

Modulus Operandi: Round-Optimal Modulus Conversion for MPC and Threshold FHE

Ivan Damgård , Sebastian Hasler , Sebastian Kolby , and Stanislas Pawlak 

Aarhus University

`ivan@cs.au.dk`, `sebastian@hasler.dev`, `sk@cs.au.dk`, `stanislas.pawlak@cs.au.dk`

Abstract. In secure multiparty computation (MPC), *modulus conversion* turns a sharing of a value x modulo q into a sharing of the same value modulo t , without leaking anything about x . One important application of modulus conversion is in threshold decryption for lattice-based cryptosystems, where a sharing modulo the ciphertext modulus has to be moved to the plaintext modulus before it can be opened. The most efficient protocols so far (Damgård et al., Asiacrypt 2026) are in the preprocessing model, assume x is bounded away from q by a constant factor and take two rounds for a prime target modulus, three rounds for arbitrary moduli, and one round when all moduli are 2-powers. We improve upon this by constructing three new modulus conversion protocols in the preprocessing model:

- Our first conversion protocol takes a single round for arbitrary moduli, consisting of just one opening.
- Our second conversion protocol is a variant of the first protocol, where in the preprocessing most of the secure multiplications are replaced by private inputs, which are checked in a batch using fully linear interactive oracle proofs (FLIOPs). This makes the preprocessing cheaper (amortized over a batch of conversions) when the number N of parties is small. This comes at the cost of making the bound on x a factor N stronger.
- Our third conversion protocol avoids the bound on x altogether but takes two rounds.

All our protocols are stated in the arithmetic black box (ABB) model, so they can run on top of a wide variety of MPC protocols, inheriting their corruption threshold and security model.

To allow the third conversion protocol to run in two rounds, we construct the first *one-round* comparison protocol in the ABB model for comparing a bitwise shared n -bit value against a public threshold with polynomial-size preprocessing. Concretely, the preprocessing size is $\mathcal{O}(n^2)$ in general, and $\mathcal{O}(n^2/\log n)$ in our case where the secret value is fixed in preprocessing.

We also construct constant-round ABB protocols *without preprocessing* for computing symmetric functions and comparisons of values shared bitwise modulo powers of arbitrary primes (e.g., 2^k), at communication quasi-linear in the number of input bits. Previously, this required either a sufficiently large prime or polynomially larger communication.

Finally, we give an *impossibility* result for non-interactive modulus conversion, demonstrating that a single round is optimal: unless t divides q , every non-interactive conversion for inputs $x < q/\beta$ is incorrect with probability at least $\approx 1/(4\beta)$.

Table of Contents

1	Introduction	3
1.1	Our Contributions	3
1.2	Related Work	5
2	Technical Overview	8
3	Preliminaries	12
3.1	Arithmetic Black Box.....	12
3.2	Comparison via Suffix-AND	14
4	New Protocols for Modulus Conversion	15
4.1	Modulus Conversion Using a Common Mask.....	16
4.2	Modulus Conversion Using Individual Masks	18
4.3	Modulus Conversion With Unbounded Input	21
5	Realization with $\text{SPD}\mathbb{Z}_{2^k}$	22
6	Benchmark and Comparison to Prior Work	26
7	One-Round Comparison with a Public Threshold	27
7.1	One-Round Symmetric Function Evaluation	28
7.2	Optimized Comparison with Preprocessed Secret	29
8	Constant-Round Protocols without Preprocessing	31
8.1	Random Bits and Digits for Prime Powers.....	32
8.2	Symmetric Functions via Galois Rings	33
8.3	Symmetric Functions via Equality Tests.....	34
9	Impossibility of Non-Interactive Modulus Conversion	35
	References	40
A	Checked Input via Fully Linear Proofs	42
A.1	Fully Linear Proof Systems	42
A.2	From Proof Systems to Checked Input	43
A.3	Checked Input of Degree-Two Languages	44
A.4	Checked Input of Bit Vectors	45
B	maBits	45
B.1	Generation	45
B.2	Consistency Check	47
B.3	Putting It Together	49
C	Security Proofs for the Conversion Protocols	50

1 Introduction

Modulus conversion is a classic problem in multiparty computation with several applications, perhaps most importantly in threshold decryption for lattice-based cryptosystems.

In a nutshell, modulus conversion starts from a setting where a set of parties hold secret shares of a value $x \in \mathbb{Z}_q$, denoted $\llbracket x \rrbracket_q$, for some modulus q . The underlying secret-sharing scheme can be Shamir's, but many other options exist. The goal now is to convert this as efficiently as possible to $\llbracket x \rrbracket_t$, where t is some other modulus, of course without leaking any information on x .¹

When doing threshold decryption for lattice-based cryptosystems, one approach is to set up a sharing of the secret key initially, then given a ciphertext it is often possible using only local computation to compute $\llbracket x \rrbracket_q$, where q is the so-called *ciphertext modulus*. In the case of BGV, the desired plaintext μ satisfies $\mu = x \bmod t$, where t is the *plaintext modulus*. Opening x is insecure, as this would leak information about the secret key. However, using modulus conversion, one can first compute $\llbracket x \rrbracket_t$ and open this, which will reveal $x \bmod t = \mu$ and no other information about x . In BFV and TFHE, the plaintext instead occupies the most significant bits of x , and the conversion then runs the other way. For simplicity of presentation, we assume that t divides q , so that $x = \mu \cdot q/t + e$ for a noise term e that is small in absolute value. The parties add a public constant that makes the noise non-negative and reduce their shares modulo q/t , both using only local computation, which yields $\llbracket e \rrbracket_{q/t}$ since q/t divides q . Converting this to $\llbracket e \rrbracket_q$ and subtracting it from $\llbracket x \rrbracket_q$ leaves a sharing of $\mu \cdot q/t$, which can be opened. Either way, threshold decryption amounts to one modulus conversion followed by one opening.

