Mercury: Constant-Round Protocols for Multi-Party Computation with Rationals

Luke Harmon and Gaetan Delavignette

Algemetric Inc.
{lharmon,gdelavignette}@algemetric.com

Abstract. Secure computation has become a necessity in the modern world. Its applications are widespread: from allowing medical researchers to compute statistics over private patient data without violating HIPAA to helping large companies like Meta and Google avoid GDPR fines. A ubiquitous and popular choice for secure computation is Multi-party Computation (MPC). Most MPC protocols work over finite fields or rings, which means that encoding techniques are required to map rational-valued data into the algebraic structure being used. Leveraging an encoding technique introduced in [16], we present Mercury — a family of protocols for addition, multiplication, subtraction, and division of rational numbers. Notably, the output of our division protocol is exact (i.e., it does not use iterative methods). Our protocols offer significant improvements in both round complexity and communication complexity when compared with prior art, and are secure for a dishonest minority of semi-honest parties. We emphasize that the encoding technique our protocols are based on is composable, so it can be paired with any MPC protocol over a prime-order field.

Keywords: secure multi-party computation · secret sharing · rational numbers · rational division.

1 Introduction

Secure computation is a tool which allows functions of private data to be evaluated without revealing those data. One such tool is homomorphic encryption (HE) (see [1] for a nice survey) which allows, e.g., a data owner to encrypt their data and outsource computations on those data. More specifically, anyone can compute reasonable functions of the encrypted data and never learn more about the data than what the output of the function reveals. However, most HE schemes are prohibitively slow, especially when computing a products. Further, most HE schemes (with the exception of CKKS[1]) do not support division.

Another well-studied form of secure computation is Multi-party Computation (MPC). In the classic setting, n mutually distrusting parties P_i possess private

[1] The CKKS scheme introduced by Cheon et al. [10] can perform division by approximating the inverse function with a Taylor polynomial $T(x) \approx x^{-1}$ and then computing $c_0/c_1 \approx c_0 T(c_1)$, for ciphertexts c_0, c_1.

data d_i and wish to jointly compute a function $F(d_1, \ldots, d_n)$ without revealing d_i to P_j for $i \neq j$. This was first studied in detail by Yao in [22]. Since then, much work has been done extending Yao's results, developing new tools for MPC, and implementing these tools in the real world (e.g., [5, 12, 13, 15, 17, 18]). Many of these protocols rely on *secret sharing*. In secret sharing, each P_i is provided masked pieces (called *shares*) of private data (henceforth, *secrets*). These shares are chosen such that only authorized sets of parties can determine a secret if they pool their shares. The parties use their shares to perform computations on the secrets by communicating (e.g., sending/receiving shares, creating and sending new shares, etc.) with one another as needed. A common primitive for secret sharing is Shamir's scheme [19] based on polynomial interpolation over a finite field. An advantage of that scheme is that it is *additively homomorphic* so that shares of two secrets can be added locally to give a sharing of the sum of those secrets. Additively homomorphic secret sharing is also used by the well-known MP-SPDZ framework [17].

Most MPC protocols are defined over finite rings or fields such as $\mathbb{Z}/n\mathbb{Z}$ or $GF(2^\ell)$, and as such require real data (often in the form of fixed-point or floating-point numbers) to be encoded as elements of the ring (field). Further, since the goal is evaluating certain functions (e.g., polynomials) over secret shared values, the encoding method must be homomorphic with respect to the operations which compose the function (e.g., addition and multiplication). Several works [7, 9, 21] encode a fixed-point or floating-point number a with precision f as the integer $a \cdot 2^f$. Other approaches [2,8] work with floating-point numbers by separately encoding the sign, exponent, and significand, along with an extra bit that is set to 0 iff the number is 0.

Our approach differs significantly from all of these. Instead of choosing a set of fixed-point numbers and designing our protocols around them, we start with a set of rationals (with bounded numerators and denominators) which can be constructed to contain a desired set of fixed-point numbers. This set of rationals, paired with an encoding that is homomorphic with respect to both addition and multiplication, forms the basis of our protocols. The range of the encoding can be any ring/field of the form $\mathbb{Z}/n\mathbb{Z}$. This means that, for the most part, our protocols are obtained by simply "attaching" the encoder to existing protocols. The exception is division which relies heavily on the structure of the aforementioned set of rationals.

All our protocols require only a constant number of rounds and have communication complexity $O(tn)$, where n is the number of parties and $t-1$ is a bound on the number of corrupted parties. Our protocols are also generic, by which we mean that they do not depend on the underlying primitive being used. For example, even though we use Shamir's scheme as the foundation, our protocols for rational arithmetic could easily be translated to use additive secret sharing (e.g., implemented with MP-SPDZ). Moreover, the encoding technique on which the protocols are based is composable, meaning that the encoding works with any MPC protocol based on secret sharing over a prime-order field.

