High-performance secure multi-party computation for data mining applications

Dan Bogdanov[1,2], Margus Niitsoo[1,2], Tomas Toft[3], and Jan Willemson[1,4]

[1] Cybernetica, Ülikooli 2, 51003 Tartu, Estonia (dan@cyber.ee, janwil@cyber.ee)
[2] University of Tartu, Institute of Computer Science, J. Liivi 2, 50409 Tartu, Estonia (margus.niitsoo@ut.ee)
[3] Aarhus University, Åbogade 34, 8200 Århus N, Denmark (ttoft@cs.au.dk)
[4] STACC, Ülikooli 2, 51003 Tartu, Estonia

Abstract. Secure multi-party computation (MPC) is a technique well-suited for privacy-preserving data mining. Even with the recent progress in two-party computation techniques such as fully homomorphic encryption, general MPC remains relevant as it has shown promising performance metrics in real-world benchmarks. SHAREMIND is a secure multi-party computation framework designed with real-life efficiency in mind. It has been applied in several practical scenarios and from these experiments, new requirements have been identified. Firstly, large datasets require more efficient protocols for standard operations such as multiplication and comparison. Secondly, the confidential processing of financial data requires the use of more complex primitives, including a secure division operation. This paper describes new protocols in the SHAREMIND model for secure multiplication, share conversion, equality, bit shift, bit extraction and division. All the protocols are implemented and benchmarked, showing that the current approach provides remarkable speed improvements over the previous work. This is verified using real-world benchmarks for both operations and algorithms.

Keywords: Secure computation, Performance, Applications

1 Introduction

The aim of secure multi-party computation is to enable a number of networked parties to carry out distributed computing tasks on private information. During the computations, no one party (and, more generally, no certain subsets of the parties) should be able to learn any information about any other party's input other than what can be inferred from the output.

The theory behind MPC is fairly well-developed, but practical solutions based on it are lagging considerably behind. In the last few years, several MPC frameworks based on various techniques have been proposed and implemented, e.g. FairPlayMP [2], VIFF [13], SEPIA [7], SecureSCM [1], VMcrypt [16], TASTY [15] and SHAREMIND [3].

Existing frameworks for MPC can broadly be split into two-party and multi-party frameworks. The former requires computational security, hence solutions are based on homomorphic, public-key encryption (HE) or garbled circuits (GC). TASTY combines the two, utilizing whichever is best for the immediate task at hand. Both solutions have

drawbacks. HE is computationally expensive, while the GC approach requires that a circuit computing the function is generated. For large scale problems – such as data mining – even simply generating and storing the garbled circuit may be challenging. VMcrypt avoids this issue by generating and evaluating the circuit on the fly.

Multi-party frameworks are generally more efficient, as the computational primitives for information theoretic solutions are simpler than those of two-party computation. For example, addition and multiplication in a ring are more efficient than public key or even symmetric key cryptography. Of the multi-party frameworks, FairPlayMP's circuit based approach suffers from the issues with large scale applications as discussed above. The passively secure VIFF and SEPIA both use Shamir's secret sharing scheme to implement MPC; both are more general than SHAREMIND, as they allow an arbitrary number of parties. while SHAREMIND is limited to three.

The first published application of MPC was the Danisco sugar beet auction held in 2008 [6]. Since then, the practical feasibility of MPC has also been shown for network anomaly detection [7] and joint analysis of financial data [5]. However, neither of the presented applications required MPC protocols to process large databases. Nevertheless, it is clear that many important areas of human endeavor, such as biomedical research and business data aggregation, do require such a capacity, often working with databases with thousands or even millions of records.

This research was mainly motivated by the need to build an MPC application suitable for the statistical analysis of financial data from competing companies, who are interested in finding the common trends in the entire sector. Several trend indicators require more complicated computational primitives, e.g. private division to find out different ratios (like turnaround to personnel size). This is impossible on many of the current systems (such as those described in [6,7]) that use only relatively simple primitive operations like multiplication and comparison of two numbers.

The primary target of this paper is to develop a set of MPC protocols with high real-life performance on large databases. We will present new primitives for secure multiplication, share conversion, equality, bit shift, bit extraction and division (both by public and private divisor).

The presented protocols have been implemented as improvements to the SHAREMIND framework [3,4]. In addition to the theoretical round and communication complexities we also present benchmark results illustrating the achieved performance improvements.

2 Preliminaries

As stated above, SHAREMIND [3,4] is a secure multi-party computation system operating on additively secret-shared values. Although general m-party protocols can be devised for such a setting (see [3]), this paper concentrates on the case with three computing parties identified as $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$. Designing and implementing protocols with a specific scenario in mind allows significant efficiency gains over general protocols; such optimizations can make the difference as to whether a problem can be solved or not.

The SHAREMIND framework uses additive secret sharing over the finite ring $\mathbb{Z}_{2^n}$. Secret-shared value $u \in \mathbb{Z}_{2^n}$ is represented as a triple $[\![u]\!] = (u_1, u_2, u_3)$, with the element u_i held by party $\mathcal{P}_i$ ($i = 1, 2, 3$) and $u_1 + u_2 + u_3 \equiv u \bmod 2^n$. In the current implementation, $n = 32$ and this value is used for the performance benchmarks. However, it is stressed that all the protocols presented here work for any choice of n.

Let $\oplus$, $\vee$ and $\wedge$ denote the bitwise XOR, OR and AND operations, respectively. Many of the protocols in this paper make use of these operations over the shared bits. In these cases, it is convenient to think of a value $u \in \mathbb{Z}_{2^n}$ as a bit vector $\overline{u} \in (\mathbb{Z}_2)^n$. We stress that both u and $\overline{u}$ represent the same value and the difference is only to denote whether it is used bitwise or as an integer, so $\overline{(\cdot)}$ is best thought of as a typing indicator. Hence, in general we have $\overline{u} = \overline{u_1 + u_2 + u_3} \neq \overline{u_1} \oplus \overline{u_2} \oplus \overline{u_3}$.

Bit-level protocols also make use of vectors that are shared bitwise. We thus introduce special notation $\overline{[\![u]\!]} = (\overline{u}_1, \overline{u}_2, \overline{u}_3)$ to refer to such a bitwise sharing that $\overline{u} = \overline{u}_1 \oplus \overline{u}_2 \oplus \overline{u}_3$. This is a fairly natural extension of the notation.

To allow elementwise access to the bit vectors, we will let $\overline{u}^{(j)}$ stand for the jth bit of $\overline{u}$. We will also use notation $\overline{[\![u]\!]}^{(j)}$ to denote the share tuple $(\overline{u}_1^{(j)}, \overline{u}_2^{(j)}, \overline{u}_3^{(j)})$, so that $\overline{u}^{(j)} = \overline{u}_1^{(j)} \oplus \overline{u}_2^{(j)} \oplus \overline{u}_3^{(j)}$.

3 Proving Security of Sharemind Protocols

The security proofs in this paper are presented in the universal composability framework of Canetti [8]. To be precise, we assume that we have three distinct computational entities $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ all of which have an ideally secure authenticated channel to the two others. Security is proven in the passive (honest-but-curious) model in which the adversary is allowed to corrupt at most one of the three parties before the execution of the protocol. The adversary is then handed both the inputs and all the incoming messages of the corrupted party ("curiosity"), but he has no control over its outputs, which are assumed to be chosen as specified in the protocols ("honesty"). This model roughly corresponds to the real-world situation where we assume the protocol implementations are fairly hard to tamper with, whereas their inputs and outputs could be eavesdropped on – which is a sensible assumption for most practical purposes.

In the following, we will use of the following definition:

Definition 1. *We say that a share computing protocol is* perfectly simulatable *if there exists an efficient universal non-rewinding simulator $\mathcal{S}$ that can simulate all protocol messages to any real world adversary $\mathcal{A}$ so that for all input shares the output distributions of $\mathcal{A}$ and $\mathcal{S}(\mathcal{A})$ coincide.*

To prove that a protocol is universally composable, it suffices to show that it is perfectly simulatable and that the outputs are independent of the inputs (Lemma 2 of [3]).

In order to prove perfect simulatability, we consider the incoming views of all the computing parties and prove that they are independent of the input shares of the other parties, hence proving existence of the simulator. We will use the sequences-of-games formalism in our proofs. Denote the distribution of the incoming view G as $(\!|G|\!)$ and let the original incoming view of the party $\mathcal{P}$ be G_0. Then we are interested in finding a

sequence $G_0, G_1, \ldots, G_n$ such that

$$(\!|G_0|\!) = (\!|G_1|\!) = \ldots = (\!|G_n|\!)$$

and that the view G_n does not contain any references to input shares of the other parties.

The main tool that allows us to construct such sequences is the following simple folk lemma.

Lemma 1. *Let the incoming view G contain incoming messages $a_1 \pm r, a_2, \ldots, a_k$ where a_1, r are elements of finite additive group $\mathcal{A}$ and where r is a uniformly random element of $\mathcal{A}$, independent from all a_i. Then*

$$(\!|G|\!) = (\!|G[a_1 \pm r/r]|\!)\,,$$

where $G[a_1 \pm r/r]$ denotes the game, where the occurrence of $a_1 \pm r$ has been replaced by r.

Proof. If $r \in \mathcal{A}$ is uniformly distributed and independent from all a_i then so is $r \pm a_1$ since $f_r(x) := r \pm x$ is a bijective mapping for $\mathcal{A}$.

In the following security proofs this lemma will be used for various groups, including $\mathbb{Z}_2$, $\mathbb{Z}_{2^n}$ and $(\mathbb{Z}_2)^n$.

The lemma is often used in combination with Lemma 1 of [3] which states that a concurrent composition of perfectly simulatable protocols is also perfectly simulatable, which allows us to use already defined perfectly simulatable protocols as subroutines without analyzing their internal messages.

All of the protocols described in the following are fully simulatable. To achieve full security in the universal composability framework, one extra step is needed to also guarantee the independence of the protocol outputs from its inputs. This resharing step was already described in [3] and is brought here for completeness as Algorithm 1. The input shared value $[\![u]\!]$ is reshared as $[\![w]\!]$ so that $u = w$, all shares w_i are uniformly distributed and u_i and w_j are independent for all i, j. This is accomplished by masking the input shares with values that are randomly generated, but shared by only two of the three parties. Sharing the random values between two parties requires one round of communication, but since the values being sent are independent of the inputs, it can generally be done in parallel with the last round of the protocol on whose outputs it is to be applied.