This approach to decryption has been used in several previous works, most recently in [DKOP26], where the so far most efficient protocols for modulus conversion were suggested. Assuming preprocessing, they show how to do modulus conversion in two rounds when the target modulus is prime, in three rounds for arbitrary moduli, and in one round when both moduli are 2-powers. In the online phase, this requires only opening of a small constant number of values and local computation. It works under the assumption that $x < q/2$, and can be generalized to $x < (1 - 1/\beta)q$ for a constant $\beta \geq 2$, where the size of preprocessing material is proportional to β . This leaves open two obvious questions: First, can we do modulus conversion in one round for all moduli, and second, can we get rid of the bound on x ? Of course, while still matching (at least) the online efficiency of the existing protocols.

1.1 Our Contributions

We settle both of the above questions:

- We construct a one-round modulus conversion Π_{ModConv1} for all moduli with smaller online complexity and smaller preprocessing material, compared to the protocol from [DKOP26]. This works under the same assumption on the input x as in previous work ($x < (1 - 1/\beta)q$). The online part of the protocol consists of opening only one value.
- We give a variant Π_{ModConv2} of this protocol where the online complexity is the same, but where the preprocessing is done differently. When amortized over a large batch of material and when the number N of parties is small, this method lowers the cost of the preprocessing per modulus conversion, at the cost of making the required bound on x a factor N stronger.
- We construct a conversion protocol Π_{ModConv3} that makes no assumption on x at all. Its online phase opens a single value, and then completes its task by securely comparing this public value to a random number that was fixed during the preprocessing.

¹ Strictly speaking, what we produce is $\llbracket x \bmod t \rrbracket_t$, but we will not include the $\bmod t$ when the context is clear.

We also show how to securely produce the preprocessing material for these protocols, using nothing but secure arithmetic modulo the involved moduli. The main tool for this is a *maBit* [RST⁺26]: a uniformly random bit that is shared over several moduli at once. For the two moduli that a conversion usually needs, this is exactly the double-shared bit of [DKOP26], which in turn is the daBit of [RW19, AOR⁺19] when one of the two moduli is 2. The maBit generation protocol of [RST⁺26] supports only additive (i.e., full threshold) secret sharing; in Appendix B we give an alternative protocol that works under every instantiation of the arithmetic black box (ABB). It comes with a new consistency check that verifies, for a whole batch of bits at once, that the sharings over the different moduli agree, using a number of openings that is independent of the size of the batch. Our check is never costlier than the check of [RST⁺26] and in many settings substantially cheaper. For instance, checking against a 64-bit 2-power modulus takes two repetitions instead of about 80 at statistical security 2^{-80} (when the check of [RST⁺26], which is stated for primes, is carried over to 2-powers).

Table 1.1 summarizes the three modulus conversion protocols and compares them to prior work. The parameter β trades the bound on the value to convert against the size of the preprocessing material. When instantiated for threshold FHE decryption with $q = 2^{63}$, $t = 2^{64}$, $L \approx q/2$, and $N = 4$ parties, as in our benchmark (Section 6), the preprocessing material for a single conversion consists of 2 sharings over $\mathbb{Z}_t$ for Π_{ModConv1} and 8 for Π_{ModConv2} . Under SPD $\mathbb{Z}_{2^k}$ sharing [CDE⁺18] with a 64-bit message authentication code (MAC), the realization of Section 5 stores 1 024 resp. 2 560 bits per party and decryption, all masks included – a 136- to 341-fold improvement over the rounding-circuit protocol of [ZZLP25], for which the same realization stores 349 458 bits. Table 2.1 in the technical overview lists what producing that material costs in terms of secure multiplications, and there the two preprocessing protocols differ in kind: the preprocessing of Π_{ModConv1} requires a number of multiplications that grows with $\log q$, whereas that of Π_{ModConv2} is independent of q but grows cubically with N .

Table 1.1. Comparison of the modulus conversion protocols of prior work with our three (Sections 4.1 to 4.3), converting from q to t among N parties. Correctness and security hold for inputs $x < L$. Material is counted in sharings stored for one conversion, and Table 2.1 lists the secure multiplications needed to produce the material. The rounds and openings are those of the conversion alone, threshold decryption adds one further round to open the plaintext, except for the any- t protocol of [DKOP26].

Protocol	Bound L	Online		Material		
		Rounds	Openings	$\mathbb{Z}_q$	$\mathbb{Z}_t$	$\mathbb{Z}_Q^a$
[ZZLP25], q, t 2-powers	q	2	2		preprocessed gates ^b	
[DKOP26], $\beta \geq 2$, prime t	$q - q/\beta$	2	3	2	$2\beta + 1$	—
[DKOP26], $\beta \geq 2$, any t	$q - q/\beta$	3	4	2	1	$2\beta + 2$
[DKOP26], q, t 2-powers	$q/2$	1	1	0 ^c	2	—
$\Pi_{\text{ModConv1}}^{q,t,\beta}$ (Section 4.1), $\beta \geq 2$	$q - \lceil q/\beta \rceil + 1$	1	1	1	β	—
$\Pi_{\text{ModConv2}}^{q,t,\beta}$ (Section 4.2), $\beta \geq N + 1$	$q - \lceil Nq/\beta \rceil + 1$	1	1	1	β	—
$\Pi_{\text{ModConv3}}^{q,t}$ (Section 4.3)	q	2	$\leq \lceil \log q \rceil$	1	$\mathcal{O}(\log^2 q / \log \log q)$	—

^a Auxiliary prime modulus $Q > 2^k q$, needed for the conversion of [DKOP26] to an arbitrary target t .

^b $\mathcal{O}(\sqrt{\log q})$ sharings over $\mathbb{Z}_t$, plus $\mathcal{O}(\sqrt{\log q} \cdot 2^{\sqrt{\log q}})$ over $\mathbb{Z}_{2^{\mathcal{O}(\sqrt{\log q})}}$ and $\mathcal{O}(2^{\sqrt{\log q}})$ over $\mathbb{Z}_{t/q}$.

