![v]\!] \in R$ such that $u = v$ and all shares $[\![v]\!]_i$ are uniformly distributed and independent of the shares $[\![u]\!]_j$. The protocol is implemented as follows: each party $\mathcal{P}_i$ generates a random value $r_i \leftarrow R$ and sends it to the next computing party $\mathcal{P}_{\mathsf{n}(i)}$, adds the generated value r_i to the input share $[\![u]\!]_i$, and subtracts the random number $r_{\mathsf{p}(i)}$ received from the previous computing party $\mathcal{P}_{\mathsf{p}(i)}$. The shares of the output $[\![v]\!]$ of the protocol are $([\![u]\!]_1 + r_1 - r_3, [\![u]\!]_2 + r_2 - r_1, [\![u]\!]_3 + r_3 - r_2)$. We see that $v = [\![v]\!]_1 + [\![v]\!]_2 + [\![v]\!]_3 = [\![u]\!]_1 + [\![u]\!]_2 + [\![u]\!]_3 = u$.

We can spot a common pattern that occurs in resharing (and in some other primitive protocols): a party generates a random number and sends it to some other party. This pattern can be optimized by letting both parties generate the same random number using a common random number generator (RNG). Analysis of our protocols shows that network communication can be reduced by 30% to 60% using this technique (exactly 60% in the case of integer multiplication protocol). This optimization is not new and has previously been used in [24]. Our toolchain around the Protocol DSL allows this optimization to be automatically introduced, with no changes to the specification of the protocols. The optimization itself is straightforward on our intermediate representation: we detect randomness nodes that are sent to one other computing party, and transform them to instead take use of shared randomness nodes.

We have manually implemented this optimization for the multiplication protocol (for which the Protocol DSL has not been used) and compared the performance to the unoptimized version to validate the effectiveness of this modification. Multiplication protocol has been chosen because of its simplicity, efficiency, ubiquity in application, and because it is one of the least computation heavy protocols. The comparison was performed using the methodology described in Sec. 5 and the results are displayed in Table 1. We see a slowdown of at most 15% on small input lengths (up to one hundred elements), but for large inputs we see a universal speedup that reaches up to 60%. The performance of 64-bit multiplication has been universally improved. The slowdown on small inputs can be explained by a slight increase in computation overhead (critical path became longer due to invoking the shared RNG in the end of the protocol) and the

Algorithm 3: Multiplication protocol.

Data: Shared values $[\![u]\!], [\![v]\!] \in R$
Result: Shared value $[\![w]\!] \in R$ such that $uv = w$.

$[\![u]\!] \leftarrow$ `Reshare(`$[\![u]\!]$`)`
$[\![v]\!] \leftarrow$ `Reshare(`$[\![v]\!]$`)`
All parties $\mathcal{P}_i$ perform the following:
    Send $[\![u]\!]_i$ and $[\![v]\!]_i$ to $\mathcal{P}_{\mathsf{n}(i)}$
    Receive $[\![u]\!]_{\mathsf{p}(i)}$ and $[\![v]\!]_{\mathsf{p}(i)}$ from $\mathcal{P}_{\mathsf{p}(i)}$
    $[\![w]\!]_i \leftarrow [\![u]\!]_i \cdot [\![v]\!]_i + [\![u]\!]_{\mathsf{p}(i)} \cdot [\![v]\!]_i + [\![u]\!]_i \cdot [\![v]\!]_{\mathsf{p}(i)}$
$[\![w]\!] \leftarrow$ `Reshare(`$[\![w]\!]$`)`
return $[\![w]\!]$

Algorithm 4: Symmetric multiplication protocol.

Data: Shared values $[\![u]\!], [\![v]\!] \in R$
Result: Shared value $[\![w]\!] \in R$ such that $uv = w$.