The paper is organized as follows:

* Section 2 discusses notation and provides an overview of Shamir's scheme.
* Section 3 introduces the rational-to-ring-element encoder, "building block" protocols, and finally Mercury: our rational addition, multiplication, subtraction, and division protocols.
* Section 4 contains a brief discussion of our (partial) compatibility with fixed-point numbers, and then investigates how to choose a subset of the rational domain of the encoder which allows (arithmetic) circuits up to a certain multiplicative depth to be evaluated on shared values.
* Section 5 discusses how the round complexity and communication complexity of Mercury can be reduced by using well-known optimizations.
* Section 6 compares Mercury with prior work handling fixed-point arithmetic in MPC [7, 9, 20, 21].
* Section 7 summarizes our results and discusses additional protocols which we hope to include in Mercury in the future.

2 Preliminaries

2.1 Notation

For a positive integer a, $\mathbb{Z}/a\mathbb{Z}$ denotes the ring of integers modulo a. In case a is prime, we write $\mathbb{F}_a$. The elements of $\mathbb{Z}/a\mathbb{Z}$ will be represented by integers $0, 1, \ldots, a-1$. For a ring R, $R[x]$ will denote the ring of polynomials with coefficients in R. We use $y \leftarrow A(x)$ to denote that a randomized algorithm A on input x outputs y. If A is deterministic, we simply write $y = A(x)$. All circuits we consider are arithmetic over a field $\mathbb{F}_a$, and have fan-in 2.

Shamir's scheme. We pause here to provide a brief overview of Shamir secret sharing (SSS), and the notation used therein. Suppose we have n parties and wish for any set of $t \leq n$ parties to be able to reconstruct a secret by pooling their shares. This is called t-out-of-n threshold scheme. One creates *Shamir shares* of a secret $s \in \mathbb{F}_p$ by generating a random polynomial $f(x) \in \mathbb{F}_p[x]$ of degree at most $t-1$ whose constant term is s (i.e., $f(0) = s$) and whose remaining coefficients are chosen uniformly from $\mathbb{F}_p$. Shares of s are the field elements $f(i)$, $i \in \mathbb{F}_p \setminus \{0\}$. We assume the i^{th} party receives the share $f(i)$. We use $[x]_i$ to denote the i^{th} party's share of $x \in \mathbb{F}_p$, and $[x]$ to denote a sharing of x among all parties. Then, for example, $[x] + [y]$ will mean "each party adds their share of x to their share of y," and $c[x]$ will mean "each party multiplies their share of x by c." Any collection of t parties can pool their shares $[s]$ and reconstruct the polynomial f using Lagrange interpolation, thereby obtaining the secret $s = f(0)$.

2.2 Framework

As mentioned above, we use t-out-of-n SSS over $\mathbb{F}_p$ for all protocols. Our protocols are built on the basic Shamir addition, and the multiplication introduced in [14], which is itself an optimization of the multiplication described in [4]. To

ensure the multiplication protocol works, we require $t-1 < n/2$. We also tolerate up to $t-1$ semi-honest (honest-but-curious) adversaries. The communication complexity of a protocol is measured by the number of field elements sent by all parties throughout the protocol. When comparing with protocols that measure communication complexity in bits, we simply multiply our communication complexities by $\log_2(p)$.

3 Protocols for Rational Numbers

We propose a family of efficient MPC protocols for performing computations with rational numbers. These protocols are obtained by pairing an encoder that maps certain rational numbers to integers with the familiar SSS protocols for addition, multiplication, and computing modular inverses. The protocols for computing the sum and product of shared fixed-point numbers remain unchanged from the analogous SSS primitives, except that rational operands are encoded to field elements before any protocols are executed. Division is an amalgam of existing protocols and relies on the fact that our mapping for encoding rational numbers to integers is induced by a ring homomorphism, and therefore preserves inverses; likewise for the decode mapping. We elaborate below.

3.1 Encoding Rationals

We use the encoding map introduced in [16] which maps a subset of rationals, whose denominators are co-prime with a prime p, into $\mathbb{F}_p$. This map is defined by $\mathsf{encode}(x/y) = xy^{-1} \bmod p$, and has as its domain the *Farey rationals*

$$\mathcal{F}_N := \big\{ x/y \,:\, |x| \le N,\, 0 < y \le N,\, \gcd(x,y) = 1, \gcd(p,y) = 1 \big\},$$

where $N = N(p) := \lfloor \sqrt{(p-1)/2} \rfloor$. encode is induced by a ring isomorphism, so both it and its inverse decode are additively and multiplicatively homomorphic as long as the composition of operands in $\mathcal{F}_N$ remains in $\mathcal{F}_N$[2]. decode can be computed efficiently using a slight modification of the Extended Euclidean Algorithm. We summarize important properties of encode, decode, and $\mathcal{F}_N$ with the following lemma.

Lemma 1. *Let p be a prime, $N = N(p)$, and* encode, decode *be the encode and decode maps, respectively.*

(i) If $x/y \in \mathcal{F}_N$, then $-x/y \in \mathcal{F}_N$.

(ii) If $x/y \in \mathcal{F}_N$ is nonzero, then $y/x \in \mathcal{F}_N$.

(iii) $\mathcal{F}_N$ is not closed under addition and multiplication.

(iv) $[-N, N] \cap \mathbb{Z} \subseteq \mathcal{F}_N$. Moreover, if $z \in [0, N] \cap \mathbb{Z}$, then $\mathsf{encode}(z) = z$.