Theorem 1. *Algorithm 1 is correct.*

Proof. It is easy to see that

$$\begin{aligned} w &= w_1 + w_2 + w_3 \\ &= u_1 + r_{12} - r_{31} + u_2 + r_{23} - r_{12} + u_3 + r_{31} - r_{23} \\ &= u_1 + u_2 + u_3 = u\,. \end{aligned}$$

Proofs of independence can be given exactly as in Lemma 1, since all the elements w_i are of the form $u_j + r - s$ for randomly generated elements r, s.

Algorithm 1: Resharing protocol $[\![w]\!] \leftarrow \mathsf{Reshare}([\![u]\!])$.

Data: Shared value $[\![u]\!]$.
Result: Shared value $[\![w]\!]$ such that $w = u$, all shares w_i are uniformly distributed and u_i and w_j are independent for $i, j = 1, 2, 3$.

$\mathcal{P}_1$ generates random $r_{12} \leftarrow \mathbb{Z}_{2^n}$.
$\mathcal{P}_2$ generates random $r_{23} \leftarrow \mathbb{Z}_{2^n}$.
$\mathcal{P}_3$ generates random $r_{31} \leftarrow \mathbb{Z}_{2^n}$.
All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
$\mathcal{P}_1$ computes $w_1 \leftarrow u_1 + r_{12} - r_{31}$.
$\mathcal{P}_2$ computes $w_2 \leftarrow u_2 + r_{23} - r_{12}$.
$\mathcal{P}_3$ computes $w_3 \leftarrow u_3 + r_{31} - r_{23}$.
Return $[\![w]\!]$.

Algorithm 2: Protocol for multiplying two shared values $[\![w']\!] \leftarrow \mathsf{Mult}([\![u]\!], [\![v]\!])$.

Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shared value $[\![w']\!]$ such that $w' = uv$.

$[\![u']\!] \leftarrow \mathsf{Reshare}([\![u]\!])$
$[\![v']\!] \leftarrow \mathsf{Reshare}([\![v]\!])$
$\mathcal{P}_1$ sends u'_1 and v'_1 to $\mathcal{P}_2$.
$\mathcal{P}_2$ sends u'_2 and v'_2 to $\mathcal{P}_3$.
$\mathcal{P}_3$ sends u'_3 and v'_3 to $\mathcal{P}_1$.
$\mathcal{P}_1$ computes $w_1 \leftarrow u'_1v'_1 + u'_1v'_3 + u'_3v'_1$.
$\mathcal{P}_2$ computes $w_2 \leftarrow u'_2v'_2 + u'_2v'_1 + u'_1v'_2$.
$\mathcal{P}_3$ computes $w_3 \leftarrow u'_3v'_3 + u'_3v'_2 + u'_2v'_3$.
Return $[\![w']\!] \leftarrow \mathsf{Reshare}([\![w]\!])$.

We stress that for practical applications, this step needs to be added to the end of all the protocols which are available to the end-user for use. However, for all the intermediate steps, perfect simulatability is enough, which is why we omit the resharing step from the descriptions of the protocols in this paper.

4 Multiplication Protocol

Instead of using the Du-Atallah multiplication protocol as in [3] and [4], we propose a new protocol which is based on the following observation. If we have two values u and v shared additively as $u = u_1 + u_2 + u_3$ and $v = v_1 + v_2 + v_3$, their product is $uv = \sum_{i=1}^{3} \sum_{j=1}^{3} u_i v_j$. The addends of the form $u_i v_i$ can be computed locally by party $\mathcal{P}_i$. In order to find an addend of the form $u_i v_j$ ($i \neq j$), the share u_i can be sent from $\mathcal{P}_i$ to $\mathcal{P}_j$ (or the share v_j from $\mathcal{P}_j$ to $\mathcal{P}_i$). Knowing the shares u_i and u_j, party $\mathcal{P}_j$ is still unable to get any information concerning u, but in order to obtain universal composability, all the shares u_i and v_j still need to be reshared.

The new multiplication protocol is presented in Algorithm 2.

Theorem 2. *Algorithm 2 is correct and secure against a passive attacker.*

Proof. For correctness we note that

$$\begin{aligned} w = w_1 + w_2 + w_3 &= u'_1v'_1 + u'_1v'_3 + u'_3v'_1 + \\ &\quad u'_2v'_2 + u'_2v'_1 + u'_1v'_2 + u'_3v'_3 + u'_3v'_2 + u'_2v'_3 \\ &= (u'_1 + u'_2 + u'_3)(v'_1 + v'_2 + v'_3) \\ &= (u_1 + u_2 + u_3)(v_1 + v_2 + v_3) = uv\,. \end{aligned}$$

To prove security, we note that Algorithm 2 is symmetric for all the parties. Thus, it will be enough to consider just the incoming view of $\mathcal{P}_1$, which consists of just two values u'_3 and v'_3. Both incoming values are uniformly distributed and independent of the private inputs other than u_1, v_1 due to Lemma 1. Therefore, the protocol is secure since we can build a perfect simulator by generating uniformly distributed values.

We will use the standard arithmetic shorthand and write $[\![w]\!] \leftarrow [\![u]\!] \cdot [\![v]\!]$ to mean $[\![w]\!] \leftarrow \mathsf{Mult}([\![u]\!], [\![v]\!])$. We note that this protocol works over any ring, so we can also use it for $\wedge$ operation for shares from $\mathbb{Z}_2$. In that case we will also use a shorthand notation to write $[\![u]\!] \wedge [\![v]\!]$ instead of calling the multiplication protocol. We stress, however, that while $+$ and $\oplus$ require only local addition of the shares and are thus essentially free, both $\cdot$ and $\wedge$ *require communication* and are hence considerably more costly.

Even though the description of the multiplication protocol involves two rounds of sending different values between the computing parties, the resharing round can be carried out as precomputation since it is independent of inputs $[\![u]\!]$ and $[\![v]\!]$. Hence the real overhead of multiplication is just one round. It is actually possible to combine the two rounds of communication into a single round, but this provided for no noticeable increase in performance in implementation so we omit the details.

5 Bit-level Protocols

Most of the high-level protocols presented in Sections 6 and 7 depend on low-level bit operations. Since the SHAREMIND virtual machine operates on values shared additively over $\mathbb{Z}_{2^n}$, accessing the bits of the shared value is a non-trivial problem.

The first step for all the protocols in the current Section is to consider the shares $u_1, u_2, u_3 \in \mathbb{Z}_{2^n}$ as the elements $\overline{u_1}, \overline{u_2}, \overline{u_3}$ of the ring $(\mathbb{Z}_2)^n$ and carry out all the operations bitwise. Recall that the value $\overline{[\![u]\!]}$ represented by the shares $\overline{u}_1, \overline{u}_2, \overline{u}_3$ is, generally speaking, not equal to u when converted back to $\mathbb{Z}_{2^n}$, since it does not take into account the carry bits that occur during addition.

The bit-level protocols used in SHAREMIND utilize the basic principles of digital circuit design [17] and build on the general bit extraction framework proposed by Damgård *et al.* [9].

One elementary protocol used is $[\![b]\!] \leftarrow \mathsf{BitConj}(\overline{[\![u]\!]})$ for finding the conjunction of all the bits of a bitwise shared vector $\overline{[\![u]\!]}$ and representing the result as a shared bit $[\![b]\!]$. This protocol can be implemented using a natural recursive split-in-half approach and achieves round complexity logarithmic in the input vector length.

Another protocol we need is $\overline{[\![s]\!]} \leftarrow \mathsf{CarryBits}(\overline{v}, \overline{[\![r]\!]})$, where the value $\overline{v} \in \mathbb{Z}_{2^n}$ is known by $\mathcal{P}_2$ and $\mathcal{P}_3$, the value r is shared bitwise over $(\mathbb{Z}_2)^n$ between $\mathcal{P}_2, \mathcal{P}_3$ (so

Algorithm 3: $\overline{[\![p']\!]} \leftarrow \mathsf{PrefixOR}(\overline{[\![p]\!]})$.

Data: Bitwise shared vector $\overline{[\![p]\!]}$.
Result: The vector $\overline{[\![p']\!]}$ which has the form $00\ldots011\ldots1$, where the initial part $00\ldots01$ coincides with the vector originally represented by $\overline{[\![p]\!]}$.

1 $l \leftarrow |\overline{[\![p]\!]}|$.
2 **if** $l = 1$ **then**
3 Return $\overline{[\![p']\!]} \leftarrow \overline{[\![p]\!]}$.
4 **else**
5 $\overline{[\![p']\!]}^{(l-1\ldots\lfloor l/2\rfloor)} \leftarrow \mathsf{PrefixOR}(\overline{[\![p]\!]}^{(l-1\ldots\lfloor l/2\rfloor)})$.
6 $\overline{[\![p']\!]}^{(\lfloor l/2\rfloor-1\ldots0)} \leftarrow \mathsf{PrefixOR}(\overline{[\![p]\!]}^{(\lfloor l/2\rfloor-1\ldots0)})$.
7 **for** $i \leftarrow 0$ **to** $\lfloor l/2\rfloor - 1$ **do**
8 $\overline{[\![p']\!]}^{(i)} \leftarrow \overline{[\![p']\!]}^{(i)} \vee \overline{[\![p']\!]}^{(\lfloor l/2\rfloor)}$.
9 **end**
10 Return $\overline{[\![p']\!]}$.
11 **end**

$\overline{r}_1 = 0$), and the output is a shared vector $\overline{[\![s]\!]}$ representing the carries occurring when the addition $v+r$ is performed. The protocol can be implemented exactly as in [4] using carry look-ahead technique which also works in a logarithmic number of rounds.

Whenever a bit-level protocol needs to $\wedge$ together two bit values, the (perfectly simulatable) protocol $\mathsf{Mult}(\cdot,\cdot)$ can be called. The security proofs for protocols $\mathsf{CarryBits}(\cdot,\cdot)$ and $\mathsf{BitConj}(\cdot)$ are standard applications of universal composability Lemmas 1 and 2 of [3] and we will skip them here.

As an example of a more involved bit-level protocol, we will present and analyze the protocol for finding the most significant non-zero bit position of a bitwise shared value $\overline{[\![u]\!]}$.

We will proceed in two steps. First, we set all the bits after the first 1 to be 1 as well by a recursive prefix-OR procedure presented as Algorithm 3. In the procedure, let $|\overline{[\![p]\!]}|$ denote the number of bits that the vector represented by $\overline{[\![p]\!]}$ contains. Also, let $\overline{[\![p]\!]}^{(i\ldots j)}$ denote the shared subvector containing the bits represented by $\overline{[\![p]\!]}^{(i)}, \ldots, \overline{[\![p]\!]}^{(j)}$ (note that we will write more significant bits to the left, so in this notation $i \geq j$).