^c Sharings over the smaller modulus come for free by reducing those over the larger.

One-Round Secure Comparison. For our third conversion protocol to run in two online rounds, we give a new *one-round* protocol for comparing a bitwise shared n -bit value against a public threshold (Section 7). The only interaction in that protocol is the parallel evaluation of several symmetric functions, i.e., Boolean functions whose outputs depend only on the number

of 1-bits in the input. Damgård et al. [DFK⁺06] evaluate a symmetric function in constant rounds by interpolating a polynomial through the possible values of the sum of the input bits. Over modulus $t = p^k$, opening the sum under an additive mask from the preprocessing turns this interpolation into a one-round protocol, provided $n < p$. However, this condition fails for $p = 2$, for instance, which is an important case to consider in practice. To solve this, we propose two new one-round protocols for symmetric function evaluation, covering the cases $p = 2$ and $p \geq 3$ (Section 7.1). As a result, our unbounded conversion Π_{ModConv3} runs in two online rounds. For the case where the secret side of the comparison is fixed by the preprocessing (as in Π_{ModConv3}), we present an additional optimization that reduces the size of the preprocessing from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^2/\log n)$ (Section 7.2).

Constant-Round Protocols Without Preprocessing. Our one-round protocols for symmetric functions and comparison work modulo any prime power, but only in the preprocessing model, and computing their lookup tables by a multiplication tree takes $\mathcal{O}(\log \log n)$ rounds for n input bits. Constant-round protocols without preprocessing are known, but they either suffer from the above-mentioned restriction on the modulus [DFK⁺06] or work over any ring at polynomially larger communication [CFIK03]. This raises the question whether constant rounds without preprocessing (or equivalently, a constant-round preprocessing) can also be achieved modulo any prime power at quasi-linear communication in the number of input bits. In Section 8, we answer this question affirmatively for both symmetric functions and comparison. The main tool is a variant of the interpolation of [DFK⁺06] over a Galois ring extension of $\mathbb{Z}_{p^k}$, which contains enough interpolation nodes for every p , combined with an additional optimization that reduces the fan-in of the interpolation from n to $\mathcal{O}(\log n)$.

Impossibility Result. Finally, we prove the *impossibility* of non-interactive modulus conversion, demonstrating that a single round is optimal: we show that, unless t divides q , every non-interactive conversion is incorrect with probability at least $\approx 1/(4\beta)$ under the promise that the secret x is uniformly chosen such that $x < q/\beta$.

1.2 Related Work

Threshold decryption for lattice-based cryptosystems is one of the central applications of modulus conversion. But the area of threshold decryption goes far beyond modulus conversion, with works broadly falling in two buckets, *noise flooding* and *MPC-based*. Our work belongs to the latter.

Noise Flooding. Bendlin and Damgård [BD10] first introduced noise flooding, where every party locally derives a partial decryption using its share of the secret key. Parties then add noise to their partial decryptions prior to publishing them, preventing leakage on their key shares. The additional noise means parameters must be set so the plaintext can be reconstructed from the noisy shares, which often results in exponentially large moduli if the noise is to statistically hide the secret key. Noise flooding approaches require little to no interaction, with Boneh et al. [BGG⁺18] proposing a threshold FHE with single round decryption and, from it, a universal thresholdizer for a large class of schemes. Cho et al. [CKL24] subsequently demonstrated flaws in the simulation security of [BGG⁺18], providing attacks in some parameter regimes.

For many schemes, locally added noise is scaled by reconstruction coefficients at final reconstruction, forcing the use of even larger moduli. Several works have reduced this overhead, assuming the set of decryptors is known beforehand [dVPS26] or by carefully choosing and analyzing Shamir share evaluation points [KLL⁺26]. Boudgoust and Scholl [BS23] provide a construction with a polynomial modulus by analyzing the flooded distribution using Rényi divergence rather than statistical distance. An attack by Passelègue and Stehlé [PS24]

demonstrated that the construction of [BS23] did not achieve adaptive security, rather it achieves a weaker selective security notion. Passelègue and Stehlé [PS24] recover small decryption shares for a generalized definition in the full threshold setting. Okada and Takagi [OT25] extend this to arbitrary thresholds without relying on $\{0, 1\}$ -linear secret sharing.

MPC-based Decryption. The second branch gives up non-interactive decryption and in exchange leaves the parameters of the cryptosystem untouched, using secure multiparty computation (MPC) to distribute the decryption algorithm. Ajax [HLW⁺25] follows an MPC-style approach, achieving two-round threshold decryption for FHEW-like cryptosystems. Their protocol computes the decryption error and removes it using a double shared mask to convert between moduli. Unfortunately, their conversion technique requires an exponential gap between the noise and the plaintext scaling factor to avoid decryption failures, imposing similar, but weaker restriction on ciphertext modulus as for the noise flooding.

Zyskind et al. [ZZLP25] construct a three-round threshold decryption protocol for LWE ciphertexts with 2-power moduli. Their protocol uses gate preprocessing techniques to efficiently realize a distributed rounding procedure. Bootland et al. [BCD⁺25] provide distributed key generation and decryption procedures for BGV, BFV and TFHE. While they present an exact decryption protocol for TFHE, [BCD⁺25] still broadly favours noise flooding approaches due to their lower round complexity.

Modulus Conversion. Damgård et al. [DKOP26] cast the challenge of threshold decryption in terms of modulus conversion, isolating it as a primitive of independent interest. Their approach extracts the decryption error in a similar fashion to Ajax [HLW⁺25] and converts the error back to the original ciphertext modulus so that it may be removed. Crucially, Damgård et al. only require a constant fraction gap between a value x and its modulus q of the form $x < (1 - 1/\beta)q$ for conversion to succeed. By foregoing the exact rounding of [ZZLP25] their protocol reduces the complexity of the required preprocessing.

