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

No Author Given

No Institute Given

Abstract. Most protocols for secure multi-party computation (MPC) work over fields or rings, which means that encoding techniques are needed to map rational-valued data into the algebraic structure being used. Leveraging an encoding technique introduced in recent work of Harmon et al. that is compatible with any MPC protocol over a prime-order field, 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 improvements in both round complexity and communication complexity when compared with prior art, and are secure for a dishonest minority of semi-honest parties.

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. A well-studied form of secure computation, and the focus of this work, is Multi-party Computation (MPC). In the classic setting, n mutually distrusting parties P_i possess private data d_i and wish to jointly compute a function $F(d_1, \ldots, d_n)$ without revealing any information about honest parties' inputs to any coalition of corrupted parties. This problem was first studied in detail by Yao [25]. 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., [7, 15, 16, 18, 20, 21]). 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 [22] 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 [20].

Most MPC protocols are defined over finite rings or fields such as $\mathbb{Z}/m\mathbb{Z}$ 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 to evaluate certain functions (e.g., polynomials) over secret shared values, the encoding method must be homomorphic with respect to the operations that compose the function (e.g., addition and multiplication). Several works [10,12,24] encode a fixed-point or floating-point number a with f digits after the radix point as the integer $a \cdot 2^f$. Other approaches [2,11] 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 that set, we start with a set of rationals (with bounded numerators and denominators) that contains some set of fixed-point numbers. This set of rationals, paired with an encoding of those rationals as elements of a ring/field 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 $\mathbb{Z}/m\mathbb{Z}$, however we focus on the case of m prime. This means that, for the most part, our protocols are obtained by simply composing the encoder with existing protocols. An exception is the way we handle division, which relies heavily on the structure of the aforementioned set of rationals.

All our protocols for the basic arithmetic operations require only a constant number of rounds and have communication complexity at most $O(tn)$ field elements, where n is the number of parties and t is a bound on the number of corrupted parties.[1] 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., as used in MP-SPDZ to tolerate all-but-one corrupted party).

The paper is organized as follows:

* Section 2 discusses notation and provides an overview of Shamir's scheme, and some "building block" protocols.
* Section 3 introduces the rational-to-ring-element encoder, and Mercury: our protocols for rational addition, multiplication, subtraction, and division. We close by discussing the security and correctness of our 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 domain of the encoder that allows for evaluation of (arithmetic) circuits up to a certain multiplicative depth. We end with an example of securely computing the kurtosis of a dataset held distributively by n parties.
* 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 [10, 12, 23, 24].

[1] After optimizations, the *online* communication complexity of our protocols is at most $O(t + n)$ field elements.

∗ 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 m, $\mathbb{Z}/m\mathbb{Z}$ denotes the ring of integers modulo m. In case m is prime, we write $\mathbb{F}_m$. The elements of $\mathbb{Z}/m\mathbb{Z}$ will be represented by integers $0, 1, \ldots, m-1$. For a ring R, $R[x_1, x_2, \ldots]$ will denote the ring of polynomials in the variables $x_1, x_2, \ldots$ with coefficients in R. For $\mathsf{p} \in \mathbb{Q}[x_1, x_2, \ldots]$, $\|\mathsf{p}\|_1$ denotes the ℓ_1 norm of p, i.e., the sum of the absolute values of the coefficients of p. 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 $\mathbb{F}_p$, and have gates with fan-in 2.

2.2 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 $t+1 \leq n$ parties to be able to reconstruct a secret by pooling their shares. This is called a $(t+1)$-out-of-n threshold scheme. One creates *Shamir shares* of a secret $s \in \mathbb{F}_p$, where $|\mathbb{F}_p| \geq n$, by generating a random polynomial $f(x) \in \mathbb{F}_p[x]$ of degree at most t 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 $f(i)$, $i \in \mathbb{F}_p \backslash \{0\}$. 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 $t+1$ parties can pool their shares $[s]$ and reconstruct the polynomial f using Lagrange interpolation, thereby obtaining the secret $s = f(0)$.