Note that the $\log_2 n$ recursive calls of Algorithm 3 require one round of multiplication each (to compute $\vee$ of the shared bits $[\![b_1]\!]$ and $[\![b_2]\!]$ as $[\![b_1]\!] \oplus [\![b_2]\!] \oplus ([\![b_1]\!] \wedge [\![b_2]\!])$), hence the overall round complexity of Algorithm 3 is $\log_2 n$.

To compute the most significant bit, it now suffices to zero all the bits that are to the right of the first 1. This can be done by a simple loop given on the lines 2 to 3 of the main protocol described as Algorithm 4. As this computation is local and requires no communication, the complexity is the same as for Algorithm 3.

Theorem 3. *Algorithm 4 is correct and secure against one passive attacker.*

Proof. Correctness of the protocol follows directly from the discussion given above.

Security of the protocol is trivial as well, since we are only composing perfectly simulatable primitives.

Algorithm 4: Protocol for the most significant non-zero bit position $\overline{[\![s]\!]} \leftarrow \mathsf{MSNZB}(\overline{[\![u]\!]})$.

Data: Bitwise shared value $\overline{[\![u]\!]}$.

Result: Shared vector $\overline{[\![s]\!]}$ such that $\overline{[\![s]\!]}^{(j)}$ represents 1, where j is the most significant position, where $\overline{[\![u]\!]}^{(j)}$ represents 1, and 0 otherwise. If all the bits of $\overline{[\![u]\!]}$ represent 0, all the shared bits of $\overline{[\![s]\!]}$ represent 0 as well.

1 $\overline{[\![u']\!]} \leftarrow \mathsf{PrefixOR}(\overline{[\![u]\!]})$.
2 **for** $i \leftarrow 0$ **to** $n-2$ **do**
3 $\quad \overline{[\![s]\!]}^{(i)} \leftarrow \overline{[\![u']\!]}^{(i)} \oplus \overline{[\![u']\!]}^{(i+1)}$.
4 **end**
5 $\overline{[\![s]\!]}^{(n-1)} \leftarrow \overline{[\![u']\!]}^{(n-1)}$.
6 Return $\overline{[\![s]\!]}$.

6 Improved High-level Protocols

We now describe the new and improved high-level protocols for the framework: the operators for share conversion, bit extraction and equality. We have also developed protocols for bit shift under a public shift which are presented in Appendix A.

6.1 Share conversion

Bit-level operations are typically used as building blocks within algorithms working with the full shares over $\mathbb{Z}_{2^n}$. Hence, the problem of converting the bits shared over $\mathbb{Z}_2$ to shares over $\mathbb{Z}_{2^n}$ arises. Converting individual shares locally does not solve the problem, since we will lose reduction modulo 2. Hence, a different approach is needed.

The routine presented as Algorithm 5 first splits the bit u as $u = m \oplus s$ so that $m = b \oplus u_1$ for a random bit b. The bit m can then be directly converted to $\mathbb{Z}_{2^n}$ by one party and the value of s can be used to select whether the real value of u should be $1 - m$ or m.

Theorem 4. *Algorithm 5 is correct and secure against one passive attacker.*

Proof. For correctness, we first note that

$$u = u_1 \oplus u_2 \oplus u_3 = b \oplus m \oplus b_{12} \oplus s_{23} \oplus b_{13} \oplus s_{32} = m \oplus s\,.$$

Hence, if $s = 1$, we have $v = v_1 + v_2 + v_3 = 1 - m_{12} - m_{13} = 1 - m$, which is equal to $m \oplus 1$ when embedded to $\mathbb{Z}_{2^n}$. If $s = 0$ we have $v = v_1 + v_2 + v_3 = m_{12} + m_{13} = m$, which is equal to $m \oplus 0$ when embedded to $\mathbb{Z}_{2^n}$.

To prove security, we will consider all the three computing parties and prove that their incoming views can be perfectly simulated. The view of party $\mathcal{P}_1$ contains no incoming messages, so the corresponding simulator is trivial. The incoming view of $\mathcal{P}_2$ can be perfectly simulated, since using Lemma 1 we see that its distribution

$$(\!|m_{12}, b_{12}, b \oplus b_{12} \oplus u_3|\!) = (\!|m_{12}, b_{12}, b|\!)$$

Algorithm 5: Protocol $[\![v]\!] \leftarrow \mathsf{ShareConv}([\![u]\!])$ for converting a share $[\![u]\!] \in \mathbb{Z}_2$ to $[\![v]\!] \in \mathbb{Z}_{2^n}$.

Data: Shared value $[\![u]\!]$ in bit shares.
Result: Shared value $[\![v]\!]$ such that $u = v$ and $[\![v]\!]$ is shared in $\mathbb{Z}_{2^n}$.

$\mathcal{P}_1$ generates random $b \leftarrow \mathbb{Z}_2$ and sets $m \leftarrow b \oplus u_1$.
$\mathcal{P}_1$ locally converts m to $\mathbb{Z}_{2^n}$, generates random $m_{12} \leftarrow \mathbb{Z}_{2^n}$ and computes $m_{13} = m - m_{12}$.
$\mathcal{P}_1$ generates random $b_{12} \leftarrow \mathbb{Z}_2$ and computes $b_{13} = b - b_{12} = b \oplus b_{12}$.
All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
$\mathcal{P}_2$ sets $s_{23} \leftarrow b_{12} \oplus u_2$.
$\mathcal{P}_3$ sets $s_{32} \leftarrow b_{13} \oplus u_3$.
All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
$\mathcal{P}_1$ sets $v_1 \leftarrow 0$
$\mathcal{P}_2$ and $\mathcal{P}_3$ set $s \leftarrow s_{23} \oplus s_{32}$
if $s = 1$ **then**
    $\mathcal{P}_2$ sets $v_2 \leftarrow (1 - m_{12})$.
    $\mathcal{P}_3$ sets $v_3 \leftarrow (-m_{13})$.
else
    $\mathcal{P}_2$ sets $v_2 \leftarrow m_{12}$.
    $\mathcal{P}_3$ sets $v_3 \leftarrow m_{13}$.
end
Return $[\![v]\!]$.

is independent of private inputs other than u_2.

Similarly, the incoming view of $\mathcal{P}_3$ can be perfectly simulated, since using Lemma 1 we see that its distribution

$$\begin{aligned} &(\!|b \oplus u_1 - m_{12}, b \oplus b_{12}, b_{12} \oplus u_2|\!) \\ &= (\!|m_{12}, b \oplus b_{12}, b_{12} \oplus u_2|\!) \\ &= (\!|m_{12}, b, b_{12} \oplus u_2|\!) \\ &= (\!|m_{12}, b, b_{12}|\!) \end{aligned}$$

is independent of private inputs other than u_3. Furthermore, the values m_{12}, b_{12} and b are uniformly distributed so we can build a perfect simulator for $\mathcal{P}_2$ and $\mathcal{P}_3$ and show security.

6.2 Bit extraction

In order to perform bit-level computations, we first need to extract the bits, which is a non-trivial task for shared values. The basic working principle of Algorithm 6 is the same as of the bitwise addition protocol explained in [4]. The initial value u is represented as the sum $v + r$, where r is a random value with a known shared bitwise decomposition. We can then use the carry look-ahead algorithm to determine the carry bits that occur in the addition $v + r$ and use them to compute the bits of u.

Theorem 5. *Algorithm 6 is correct and secure against one passive attacker.*

Algorithm 6: Protocol $\overline{[\![w]\!]} \leftarrow \mathsf{BitExtr}([\![u]\!])$ for bit extraction.

Data: Additively shared value $[\![u]\!]$.
Result: Bitwise shared vector $\overline{[\![w]\!]}$ representing the bits of u.

1 $\mathcal{P}_1$ generates random $r, r_2 \leftarrow \mathbb{Z}_{2^n}$, $\overline{q}_2 \leftarrow (\mathbb{Z}_2)^n$, sets $\overline{q}_1 = \overline{0}$ and computes $r_3 \leftarrow u_1 - r - r_2$, $\overline{q}_3 \leftarrow \overline{r} \oplus \overline{q}_2$.
2 $\mathcal{P}_1$ sends $r_i, \overline{q}_i$ to $\mathcal{P}_i$ $(i = 2, 3)$.
3 $\mathcal{P}_i$ $(i = 2, 3)$ computes the share $v_i \leftarrow u_i + r_i$ and sends it to $\mathcal{P}_{6/i}$.
4 $\mathcal{P}_2, \mathcal{P}_3$ compute $v = v_2 + v_3$.
5 $\overline{[\![s]\!]} \leftarrow \mathsf{CarryBits}(\overline{v}, \overline{[\![q]\!]})$.
6 Define $s_i^{(-1)} = 0$, $(i = 1, 2, 3)$.
7 **for** $j \leftarrow 0$ **to** $n - 1$ **do**
8 In $\mathcal{P}_1$: $\overline{w}_1^{(j)} \leftarrow \overline{s}_1^{(j-1)}$.
9 In $\mathcal{P}_2$: $\overline{w}_2^{(j)} \leftarrow \overline{v}^{(j)} \oplus \overline{q}_2^{(j)} \oplus \overline{s}_2^{(j-1)}$.
10 In $\mathcal{P}_3$: $\overline{w}_3^{(j)} \leftarrow \overline{q}_3^{(j)} \oplus \overline{s}_3^{(j-1)}$.
11 **end**
12 Return $\overline{[\![w]\!]}$.

Proof. During the initial stage, u is represented as

$$\begin{aligned} u &= u_1 + u_2 + u_3 = (r + r_2 + r_3) + (v_2 - r_2) + (v_3 - r_3) \\ &= v_2 + v_3 + r = v + r\,, \end{aligned}$$

where r has a known shared bit decomposition $r = \overline{q}_2 \oplus \overline{q}_3$. Thus, in order to find the bits of u, we can use bitwise addition to compute the bits of $v + r$. To do that, one needs to compute the carry bits, and this is done by calling the Algorithm $\mathsf{CarryBits}(\cdot, \cdot)$ (laid out in [4]).