Our Π_{ModConv3} (Section 4.3) removes the bound on x at the cost of a secure comparison in the online phase. Like Π_{ModConv3} , the conversion of SecureNN [WGC19] from $\mathbb{Z}_{2^{64}}$ to $\mathbb{Z}_{2^{64}-1}$ rests on a secure comparison, but it is restricted to two parties plus an online helper.

Secure Comparison. The online phase of Π_{ModConv3} compares a bitwise shared value against a public value, which is a well-studied primitive. It appears as BIT-LT in [DFK⁺06], as a subprotocol of [NO07], as the BitLT family of [CdH10] and as Π_{LTBits} in Rabbit [MRVW21]. These protocols combine the per-position agreement bits online by a prefix-OR, either by a circuit of logarithmic depth or in constant rounds over a field, but even the constant-round variants need several online rounds, namely two for that of [CdH10] after precomputation, eight for that of [NO07] and 19 for that of [DFK⁺06]. SecureNN [WGC19], Falcon [WTB⁺21] and SC-DT [ZCZC26] instead test masked values for a zero, in the honest-majority setting. SecureNN runs in one round but among three parties, handing the masked result to a helper party, and Falcon takes a logarithmic number of rounds. SC-DT needs three online rounds, or two when the bitwise shared value comes from the preprocessing, and it requires Shamir sharing over a prime field.

The prefix-OR protocols above do not utilize that the bitwise shared value can be available long before the public one, as in our application. Four works do: Reistad [Rei09] compares two bits at a time, storing the product of each pair of shared bits in a preprocessing phase, Malten, Ugurbil and de Vega [MUdV23] generalize this to blocks of ν bits with all $2^\nu - \nu - 1$ products stored, Zyskind et al. [ZZLP25] store the products within digits of roughly $\sqrt{n}$ bits, and SC-DT [ZCZC26] precomputes the products of the shared bits with the random bit that blinds its result, which saves one of its online rounds. In the first three, the comparison within a block is then local once the public value arrives.

[Rei09], [MUdV23] and [ZZLP25] differ in how they combine the blocks. [Rei09] and [MUdV23] combine them by a prefix multiplication and extract a least significant bit from its result, which needs an opening of its own and hence a second round. [ZZLP25] expand the products into one lookup table per digit and feed the digit results into a second table of 2^{d+1} entries for d digits, so balancing the two tables puts both the digit size and d near $\sqrt{n}$ and makes the preprocessing exponential in $\sqrt{n}$. We combine the blocks differently, in one online round and with preprocessing quadratic rather than exponential in the number of blocks, as Section 2 explains.

Yu [Yu11] works over specially chosen prime fields in which the OR of a block of $\log \log p$ bits costs one online multiplication, and combines the blocks in four online rounds. Ohata and Nuida [ON20] also split the bits into blocks and take one AND across them, but through a generic multi-input AND gate whose preprocessing grows exponentially in the fan-in, which caps the number of blocks at a small constant. Blocks of bits also appear in SIM [YLM23], whose per-block tables are indexed by the public value but recombined by a table exponential in the number of blocks over three rounds.

Constant-Round Protocols over Rings. Damgård et al. [DFK⁺06] compare bitwise shared values in constant rounds without preprocessing by evaluating symmetric functions of the agreement bits, each by interpolating a polynomial over $\mathbb{Z}_q$ through the possible values of the sum of its inputs. This requires those values to have invertible pairwise differences, which holds for a prime q larger than the fan-in but fails over $\mathbb{Z}_{2^k}$. Cramer et al. [CFIK03] evaluate any branching program over any finite ring in constant rounds, which covers symmetric functions and comparison of bitwise shared values over arbitrary finite rings, but at communication quadratic in the size of the branching program, e.g., $\mathcal{O}(n^4)$ for the branching program that counts the ones among n bits. In Section 8, we evaluate symmetric functions and compare bitwise shared values over any prime-power modulus at communication quasi-linear in the number of input bits.

daBits and maBits. A daBit [RW19, AOR⁺19] is a random bit shared over two moduli at once, and a maBit [RST⁺26] is its generalization to any number of moduli. Rotaru et al. [RST⁺26] generate the bits over a single auxiliary modulus and move them to the target moduli by a rounding trick that works only for additive (i.e., full threshold) secret sharing; consistency is then verified by opening masked random linear combinations. Our generation (Appendix B) instead forms the sharings inside each modulus separately, which lifts the restriction to additive sharing, and it guarantees that the vectors entering the consistency check are bit vectors over every modulus, which reduces the number of maBits the check sacrifices whenever a modulus is a proper prime power.

Damgård et al. [DKOP26] generate maBits for two moduli (one of them prime) using only the ABB: Each party inputs its bit over both moduli and proves consistency by cut-and-choose. The cut-and-choose sacrifices κ auxiliary pairs per bit, whereas our check sacrifices a number of maBits that grows only logarithmically in the size of the batch.

Local Share Conversion. Cramer, Damgård and Ishai [CDI05] study share conversion where the secret domain is kept constant. Ben-David et al. [DNNP25] study which schemes convert into which, and show, among other things, that a bit shared by replicated sharing over one prime field cannot be turned into a linear sharing over another by local computation alone; they leave open what becomes possible once a small correctness error is allowed. Section 9 shows that no local computation can change the modulus unless t divides q .

2 Technical Overview

Arithmetic Black Box. We present our protocols in the arithmetic black box (ABB) model [DN03]. The ABB is an ideal functionality that models an MPC protocol capable of doing secure arithmetic modulo both the source and the target modulus, as well as a few additional auxiliary operations. This demonstrates that our constructions will work, based on any MPC protocol that supports the ABB we define. They thereby inherit the security notion of that protocol. This even includes guaranteed output delivery (GOD): over an ABB instantiation that guarantees output, all three of our protocols deliver output as well, as we discuss in Section 3.1.