2.3 Framework

We use $(t+1)$-out-of-n SSS over $\mathbb{F}_p$ for all protocols, and assume that all parties are connected by pair-wise secure channels which they use to send and receive shares when necessary. The communication complexity of a protocol is measured by the number of field elements sent by all parties in the protocol. When comparing our work with existing protocols that measure communication complexity in bits, we simply multiply our communication complexities by $\log_2(p)$. All adversaries are assumed to be semi-honest (honest-but-curious), and we tolerate at most t of them. As previously mentioned, given sharings $[x]$ and $[y]$, the parties can compute a sharing $[x+y]$ of their sum without interaction by computing $[x] + [y]$. For multiplication of shared values we use the protocol of Gennaro et al. [17], which is itself an optimization of the multiplication subroutine in the BGW protocol [6]. To ensure the multiplication protocol works, we require $t < n/2$. We present the details of the relevant building block protocols in the next section.

2.4 Building Blocks

SSS primitives will be the algorithms/protocols Share, Add, $\mathsf{ScalarMult}$, and Mult. Respectively, these create shares of a secret value $s \in \mathbb{F}_p$, 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. Add and $\mathsf{ScalarMult}$ are non-interactive. Borrowing from [2], we use $s \leftarrow \mathsf{Output}([s])$ to mean that each of a set of t parties send their share of s to another party, which subsequently reconstructs s from the shares $[s]$ and sends s to the other $n-1$ parties. We pause briefly to describe the multplication protocol from [17].

Fig. 1: Multiplying shared secrets without revealing.

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

1. For $i = 1, \ldots, n$, the $i^{\text{th}}$ party $P_i$ computes $z_i = [x]_i \cdot [y]_i$;
2. For $i = 1, \ldots, 2t+1$, $P_i$ computes $[z_i] \leftarrow \mathsf{Share}(z_i)$, and sends the $j^{\text{th}}$ share
   to $P_j$;
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, $\mathsf{RandInt}$ and Inv ([3], Lemma 6), are required for our rational division protocol. These protocols allow all parties to obtain shares of a random field element (Figure 2) and compute shares of the multiplicative inverse of a field element (Figure 3), respectively.

Fig. 2: Generating shares of a random element of $\mathbb{F}_p$.