As a result, the bitwise shared vector $\overline{[\![s]\!]}$ will represent exactly the carry bits from the corresponding positions when computing $v + r$, hence it remains to add these bits to the shared bitwise $\oplus$ of v and r, which is done on lines 6 to 10. Indeed, we see that

$$\begin{aligned} \overline{w}^{(j)} &= \overline{w}_1^{(j)} \oplus \overline{w}_2^{(j)} \oplus \overline{w}_1^{(j)} \\ &= \overline{s}_1^{(j-1)} \oplus \overline{v}^{(j)} \oplus \overline{q}_2^{(j)} \oplus \overline{s}_2^{(j-1)} \oplus \overline{q}_3^{(j)} \oplus \overline{s}_3^{(j-1)} \\ &= \overline{v}^{(j)} \oplus \overline{r}^{(j)} \oplus \overline{s}^{(j-1)}\,. \end{aligned}$$

To prove security, we will consider all three computing parties and prove that their incoming views can be perfectly simulated. The incoming view of party $\mathcal{P}_1$ contains no other incoming messages than the ones determined by Algorithm $\mathsf{CarryBits}(\cdot, \cdot)$, which can be perfectly simulated. The incoming view of $\mathcal{P}_2$ looks mostly the same as that of $\mathcal{P}_1$, only the initial part differs. We use Lemma 1 again to see that

$$(\!|r_2, \overline{q}_2, u_3 + r_3, \ldots|\!) = (\!|r_2, \overline{q}_2, r_3, \ldots|\!),$$

which does not depend on any input values and can hence be perfectly simulated.

Algorithm 7: Protocol $[\![w]\!] \leftarrow \mathsf{Equal}([\![u]\!], [\![v]\!])$ for evaluating the equality predicate.

Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shared value $[\![w]\!]$ such that $w = 1$ if $u = v$, and 0 otherwise.

1 $\mathcal{P}_1$ generates random $r_2 \leftarrow \mathbb{Z}_{2^n}$ and computes $r_3 \leftarrow (u_1 - v_1) - r_2$.
2 $\mathcal{P}_1$ sends r_i to $\mathcal{P}_i$ $(i = 2, 3)$.
3 $\mathcal{P}_i$ computes $e_i = (u_i - v_i) + r_i$ $(i = 2, 3)$.
4 $\mathcal{P}_1$ sets $\overline{p}_1 \leftarrow 2^n - 1 = 111 \ldots 1$.
5 $\mathcal{P}_2$ sets $\overline{p}_2 \leftarrow e_2$.
6 $\mathcal{P}_3$ sets $\overline{p}_3 \leftarrow (0 - e_3)$.
7 Return $[\![w]\!] \leftarrow \mathsf{BitConj}(\overline{[\![p]\!]})$.

Similarly, for party $\mathcal{P}_3$ we have the initial part of the incoming view

$$\begin{aligned} &(\!|u_1 - r - r_2, \overline{r} \oplus \overline{q}_2, u_2 + r_2, \ldots|\!) \\ &= (\!|u_1 - r - r_2, \overline{q}_2, u_2 + r_2, \ldots|\!) \\ &= (\!|r, \overline{q}_2, u_2 + r_2, \ldots|\!) \\ &= (\!|r, \overline{q}_2, r_2, \ldots|\!), \end{aligned}$$

which does not depend on any input values once again.

We note that this protocol can also be used to improve the efficiency of comparison protocols which usually use bit extraction as a subroutine [4].

6.3 Equality testing

Equality testing can be accomplished fairly easily via bit extraction. However, since equality comparison is used quite often in practical applications, it makes sense to provide a separate and more efficient protocol specifically designed for that task. Algorithm 7 first shares the difference $u - v$ as $e_2 + e_3$ between $\mathcal{P}_2$ and $\mathcal{P}_3$. Then it remains to determine, whether $e_2 + e_3 = 0$, which can be done by comparing e_2 and $-e_3$ bitwise.

Theorem 6. *Algorithm 7 is correct and secure against one passive attacker.*

Proof. For correctness note first that

$$\begin{aligned} e_2 + e_3 &= (u_2 - v_2) + r_2 + (u_3 - v_3) + r_3 \\ &= (u_2 - v_2) + (u_3 - v_3) + (u_1 - v_1) = u - v\,, \end{aligned}$$

hence $u = v$ iff $e_2 = 0 - e_3$. Algorithm 7 compares u and v by comparing $\overline{p}_2 = e_2$ and $\overline{p}_3 = (0 - e_3)$ bitwise. For that we analyze the bitwise sum (XOR) of $\overline{p}_1 = 2^n - 1 = 111 \ldots 1$, $\overline{p}_2$ and $\overline{p}_3$. We see that $u = v$ iff all the bits represented $\overline{[\![p]\!]}$ are 1, which is exactly the case when the conjunction $[\![w]\!] = \wedge_{i=0}^{n-1} \overline{[\![p]\!]}^{(i)}$ is 1. This is exactly what is verified by calling Algorithm $\mathsf{BitConj}(\cdot)$.

To prove security, we will consider all the three computing parties and prove that their incoming views can be perfectly simulated.

The incoming view of party $\mathcal{P}_1$ coincides with its view in Algorithm $\mathsf{BitConj}(\cdot)$ and can hence be simulated. The incoming view of $\mathcal{P}_2$ is almost equivalent to that of $\mathcal{P}_1$ with the only exception of receiving one extra independently and uniformly chosen element r_2, which is trivial to simulate. The same holds for $\mathcal{P}_3$ who receives $r_3 = (u_1 - v_1) - r_2$ which can be replaced by r_2 by Lemma 1.

7 Division protocols

In this paper we introduce two division protocols – one where the divisor is a public constant and the other, where the divisor is a shared value. The protocols make use of two subroutines $\mathsf{ReshareToTwo}(\cdot)$ and $\mathsf{Overflow}(\cdot)$ meant for resharing a value to two parties and for computing the overflow bit once the values are shared in this way, respectively. As both protocols are fairly straightforward, the exact details for them are given in Appendix A.

7.1 Division by a public value

The main idea for the division protocol comes from [14] and essentially consists of publicly finding the inverse $d' = 1/d$ of the divisor d and then multiplying the dividend a with the previously found inverse value d'. This trick reduces division to multiplication by a constant, which is often used on normal processors to speed up batch division of many numbers with a single value. In our case, however, it allows to do the division publicly and then only perform a secret multiplication, which is fairly efficient.

Since we have access to only integer arithmetic, it makes sense to denote the inverse value $d' \approx c2^{-k}$ where we choose k in such a way that c is an integer. It is shown in [14] that one can choose c and k in such a way that the final outcome w of the protocol is equal to $\lfloor \frac{u}{d} \rfloor$. All that is left to do is to compute the division by multiplying c and u and then shifting the result cu right k positions.

It is shown in [14] that if we are working with integers from $\mathbb{Z}_{2^n}$, it suffices to choose $k = n + 1$ and the problem can thus be reduced to just finding the highest n bits of the $2n + 1$ bit multiplication result.[5]

In our setting, multiplying two n bit values means that the higher bits are thrown away. To avoid that, both values would temporarily need to be converted to $2n + 1$-bit values, then multiplied and then have the result truncated back to n bit value.

Converting a secret n bit value $[\![u]\!]$ into a value of $m > n$ bits requires that we know whether the sum $u_1 + u_2 + u_3$ produces a carry into the higher order bits when viewed in $\mathbb{Z}_{2^m}$. We can use Algorithm 11 to obtain the carry bit. We can then carry out the multiplication and use Algorithm 11 again to perform the truncation.[6]

[5] In [14] the authors actually transform the problem so that it is enough to use just $2n$ bits. However, the transformation assumes that bit shifts are cheap, making it impractical in the current MPC setting.

[6] We do not need to use Algorithm 12 because we do not introduce new digits on the left like we would in the case of a normal bit shift.

Algorithm 8: Protocol $[\![w]\!] \leftarrow \mathsf{PubDiv}([\![u]\!], d)$ for division by a public value d.

Data: Shared value $[\![u]\!]$ and a public divisor d.
Result: Shares $[\![w]\!]$ of the value $w = \lfloor \frac{u}{d} \rfloor$.

1 $[\![u']\!] \leftarrow \mathsf{ReshareToTwo}(u)$.
2 $\mathcal{P}_2, \mathcal{P}_3$ find $c \in \mathbb{Z}_{2^{n+1}}$ such that $c2^{-(n+1)} \approx \frac{1}{d}$ as in [14].
3 $\mathcal{P}_1$ sets $v_1^1, v_1^2 \leftarrow 0 \in \mathbb{Z}_{2^{2n+1}}$.
4 $\mathcal{P}_2$ sets $v_2^1 \leftarrow cu_2'$, $v_2^2 \leftarrow c(u_2' - 2^n) \in \mathbb{Z}_{2^{2n+1}}$.
5 $\mathcal{P}_3$ sets $v_3^1, v_3^2 \leftarrow cu_3' \in \mathbb{Z}_{2^{2n+1}}$.
6 $[\![\lambda]\!] \leftarrow \mathsf{Overflow}([\![u']\!])$.
7 $[\![\lambda^1]\!] \leftarrow \mathsf{Overflow}([\![v^1 \ll n]\!])$.
8 $[\![\lambda^2]\!] \leftarrow \mathsf{Overflow}([\![v^2 \ll n]\!])$.
9 Every $\mathcal{P}_i$ sets $w_i^1 \leftarrow v_i^1 \gg (n+1)$ and $w_i^2 \leftarrow v_i^2 \gg (n+1)$.
10 Return $[\![w]\!] \leftarrow (1 - [\![\lambda]\!])([\![w^1]\!] + [\![\lambda^1]\!]) + [\![\lambda]\!]([\![w^2]\!] + [\![\lambda^2]\!])$.

However, the crucial observation is that we can parallelize determining the carry bit and truncation – since there are just two different choices for the carry and we can just compute the results for both and only decide in the end which of the two to use. This approach leads to Algorithm 8.

Theorem 7. *Algorithm 8 is correct and secure against one passive attacker.*

Proof. In order to compute $\frac{u}{d} = u \cdot \frac{1}{d}$, the algorithm first chooses c such that $\frac{1}{d} \approx c2^{-(n+1)}$ and then computes cu. After running Algorithm 10, we have either $u = u_2' + u_3'$ or $u = u_2' + u_3' - 2^n$. Hence the values

$$\begin{aligned} v^1 &= v_1^1 + v_2^1 + v_3^1 = 0 + cu_2' + cu_3' = c(u_2' + u_3') \text{ and} \\ v^2 &= v_1^2 + v_2^2 + v_3^2 = 0 + c(u_2' - 2^n) + cu_3' \\ &= c(u_2' + u_3' - 2^n) \end{aligned}$$

are the two candidates for cu. Now it remains to divide both of these values by 2^{n+1} and choose the correct one. The division is performed by right shift on line 9 and the correct value is chosen based on the value of the bit λ on line 10. Note that when performing the right shift, we may still need to add 1 to compensate for the carry we lose when truncating; this is achieved by running Algorithm 10 first to reshare the values $[\![v^i]\!] \ll (n+1)$ and then Algorithm 11 in order to obtain known carries.

To prove security we note that sending messages only occurs within subprotocols proven above to be perfectly simulatable, hence building the required simulator is trivial.