Accordingly, in the following, the notation $\llbracket x \rrbracket_q$ will mean that the ABB holds a variable containing the value $x \bmod q$. Of course, for most implementations of the ABB, this really means that the parties running the protocol hold shares of x , but the ABB abstraction makes protocols easier to specify and prove secure.

Our Main Protocol. Recall that we are given $\llbracket x \rrbracket_q$ and the goal is to compute $\llbracket x \rrbracket_t$. Let us first assume for simplicity that $x \leq q/2$. On the highest level, our approach to this is the same as previous work [DKOP26]: preprocess a pair $\llbracket m \rrbracket_q, \llbracket m \rrbracket_t$ where m is chosen uniformly in $\mathbb{Z}_q$. Open $y := x + m \bmod q$ which, by assumption on x , satisfies $y = x + m - cq$ where c is 0 or 1. It is now straightforward to compute $y - \llbracket m \rrbracket_t = \llbracket x - cq \rrbracket_t$. The final step is to use additional preprocessing material to figure out the value of c securely and correct our result to get $\llbracket x \rrbracket_t$. It is in this step that our new idea comes in.

Assume for now, for simplicity, that q is even, and consider the intervals $I_0 := [0..q/2)$ and $I_1 := [q/2..q)$. For any $a \in \mathbb{Z}_q$, denote by $\alpha(a)$ the index such that $a \in I_{\alpha(a)}$. The key observation is that, if one is given $\alpha(m)$ and $\alpha(y)$, one can compute c . Namely, writing $b := \alpha(m)$, we have $c = b$ if $\alpha(y) = 0$, and $c = 1 - b$ if $\alpha(y) = 1$.

Observe that, when we define $u_0 := m - bq \bmod t$ and $u_1 := m \bmod t$, it holds that $m - cq = u_{\alpha(y)}$. Hence, if we preprocess $\llbracket u_0 \rrbracket_t$ and $\llbracket u_1 \rrbracket_t$, then we compute the output as $\llbracket y \rrbracket_t - \llbracket u_{\alpha(y)} \rrbracket_t$ in the online phase. For correctness, observe that

$$y - \llbracket u_{\alpha(y)} \rrbracket_t = y - \llbracket m - cq \rrbracket_t = \llbracket x \rrbracket_t.$$

The preprocessing material for this protocol consists of only 3 values: $\llbracket m \rrbracket_q$, $\llbracket u_0 \rrbracket_t$, and $\llbracket u_1 \rrbracket_t$.

Note that the preprocessing must know $\alpha(m)$ in order to compute u_0 , so it must produce the mask along with the index of its interval. Our preprocessing protocol does so from shared random bits: it takes a random bit idx and a random value m_{off} of $\lceil \log q \rceil - 1$ bits, sets $m := m_{\text{off}} + (q/2) \cdot \text{idx}$, and rejects the pair unless $m_{\text{off}} < q/2$, as decided by a secure comparison. An accepted m is uniform on $\mathbb{Z}_q$ and has $\alpha(m) = \text{idx}$, and each attempt is accepted with probability more than $1/2$.

This use of rejection sampling is what allows us to extend the techniques of [DKOP26] beyond the 2-power regime. For odd q the argument goes through unchanged once we set $I_0 := [0..\lceil q/2 \rceil)$ and $I_1 := [\lceil q/2 \rceil..q)$, the assumption on x being $x \leq \lfloor q/2 \rfloor$.

Furthermore, we can weaken the bound on x to $x \leq (1 - 1/\beta)q$ for a natural number $\beta \geq 2$. The approach will be very similar: We now split $\mathbb{Z}_q$ into β disjoint intervals and from the mask m , we precompute a correction value for each interval, and decide online which one to use based on the value of y .

Variant With Individual Masks. In the protocol above, the preprocessing assembles the mask from $\lceil \log q \rceil$ shared random bits, each of which costs at least one secure multiplication [RST⁺26]. For large q , these bits dominate the cost of the preprocessing. When the number N of parties is small, our second protocol needs far fewer of them: every party σ chooses a mask $m^\sigma \xleftarrow{\$} \mathbb{Z}_q$ on its own, and the mask is the sum $m := \sum_{\sigma=1}^N m^\sigma$ over the integers. Each

party knows the bits of its own mask and the index of its interval, so it inputs them along with a distributed zero-knowledge proof on secret-shared data [BBC⁺19a] that the two are consistent. The complication is that $x + m$ may now wrap around $\mathbb{Z}_q$ up to N times. The number of wrap-arounds, C , is still determined by the public $\alpha(y)$ together with a value that the preprocessing knows, namely the sum $s := \sum_{\sigma=1}^N \alpha(m^\sigma)$ of the individual interval indices: we show that $C = \lceil (s - \alpha(y))/\beta \rceil$ whenever $x \leq (1 - N/\beta)q$. The preprocessing therefore stores the corrections $u_i := m - \lceil (s - i)/\beta \rceil q \bmod t$ for $i \in [\beta]$, and the online phase is unchanged. It computes s by adding the indices modulo β and keeping the carries, with the additions being performed over one-hot encodings of the indices, which takes $(N - 1)\beta^2$ secure multiplications. All these multiplications are over the bits, so they can be carried out over an auxiliary modulus as small as 2, decoupling the number of modulo q and t multiplications from the size of q . This all comes at the cost of a stronger bound on x : reaching a given bound takes N times as many intervals, and hence N times as many corrections. For a given bound, β therefore grows linearly with N , and the $(N - 1)\beta^2$ multiplications grow cubically. Moreover, the checks of the inputs are cheap only when amortized over a batch of conversions. Hence, the variant lowers the cost of the preprocessing only for a small number of parties and when amortized over a batch.