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

1. Each of a set of $t+1$ parties select a uniform $r_i \in \mathbb{F}_p$, $i = 1, \ldots, t+1$;
2. each of the $t+1$ parties computes $[r_i] \leftarrow \mathsf{Share}(r_i)$, and sends the $j^{\text{th}}$ share
   to $P_j$;
3. $[r] = [r_1] + \cdots + [r_{t+1}]$;
4. return $[r]$.
```

Fig. 3: Calculating shares of a multiplicative inverse in $\mathbb{F}_p$.

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

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	$n-1$
Output	2	$t+(n-1)$
Add	0	0
Mult	1	$(2t+1)(n-1)$
ScalarMult	0	0
RandInt	1	$(t+1)(n-1)$
Inv	4	$(t+1)(3n-2)-1$

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 mapping certain rational numbers to field elements with compositions of the building block protocols described in Section 2.4. The protocols for computing the sum and product of shared fractions remain unchanged from the analogous SSS primitives, except that rational operands are encoded to field elements before those protocols are executed. Subtraction and Division are an amalgam of the building blocks. Division, in particular, 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 [19] which maps a subset of rationals, whose denominators are co-prime with a prime p, into $\mathbb{F}_p$. This map is defined by $\mathsf{encode}(x/y) = xy^{-1} \bmod p$, with 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 $N = N(p) := \lfloor\sqrt{(p-1)/2}\rfloor$. Notice that $\mathcal{F}_N$ is *not* closed under addition and multiplication.
The map 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] The inverse operations decode can be computed efficiently using a slight modification of the Extended Euclidean Algorithm. We summarize important properties of encode, decode, and $\mathcal{F}_N$ in 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) $[-N, N] \cap \mathbb{Z} \subseteq \mathcal{F}_N$. Moreover, if $z \in [0, N] \cap \mathbb{Z}$, then $\mathsf{encode}(z) = z$.
(iv) 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)-(iii) are obvious. (iv) is proved in [19, Proposition 2].

3.2 Rational Addition, Multiplication, Subtraction, and Division

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

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

protocols, we use the field $\mathbb{F}_p$, and assume $x_0/y_0, x_1/y_1 \in \mathcal{F}_N$. Our addition and multiplication protocols HgAdd and HgMult are obtained by simply pairing the encoder with Add and Mult, respectively. As such, we omit the descriptions of both protocols. The remaining two protocols, $\mathsf{HgSubtr}$ and HgDiv are introduced below.

Remark 1. We use the prefix "Hg" for our protocols because it is the chemical symbol for the element Mercury.

Let $\mathsf{enc}_0 = \mathsf{encode}(x_0/y_0)$ and $\mathsf{enc}_1 = \mathsf{encode}(x_1/y_1)$.

Fig. 4: Mercury subtraction protocol.

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

1. All parties compute $\mathsf{encode}(-1) = -1_{\text{field}} \in \mathbb{F}_p$;
2. all parties compute $[-\mathsf{enc}_1] = \mathsf{ScalarMult}(-1_{\text{field}}, [\mathsf{enc}_1])$;
3. $[\mathsf{enc}_0 - \mathsf{enc}_1] = \mathsf{HgAdd}([\mathsf{enc}_0], [-\mathsf{enc}_1])$;
4. return $[\mathsf{enc}_0 - \mathsf{enc}_1] = [\mathsf{encode}(x_0/y_0 - x_1/y_1)]$.

Fig. 5: Mercury division protocol.

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

1. $[\mathsf{enc}_1^{-1}] \leftarrow \mathsf{Inv}([\mathsf{enc}_1])$;
2. $[\mathsf{enc}_0 \cdot \mathsf{enc}_1^{-1}] \leftarrow \mathsf{HgMult}([\mathsf{enc}_0], [\mathsf{enc}_1^{-1}])$;
3. return $[\mathsf{enc}_0 \cdot \mathsf{enc}_1^{-1}] = [\mathsf{encode}(x_0y_1/y_0x_1)]$.

Remark 2. $\mathsf{ScalarMult}$ can be turned into $\mathsf{HgScalarMult}$ 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 $\mathsf{HgScalarMult}$ also serves as a *division by public divisor* protocol - simply replace $\alpha \neq 0$ by $1/\alpha$.

Table 2: *Communication complexity (in field elements) of* Mercury *protocols.*

Mercury

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

3.3 Security and Correctness