7.2 Division by a shared value

In order to implement the protocol we will use the Goldschmidt iteration method, which is an adaptation of Newton iteration designed especially for efficient implementation in digital computers. When dividing u by v, the algorithm keeps track of N and D so that

$\frac{N}{D} = \frac{u}{v}$ but where $D \to 1$ as the number of iterations increases, which guarantees that $N \to \frac{u}{v}$ as well.

To be precise, the method works by starting with $N_0 = c_0 u$ and $D_0 = c_0 v$ where c_0 is a scaling constant designed to guarantee $0.5 \leq D_0 < 1$. In each step of the iteration, a scaling coefficient $F_i = 2 - D_{i-1}$ is computed after which the new values $D_i = F_i D_{i-1}$ and $N_i = F_i N_{i-1}$ can be calculated.

Goldschmidt iteration method has many desirable properties. First, it was conceived with parallelism in mind, so that the two multiplications in each iteration can be done in parallel. Secondly, it has quadratic convergence, which means that if $0.5 \leq D_0 < 1$ then $1 - 2^{-2^i} \leq D_i < 1$ for all $i \geq 0$. Consequently, the relative error $\frac{u/v - N_i}{u/v} = 1 - D_i \in (0, 2^{-2^i}]$, implying that $\log_2 n$ iterations suffice for n bits of precision (see Appendix B for details). Thirdly, the convergence is monotonic so that $N_0 < N_1 < \ldots < N_i < \ldots < \frac{u}{v}$. This becomes crucial when one is interested in division that always rounds in a fixed direction (i.e. either always upwards or downwards).

In practice, the method is most often used for floating-point division. However, analogously to the public division, it is pretty straightforward to convert everything to purely integer arithmetic by simply emulating fix-point arithmetic with corresponding integer operations followed by the appropriate bit shifts. The details of such an approach can be found in [18].

However, some interesting technical problems arise when attempting an efficient implementation of such an iteration method within SHAREMIND. The key difference between the standard model and our MPC setting is the cost of bit operations – they are virtually costless in the standard model, but extremely costly in SHAREMIND.

This causes the most problems for computing the initial scaling constant c_0, which is usually done by just finding the most significant bit position h_v of v and taking $c_0 = 2^{-h_v - 1}$. We will follow the same approach (combining Algorithms 6 and 4), but note that doing so is very costly – finding c_0 constitutes roughly one third of the whole protocol in terms of both round and communication complexity. Nevertheless, there seems to be no obvious way around it as the iteration methods converge very slowly if an initial estimate of comparable quality is not used and the relevant literature does not seem to discuss any other methods of finding such an estimate.

Second problem arises when we note that emulating fractional multiplication requires an efficient shift right operator. However, this is again something that the current framework does not provide, as Algorithm 12 is rather costly. The same is true for modulus expansion, which is required to get the high bits of the multiplication result.

However, these problems can be solved fairly efficiently. Firstly, the ring is expanded to m bits just once before the iterations, and m is simply chosen large enough so that no further expansions would be necessary. Since this can be done in parallel with finding c_0, doing so is essentially free in terms of rounds. Since exact truncation is also very expensive, we will replace it with an imprecise one where all the shares are truncated individually without worrying about the possible carry bit. This introduces additional imprecision into the computation, but that can be dealt with by slightly increasing the precision of the arithmetic from 2^n to $2^{n'}$ and adding an additional iteration. The details of the corresponding error analysis and the details of the choice of m and n' are presented in Appendix B. Using these two tricks brings the cost of each itera-

tion step down to just two parallel (large-modulus) multiplications, making it relatively fast and efficient.

Additional care needs to be taken to enforce strict downward rounding. As mentioned before, Goldschmidt iteration ensures monotonic convergence from below. This means that N will always be less than the real value $\frac{u}{v}$, which also means that generally we expect $\lfloor N \rfloor = \lfloor \frac{u}{v} \rfloor$ to hold. However, there are cases where we can get $\lfloor N \rfloor < \lfloor \frac{u}{v} \rfloor$. This can be easily fixed by adding a suitably small value Δ to N before truncation. The details of choosing Δ are presented in Appendix B along with the analysis of error terms introduced by imprecise truncation during the iteration steps.

To make convergence faster, we will alter the first iteration a bit by setting $F_1 = 2\sqrt{2} - 2D_0$, which is standard practice for Newton iteration, but somewhat less used in the case of Goldschmidt algorithm. Assuming $\sqrt{2} - 1 < D_0 < 1$, it is easy to see that $2\sqrt{2} - 2 < D_1 < 1$. Note that $\sqrt{2} - 1 \approx 0.41 < 0.5$, but $2\sqrt{2} - 2 \approx 0.83$, which gives us a better estimate compared to $1 - 2^{-2^1} = 0.75$ provided by the original first iteration of Goldschmidt method. This will guarantee sufficient extra precision to compensate for the additional errors introduced with truncation, so no extra iteration step is needed.

The protocol for division is formalized as Algorithm 9. In this protocol, we will be working with values from several different domains. The inputs u, v and the output w belong to $\mathbb{Z}_{2^n}$. At the first stage, the input values and the initial approximation c_0 will be converted to $\mathbb{Z}_{2^m}$ using the procedures $\mathsf{ShareConv}^{\mathsf{m}}(\cdot)$ and $\mathsf{Overflow}^{\mathsf{m}}(\cdot)$. They behave exactly as $\mathsf{ShareConv}(\cdot)$ and $\mathsf{Overflow}(\cdot)$ with their outputs considered to be shared over $\mathbb{Z}_{2^m}$.

The intermediate values are essentially fixed point numbers of the form $2^{-n'} \cdot \widehat{x}$ with $\widehat{x} \in \mathbb{Z}_{m'}$ for some m'. When such values are multiplied, we also get numbers of the form $2^{-2n'} \cdot \widehat{\widehat{x}} \in \mathbb{Z}_{m''}$. Since SHAREMIND can only handle integer values, these numbers will be represented by the values $\widehat{x}$ and $\widehat{\widehat{x}}$, respectively. In order to retain constant precision, the numbers $2^{-2n'} \cdot \widehat{\widehat{x}} \in \mathbb{Z}_{m''}$ need to be converted back to the form $2^{-n'} \cdot \widehat{x}$. This is done by right-shifting all the shares of $\widehat{\widehat{x}}$ by n' positions and rounding, if necessary (see line 13). As a result, the modulus will also be decreased by n', i.e. $m' = m'' - n'$. This truncation will introduce rounding errors; see Appendix B for details on how they are accounted for.

Theorem 8. *Algorithm 9 is correct and secure against one passive attacker.*

Proof. Correctness directly follows from the discussion above and error computations presented in Appendix B. Security of the protocol is trivial as well, since we are only using perfectly simulatable building blocks.

8 Performance analysis

8.1 Complexity of protocols

Communication and round complexities of the described protocols are presented in Table 1. Here, $\ell = \log_2 n$ and the details of selecting n' and m for the general division protocol are presented in Appendix B.

Algorithm 9: Protocol $[\![w]\!] \leftarrow \mathsf{Div}([\![u]\!], [\![v]\!])$ for division.

Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shares $[\![w]\!]$ such that $w = \lfloor \frac{u}{v} \rfloor$.

$\overline{[\![v']\!]} \leftarrow \mathsf{BitExtr}([\![v]\!])$.
$\overline{[\![s]\!]} \leftarrow \mathsf{MSNZB}(\overline{[\![v']\!]})$.
Compute $[\![c^i]\!] \leftarrow \mathsf{ShareConv}^{\mathsf{m}}(\overline{[\![s]\!]}^{(i)})$ $(i = 0, \ldots, n-1)$.
Set $[\![\widehat{c_0}]\!] \leftarrow \sum_{i=0}^{n-1} 2^{n'-i-1}[\![c^i]\!]$.
$[\![u']\!] \leftarrow \mathsf{ReshareToTwo}([\![u]\!])$, $[\![\lambda^1]\!] \leftarrow \mathsf{Overflow}^{\mathsf{m}}([\![u']\!])$.
$[\![v']\!] \leftarrow \mathsf{ReshareToTwo}([\![v]\!])$, $[\![\lambda^2]\!] \leftarrow \mathsf{Overflow}^{\mathsf{m}}([\![v']\!])$.
$[\![u'']\!] \leftarrow [\![u']\!] - 2^n[\![\lambda^1]\!] \in \mathbb{Z}_{2^m}$.
$[\![v'']\!] \leftarrow [\![v']\!] - 2^n[\![\lambda^2]\!] \in \mathbb{Z}_{2^m}$.
Compute $[\![\widehat{N_0}]\!] \leftarrow [\![u'']\!] \cdot [\![\widehat{c_0}]\!]$ and $[\![\widehat{D_0}]\!] \leftarrow [\![v'']\!] \cdot [\![\widehat{c_0}]\!]$.
Set $[\![\widehat{F_1}]\!] \leftarrow \lfloor 2^{n'} \cdot 2\sqrt{2} \rfloor - 2[\![\widehat{D_0}]\!]$.
Compute $[\![\widehat{\widehat{N_1}}]\!] \leftarrow [\![\widehat{N_0}]\!] \cdot [\![\widehat{F_1}]\!]$ and $[\![\widehat{\widehat{D_1}}]\!] \leftarrow [\![\widehat{D_0}]\!] \cdot [\![\widehat{F_1}]\!]$.
for $k \leftarrow 1$ **to** $\log_2 n$ **do**
    Each party $\mathcal{P}_i$ computes $(\widehat{N_k})_i \leftarrow ((\widehat{\widehat{N_k}})_i \gg n') + 1$ and $(\widehat{D_k})_i \leftarrow (\widehat{\widehat{D_k}})_i \gg n'$ $(i = 1, 2, 3)$.
    Set $[\![\widehat{F_{k+1}}]\!] \leftarrow 2^{n'} \cdot 2 - [\![\widehat{D_k}]\!]$.
    Compute $[\![\widehat{\widehat{N_{k+1}}}]\!] \leftarrow [\![\widehat{N_k}]\!] \cdot [\![\widehat{F_{k+1}}]\!]$ and $[\![\widehat{\widehat{D_{k+1}}}]\!] \leftarrow [\![\widehat{D_k}]\!] \cdot [\![\widehat{F_{k+1}}]\!]$.
end
Compute $[\![R]\!] \leftarrow [\![\widehat{\widehat{N_{\log_2 n+1}}}]\!] + \Delta$.
Return $[\![w]\!] \leftarrow \mathsf{ShiftR}^{\mathsf{n+n'}}([\![R]\!], n') \bmod 2^n$.