Conversion with Unbounded Input. Finally, we sketch how the conversion with no assumption on x works. This requires a different approach than the first two protocols. Consider a preprocessed pair $\llbracket m \rrbracket_q, \llbracket m \rrbracket_t$ and observe that the wrap-around bit, still defined by $y = x + m - cq$, is determined by the public y and the mask m with no interval structure involved at all, since $c = 1\{y < m\}$: if $c = 1$, then $y - m = x - q < 0$, and if $c = 0$, then $y = x + m \geq m$. So the parties open y , compute $\llbracket c \rrbracket_t$ from the public y and the preprocessed mask, and return $y - \llbracket m \rrbracket_t + q \cdot \llbracket c \rrbracket_t$. This is correct for every $x \in \mathbb{Z}_q$, and the only value that the protocol opens is y , which is uniform and independent of x . The price is that the correction is now computed online, by comparing the secret m against the public y . This adds further interaction, so this protocol no longer runs in a single round.

Let $(m_i)_{i \in [n]}$ and $(y_i)_{i \in [n]}$ be the bits of m and y , respectively, where $n := \lceil \log q \rceil$. Assuming we have shares $(\llbracket m_i \rrbracket)_i$ from preprocessing, a standard technique [DFK⁺06, NO07, Yu11, ON20, MRVW21] is to compute the comparison result $\llbracket 1\{y < m\} \rrbracket$ as a public linear combination of the suffix-AND over $(e_i)_i := (\llbracket m_i \rrbracket \oplus y_i)_i$, i.e., the bits indicating in which positions m and y agree (cf. Section 3.2). The suffix-AND of bits $(e_i)_i$ is the sequence $(E_j)_j$ with $E_j := \prod_{i=j}^{n-1} e_i$, i.e., E_j is the AND of the bit at position j and all more significant bits. Each E_j is a symmetric function of the bits e_i with $i \geq j$, i.e., a function that depends only on how many of its inputs are 1. Damgård et al. [DFK⁺06] evaluate a symmetric function of u shared bits by interpolating a polynomial over $\mathbb{Z}_t$ through the $u + 1$ possible values of the sum of the bits. With the sum opened under an additive mask from the preprocessing, the powers of the sum are local, so the parties could compute all E_j in parallel in a single round. However, the interpolation requires $u < p$, where p is the prime dividing t , so it fails for small p , in particular for $p = 2$, since E_0 is an AND over all bits of m .

To resolve this issue, we give two one-round protocols for symmetric functions that read the result off a lookup table instead of interpolating, one for $p = 2$ and one for $p \geq 3$ (Section 7.1). Let ϕ be a symmetric function of u shared bits, written as a function of their sum B and extended by $\phi(h) := 0$ for $h > u$, and let $\ell := \lceil \log_2(u + 1) \rceil$. The preprocessing draws a random $r \in [2^\ell]$ and stores $\llbracket r \rrbracket$ together with the lookup table $\llbracket \phi(z - r \bmod 2^\ell) \rrbracket$ for $z \in [2^\ell]$.² Online, the parties locally compute $\llbracket B \rrbracket$ and open $s := B + r \bmod 2^\ell$, which is uniform and hence independent of B . Since $B \leq u < 2^\ell$, the entry for $z = s$ is a sharing of $\phi(s - r \bmod 2^\ell) = \phi(B)$. The table has $2^\ell < 2(u + 1)$ entries. For $p \geq 3$, the parties cannot

² Actually, for $p = 2$, the preprocessing stores the one-hot vector of r instead, of which r and every entry of the table are public linear combinations, so that the mask need not be stored separately.

reduce modulo 2^ℓ before opening, so they instead mask B with a random number of $\ell + \kappa$ bits whose lowest ℓ bits are r , for a statistical security parameter κ , and open over a power of p large enough that the masked sum does not wrap around.

In the comparison of m against the public y , the mask m is produced by the preprocessing, so the parties can prepare more than its bits. For each block of g consecutive positions, they store the products of all subsets of the mask bits in the block, $2^g - g - 1$ sharings per block. Once y is public, the product of the agreement indicators over a block is a linear combination of these stored products with public coefficients, so it is local. What remains for each suffix is an AND with one input per block, i.e., of fan-in roughly $(n - j)/g$ instead of $n - j$, and this AND is again a symmetric function. Hence the online phase still consists of one round, in which all suffix products are evaluated in parallel, and the preprocessing for the ANDs shrinks by a factor of g . The products within the blocks grow as 2^g , so we take $g \approx \log n$, which yields $\mathcal{O}(n^2/\log n)$ sharings in total instead of $\mathcal{O}(n^2)$ (Section 7.2). Storing the products within blocks also appears in the comparison protocols of [Rei09, MUdV23, ZZLP25]; the difference lies in how the blocks are combined, see Section 1.2.

Table 2.1. Secure multiplications for producing the preprocessing material of one conversion, for the bound $x \leq q/2$, i.e., $\beta = 2$ for Π_{ModConv1} and $\beta = 2N$ for Π_{ModConv2} , using auxiliary modulus $Q = 2$. Here $n := \lceil \log q \rceil$ is the number of maBits of a mask. ν is the preprocessing of the online comparison of Π_{ModConv3} (Protocol 7.4), i.e., the products within blocks of $g = \lceil \log n - \log \log n \rceil$ bits and the preprocessing of the $n - g$ ANDs across blocks. For a 2-power t , these ANDs use lookup tables (Section 7.1). For t not a 2-power, we assume that $p > \lceil n/g \rceil$ for the prime p dividing t , so that these ANDs use the interpolation of [DFK⁺06], and Section 7.2 covers smaller p . In both cases, $\nu = \mathcal{O}(n^2/\log n)$, e.g., $\nu = 878$ for a 2-power t and $n = 63$.