It is well-known (see, e.g., [5]) that SSS is perfectly secure in the sense that possession of fewer than the threshold number of shares does not reveal any information about the secret. It is also easy to see that the building block protocols Share, Output, Add, ScalarMult, and Mult do not reveal any information, as the only information received by the parties are shares and no party ever receives enough shares to reconstruct. By invoking Canetti's composition theorem [9], which roughly states that a composition of secure protocols yields a secure protocol, we see that both RandInt and Inv are also secure.

The authors of [19] remark that for p an odd prime and $N = N(p)$, $\mathcal{F}_N$ is not in bijective correspondence with $\mathbb{F}_p$. In fact, $|\mathcal{F}_N| \approx 0.6p$. A consequence of this is that an attacker can reduce the set of possible secret encodings in $\mathbb{F}_p$ to $\mathsf{encode}(\mathcal{F}_N) \subsetneq \mathbb{F}_p$. This is not problematic, however, as each value in $\mathsf{encode}(\mathcal{F}_N)$ is equally likely to be the secret.
The following theorem provides necessary conditions for correctness of the Mercury protocols.

Theorem 1 (Correctness of Mercury protocols). *Let p be an odd prime and $N = N(p)$. Suppose $x_i/y_i \in \mathcal{F}_N$ with $\alpha_i = \mathsf{encode}(x_i/y_i)$, for $i = 0, 1$.*

(i) $\mathsf{decode}\big(\mathsf{HgAdd}(\alpha_0, \alpha_1)\big) = x_0/y_0 + x_1/y_1$ *as long as* $x_0/y_0 + x_1/y_1 \in \mathcal{F}_N$.

(ii) $\mathsf{decode}\big(\mathsf{HgMult}(\alpha_0, \alpha_1)\big) = x_0/y_0 \cdot x_1/y_1$ *as long as* $x_0/y_0 \cdot x_1/y_1 \in \mathcal{F}_N$.

(iii) $\mathsf{decode}\big(\mathsf{HgSubtr}(\alpha_0, \alpha_1)\big) = x_0/y_0 - x_1/y_1$ *as long as* $x_0/y_0 - x_1/y_1 \in \mathcal{F}_N$.

(iv) $\mathsf{decode}\big(\mathsf{HgDiv}(\alpha_0, \alpha_1)\big) = x_0/y_0 \div x_1/y_1$ *as long as* $x_0/y_0 \div x_1/y_1 \in \mathcal{F}_N$.

Proof. HgAdd is trivially correct if we ignore the encoded fractions and only consider field elements. That is, $\mathsf{HgAdd}(\alpha_0, \alpha_1) = \alpha_0 + \alpha_1$. So correctness is guaranteed as long as $\mathsf{decode}(\alpha_0 + \alpha_1) = x_0/y_0 + x_1/y_1$. Now, suppose $x_0/y_0 + x_1/y_1 \in \mathcal{F}_N$. Since decode is additively homomorphic when the sum remains in $\mathcal{F}_N$, $\mathsf{decode}(\alpha_0 + \alpha_1) = \mathsf{decode}(\alpha_0) + \mathsf{decode}(\alpha_1) = x_0/y_0 + x_1/y_1$, as desired. The correctness of the remaining Mercury protocols follows *mutatis mutandis.*

4 Which Rational Numbers Can We Use?

All our protocols use the aforementioned Farey rationals. As mentioned in Lemma 1, $\mathcal{F}_N$ is closed under additive inverses and multiplicative inverses, but is not closed under addition and multiplication. This means that for applications to MPC 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 those fractions yields fractions that remain in $\mathcal{F}_N$. Following closely the analysis of [19], this set will be chosen as

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

for some X, Y to be specified.

4.1 Fixed-Point Numbers

We now highlight our (partial) compatibility with fixed-point numbers. 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 given set of fixed-point numbers.

Fixed-point numbers 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 (called the *precision*). 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

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

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 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 truncation), while $\mathcal{F}_N$ allows the precision to increase until overflow occurs and the output of a computation is no longer correct. We will use the fact that $\mathcal{F}_N$ contains certain fixed-point numbers in Section 6 when we compare our protocols with prior work.

4.2 Compatible Circuits

Again borrowing from [19], for positive integers d, τ we define the class of (arithmetic) (d, τ)-*circuits* over $\mathbb{Q}$ to be those that compute a polynomial