(v) encode *and* decode *are homomorphic w.r.t. addition and multiplication as long as the composition of operands in $\mathcal{F}_N$ remains in $\mathcal{F}_N$.*

Proof. (i)-(iv) are obvious. (v) is proved in [16, Proposition 2].

[2] E.g., $\mathsf{encode}\left(\frac{x_0}{y_0} + \frac{x_1}{y_1}\right) = \mathsf{encode}\left(\frac{x_0}{y_0}\right) + \mathsf{encode}\left(\frac{x_1}{y_1}\right)$ iff $\frac{x_0}{y_0}, \frac{x_1}{y_1}, \frac{x_0}{y_0} + \frac{x_1}{y_1} \in \mathcal{F}_N$.

3.2 Building Blocks

SSS primitives will be the algorithms/protocols Share, Recon, Add, ScalarMult, and Mult. Respectively, these create shares of a secret value $s \in \mathbb{F}_p$, reconstruct a secret value given enough shares $[s]$, compute the shares of the sum of two secrets without revealing either, compute shares of the product of a secret value and a known value without revealing the secret value, and compute shares of the product of two secrets without revealing either. Note that Recon, Add, and ScalarMult are all performed locally[3]. Borrowing from [2], we use $s \leftarrow \mathsf{Output}([s])$ to mean that each of a set of $t-1$ parties sends their share of s to another party, which subsequently computes $s = \mathsf{Recon}([s])$ and sends s to the other $n-1$ parties. We pause briefly to describe the multplication protocol from [14].

$[xy] \leftarrow \mathsf{Mult}([x], [y])$

---

1. For $i = 1, \ldots, n$, the i^{th} party P_i computes $z_i = [x]_i \cdot [y]_i$;
2. for $i = 1, \ldots, 2t-1$, P_i computes $[z_i] \leftarrow \mathsf{Share}(z_i)$[4];
3. once each party $P_1, \ldots, P_n$ has received $2t-1$ shares, each party P_i locally computes $[xy]_i$ using Lagrange interpolation on the received shares;
4. output $[xy]$.

Two additional protocols, RandInt and Inv, are required for our rational division protocol. These protocols allow all parties to obtain shares of a random ring element and compute shares of the multiplicative inverse of a ring element, respectively.

$[r] \leftarrow \mathsf{RandInt}()$

---

1. Each of a set of t parties select a uniformly random $r_i \in \mathbb{F}_p$, $i = 1, \ldots, t$;
2. each of the t parties do $[r_i] \leftarrow \mathsf{Share}(r_i)$, and send the j^{th} share to P_j;
3. $[r] \leftarrow \mathsf{Add}([r_1], \ldots, [r_t])$;
4. return $[r]$.

[3] By "performed locally" we mean that the parties use their shares execute the protocol without communicating with each other. E.g. for Recon a party (or the dealer, if one exists) in possession of at least t shares reconstructs the secret using polynomial interpolation.

[4] One could use any set of $2t-1$ parties for this step.

$[x^{-1}] \leftarrow \mathsf{Inv}([x])$

1. $[r] \leftarrow \mathsf{RandInt}()$;
2. $[rx] \leftarrow \mathsf{Mult}([r], [x])$;
3. $rx = \mathsf{Output}([rx])$;
4. abort and restart if $rx = 0$, otherwise continue;
5. each party locally computes $(rx)^{-1} = x^{-1}r^{-1} \bmod p$;
6. each party does $[x^{-1}] = \mathsf{ScalarMult}(x^{-1}r^{-1}, [r])$;
7. return $[x^{-1}]$.

Table 1: *Total communication complexity (measured in field elements) of SSS building block protocols.*

Protocol	Rounds	Comm. Complexity
Share	1	$n-1$
Recon	0	0
Output	2	$(n-1)+(t-1)$
Add	0	0
Mult	1	$(2t-1)(n-1)$
ScalarMult	0	0
RandInt	1	$t(n-1)$
Inv	4	$t(3n-2)-1$

Note that Share has communication complexity n (initial sharing of a secret), unless one party is sharing with the remaining parties, in which case it has communication complexity $n-1$. The latter is the only one relevant to our protocols.

3.3 Rational Addition, Multiplication, Subtraction, and Division

Addition and multiplication are obtained by simply pairing the encoder encode with Add and Mult. To represent shares of the encoding of $x/y \in \mathcal{F}_N$, we write $\big[\mathsf{encode}(x/y)\big]$. We first present the four protocols, and then list their complexities in table 2. For all protocols, we use the field $\mathbb{F}_p$, and $\dfrac{x_0}{y_0}, \dfrac{x_1}{y_1} \in \mathcal{F}_N$ such that $\dfrac{x_0}{y_0} \pm \dfrac{x_1}{y_1}, \dfrac{x_0}{y_0} \cdot \dfrac{x_1}{y_1}, \dfrac{x_0}{y_0} \div \dfrac{x_1}{y_1} \in \mathcal{F}_N$. Let $t_0 = \mathsf{encode}(x_0/y_0)$ and $t_1 = \mathsf{encode}(x_1/y_1)$.