8.2 Experimental setup

Since we had access to the source code of the SHAREMIND virtual machine, we extended it by implementing the described protocols as primitive operations. We conducted a series of experiments to verify that the new protocols are an improvement over the previous protocols presented in [3].

Since SHAREMIND is designed to be a data mining platform, its instruction set follows the SIMD (single instruction, multiple data) principle. This requires protocols to accept vectors of integers as inputs and provide vectors as outputs. Previous tests conducted on the platform showed that vector operations can be more efficient than single operations. Additionally, we wanted to verify the scalability of the implementation by testing large input vectors. Therefore, we benchmarked each operation with input vectors with sizes ranging from 1 up to 10^8. The input vectors consisted of random values.

The order of the experiments was randomized to reduce the impact of outside factors such as flow control and low-level processes of the operating system. For comparison, we also benchmarked the protocols from [3]. Not all vector sizes could be tested for the old protocols, as their inefficiency overloaded the networking layer and the timeouts caused SHAREMIND to cancel the protocol.

Protocol	Rounds	Communication
Mult	1	$15n$
ShareConv	2	$5n + 4$
Equal	$\ell + 2$	$22n + 6$
ShiftR	$\ell + 3$	$12(\ell + 4)n + 16$
BitExtr	$\ell + 3$	$5n^2 + 12(\ell + 1)n$
PubDiv	$\ell + 4$	$(108 + 30\ell)n + 18$
Div	$4\ell + 9$	$2mn + 6m\ell + 39\ell n + 35\ell n' + 126n + 32n' + 24$

Table 1. Complexities of protocols

Protocol	Single op.	n_{s}	Saturated op.	Old
ShareConv	15.3 ms	24000	0.8 μs	18 μs
Mult	25.9 ms	15000	1.8 μs	3.5 μs
Equal	101 ms	27000	5.0 μs	2225 μs
BitExtr	113 ms	2600	51 μs	1426 μs
ShiftR	122 ms	12000	15.7 μs	-
PubDiv	124 ms	3500	44 μs	-
Div	390 ms	800	534 μs	-

Table 2. Overview of experimental results

We performed the benchmarks on a high performance computation cluster. The servers run the Debian Linux operating system, contain 12-core Intel® Xeon® processors, have 48 GB of memory and are connected with network interface cards allowing for speeds up to 1Gb/s. We used three of these servers to run the SHAREMIND virtual machine.

SHAREMIND can also run successfully, albeit with lower performance, on weaker hardware and with smaller communication channels. We note, that on each machine, SHAREMIND used one core, leaving the other cores with no load. This is because the performance of SHAREMIND is communication-bound, as opposed to circuit-based solutions. Further experiments must be conducted to measure the effect of weaker communication channels on the computation speed.

8.3 Benchmark results

After conducting the experiments, we fitted the protocol execution times using linear regression. Two distinct lines emerged. One of the regression lines was a fit for input vector size smaller than the point n_{s} and the other for input vector sizes larger than n_{s}. This point n_{s} was identified for each protocol. We call this point the *saturation point* of a protocol.

According to our tests, the saturation point depends on the communication complexity of the protocol. For smaller input vectors, the required network messages fit into the network channel without fragmentation. When the network traffic exceeds the available bandwidth, the flow control algorithms on the SHAREMIND network layer start to work.

However, this reduces the efficiency of the protocol and further growth is characterized by a different linear function.

A direct result of this is that practical applications should try to run each protocol with vector sizes equal or larger to the saturation point. This will guarantee that network is used to its full capabilities and private operations are run on their maximum efficiency.

We note that for input sizes larger than 10^7 the implementation started using large amounts of memory and this affected the running time. This can be seen best on Figure 3 where a third line seems to emerge. However, since we propose that the implementation of algorithms run the algorithms in batches with the size of the the saturation point n_{s}, we do not consider the performance of huge vectors a major issue. It may also be reduced by further fine-tuning of the implementation.

Table 2 presents an overview of the benchmark results. For each benchmarked protocol, the table contains the time needed to process a single input, the estimated input size that causes saturation in the communication channel and finally, the time needed to process a single value in input vectors larger than the saturation point (which is presented in microseconds as it is generally at least a thousand times smaller than the time needed in the case of a single value operation). The values are taken from the linear fits and are therefore estimates.

For comparison, Table 2 also contains the saturated operation cost for the previous generation of protocols ("Old"). It is clear from the data that the speed of all complex protools has increased by several orders of magnitude. For a visual comparison of the new and old protocols, please refer to Appendix C.

We conclude that the protocols presented in this paper are significantly more efficient than the ones in [3]. The improvement is most visible for the bit extraction and comparison protocols, where the new protocols are more than a hundred times faster than the previous ones. We also note that our implementation actually achieves a speed of 1 MIPS (million instructions per second) for the private share conversion operations. This is a significant milestone for secure multi-party computation, as data miners typically work with large datasets.

Another goal of our experiments was to show that it is possible to run secure multiparty computation protocols with large input vectors containing up to 100 million values. This demonstrates the robustness of the implementation and therefore its suitability for practical applications.

8.4 Secure k-means clustering

k-means clustering is a cluster analysis algorithm for partitioning a set of points into k clusters according to their distances from each other. Cluster analysis helps to identify similar object groups and is used in a range of areas from business intelligence to computational biology. k-means clustering is a fitting benchmark for the protocols in this paper, as it requires multiplications, greater-than comparison and division.

Our implementation of k-means uses secure computation to hide the values of the clustered data. We did not hide the size of the clusters or the assignment of points to the clusters. However, this is not a significant limitation, since the cluster sizes would typically be published anyway. While certain techniques could be used to hide the movement of points between clusters, they would significantly lower the performance of the

Dataset	k	Time	Iter.	Multiplications	Less-thans	Divisions
iris 150×4	3	1 s	4	12.6% (9600 ops)	42.9% (5400 ops)	38.5% (44 ops)
synthetic 600×60	3	3 s	5	44.4% ($7.2 \cdot 10^5$ ops)	29% ($2.7 \cdot 10^4$ ops)	21.7% (900 ops)
	5	6 s	8	41.2% ($1.7 \cdot 10^6$ ops)	33% ($1.2 \cdot 10^5$ ops)	16% (2400 ops)
	8	8 s	7	42.2% ($2.3 \cdot 10^6$ ops)	44.2% ($2.7 \cdot 10^5$ ops)	10.7% (3360 ops)
plants 34781×70	3	4 min 58 s	12	75.8% ($1.2 \cdot 10^8$ ops)	21.1% ($3.8 \cdot 10^6$ ops)	0.6% (2520 ops)
	5	22 min 42 s	28	41.2% ($4.1 \cdot 10^8$ ops)	33% ($2.4 \cdot 10^7$ ops)	0.4% (9800 ops)
	10	36 min 35 s	17	51.3% ($4.6 \cdot 10^8$ ops)	47.6% ($5.9 \cdot 10^7$ ops)	0.2% (11900 ops)

Table 3. Privacy-preserving k-means clustering performance

computation. It is a generally accepted trade-off between privacy and efficiency also taken by previous implementations [19,10]. We modified the algorithm to use vector operations as much as possible to take advantage of the increased performance.

The points are distributed into initial clusters on a round-robin basis (the initial cluster number of point i is $i \bmod k$). Note that different initial cluster numbers can affect the number of iterations needed. We did not attempt to achieve more favourable initial clusters. The algorithm runs until stabilizing.

The benchmarking results presented in Table 3 are based on the *iris*, *synthetic* and *plants* datasets from the UCI Machine Learning Repository [12]. The databases were stored in secret-shared form after scaling fractional values to allow integer computation; the scaling did not affect the final clustering. Each row shows one experiment: the parameters, overall number of operations, and measured runtime. The latter is further broken down by secure operation. Note that the percentages do not add up to 100% as a fraction of the time was also used for disk operations and secret sharing.

The synthetic control chart time series data set was also used by Doganay et al. in [10] to benchmark k-means clustering algorithms developed by themselves and by Vaidya and Clifton [19]. Their algorithms only work on vertically partitioned data, which is a considerably weaker security model compared to the one used by SHAREMIND. Despite that, the time they required to cluster the *synthetic* dataset were considerably larger compared to our implementation. Whereas SHAREMIND required 3-8 seconds for this task, the implementation of the algorithms introduced in [10] needed roughly 30 seconds, and the time required by the algorithm of [19] was even several orders of magnitude larger.

9 Conclusions

In this paper, we have presented numerous advancements for computational primitives used in the SHAREMIND multi-party computation engine. Compared to the original implementation presented in [3], the performance of all the primitives (multiplication, share conversion, bit extraction, equality testing, comparison) has been increased. Additionally, new protocols for right shift by a public offset and division by both public and private value have been implemented and benchmarked. All the proposed protocols

have been proven secure in the semi-honest model with one passive adversary and a convenient game-based framework for improving the readability of the proofs has been presented.

The original motivation for developing SHAREMIND has come from the needs of mining large volumes of data. Our benchmarks show that with the current improvements, input vectors of up to 10^8 elements can be processed in reasonable time and that speeds up to 1 MIPS can be achieved for basic primitives on state-of-the-art hardware. Real performance of the primitives has improved up to 100 times. Benchmarks with the k-means clustering algorithm show that secure MPC is ready to handle real-world data-mining tasks, as algorithm runs needing hundreds of millions of private operations can be executed in reasonable time.