$\mathsf{p} \in \mathbb{Q}[x_1, x_2, \ldots]$ such that p satisfying: (i) ℓ_1 norm is at most τ, (ii) total degree is at most d, and (iii) all nonzero coefficients have absolute value greater than or equal to 1. Note that nonzero polynomials $\mathsf{p} \in \mathbb{Z}[x_1, x_2, \ldots]$ with $\|\mathsf{p}\|_1 \le \tau$ and $\deg(\mathsf{p}) \le d$ satisfy (iii). Let $\mathcal{C}_{d,\tau}$ be the set of (d,τ)-circuits, and $\mathcal{P}_{d,\tau}$ be the set of polynomials those circuits compute. We obtain the following by slight modification of the proof of [19, Proposition 7], which allow us to determine d, τ so that evaluating any (d,τ)-circuit on inputs from $\mathcal{G}_{X,Y}$ will have output in $\mathcal{F}_N$.

Proposition 1. *Let $d, \tau \ge 1$. If x/y is the output of $C \in \mathcal{C}_{d,\tau}$ evaluated on inputs from $\mathcal{G}_{X,Y} \subseteq \mathcal{F}_N$, then $|x| \le \tau X^d Y^{d(\tau-1)}$ and $|y| \le Y^{d\tau}$.*

Proof. See Appendix A.

Intuitively, the bound on x is larger than the bound on y because the numerator grows faster than the denominator when fractions are summed (since $a/b + c/d = (ad+bc)/bd$), whereas they grow at the same rate when multiplied.
The following table shows some possible choices for d and τ if we use $\mathcal{G}_{2^{32},2^{14}} \subsetneq \mathcal{F}_{2^{1024}}$. Note that in this case, $p \approx 2^{2048}$.

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

|num| $\le 2^{32}$, denom $\le 2^{14}$

d	1	2	3	4	5	10
τ	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. In such cases, we can significantly improve the bounds on d and τ. In general, if the fractional data all have the same denominator, then Proposition 1 yields the following corollary.

Corollary 1. *Let $C \in \mathcal{C}_{d,\tau}$ with inputs from $\mathcal{F}_N$ whose denominators are all some fixed power e of an integer base b, $0 < b^e \le N$, and whose numerators are bounded in absolute value by $X \le N$. If x/y is the output of C, then $|x| \le \tau(Xb^e)^d$ and $y \le b^{ed}$.*

Proof. Note that $\mathsf{p} \in \mathcal{P}_{d,\tau}$ can be written as $\mathsf{p} = \sum_i c_i p_i$, where $\sum_i |c_i| \le \tau$, each $|c_i| \ge 1$, and each p_i is a monomial of degree at most d.
Let $\mathsf{p} = \sum_{i=1}^{I} c_i p_i$, and suppose we have k inputs x_i/b^e.

Since $\deg(p_i) \leq d$,

$$p_i\big(x_1/b^e, \ldots, x_k/b^e\big) = \frac{x_{i_1}x_{i_2}\cdots x_{i_\ell}}{b^{e\ell}}, \text{ for some } \ell \leq d \text{ and } \{i_1, \ldots, i_\ell\} \subseteq \{1, \ldots, k\}.$$

As each $|x_i| \leq X$, we have $|x_{i_1}x_{i_2}\cdots x_{i_\ell}| \leq X^\ell \leq X^d$.
Now, if $x/y = \sum_{i=1}^{I} c_i \cdot p_i\big(x_1/b^e, \ldots, x_k/b^e\big)$, then

$$x = (c_1a_1)b^{e(I-1)} + (c_2a_2)b^{e(I-1)} + \cdots + (c_Ia_I)b^{e(I-1)} \text{ and } y = b^{eI}.$$

It follows that $|x| \leq \sum_{i=1}^{I} |c_i|(Xb^e)^I \leq \tau \cdot (Xb^e)^I$ and $|y| \leq b^{eI}$. The proof is completed by observing that $|c_\alpha| \geq 1$, for all α, implies $I \leq \tau$.

Rehashing the above example ($X = 2^{32}$ and $N \approx 2^{1024}$) with $b^e = 2^{14}$ we get $\tau(2^{32}2^{14})^d \leq 2^{1024} \implies \log_2(\tau) \leq 1024 - 46d$ 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 $\mathcal{P}_{d,\tau}$ is at least 1, $\log_2(\tau) \geq 0 \implies 1024 - 46d \geq 0 \implies 22 \geq d$. This yields the following table.

Table 4: *Possible values of d (total degree of polynomial computed by a (d,τ)-circuit) and τ (ℓ_1 norm of polynomial computed by (d,τ)-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
τ	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 enourmous 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 $C \in \mathcal{C}_{d,\tau}$ be degree-constant with inputs from $\mathcal{F}_N$ whose denominators are all b^e and whose numerators are bounded in absolute value by $X > 0$. If x/y is the output of C, then $|x| \leq \tau X^d$ and $y \leq b^{ed}$*

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^e.

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

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

$|\text{num}| \leq 2^{32}$, denom $= 2^{14}$

d	1	2	10	15	25	31
τ	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.

4.3 An Application: Computing Excess Kurtosis

The *excess kurtosis* of a distribution is a measure of its "tailedness" relative to a normal distribution: low excess kurtosis (< 0) means the distribution has thin tails while high excess kurtosis (> 0) means the distribution has thick tails. This measure is frequently-used in descriptive analytics and rather involved to calculate, which makes it a good candidate computation for Mercury. We derive below the parameters necessary to guarantee that the excess kurtosis of a sample of size k remains in $\mathcal{F}_N$. The excess kurtosis EK_s is defined as