Remark 1. We use the prefix "Hg" for our protocols because it is the chemical symbol for Mercury in the Periodic Table of Elements.

$[\mathsf{encode}(x_0/y_0 + x_1/y_1)] = \mathsf{HgAdd}\big([\mathsf{encode}(x_0/y_0)], [\mathsf{encode}(x_1/y_1)]\big)$

1. $[t_0 + t_1] = \mathsf{Add}([t_0], [t_1])$;
2. return $[t_0 + t_1] = [\mathsf{encode}(x_0/y_0 + x_1/y_1)]$.

$[\mathsf{encode}(x_0x_1/y_0y_1)] = \mathsf{HgMult}\big([\mathsf{encode}(x_0/y_0)], [\mathsf{encode}(x_1/y_1)]\big)$

1. $[t_0t_1] \leftarrow \mathsf{Mult}([t_0], [t_1])$;
2. return $[t_0t_1] = [\mathsf{encode}(x_0x_1/y_0y_1)]$.

$[\mathsf{encode}(x_0/y_0 - x_1/y_1)] = \mathsf{HgSubtr}\big([\mathsf{encode}(x_0/y_0)], [\mathsf{encode}(x_1/y_1)]\big)$

1. All parties compute $\mathsf{encode}(-1) = -1_{\text{field}} \in \mathbb{F}_p$;
2. all parties compute $[-t_1] = \mathsf{ScalarMult}(-1_{\text{field}}, [t_1])$;
3. $[t_0 - t_1] = \mathsf{HgAdd}([t_0], [-t_1])$;
4. return $[t_0 - t_1] = [\mathsf{encode}(x_0/y_0 - x_1/y_1)]$.

$[\mathsf{encode}(x_0y_1/y_0x_1)] = \mathsf{HgDiv}\big([\mathsf{encode}(x_0/y_0)], [\mathsf{encode}(x_1/y_1)]\big)$

1. $[t_1^{-1}] \leftarrow \mathsf{Inv}([t_1])$;
2. $[t_0t_1^{-1}] \leftarrow \mathsf{HgMult}([t_0], [t_1^{-1}])$;
3. return $[t_0t_1^{-1}] = [\mathsf{encode}(x_0y_1/y_0x_1)]$.

Table 2: *Total communication complexity (measured in field elements) of rational addition, multiplication, subtraction, and division protocols.*

Mercury		
Protocol	Rounds	Comm. Complexity
HgAdd	0	0
HgMult	1	$(2t-1)(n-1)$
HgSubtr	0	0
HgDiv	5	$t(5n-4)-n$

Remark 2. ScalarMult can be turned into HgScalarMult by simply encoding a public element $\alpha \in \mathcal{F}_N$, and then computing $\big[\mathsf{encode}(\alpha)s\big] = \mathsf{ScalarMult}\big(\mathsf{encode}(\alpha), [s]\big)$. Note that HgScalarMult also serves as a *division by public divisor* protocol - simply replace $\alpha \neq 0$ by $1/\alpha$.

4 Which rational numbers can we use?

All of our protocols use the aforementioned Farey rationals. As mentioned in lemma 1, $\mathcal{F}_N$ is closed under additive inverses and under multiplicative inverses, but it is not closed under addition and multiplication. This means that a suitable subset of $\mathcal{F}_N$ must be chosen as the set of rational inputs. In particular, we must include fractions with "small" numerators and denominators so that adding/multiplying these fractions yields fractions that remain in $\mathcal{F}_N$. Following closely the analysis of [16], this set will be chosen as

$$\mathcal{G}_{X,Y} := \big\{x/y \in \mathcal{F}_N \mid \ X, Y \in [0, N],\, |x| \leq X,\, 0 < y \leq Y\big\}.$$

4.1 Compatibility with fixed-point numbers

We now highlight our (partial) compatibility with fixed-point arithmetic. Further, many previous works designed their protocols with fixed-point arithmetic in mind. So, to facilitate comparison with prior art, we briefly discuss conditions under which $\mathcal{F}_N$ contains a set of fixed-point numbers.

Fixed-point numbers. These are rational numbers represented as a list of digits split by a radix point, and are defined by an integer (represented in a particular base b) in a given range along with a fixed scaling factor f. For example, we can represent decimal numbers with integral part in the range $(-10^{\ell+1}, 10^{\ell+1})$ and up to f decimal places after the radix point as $a \cdot 10^{-f} = a/10^f$, $a \in (-10^{\ell+f+1}, 10^{\ell+f+1})$. We will represent a set of fixed point numbers with a tuple of the form (b, ℓ, f), where b is the base, $(-b^{\ell+1}, b^{\ell+1})$ is range of the integer part, and up to f base-b digits are allowed after the radix point. The set of Farey rationals $\mathcal{F}_N$ contains the fixed-point numbers given by (b, ℓ, f) as long as

$$N \geq \max\{b^{\ell+f+1} - 1, b^f - 1\}. \tag{1}$$

Of course, N should be sufficiently large to ensure that adding/multiplying the fixed-point numbers does not cause overflow. While $\mathcal{F}_N$ can be made to contain a set of fixed-point numbers with precision f, addition and multiplication of Farey rationals does not quite coincide with addition and multiplication of fixed-point numbers. This is because the fixed-point representation requires the precision to remain f after each operation (this necessitates a truncation protocol), while $\mathcal{F}_N$ allows the precision to increase until overflow occurs and the output of a computation is no longer correct. However, this disparity mostly vanishes if the system running the protocols only allows precision f, because the "extra" bits that would be removed via a truncation protocol are removed by the system itself.