Moduli	$\mathbb{Z}_q$	$\mathbb{Z}_t$	$\mathbb{Z}_2$
Preprocessing for $\Pi_{\text{ModConv1}}^{q,t,2}$ (Section 4.1)			
q and t 2-powers	0	n	0
q a 2-power, t not	n	$(N - 1)n$	0
t a 2-power, q not	$(N - 1)n$	n	$n - 1$
neither a 2-power	$(N - 1)n$	$(N - 1)n$	$n - 1$
Preprocessing for $\Pi_{\text{ModConv2}}^{q,t,2N}$ (Section 4.2)			
t a 2-power	0	$3N - 2$	$4N^2(N - 1)$
t not a 2-power	0	$(N - 1)(3N - 2)$	$4N^2(N - 1)$
Preprocessing for $\Pi_{\text{ModConv3}}^{q,t}$ (Section 4.3)			
q and t 2-powers	0	$n + \nu$	0
q a 2-power, t not	n	$(N - 1)n + \nu$	0
t a 2-power, q not	$(N - 1)n$	$n + \nu$	$n - 1$
neither a 2-power	$(N - 1)n$	$(N - 1)n + \nu$	$n - 1$

Offline Cost. In Table 2.1, we list what producing the preprocessing material costs in terms of secure multiplications. All three protocols use the auxiliary modulus $Q = 2$ and the bound $x \leq q/2$, achieved via $\beta = 2$ in Π_{ModConv1} and $\beta = 2N$ in Π_{ModConv2} . A shared random bit counts as one multiplication (Section 8.1). The comparisons in the preprocessing (Protocol 3.4) multiply the bits sequentially, so comparing an n -bit number takes $n - 1$ multiplications and as many rounds. The online comparison of Π_{ModConv3} instead preprocesses the products within blocks of $g = \lceil \log n - \log \log n \rceil$ bits and evaluates each of the remaining $n - g$ ANDs across blocks as a symmetric function (Section 7.2). The numbers depend on whether q and t are 2-powers, for three reasons. First, a maBit over a 2-power modulus costs one shared

random bit, whereas over any other modulus it costs the $N - 1$ multiplications of an N -party XOR (Appendix B). (Under additive secret sharing, the maBit protocol of [RST⁺26] replaces the $(N - 1)n$ multiplications over q or t by n multiplications over its auxiliary modulus.) Second, for a 2-power q none of the protocols rejects a mask and the comparison that would test it is not needed; for other q , the numbers count one attempt, and since each attempt is accepted with probability $> 1/2$, fewer than two are needed in expectation. Third, if q and t are both 2-powers, then q divides t , and the mask modulo q need not be stored. The preprocessing of Π_{ModConv2} carries out no multiplication over q , so its numbers depend on t alone. The numbers are amortized over a batch of conversions, over which the cost of the consistency check of Appendix B vanishes.

Constant-Round Protocols without Preprocessing. Our one-round protocols for symmetric functions and comparison read the result off a lookup table prepared in the preprocessing, and computing that table by a multiplication tree takes $\mathcal{O}(\log \log u)$ rounds for u input bits. Constant-round protocols for symmetric functions without preprocessing are known, but each comes with a drawback. Recall that Damgård et al. [DFK⁺06] interpolate a public polynomial f over $\mathbb{Z}_Q$ with $f(i) = \phi(i)$ for $i \in \{0, \dots, u\}$ and evaluate it at the shared sum B of the input bits, which takes constant rounds since the powers of B can be computed in constant rounds [BIB89]. However, the interpolation needs $u + 1$ nodes with invertible pairwise differences, which exist in $\mathbb{Z}_Q$ only if $p > u + 1$. The protocol of [CFIK03] evaluates any branching program over any finite ring in constant rounds, at communication quadratic in the size of the program. A symmetric function of u bits can be computed by a branching program of size $\mathcal{O}(u^2)$ that counts the ones among its inputs, so this costs $\mathcal{O}(u^4)$. In Section 8, we obtain constant rounds without preprocessing modulo any prime power $Q = p^k$ at communication quasi-linear in u .

Our starting point is to move the interpolation to a Galois ring extending $\mathbb{Z}_Q$, where an extension of degree $\lceil \log_p(u + 1) \rceil + 1$ contains enough nodes with invertible pairwise differences for any p . However, the sum B is not an interpolation node of the ring: the nodes are the elements whose coefficients are digits modulo p , so computing the node of B would require the base- p digits of B . We resolve this by masking B with random digits modulo p . The node of the mask r is a local function of its digits, and after opening $s = r + B$, the parties interpolate the public polynomial that takes the value $\phi(i)$ at the node of $s - i$ for every $i \leq u$. Evaluating it at the node of r yields $\phi(B)$. The protocol takes $\mathcal{O}(u)$ multiplications and openings in the Galois ring, i.e., $\mathcal{O}(u \log^2 u)$ interactive commands over $\mathbb{Z}_Q$.

We then reduce the number of interpolation nodes from $u + 1$ to $\mathcal{O}(\log u)$ for sparse symmetric functions, i.e., functions that output 1 for few values of the sum. Recall that our one-round protocols open a masked sum of the input bits and read the entry of a preprocessed lookup table at that sum, where every entry is a public linear combination of the one-hot vector of the mask. With the mask r and the opened s from above, this entry is $\phi(B) = \sum_{i: \phi(i)=1} 1\{r = s - i \bmod p^\ell\}$, which involves only the one-hot entries at the positions $s - i$ with $\phi(i) = 1$. Instead of preparing the whole table, we therefore compute these few one-hot entries on demand once s is known. Each entry is an equality test between r and a public value, i.e., an AND of $\mathcal{O}(\log u)$ bits derived locally from the bits of the mask digits. We evaluate this AND by the Galois ring protocol, which then interpolates through $\mathcal{O}(\log u)$ nodes (reducing both the polynomial degree and the extension degree). For an AND, a single entry is needed, and the communication drops from quasi-linear to polylogarithmic in u . Since our comparison protocol amounts to n such ANDs evaluated in parallel, we obtain a constant-round comparison of a bitwise shared n -bit value without preprocessing at communication quasi-linear in n .