$$EK_s = \frac{k(k+1)\sum_{i=1}^{k}\left(x_i - \bar{x}\right)^4}{(k-1)(k-2)(k-3)s^4} - \frac{3(k-1)^2}{(k-2)(k-3)} \text{ for } k \geq 4$$

where k is the size of the sample, s^2 the variance, and $\bar{x}$ is the mean of the sample. For simplicity, and to avoid calculating s^2 and $\bar{x}$ separately, the formula can be rewritten as

$$EK_s = \frac{k(k+1)(k-1)\sum_{i-1}^{k}\Big(kx_i - \sum_{i=1}^{k} x_i\Big)^4 - 3\big(k-1\big)^2\Big(\sum_{i=1}^{k}\big(kx_i - \sum_{i=1}^{k} x_i\big)^2\Big)^2}{\big(k-2\big)\big(k-3\big)\Big(\sum_{i=1}^{k}\big(kx_i - \sum_{i=1}^{k} x_i\big)^2\Big)^2}$$

Assuming that we need to compute the excess kurtosis of a sample of about one billion ($\approx 2^{30}$), and using data with denominators 2^{14} and numerators less than 2^{32}, as in table 5. We determine that the numerator of EK_s is bounded by $k^8 2^{134}$ and the denominator is bounded by $k^6 2^{132}$. Therefore, to guarantee $EK_s \in \mathcal{F}_N$, it suffices to take $N \geq k^8 2^{134}$. Using $N \geq \big(2^{30}\big)^8 2^{134}$, we get $N \geq 2^{374}$, or a p on the order of 2^{749}.

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 [4]) for more efficient multiplication, and Pseudo-Random Secret Sharing (PRSS, [14]) to generate random field elements without 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 $n-(t+1)$ parties) to locally compute shares of a random field element s using Replicated Secret Sharing (see [14] for details). The use of PRSS reduces the online round and communication complexity of RandInt from 1 and $(t+1)(n-1)$ 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 with only two online rounds of interaction. In particular, shares $[xy]$ can be obtained from $[x]$ and $[y]$ using only Add, ScalarMult, 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.*

Optimizations	Protocol	Online Rounds	Offline Rounds	Online Comm.	Offline Comm.
PRSS and Beaver triples	HgAdd	0	0	0	0
	HgSubtr	0	0	0	0
	HgMult	2	1	$t + (n-1)$	$(2t+1)(n-1)$
	HgDiv	6	2	$3t + 3(n-1)$	$2(2t+1)(n-1)$
PRSS	HgMult	1	0	$(2t+1)(n-1)$	0
	HgDiv	4	0	$(4t+3)(n-1)$	0

The reader may notice that optimizing Mult actually *increases* the round complexity in the online phase from 1 to 2. This results from invocations of Output (executed in parallel) which requires 2 rounds per invocation, and is the cost of reducing the online communication from $O(tn)$ to $O(t + n)$. A user preferring instead to minimize the round complexity can do so by not using Beaver triples. Table 6 lists the optimized complexities of the Mercury protocols, along with the complexities of HgMult and HgDiv obtained by using PRSS but not Beaver triples.

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 [12], 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 [10]. 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 [10].

Let $n = 3$ and $t = 1$, so two parties can reconstruct. In an example, the authors choose $f \in [32, 56]$ and $\theta = 5$, which results in a 14 round division with online communication complexity $330n = 990$ field elements. In contrast, our division requires 8 rounds, and has online communication complexity 9 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

Table 7: *Complexity comparison between our work (optimized with PRSS and Beaver triples) and that of [10]. 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	3	$t + (n-1)$	$(2t+1)(n-1)$
	Division	8	$3t + 3(n-1)$	$2(2t+1)(n-1)$
[10]	Multiplication	1	n	nf
	Division	$9+\theta$	$n(10f + 2\theta)$	$n(16f + 4\theta f)$

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 the beginning of section 4, for X, Y sufficiently smaller than N, or (ii) use a larger field than [10]. 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 [23]. They present 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 $t = 1$, and use the smallest prime field $\mathbb{F}_p$ allowed by [23]:

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