We will use the fact that $\mathcal{F}_N$ contains certain fixed-point numbers in Section 6 where we compare our protocols with prior work.

4.2 Compatible Circuits

Again borrowing from [16], for positive integers d, t we define (d,t)-*permitted (arithmetic) circuits* over $\mathbb{Q}$ to be those that compute a polynomial $\mathsf{p} \in \mathbb{Q}[x_1, x_2, \ldots]$ such that: (i) p has ℓ_1 norm at most t, (ii) total degree at most d, and (iii) coefficients with absolute value greater than or equal to 1. Note that nonzero polynomials $\mathsf{p} \in \mathbb{Z}[x_1, x_2, \ldots]$ with $\|\mathsf{p}\|_1 \leq t$ and $\deg(\mathsf{p}) \leq d$ satisfy (iii). Let $\mathcal{C}_{d,t}$ be the set of (d,t)-permitted circuits, and $\mathcal{P}_{d,t}$ be the set of polynomials the circuits compute. We obtain the following from [16, Proposition 7].

Lemma 2. *Let $C \in \mathcal{C}_{d,t}$ with inputs from $\mathcal{G}_{X,Y} \subseteq \mathcal{F}_N$. If x/y is the output of C, then $|x| \leq tX^dY^{d(t-1)}$ and $|y| \leq Y^{dt}$.*

Proof. Slight modification of the proof of [16, Proposition 7].

The bounds given in lemma 2 allow us to determine the (d,t)-permitted circuits with output fractions in $\mathcal{F}_N$ given inputs from $\mathcal{G}_{X,Y}$. Intuitively, the bound on x is larger than the bound on y because the numerator grows faster than the denominator when fractions are summed versus when they are multiplied. E.g. $a/b + c/d = (ad+bc)/bd$ versus $(a/b)(c/d) = ac/bd$. Some "space" is wasted if, for example, one chooses $X = Y$, because then the numerator bound becomes the limiting factor for ensuring the output of a permitted circuit remains in $\mathcal{F}_N$. For fixed d and t, this can be avoided by judiciously choosing X to be smaller than Y. In particular, X and Y should satisfy $\log_2(t)/d + \log_2(X) = \log_2(Y)$. Of course this is not possible for many applications, so typically the numerator bound will remain the limiting factor for depth of compatible circuits.

For example, suppose p is a prime such that $N = \lfloor\sqrt{(p-1)/2}\rfloor$ is 1024 bits, and let $X = 2^{32}$, $Y = 2^{14}$. According to Lemma 2, we can only evaluate circuits in $\mathcal{C}_{d,t}$ with inputs in $\mathcal{G}_{X,Y}$ if $t(2^{32})^d(2^{14})^{d(t-1)} \leq 2^{1024}$ or, equivalently, if $\log_2(t) + 18d + 14dt \leq 1024$. Notice how the numerator bound from Lemma 2 is the only one needed to determine d and t. The following table shows some choices for d and t.

Table 3: *Possible values of d (total degree of polynomial computed by permitted circuit) and t (ℓ_1 norm of polynomial computed by permitted circuit) for fractions with numerators bounded in absolute value by 2^{32} and denominators bounded by 2^{14}.*

$\lvert\text{num}\rvert \leq 2^{32}$, denom $\leq 2^{14}$						
d	1	2	3	4	5	10
t	71	35	22	16	13	6

This is not particularly useful, as many applications require thousands or even millions of additions to be performed on shared values. However, for many applications one is likely to work with decimal numbers with a small number of

significant digits; e.g., fractions with denominators a power of 10 between 7 bits and 17 bits. In such cases, we can significantly improve the bounds on d and t. In general, if the fractional data all have the same denominator, then Lemma 2 yields the following corollary.

Corollary 1. *Let $C \in \mathcal{C}_{d,t}$ with inputs from $\mathcal{F}_N$ whose denominators are all some power of integer base $b > 0$, say $B = b^e$, and whose numerators are bounded in absolute value by X. If x/y is the output of C, then $|x| \leq t(XB)^d$ and $y \leq B^d$.*

Rehashing the above example ($X = 2^{32}$ and N 1024 bits) with $B = 2^{14}$ we get $t(2^{32}2^{14})^d \leq 2^{1024} \implies \log_2(t) \leq 1024 - 46d$ and $(2^{14})^d \leq 2^{1024} \implies d \leq 73$. The bound on d is in fact even smaller: since the ℓ_1 norm of a polynomial in $\mathcal{P}_{d,t}$ is at least 1, $\log_2(t) \geq 0 \implies 1024 - 46d \geq 0 \implies 22 \geq d$. This yields the following table.