$[\![u]\!] \leftarrow$ `Reshare(`$[\![u]\!]$`)`
$[\![v]\!] \leftarrow$ `Reshare(`$[\![v]\!]$`)`
All parties $\mathcal{P}_i$ perform the following:
    Send $[\![u]\!]_i$ to $\mathcal{P}_{\mathsf{n}(i)}$ and $[\![v]\!]_i$ to $\mathcal{P}_{\mathsf{p}(i)}$
    Receive $[\![u]\!]_{\mathsf{p}(i)}$ from $\mathcal{P}_{\mathsf{p}(i)}$ and $[\![v]\!]_{\mathsf{n}(i)}$ from $\mathcal{P}_{\mathsf{n}(i)}$
    $[\![w]\!]_i \leftarrow [\![u]\!]_i \cdot [\![v]\!]_i + [\![u]\!]_{\mathsf{p}(i)} \cdot [\![v]\!]_i + [\![u]\!]_{\mathsf{p}(i)} \cdot [\![v]\!]_{\mathsf{n}(i)}$
$[\![w]\!] \leftarrow$ `Reshare(`$[\![w]\!]$`)`
return $[\![w]\!]$

speedup can be explained by the decrease in network communication. In fact, network communication is reduced by exactly 60%.

4.2 Symmetric protocols

Multiplication protocol in additive schemes is commonly presented as Alg. 3 such as in [10] and [27]. The given protocol is perfectly reasonable when the SRNG optimization is not used: the resharing sub-protocol sends the network messages in one direction and the multiplication protocol itself in the other. As a result the communication channels are under similar workload. However, using the SRNG optimization results in a protocol that sends network messages only over one of the two network channels. We propose a small modification in the form of Alg. 4 as an alternative multiplication protocol that uses the network in a balanced manner. The correctness and security of the algorithm can be shown the same way as it was shown for the multiplication protocol in [10].

The symmetric protocol provides a small performance gain over the SRNG optimized protocol. The comparison against our legacy multiplication protocol (see Table 1) shows better results and disappearance of the slowdown present with only the SRNG optimization. Only the 8-bit multiplication experiences a small slowdown in a few cases. We predict that the speedups will be greater in

Table 2. Speedup of optimized floating-point protocols.

Operation	Precision	Speedup on given input length						
		10^0	10^1	10^2	10^3	10^4	10^5	10^6
$[\![x]\!] + [\![y]\!]$	single	1.04	1.13	1.48	1.94	1.73	1.71	1.73
	double	0.97	1.03	1.38	1.67	1.61	1.69	1.77
$[\![x]\!] \times [\![y]\!]$	single	0.91	0.92	1.04	1.42	1.60	1.45	1.57
	double	1.03	1.08	1.28	1.81	1.82	1.80	1.79
$\sqrt{[\![x]\!]}$	single	0.91	0.98	1.33	1.82	1.73	1.66	1.64
	double	1.06	1.22	1.71	1.86	1.85	1.85	1.87

a setting where network latency is worse or the available bandwidth is smaller because in these cases the network will become the dominant bottleneck. This claim is supported by the evidence that the speedups improve as the protocols need to send more data over the network (larger bit-widths or larger input vectors).

This modification can be applied to many other protocols, but a few of the protocols are inherently asymmetric (such as squaring a value, or finding the bitwise conjunction of a single bit with a 64-bit integer). For all asymmetric protocols we can implement two versions that are unbalanced in different directions, and pick versions of them such that overall the communication is roughly balanced (we do not expose this facility to the end user). This optimization has been applied manually as the set of primitive protocols is manageable and the protocol DSL enables such changes easily. We have not explored the possibility of automatically performing communication balancing.

4.3 Speedup over previous results

We have applied the systematic optimizations presented in this section to all our protocols and compared the results against operations without those optimizations. In addition to the optimizations mentioned previously we have also eliminated many resharing calls (this optimization does not reduce network communication) as allowed by [7] and verified the security of resulting protocols using our privacy analyser [32]. Table 2 shows comparison results for floating-point addition, multiplication and square root. These protocols provide a rough idea of how the optimizations fare across all protocols.

Table 2 shows an almost universal improvement in performance. In a few cases single-precision floating-point operations perform slightly worse (less than 10%) but only on small input sizes. In the case of inputs of length 100 and more we see significant speedups across the board. In a few cases speedups reach over 80%.

5 Large-scale performance evaluation

Benchmarking was performed on a dedicated cluster of three computers connected with 10Gbps Ethernet. Each computer was equipped with 128GB DDR4 memory, two 8-core Intel Xeon (E5-2640 v3) processors and was running Debian 8.2 Jessie (15^{th} Sep 2015). Both memory overcommit and swap were disabled. During benchmarking only the necessary system processes and some low overhead services (such as SSH and monitoring) were enabled.