References

1. SecureSCM. Technical report D9.1: Secure Computation Models and Frameworks. `http://www.securescm.org` (2008)
2. 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, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1455770.1455804
3. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: ESORICS 2008: Proceedings of the 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008, *LNCS*, vol. 5283, pp. 192–206. Springer (2008)
4. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: a framework for fast privacy-preserving computations. Cryptology ePrint Archive, Report 2008/289 (2008). `http://eprint.iacr.org/`
5. Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis. Submitted. (2011)
6. 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: FC '09: Proceedings of the 13th Thirteenth International Conference on Financial Cryptography, pp. 325–343 (2009)
7. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In: Proceedings of the USENIX Security Symposium '10, pp. 223–239. Washington, DC, USA (2010)
8. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS '01: 42nd Annual Symposium on Foundations of Computer Science, pp. 136–145 (2001)
9. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Proceedings of The Third Theory of Cryptography Conference, TCC 2006, *LNCS*, vol. 3876. Springer (2006)
10. Doganay, M.C., Pedersen, T.B., Saygin, Y., Savaş, E., Levi, A.: Distributed privacy preserving k-means clustering with additive secret sharing. In: Proceedings of the 2008 International Workshop on Privacy and Anonymity in Information Society, PAIS '08, pp. 3–11 (2008)
11. Even, G., Seidel, P.M., Ferguson, W.E.: A parametric error analysis of Goldschmidt's division algorithm. J. Comput. Syst. Sci. **70**(1), 118–139 (2005)

12. Frank, A., Asuncion, A.: UCI machine learning repository (2010). URL http://archive.ics.uci.edu/ml
13. Geisler, M.: Cryptographic protocols: Theory and implementation. Ph.D. thesis, Aarhus University (2010)
14. Granlund, T., Montgomery, P.L.: Division by invariant integers using multiplication. In: PLDI '94: Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, pp. 61–72 (1994)
15. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: CCS '10: Proceedings of the 17th ACM conference on Computer and Communications Security, pp. 451–462. ACM (2010)
16. Malka, L., Katz, J.: VMCrypt - modular software architecture for scalable secure computation. Cryptology ePrint Archive, Report 2010/584 (2010). http://eprint.iacr.org/
17. Parhami, B.: Computer Arithmetic: Algorithms and Hardware Designs, 2nd edn. Oxford University Press (2010)
18. Rodeheffer, T.: Software integer division. Microsoft Research Tech Report MSR-TR-2008-141 (2008)
19. Vaidya, J., Clifton, C.: Privacy-preserving k-means clustering over vertically partitioned data. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data mining, KDD '03, pp. 206–215 (2003)

A Bit Shift Protocols under a Public Shift

The protocols in this section allow us to perform two more standard bit-level operations on shared values, namely left and right shifts ($\ll$ and $\gg$).[7]

First, note that the left shift protocol is actually trivial, since left shift by p positions can be accomplished by multiplying the shared value by a public constant 2^p. This, in turn, can be done by locally multiplying all the shares by the same constant. Since no messages are exchanged, the protocol is trivially secure against a passive adversary.

Right shift, on the other hand, is more complicated because of the unknown overflow carry modulo 2^n. Thus, in order to build a right shift protocol, we first need a protocol to compute the overflow. This is considerably easier to do if the value in question is (temporarily) secret-shared between just two parties, because then the overflow is guaranteed to be either 0 or 1. We thus present two routines: Algorithm 10 for resharing a value to just two parties and Algorithm 11 for computing the overflow bit once the values are shared in this way.

Theorem 9. *Algorithm 10 is correct and secure against one passive attacker.*

Proof. Correctness of the Algorithm is straightforward:

$$\begin{aligned} u' &= u'_1 + u'_2 + u'_3 = 0 + u_2 + r_2 + u_3 + r_3 \\ &= u_2 + r_2 + u_3 + u_1 - r_2 = u. \end{aligned}$$

For security note that $\mathcal{P}_1$ has no incoming messages, whereas the only incoming messages for $\mathcal{P}_2$ and $\mathcal{P}_3$ are r_2 and $u_1 - r_2$, respectively. These messages can be easily simulated with a random value.

Algorithm 10: Protocol $[\![u']\!] \leftarrow \mathsf{ReshareToTwo}([\![u]\!])$ for resharing a value $[\![u]\!]$ between the parties $\mathcal{P}_2$ and $\mathcal{P}_3$.

Data: Shared value $[\![u]\!]$.
Result: Shared value $[\![u']\!]$ so that $u = u'$ and $u'_1 = 0$.
1 $\mathcal{P}_1$ generates random $r_2 \leftarrow \mathbb{Z}_{2^n}$ and computes $r_3 \leftarrow u_1 - r_2$.
2 $\mathcal{P}_1$ sets $u'_1 = 0$ and sends r_i to $\mathcal{P}_i$ $(i = 2, 3)$.
3 $\mathcal{P}_i$ computes $u'_i \leftarrow u_i + r_i$ $(i = 2, 3)$.
4 Return $[\![u']\!]$.

Algorithm 11: Protocol $[\![\lambda]\!] \leftarrow \mathsf{Overflow}([\![u']\!])$ for obtaining the overflow bit $[\![\lambda]\!]$ for $[\![u']\!]$ if share $u'_1 = 0$.

Data: Shared value $[\![u']\!]$ where $u'_1 = 0$.
Result: Share $[\![\lambda]\!]$ so that $u' = u'_2 + u'_3 - \lambda 2^n$.
1 $\mathcal{P}_1$ sets $p_1 = 0$.
2 $\mathcal{P}_2$ sets $p_2 = u'_2$.
3 $\mathcal{P}_3$ sets $p_3 = -u'_3$.
4 $\overline{[\![s]\!]} \leftarrow \mathsf{MSNZB}(\overline{[\![p]\!]})$.
5 Share the value $-u'_3$ bitwise as a vector $\overline{[\![-u'_3]\!]}$.
6 $[\![\lambda^0]\!] \leftarrow 1 \oplus \bigoplus_{i=0}^{n-1} \overline{[\![s]\!]}^{(i)} \wedge \overline{[\![-u'_3]\!]}^{(i)}$.
7 $\mathcal{P}_3$ checks whether $u'_3 = 0$. If so, $\lambda^0_3 = 1 \oplus \lambda^0_3$.
8 Return $[\![\lambda]\!] \leftarrow \mathsf{ShareConv}([\![\lambda^0]\!])$.

The correctness proof for Algorithm 11 is somewhat more complicated.

Theorem 10. *Algorithm 11 is correct and secure against one passive attacker.*

Proof. To prove correctness, we need to compute the overflow bit λ. The overflow occurs exactly when $u'_2 + u'_3 \geq 2^n$, or equivalently $u'_2 \geq 2^n - u'_3$. Note that modulo 2^n the value $2^n - u'_3$ is represented just as $-u'_3$ (unless $u'_3 = 0$, which has to be treated separately). Thus

$$\lambda = 1 \quad \Longleftrightarrow \quad u'_2 \geq (-u'_3) \bmod 2^n \ \wedge \ u'_3 \neq 0\,.$$

In order to perform the comparison between u'_2 and $-u'_3$, we first run Algorithm 4 and obtain a bitwise shared vector $\overline{[\![s]\!]}$, which contains all zeroes if $u'_2 = -u'_3 \bmod 2^n$, or has just one bit in the highest position where they differ. Thus, the dot product $\bigoplus_{i=0}^{n-1} \overline{[\![s]\!]}^{(i)} \wedge \overline{[\![-u'_3]\!]}^{(i)} = 1$ iff $u'_2 < -u'_3 \bmod 2^n$ and hence $\lambda^0 = 1$ iff $u'_2 \geq -u'_3 \bmod 2^n$, as required. The only exception appears when $u'_3 = 0$, in which case no overflow can occur, but λ^0 is set to 1. This mistake is easy to correct locally by $\mathcal{P}_3$ who has the original u'_3 and can flip his own share of λ^0 in case u'_3 happens to be 0.

The security of the protocol is still trivial as it is just a composition of perfectly simulatable protocols.

[7] Note that a bit shift can be used for efficient comparison as the highest bit of x is just $x \gg 31$.

We are now ready to present the right shift protocol. The main idea behind the public right-shift protocol is to convert the input to a sum of two values (known to two of the parties) and then shift these down. This leaves us with two problems. First, discarding the low bits discards the carry-bit for the least significant position that is retained. Second, the top carry-bit of the addition would previously implicitly disappear as we consider addition modulo 2^n. Since the values have been shifted down, the carry-bit will be present. The bulk of the work of the protocol consists of determining and correcting for these two carry-bits.

The protocol itself is presented as Algorithm 12

Algorithm 12: Protocol $[\![w]\!] \leftarrow \mathsf{ShiftR}([\![u]\!], p)$ for evaluating right shift.

Data: Shared value $[\![u]\!]$ and a public shift p.
Result: Shares $[\![w]\!]$ such that $w = u \gg p$.

1 $[\![u']\!] \leftarrow \mathsf{ReshareToTwo}([\![u]\!])$.
2 $[\![s]\!] \leftarrow [\![u' \ll n - p]\!]$ (locally).
3 $[\![\lambda_1]\!] \leftarrow \mathsf{Overflow}(u')$.
4 $[\![\lambda_2]\!] \leftarrow \mathsf{Overflow}(s)$.
5 $\mathcal{P}_i$ computes $v_i \leftarrow u'_i \gg p$.
6 Return $[\![w]\!] = [\![v]\!] - 2^{n-p}[\![\lambda_1]\!] + [\![\lambda_2]\!]$.

Theorem 11. *Algorithm 12 is correct and secure against one passive attacker.*

Proof. Correctness of the algorithm follows from the discussion above. Since $u'_2 + u'_3 = u + \lambda_1 2^n$, we have

$$\begin{aligned} v = v_1 + v_2 + v_3 \bmod 2^n &= (u'_2 \gg p) + (u'_3 \gg p) \bmod 2^n \\ &= u \gg p + \lambda_1 2^{n-p} - \lambda_2 \bmod 2^n, \end{aligned}$$

hence

$$u \gg p = v - \lambda_1 2^{n-p} + \lambda_2 \bmod 2^n .$$

For security note that we are only composing perfectly simulatable subroutines.

This protocol can also be used for extracting the most significant bit for comparison purposes. As it is also slightly more efficient than the full bit extraction, we use it as the basis of the comparison in the current implementation for the comparison operator.

B Error calculation of Goldschmidt division

We will present an analysis of the effects of rounding errors. This is done by looking at the divergence from the "ideal" computation where no rounding takes place and for which the error terms can be fairly easily estimated. A similar analysis was performed in [11]. Their analysis was more detailed, but relied on using floating point numbers, making it hard to apply it here directly.

Let N_i, D_i, F_i, c_0 denote the actual real numbers encountered during the run of Newton Goldschmidt iterations as described in Section 9. In SHAREMIND, we are using the approximations of values x by fixed point numbers $\widetilde{x} = 2^{-n'} \cdot \widehat{x}$ being represented by $\widehat{x} \in \mathbb{Z}_{m'}$ for some m'.

Recall that both the sequences (N_i) and (D_i) were converging from below to $\frac{u}{v}$ and 1, respectively. To preserve the convergence from below in the presence of errors, extra care needs to be taken with rounding errors to make sure they are also one-sided.

Let the differences between the real values N_i, D_i and their approximations be ΔN_i and ΔD_i selected so that $\widetilde{N_i} = N_i + \Delta N_i$ and $\widetilde{D_i} = D_i - \Delta D_i$. Note that on line 13 of Algorithm 9 the value of $\widehat{N_k}$ is always rounded up and the value of $\widehat{D_k}$ is always rounded down. This guarantees that we have $\Delta N_k, \Delta D_k \geq 0$ for all $k \geq 1$.