Table 4: *Possible values of d (total degree of polynomial computed by a permitted circuit) and t (ℓ_1 norm of polynomial computed by permitted circuit) for fractions with numerators bounded in absolute value by 2^{32} and denominators all equal to 2^{14}*

$\lvert\text{num}\rvert \leq 2^{32}$, denom $= 2^{14}$						
d	1	2	10	15	20	22
t	2^{978}	2^{932}	2^{564}	2^{334}	2^{104}	2^{12}

So if we restrict inputs to have the same denominators, we can perform an enormous number of additions and a reasonable number of multiplications before the output lands outside of $\mathcal{F}_N$. We can do even better though.

Degree-constant circuits Each gate of an arithmetic circuit computes a polynomial over (some of) the inputs. We define a *degree-constant (arithmetic) circuit* to be one in which every gate computes a polynomial whose monomial summands all have the same degree; e.g., a dot product. The goal of introducing these circuits is to ensure that whenever two fractions are summed, they already have a common denominator.

Corollary 2. *Let $C \in \mathcal{C}_{d,t}$ be degree-constant with inputs from $\mathcal{F}_N$ whose denominators are all $B = b^e$ and whose numerators are bounded in absolute value by $X > 0$. If x/y is the output of C, then $|x| \leq tX^d$ and $y \leq B^d$*

Proof. This follows easily from the fact that whenever two terms are added during the evaluation of a degree-constant circuit, they already have a common denominator which is a power of B.

Again, using a 1024 bit N, $X = 2^{32}$, and $B = 2^{14}$, we get the inequalities $\log_2(t) \leq 1024 - 32d$ and $d \leq 32$, yielding the following table.

Table 5: *Possible values of d (total degree of polynomial computed by a permitted circuit) and t (ℓ_1 norm of polynomial computed by permitted circuit) for degree-constant $C \in \mathcal{C}_{d,t}$ taking inputs from $\mathcal{F}_N$ with numerators bounded in absolute value by 2^{32} and denominators all equal to 2^{14}*

$\lvert\text{num}\rvert \leq 2^{32}$, $\text{denom} = 2^{14}$						
d	1	2	10	15	25	31
t	2^{992}	2^{960}	2^{704}	2^{544}	2^{384}	2^{32}

Incorporating division Once we allow divisions, the bounds given in Corollary 1 and Corollary 2 no longer apply, since the numerator of the divisor becomes a factor of denominator of the quotient. This means we should perform any necessary divisions as late as possible relative to other operations.

5 Optimizations

The complexity of HgMult and HgDiv can be reduced by executing parts of the protocols asynchronously in an *offline phase*. This allows certain parts of the protocols to be executed before the desired computation, thereby reducing the *online* complexity. The complexity of the offline phase depends on chosen primitives, existence of a trusted dealer, etc. Henceforth, we emphasize the online round complexity and the online communication complexity.

We utilize two ubiquitous tools for optimization: Beaver triples (introduced in [3]) for more efficient multiplication, and Pseudo-Random Secret Sharing (PRSS, [11]) to generate random field elements *sans* interaction.

In PRSS, the parties agree on a pseudo-random function (PRF) $\psi_\cdot(\cdot)$ and a common input a. They then use pre-distributed keys r_A (one for each set A of $n-t$ parties) to locally compute shares of a random field element s using Replicated Secret Sharing (see [11] for details). The use of PRSS reduces the online round and communication complexity of RandInt from 1 and $t(n-1)$ to 0 and 0, respectively. Further, we assume that the PRF, common input, and keys are agreed upon and distributed during a *set-up phase*, whence using PRSS makes the offline round and communication complexity of RandInt both 0.

Beaver triples are 3-tuples of shares $([a],[b],[c])$ satisfying $ab = c$, and can be generated asynchronously in the offline phase using PRSS and Mult. These triples can be used to multiply secrets without interaction. In particular, shares $[xy]$ can be obtained from $[x]$ and $[y]$ using only Add, ScalarMult, Recon, Output, and one Beaver triple $([a],[b],[c])$:

$$[xy] = (x+a)[y] - (y+b)[a] + [c].$$

Used triples must be discarded, else information is leaked. This means that a sufficiently-large reservoir of Beaver triples should be maintained to allow the desired functions to be computed.

These optimizations reduce the online complexities of HgMult and HgDiv, and leave the complexities of HgSubtr and HgAdd the same. Table 6 below summarizes the improvements.

Table 6: *Optimized round and communication complexities for our protocols. Note that the nonzero offline communication complexities come from* Mult *during Beaver triple generation.*

Protocol	Online Rounds	Offline Rounds	Online Comm.	Offline Comm.
HgAdd	0	0	0	0
HgMult	0	0	0	$(2t-1)(n-1)$
HgSubtr	0	0	0	0
HgDiv	2	2	$(t-1)+(n-1)$	$2(2t-1)(n-1)$

We use the complexities listed in table 6 for the comparisons in section 6. Henceforth, "rounds" will mean "online rounds + offline rounds", and "total communication" will mean "online communication + offline communication".