A single run-time measurement was computed by taking the running times of each of the computing parties and finding the maximum of those. This is necessary as a protocol may terminate faster for some participants and the maximum reflects the time it takes for the result of the operation to become available to all. The average running time was estimated by computing the mean of all the measurements. On every input length we performed at least 5 repetitions (10 for integer operations) and, to reduce variance, significantly more on small input lengths (up to 10000 repetitions). Measurements were performed in a randomized order because we found that running the tests sequentially in an increasing size of inputs gave significantly better performance results. Sequential order results in a steady increase of network load which is predictable for the networking layer but is not a very realistic scenario for all SMC applications.

Performance results for floating-point operations are presented in Table 3. We have measured addition, multiplication, comparison, reciprocal, square root, exponentiation, natural logarithm, sine, and error function from 1 element input to one million element input vectors. All the results have been presented in operations per millisecond (thousands of operations per second). Looking at the table, it is clear that performance scales very well with vectorization: only a few hundred scalar operations can be executed per second but by computing on many inputs in parallel we can perform hundreds of thousands of operations per second.

We have also thoroughly measured the performance of integer and fixed-point multiplication operations (Table 4). The fixed-point operations, especially addition and multiplication, have turned out to be useful tools in implementing efficient higher-level applications. As the respective floating-point operations are rather slow, the computations relying heavily on them may become impractical (for example, floating-point addition [23, Alg. 4] requires private shifts which makes it a costly operation). While not a universal solution, efficient signed fixed-point operations alleviate the problem in many cases.

We have also evaluated private integer multiplication to establish a baseline, against which to compare more complex protocols when choosing the operations to be used in a larger application. We have limited the performance evaluation of multiplication to 10^9 element input vectors. This is due to memory limitations: a single 10^{10} element vector of 64-bit integers takes roughly 80 gigabytes of RAM (it would be possible to only allocate a single vector and use that as both input and output, but this would compute square and not product). Capability to handle arrays of 10^9 elements with ease demonstrates the robustness of our platform.

Table 3. Performance (in operations per millisecond) of optimized floating-point operations. Combing all manual and automatic optimizations presented in this work. Variables x and y denote floating-point numbers.

Operation	Precision	OP/ms on given input length						
		10^0	10^1	10^2	10^3	10^4	10^5	10^6
$[\![x]\!] + [\![y]\!]$	single	0.32	3.0	20.4	54.3	60.6	52.8	53.1
	double	0.27	2.4	12.9	24.5	22.9	23.9	25.3
$[\![x]\!] \times [\![y]\!]$	single	0.52	4.8	36.1	140	231	172	185
	double	0.54	4.8	32.2	111	131	107	106
$[\![x]\!] < [\![y]\!]$	single	1.14	10.5	78.8	210	237	199	209
	double	0.97	9.0	62.3	133	120	111	118
$[\![x]\!]^{-1}$	single	0.30	2.7	18.1	48.7	52.4	45.5	49.4
	double	0.23	1.9	9.0	16.8	16.9	17.6	18.6
$\sqrt{[\![x]\!]}$	single	0.26	2.4	16.4	44.8	48.7	45.1	44.1
	double	0.21	1.7	6.5	10.4	9.4	11.2	11.2
$\exp [\![x]\!]$	single	0.18	1.7	11.4	28.8	33.1	30.4	29.2
	double	0.16	1.3	5.4	9.1	8.8	9.5	9.9
$\ln [\![x]\!]$	single	0.14	1.2	6.8	12.3	12.0	11.2	11.1
	double	0.12	1.0	3.3	4.2	4.0	4.4	4.6
$\sin [\![x]\!]$	single	0.14	1.2	6.3	9.4	8.4	8.8	9.3
	double	0.12	0.9	2.7	2.8	2.8	3.3	3.4
$\mathrm{erf} [\![x]\!]$	single	0.23	2.0	12.1	24.2	24.1	23.5	23.7
	double	0.18	1.3	4.2	5.9	5.5	6.7	6.8

We have compared the performance of arithmetic operations and square root against previous works. Unfortunately it was not possible to provide comparison in an identical setups as both previous works we compare against have the performance measures on a 1 Gbps Ethernet connection over LAN (opposed to our 10 Gbps connection over LAN). However, we found that we never came close to saturating a 1 Gbps of the connection. Performance in [33] was measured on cluster of three nodes each equipped with 48 GB of RAM and 12-core 3 GHz Intel CPUs supporting AES-NI and HyperThreading. Performance in [18] was measured on two desktop computers each equipped with a 3.5 GHz Intel Core i7 CPU and 16 GB of RAM (the number of cores was unspecified).