When the shares of $\widehat{\widehat{N_k}}$ and $\widehat{\widehat{D_k}}$ are right-shifted to convert the elements back to the precision $2^{-n'}$ (Algorithm 9, line 13), additional truncation errors are introduced. Since there are three computing parties and we shift by n' positions, the errors occurring at both upwards and downwards rounding are bounded by $\delta = 3 \cdot 2^{-n'}$. Thus, for $k \geq 1$ we obtain

$$\begin{aligned}\widetilde{D_{k+1}} &> \widetilde{D_k} \cdot \widetilde{F_{k+1}} - \delta = \widetilde{D_k} \cdot (2 - \widetilde{D_k}) - \delta \\ &= (D_k - \Delta D_k) \cdot (2 - D_k + \Delta D_k) - \delta \\ &= D_{k+1} - 2\Delta D_k(1 - D_k) - (\Delta D_k)^2 - \delta\end{aligned}$$

and

$$\begin{aligned}\widetilde{N_{k+1}} &\leq \widetilde{N_k} \cdot \widetilde{F_{k+1}} + \delta = \widetilde{N_k} \cdot (2 - \widetilde{D_k}) + \delta \\ &= (N_k + \Delta N_k) \cdot (2 - D_k + \Delta D_k) + \delta \\ &= N_{k+1} + N_k \Delta D_k + \Delta N_k(2 - D_k + \Delta D_k) + \delta\,.\end{aligned}$$

This implies

$$\begin{aligned}\Delta D_{k+1} &= D_{k+1} - \widetilde{D_{k+1}} < 2\Delta D_k(1 - D_k) + (\Delta D_k)^2 + \delta \\ &\leq 2\Delta D_k 2^{-2^k} + (\Delta D_k)^2 + \delta\end{aligned}$$

and

$$\begin{aligned}\Delta N_{k+1} &= \widetilde{N_{k+1}} - N_{k+1} \\ &\leq \Delta N_k(2 - D_k + \Delta D_k) + N_k \Delta D_k + \delta \\ &< \Delta N_k(1 + 2^{-2^k} + \Delta D_k) + \frac{u}{v}\Delta D_k + \delta\,.\end{aligned}$$

Since the first rounding error is introduced only after multiplication by F_1, we have $\Delta D_1, \Delta N_1 \leq \delta$. Thus we can iterate these recurrent inequalities to get bounds for $\Delta D_k, \Delta N_k$ in terms of $\frac{u}{v}$ and δ.

In order to guarantee that truncation of the result will lead to a proper value, we will have to ensure that the end result R satisfies $\lfloor \frac{u}{v} \rfloor \leq \lfloor R \rfloor < \lfloor \frac{u}{v} \rfloor + 1$. Let $1 - D_k < 2^{-p}$, in which case $N_k < \frac{u}{v}(1 - 2^{-p})$ since $\frac{N_k}{D_k} = \frac{u}{v}$. Recall that $\widehat{c_0}$ was chosen so that

$0.5 \leq v\widetilde{c_0} < 1$, hence $\widetilde{c_0} < \frac{1}{v} \leq 2\widetilde{c_0}$. Consequently, $\widetilde{N_k} \geq N_k < \frac{u}{v}(1-2^{-p}) > \frac{u}{v} - u\widetilde{c_0}2^{-p+1}$. Taking $R = \widetilde{N_k} + \Delta$ where $\Delta = u\widetilde{c_0}2^{-p+1}$ thus guarantees $R \geq \frac{u}{v}$ and $\lfloor R \rfloor \geq \lfloor \frac{u}{v} \rfloor$.

We are left to show that $R < \lfloor \frac{u}{v} \rfloor + 1$. Let $\Delta N_k < a + b\frac{u}{v}$ as obtained after iterating the above recurring inequalities k times. Then $R = \widetilde{N_k} + u\widetilde{c_0}2^{-p+1} < N_k + a + 2u\widetilde{c_0}(2^{-p}+b)$. Since $N_k < \frac{u}{v} \leq (\lfloor \frac{u}{v} \rfloor + 1) - \frac{1}{v} < (\lfloor \frac{u}{v} \rfloor + 1) - \widetilde{c_0}$, it suffices to show that $a + 2u\widetilde{c_0}(2^{-p}+b) < \widetilde{c_0}$, or equivalently $\frac{a}{\widetilde{c_0}} + 2ub + u2^{-p+1} < 1$. Since $2^{-n} \leq \widetilde{c_0} < 1$ and $0 \leq u < 2^n$, this can be achieved by showing $2^n(a + 2b + 2^{-p+1}) < 1$.

For $n = 32$ the required inequality can be guaranteed by taking $k = 5$, $n' = 37$, in which case $p > 40.68$ (if the first iteration is done with $F_1 = 2\sqrt{2} - 2D_0$), $a, b < 0.2 \times 2^{-32}$. These choices imply $m = 32 + (5+1) \times 37 = 254$.

C Benchmark diagrams

Figures 1, 2, 3, 4, 5 and 6 compare the running times for the protocols in this paper with the protocols in [3]. The range between the minimal and maximal result is shown where multiple experiments were conducted. Missing data points indicate that the protocol was too inefficient to perform at that input size. The axes on the diagrams are drawn on a logarithmic scale. Since the right shift protocol is also used to implement greater-than comparisons, we compared it with the greater-than comparison protocol from [3]. This is an honest comparison, since the greater-than comparison can be implemented in computing the difference on two values and finding the highest bit using the right shift operation.

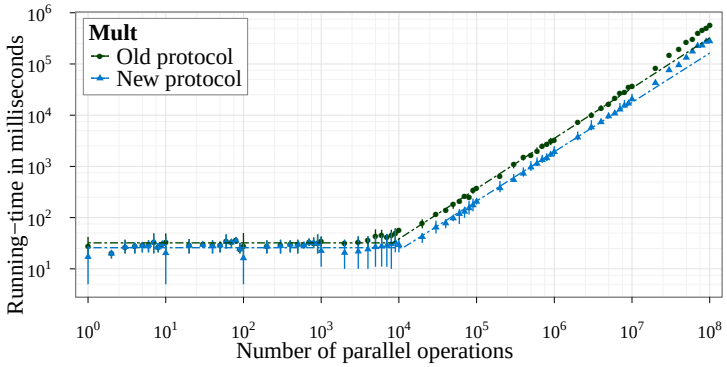

Fig. 1. Benchmark results for the multiplication conversion operation

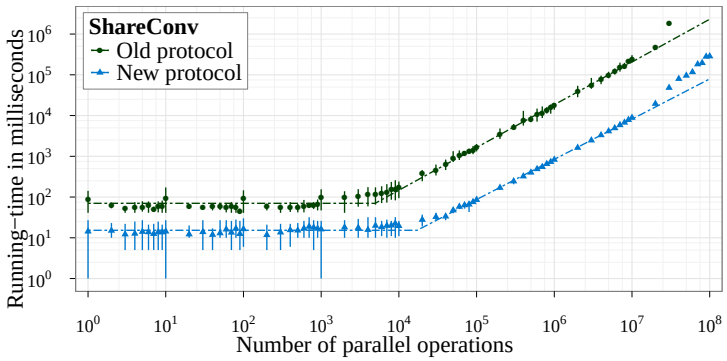

Fig. 2. Benchmark results for the share conversion operation

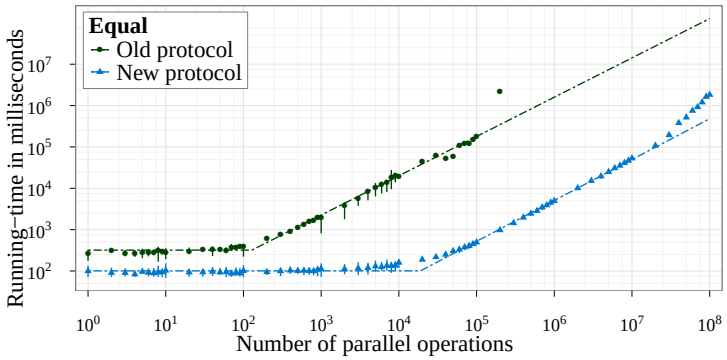

Fig. 3. Benchmark results for the equality comparison operation

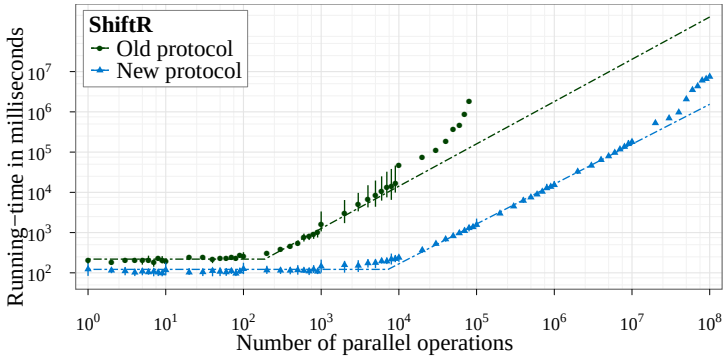

Fig. 4. Benchmark results for the greater-than comparison operation

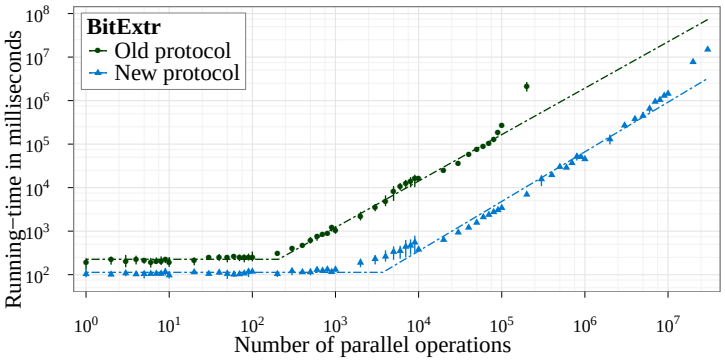

Fig. 5. Benchmark results for the bit extraction operation

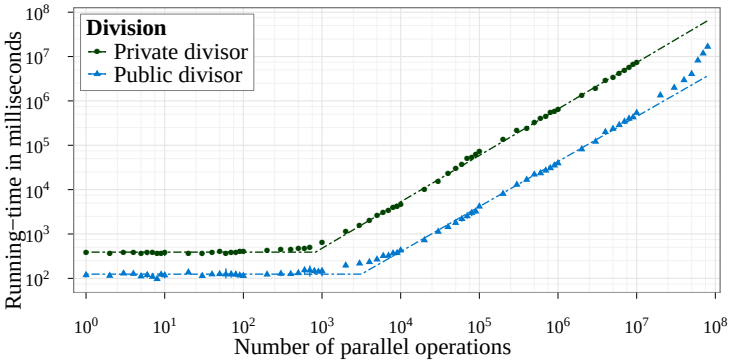

Fig. 6. Benchmark results for the division operations