6 Comparison with Prior Work

In [9], Catrina and Saxena introduced semi-honest secure protocols for fixed-point multiplication and division - their division is based on (the iterative) Goldschmidt's method. A variant of their protocol is used by MP-SPDZ for fixed-point division. Catrina subsequently improved the round and communication complexities in [7]. To measure the complexities of their protocols, they use the set of fixed-point numbers given by $(2, 2f, f)$; i.e., the set of rationals $a \cdot 2^{-f}$ with $a \in [-2^{-2f}, 2^{2f}) \cap \mathbb{Z}$. Their fixed-point encoding technique requires a field $\mathbb{F}_q$ with $q > 2^{2f+\kappa}$, κ a statistical security parameter. For our protocols, we use the same field $\mathbb{F}_q$, whence our set of rationals is $\mathcal{F}_N$ with $N = N(q)$; specifically $\log_2(N) \geq f + \kappa/2$. Table 7 shows that for reasonable values of n and t (e.g. $n = 3$, $t = 2$), our protocols far outperform those of [7].

Table 7: *Complexity comparison between our work and that of [7]. Both the online and offline communication complexity are measured in elements of $\mathbb{F}_q$ sent among **all** parties throughout a protocol. n and t are the number of parties and the threshold, resp., θ is the number of iterations of Goldschmidt's method, and f is the fixed-point precision.*

	Protocol	Rounds	Online Comm.	Offline Comm.
Mercury	Multiplication	0	0	$(2t-1)(n-1)$
	Division	3	$(t-1)+(n-1)$	$2(2t-1)(n-1)$
[7]	Multiplication	1	1	f
	Division	$9+\theta$	$10f+2\theta$	$16f+4\theta f$

For example, the authors suggest $f = 32$ and $\theta = 5$, which results in a 14 round division with online communication complexity 320 field elements. Taking $n = 3$ and $t = 2$, our division requires 2 rounds, and has online communication complexity 3 field elements. There is, however, a bit more subtlety to this comparison. As mentioned in Section 4.1, operations on fixed-point numbers require a truncation, and the protocols of Catrina et al. use truncation. Consequently, there is no limit to how many times they can multiply/divide two fixed-point numbers. However, there is a number of multiplications, say, that will render their outputs of little use because so many bits have been truncated. Our limitation, on the other hand, is overflow: computations over $\mathcal{F}_N$ are only meaningful if all intermediate outputs and the final output are in $\mathcal{F}_N$. We can address this in two ways: (i) only take inputs from the subset $\mathcal{G}_{X,Y} \subseteq \mathcal{F}_N$ defined in section 4, for X, Y sufficiently smaller than N, or (ii) use a larger field than [7]. As long as we don't choose too large a field, (ii) will preserve our complexity advantage.

Another interesting solution, albeit only for integer division, was proposed by Veugen and Abspoel in [20]. They propose three division variations: public divisor, private divisor (only one party knows the divisor), and secret divisor (hidden from all parties). Their protocols are implemented using MP-SPDZ with three parties, and runtimes along with communication complexities (in MB) for dividing a k-bit integer by a $k/2$-bit integer are provided. Even though our division protocol uses rationals in general, comparison makes sense because $\mathcal{F}_N$ contains the integers $[-N, N] \cap \mathbb{Z}$ (see lemma 1). For comparison, we use $n = 3$ and $t = 2$, and use the smallest prime field $\mathbb{F}_p$ allowed by [20]:

$$\log_2(p) \approx 4 \max\left\{ \log_2(\text{dividend}), \log_2(\text{divisor}) \right\} + 40$$

E.g., this means that for a 64 bit dividend and 32 bit divisor, we have $\log_2(p) = 296$ and $N = N(p) \approx 148$ bits.

The reader should take the comparison in table 8 with a grain of salt since our numbers were estimated and not obtained from implementation.

Table 8: *Total communication complexity in megabytes (MB) of our division protocol (applied to integers) vs. the (secret divisor) integer division protocol of [20]. The communication complexities for our work were estimated using table 7.*

dividend bits	8	16	32	64
divisor bits	4	8	16	32
Mercury	0.001MB	0.0015MB	0.0024MB	0.0042MB
[20] (semi-honest security)	8.6MB	32.6MB	121.0MB	492.7MB

The last comparison we shall show is against Pika [21]. Pika uses Function Secret Sharing [6] to construct a three-party protocol (one party is used only to generate correlated randomness) for computing functions such as reciprocal, sigmoid, and square root. Their protocol Pika takes as inputs (binary) fixed-point numbers x with precision f, such that $x \cdot 2^f \in (-2^{k-1}, 2^{k-1}]$, and creates shares in the ring $\mathbb{Z}_{2^\ell}$, where $\ell \geq 2k$. For comparison, we choose $N = N(p) = 2^{k-1}$ (meaning we share secrets over $\mathbb{F}_p$ with $p \approx 2^{2k}$). This guarantees that $\mathcal{F}_N$ contains the fixed-point numbers used by Pika regardless of the chosen precision f. As with the preceding comparisons, we again take $n = 3$ and $t = 2$.