In the case of additive 3-party secret sharing the best results so far have been obtained in [33]. In the case of scalar operations our results show 132 fold speedup for addition, 67 fold speedup for multiplication and 618 fold speedup for square root. The speedups also remain good for 10^4 element input vectors: 16, 14 and 416 fold respectively. Additionally [33] reports the performance of garbled circuit based on IEEE 754 floating-point numbers. Compared to those

Table 4. Performance of optimized integer and signed fixed-point multiplication. Numbers are provided in operations per second with suffix K denoting thousands and M denoting millions.

Type	10^0	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
uint8	7.4K	71.6K	703.5K	5.8M	24.1M	38.9M	40.1M	28.0M	37.9M	41.5M
uint16	7.0K	68.4K	663.4K	5.4M	22.0M	34.0M	29.5M	29.3M	35.0M	37.1M
uint32	6.6K	65.7K	629.7K	5.0M	17.1M	22.4M	18.8M	20.7M	22.1M	21.4M
uint64	6.4K	63.5K	586.9K	4.3M	11.2M	12.1M	10.5M	12.1M	13.7M	13.3M
fix32	640	6.0K	51.2K	270.8K	435.4K	344.1K	361.1K	369.4K	351.6K	
fix64	680	6.2K	46.6K	187.3K	226.2K	184.3K	186.0K	187.6K	179.0K	

we provide 13, 20 and 27 fold speedups in the case of scalars and 102, 364 and 495 fold speedups in the case of 10^4 element input vectors.

While garbled circuit approach is not directly comparable to secret sharing we also compare our results against [18] which provides, to our knowledge, as of now, the best performance for 2-party garbled circuit approach. For scalar operations we are, at worst, 80% slower, and in case of 10^4 element input vectors at worst 50% slower, and at best 4.6 times faster. This is considering only online time. When offline time is also taken into account we report similar performance for scalar operations and significant speedups for vectorized ones (over 40 fold). These comparisons are against the better of GMW (vector operations) and Yao (scalar operations).

6 Conclusions

We have demonstrated the current state of the art in the performance of SMC protocols for numeric computations. Our results show that with careful design and the right set of tools, significant performance improvements are still possible. But currently, as Table 4 shows, the performance of SMC operations on modern but reasonably-spec'd hardware is comparable to a computer with a 80386 processor.

References

1. 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. ACM (2013)
2. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015. ACM (2015)
3. Aliasgari, M., Blanton, M., Zhang, Y., Steele, A.: Secure computation on floating point numbers. In: 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013. The Internet Society (2013)

4. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: CCS '08: Proceedings of the 15th ACM conference on Computer and communications security. pp. 257–266. ACM (2008)
5. Bogdanov, D., Jõemets, M., Siim, S., Vaht, M.: A Short Paper on How the National Tax Office Evaluated a Tax Fraud Detection System Based on Secure Multi-party Computation. In: Proceedings of 19th International Conference on Financial Cryptography and Data Security (2015)
6. Bogdanov, D., Kamm, L., Laur, S., Pruulmann-Vengerfeldt, P., Talviste, R., Willemson, J.: Privacy-preserving statistical data analysis on federated databases. In: Privacy Technologies and Policy - Second Annual Privacy Forum, APF 2014, Athens, Greece, May 20-21, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8450, pp. 30–55. Springer (2014)
7. Bogdanov, D., Laud, P., Laur, S., Pullonen, P.: From input private to universally composable secure multi-party computation primitives. In: IEEE 27th Computer Security Foundations Symposium, CSF 2014. pp. 184–198. IEEE (July 2014)
8. Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic programming of privacy-preserving applications. In: Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014. p. 53. ACM (2014)
9. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: ESORICS. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008)
10. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multiparty computation for data mining applications. Int. J. Inf. Sec. 11(6), 403–418 (2012)
11. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T.P., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers. Lecture Notes in Computer Science, vol. 5628, pp. 325–343. Springer (2009)
12. Burden, R.L., Faires, J.D.: Numerical Analysis, 9th edition. Brooks/Cole (2011)
13. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In: USENIX Security Symposium. pp. 223–239. Washington, DC, USA (2010)
14. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6052, pp. 35–50. Springer (2010)
15. Cramer, R., Damgård, I., Maurer, U.M.: General secure multi-party computation from any linear secret-sharing scheme. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 1807, pp. 316–334. Springer (2000)
16. Cramer, R., Damgård, I., Nielsen, J.B.: Multiparty computation from threshold homomorphic encryption. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 2045, pp. 280–299. Springer (2001)
17. Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous Multiparty Computation: Theory and Implementation. In: Public Key Cryptography. Lecture Notes in Computer Science, vol. 5443, pp. 160–179. Springer (2009)
18. Demmler, D., Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., Zeitouni, S.: Automated synthesis of optimized circuits for secure computation. In: Proceed-

ings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015 [2], pp. 1504–1517
19. Goldreich, O., Micali, S., Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: STOC. pp. 218–229. ACM (1987)
20. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS'10. pp. 451–462. ACM (2010)
21. Ishai, Y., Paskin, A.: Evaluating branching programs on encrypted data. In: TCC. Lecture Notes in Computer Science, vol. 4392, pp. 575–594. Springer (2007)
22. Kamm, L.: Privacy-preserving statistical analysis using secure multi-party computation. Ph.D. thesis, University of Tartu (2015)
23. Kamm, L., Willemson, J.: Secure floating point arithmetic and private satellite collision analysis. Int. J. Inf. Sec. 14(6), 531–548 (2015)
24. Keller, M., Scholl, P., Smart, N.P.: An architecture for practical actively secure MPC with dishonest majority. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013 [1], pp. 549–560
25. Kerschbaum, F., Schröpfer, A., Zilli, A., Pibernik, R., Catrina, O., de Hoogh, S., Schoenmakers, B., Cimato, S., Damiani, E.: Secure collaborative supply-chain management. IEEE Computer 44(9), 38–43 (2011)
26. Krips, T., Willemson, J.: Hybrid model of fixed and floating point numbers in secure multiparty computations. In: Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8783, pp. 179–197. Springer (2014)
27. Laud, P., Randmets, J.: A Domain-Specific Language for Low-Level Secure Multiparty Computation Protocols. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015 [2], pp. 1492–1503
28. Lindell, Y., Pinkas, B.: A Proof of Security of Yao's Protocol for Two-Party Computation. J. Cryptology 22(2), 161–188 (2009)
29. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.W.: Automating efficient ram-model secure computation. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 623–638. IEEE Computer Society (2014)
30. Liu, Y.C., Chiang, Y.T., Hsu, T.S., Liau, C.J., Wang, D.W.: Floating point arithmetic protocols for constructing secure data analysis application. Procedia Computer Science 22, 152 – 161 (2013), 17th International Conference in Knowledge Based and Intelligent Information and Engineering Systems - KES2013
31. Malka, L.: VMCrypt: modular software architecture for scalable secure computation. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011. pp. 715–724. ACM (2011)
32. Pettai, M., Laud, P.: Automatic Proofs of Privacy of Secure Multi-Party Computation Protocols Against Active Adversaries. In: 2015 IEEE 28th Computer Security Foundations Symposium (CSF 2015) (2015)
33. Pullonen, P., Siim, S.: Combining secret sharing and garbled circuits for efficient private ieee 754 floating-point computations. In: Financial Cryptography and Data Security - FC 2015 Workshops, BITCOIN, WAHC and Wearable 2015, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers, LNCS, vol. 8976, pp. 172–183. Springer (2015)

34. Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979)
35. Zhang, Y., Steele, A., Blanton, M.: PICCO: a general-purpose compiler for private distributed computation. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013 [1], pp. 813–826