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 last comparison we shall show is against Pika [24]. Pika uses Function Secret Sharing [8] 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 $\mathcal{F}_N$

Table 8: *Total communication complexity in megabytes (MB) of our division protocol (applied to integers) vs. the (secret divisor) integer division protocol of [23]. The communication complexity for (fully optimized)* HgDiv *was estimated using table 6.*

dividend bits	8	16	32	64
divisor bits	4	8	16	32
Mercury	0.00018MB	0.00026MB	0.00042MB	0.00074MB
[23] (semi-honest security)	8.6MB	32.6MB	121.0MB	492.7MB

contains the fixed-point numbers used by Pika regardless of the chosen precision f. As with the preceding comparisons, we take $n = 3$ and $t = 1$.
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 6 rounds (one offline) with communication complexity $21 \log_2(p) \approx 21 \cdot 2k = 672$ 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 ([10,23,24]), 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 mentioned 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.

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. Bar-Ilan, J., Beaver, D.: Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In: Rudnicki, P. (ed.) 8th ACM PODC. pp. 201–209. ACM (Aug 1989). https://doi.org/10.1145/72981.72995
4. 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
5. Beimel, A.: Secret-sharing schemes: A survey. pp. 11–46 (05 2011). https://doi.org/10.1007/978-3-642-20901-7_2
6. 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
7. 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
8. 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
9. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology **13**(1), 143–202 (Jan 2000). https://doi.org/10.1007/s001459910006
10. 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
11. Catrina, O.: Efficient secure floating-point arithmetic using shamir secret sharing. In: International Conference on E-Business and Telecommunication Networks (2019)
12. 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)
13. 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
14. 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
15. Cramer, R., Damgård, I.B., et al.: Secure multiparty computation. Cambridge University Press (2015)
16. 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

17. 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
18. 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
19. 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)
20. 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
21. Lindell, Y.: Secure multiparty computation (MPC). Cryptology ePrint Archive, Report 2020/300 (2020), https://eprint.iacr.org/2020/300
22. Shamir, A.: How to share a secret. Communications of the Association for Computing Machinery **22**(11), 612–613 (Nov 1979)
23. 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
24. Wagh, S.: Pika: Secure computation using function secret sharing over rings. Cryptology ePrint Archive, Report 2022/826 (2022), https://eprint.iacr.org/2022/826
25. 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

A Proofs

Proof (of Proposition 1). Note that $\mathsf{p} \in \mathcal{P}_{d,\tau}$ can be written as $\mathsf{p} = \sum_i c_i p_i$, where $\sum_i |c_i| \leq \tau$, each $|c_i| \geq 1$, and each p_i is a monomial of degree at most d. Let $\mathsf{p} = \sum_{i=1}^{I} c_i p_i$. Since $\deg(p_i) \leq d$, the output $p_i\big(x_1/y_1, \ldots, x_k/y_k\big)$ is a fraction of the form

$$\frac{a_i}{b_i} = \frac{x_{i_1} x_{i_2} \cdots x_{i_\ell}}{y_{i_1} y_{i_2} \cdots y_{i_\ell}}, \text{ for some } \ell \leq d \text{ and } \{i_1, \ldots, i_\ell\} \subseteq \{1, \ldots, k\}.$$

As each $x_i/y_i \in \mathcal{G}_M$, we have $|a_i| \leq X^\ell \leq X^d$ and $|b_i| \leq Y^\ell \leq Y^d$.

Since $x/y = \sum_{i=1}^{I} c_i \cdot a_i/b_i$,

$$x = (c_1 a_1) b_2 b_3 \cdots b_I + b_1 (c_2 a_2) b_3 \cdots b_I + b_1 b_2 \cdots b_{I-1} (c_I a_I) \text{ and}$$
$$y = b_1 b_2 \cdots b_I.$$

It follows from $\sum |c_i| \leq \tau$ and the above bound on $|a_i|, |b_i|$ that

$$|x| \leq \sum_{i=1}^{I} |c_i| (X^d)(Y^d)^{I-1} \leq \tau \cdot X^d Y^{d(I-1)} \text{ and } |y| \leq Y^{d(I-1)}.$$

The proof is completed by observing that $|c_\alpha| \geq 1$, for all α, implies $I \leq \tau$.