Using the same parameter values for (semi-honest secure) Pika as the author, we found that that total communication complexity for securely computing the reciprocal with $k = 16$ and $\ell = 32$ was 8524 bits over three rounds (one offline). In contrast, we can compute the reciprocal of an element of $\mathcal{F}_{2^{15}}$ in three rounds (one offline) with communication complexity $9\log_2(p) \approx 18k = 288$ bits.

7 Conclusions and Future Work

Conclusion This work uses Shamir Secret Sharing with a minority of semi-honest adversaries, but Mercury is flexible in the sense that it can be easily realized over other primitives with better security assumptions; e.g. additive secret sharing *à la* MP-SPDZ along with a majority of malicious adversaries. Mercury provides an efficient low round and communication complexity solution to exact computation over rational numbers using MPC. A cost of exactness, though, is that our protocols are not intrinsically compatible with fixed-point arithmetic. Instead of truncating after every operation to not exceed the chosen fixed-point precision, we allow the precision to grow until overflow occurs. This means that we may need to work over a larger field $\mathbb{F}_p$ than prior art ([7,20,21]), but our communication and round complexity are sufficiently low as to make using a slightly larger field not problematic.

Future work Our first step is to implement Mercury to facilitate more comprehensive comparison with existing protocols. As alluded to in the introduction, even though Mercury is built on SSS, its protocols could easily be adapted to use

additive secret sharing, meaning we can implement Mercury using MP-SPDZ. We are also investiging an idea for a novel truncation protocol which will add the flexibility for Mercury to use either fixed-point or floating-point numbers in any base. Such a protocol would serve as a building block for other protocols (equality testing and comparison, for e.g.) that will make Mercury a more complete and versatile family of protocols for secure computation over rational numbers.

Acknowlegments

The authors warmly thank Professor Jonathan Katz for reading early drafts of this paper, and providing helpful insights and suggestions. This work is fully supported by Algemetric Inc.

References

1. Acar, A., Aksu, H., Uluagac, A.S., Conti, M.: A survey on homomorphic encryption schemes: Theory and implementation. ACM Computing Surveys (CSUR) **51**(4), 1–35 (2018)
2. Aliasgari, M., Blanton, M., Zhang, Y., Steele, A.: Secure computation on floating point numbers. In: NDSS 2013. The Internet Society (Feb 2013)
3. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO'91. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (Aug 1992). https://doi.org/10.1007/3-540-46766-1_34
4. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: 20th ACM STOC. pp. 1–10. ACM Press (May 1988). https://doi.org/10.1145/62212.62213
5. Bogetoft, P., Christensen, D.L., Damgard, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M., Toft, T.: Multiparty computation goes live. Cryptology ePrint Archive, Report 2008/068 (2008), https://eprint.iacr.org/2008/068
6. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 337–367. Springer, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46803-6_12
7. Catrina, O.: Round-efficient protocols for secure multiparty fixed-point arithmetic. In: 2018 International Conference on Communications (COMM). pp. 431–436 (2018). https://doi.org/10.1109/ICComm.2018.8484794
8. Catrina, O.: Efficient secure floating-point arithmetic using shamir secret sharing. In: International Conference on E-Business and Telecommunication Networks (2019)
9. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 35–50. Springer, Heidelberg (Jan 2010)
10. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Report 2016/421 (2016), https://eprint.iacr.org/2016/421
11. Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 342–362. Springer, Heidelberg (Feb 2005). https://doi.org/10.1007/978-3-540-30576-7_19

12. Cramer, R., Damgård, I.B., et al.: Secure multiparty computation. Cambridge University Press (2015)
13. Damgård, I., Ishai, Y.: Scalable secure multiparty computation. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 501–520. Springer, Heidelberg (Aug 2006). https://doi.org/10.1007/11818175_30
14. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In: Coan, B.A., Afek, Y. (eds.) 17th ACM PODC. pp. 101–111. ACM (Jun / Jul 1998). https://doi.org/10.1145/277697.277716
15. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC. pp. 218–229. ACM Press (May 1987). https://doi.org/10.1145/28395.28420
16. Harmon, L., Delavignette, G., Roy, A., Silva, D.: Pie: p-adic encoding for high-precision arithmetic in homomorphic encryption. In: Tibouchi, M., Wang, X. (eds.) Applied Cryptography and Network Security. pp. 425–450. Springer Nature Switzerland, Cham (2023)
17. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1575–1590. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3417872
18. Lindell, Y.: Secure multiparty computation (MPC). Cryptology ePrint Archive, Report 2020/300 (2020), https://eprint.iacr.org/2020/300
19. Shamir, A.: How to share a secret. Communications of the Association for Computing Machinery **22**(11), 612–613 (Nov 1979)
20. Veugen, T., Abspoel, M.: Secure integer division with a private divisor. PoPETs **2021**(4), 339–349 (Oct 2021). https://doi.org/10.2478/popets-2021-0073
21. Wagh, S.: Pika: Secure computation using function secret sharing over rings. Cryptology ePrint Archive, Report 2022/826 (2022), https://eprint.iacr.org/2022/826
22. Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS. pp. 162–167. IEEE Computer Society Press (Oct 1986). https://doi.org/10.1109/SFCS.1986.25