The random digits modulo p are built from random shared bits, which have to be generated in constant rounds as well. For odd p , we use the protocol of [DFK⁺06], and for 2-powers that

of [DEF⁺19], which is stated for additive sharing. We observe that the latter runs unchanged over the Galois ring extension used by Shamir sharing modulo a 2-power [ACD⁺19], so our protocols work under the standard realizations of $\mathcal{F}_{\text{ABB}}$ for every prime power.

Impossibility of Non-Interactive Modulus Conversion. We show that no linear secret sharing scheme (in which sets of parties that cannot reconstruct learn nothing about the secret) modulo q can be locally converted to shares of the same value modulo t unless $q \bmod t = 0$. The error probability must be (essentially) $1/4$, and $L/(4q)$ if the secret is promised to be less than L . We first show that a local conversion scheme for any linear scheme implies one that works for additive sharing, and then that this implies local conversion for two-party subtractive sharing, all with the same error probability. Finally, we prove the above-mentioned bounds for two-party subtractive sharing.

The intuition of why two-party conversion is not possible is as follows: we have shares x_1, x_2 with $x = x_1 - x_2 \bmod q$ where x is the secret. The problem is that, given only one of the shares, one cannot know with certainty if $x = x_1 - x_2$ or $x = x_1 - x_2 + q$, and so a hypothetical conversion scheme needs to “guess” if $q \bmod t \neq 0$ should be added to the output shares. The proof is essentially a technical exercise in quantifying the probability that any such guess is wrong. To see why probability $1/4$ is the important threshold, assume for simplicity that q is even and that there is no promise on the secret. Consider the four pairs of shares $(x_1, x_2), (x_1 + q/2, x_2), (x_1, x_2 + q/2), (x_1 + q/2, x_2 + q/2)$ where we assume that $x_1 < q/2, x_2 < q/2$. The first and last pair reconstruct modulo q to $x_1 - x_2$, while the other two pairs correspond to a different secret. From this, it is not hard to see that a conversion scheme cannot be correct on all four pairs.

3 Preliminaries

Notation. For $n \in \mathbb{N}$, let $[n] := \{0, \dots, n-1\}$, and let $\equiv_n$ denote congruence modulo n . For vectors $\mathbf{x}$ and $\mathbf{y}$, we write $\mathbf{x} \parallel \mathbf{y}$ for their concatenation and $\langle \mathbf{x}, \mathbf{y} \rangle$ for their inner product.

Functionalities. We prove our protocols secure in the universal composability (UC) framework [Can01, Can00], and refer there for the definitions of the real and the ideal execution. A protocol *statistically realizes* a functionality in a hybrid model *with simulation error* ϵ if there is a simulator such that, for every unbounded environment, the outputs of the two executions are ϵ -statistically close; it *perfectly realizes* it if $\epsilon = 0$. Where we state the error per command, the error of an execution is the sum over the commands issued. Several of our protocols draw public randomness, for which we assume an ideal coin-tossing functionality $\mathcal{F}_{\text{Rand}}$ that, on input a finite set S , returns a fresh uniformly random element of S to all parties.

3.1 Arithmetic Black Box

We describe our protocols in the ABB model of Damgård and Nielsen [DN03], so that they are independent of the concrete MPC protocol used to instantiate them. An ABB is an ideal functionality that stores values under identifiers and lets the parties operate on them by issuing commands, learning nothing except what they explicitly open. Any secure realization of the ABB therefore yields a realization of our protocols, and we inherit whatever corruption threshold and security notion that realization provides.

Our functionality $\mathcal{F}_{\text{ABB}}^M$ (Functionality 3.1) is parameterized by a finite set M of prime powers and stores values in $\mathbb{Z}_q$ for every $q \in M$. The presentation of the functionality is essentially that of [DKOP26], but we restrict M to prime powers and provide additional commands `CheckedInput` and `maBit`. We write $\llbracket x \rrbracket_q$ to denote that the functionality holds the value $x \in \mathbb{Z}_q$ under some identifier, and we freely write linear expressions in such handles,

each of which stands for a **Comb** command. A product of two handles likewise stands for a **Mul** command. When counting rounds, we assume that **Comb** is realized without interaction, as is supported by every linear secret-sharing scheme (LSSS). The interactive commands are **Input**, **Mul**, **Open**, and those that produce randomness.

Functionality 3.1. The Arithmetic Black Box $\mathcal{F}_{\text{ABB}}^M$

This functionality is parameterized by a finite set M of prime powers. It stores triples (id, q, x) with $q \in M$ and $x \in \mathbb{Z}_q$, and accepts the following commands. A command is executed once it has been received from all honest parties. In each case, the modulus arguments lie in M , and the functionality finally notifies the adversary that the command was issued.

Input. On $(\text{Input}, q, P_i, \text{id})$, wait for $x \in \mathbb{Z}_q$ from P_i and store (id, q, x) .

Checked Input. On $(\text{CheckedInput}, q, P_i, \mathcal{L}, \text{id}_1, \dots, \text{id}_n)$, where the language $\mathcal{L} \subseteq \mathbb{Z}_q^n$ is public, wait for $\mathbf{x} \in \mathbb{Z}_q^n$ from P_i . If $\mathbf{x} \in \mathcal{L}$, store (id_j, q, x_j) for $j \in \{1, \dots, n\}$ and send **accept** to all parties; otherwise send **abort** to all parties and halt. How to realize this command is the subject of [Appendix A](#).

Linear Combination. On $(\text{Comb}, q, (c_0, \dots, c_\ell), (\text{id}_1, \dots, \text{id}_\ell), \text{id}')$ with public $c_0, \dots, c_\ell \in \mathbb{Z}_q$, retrieve (id_j, q, x_j) for $j \in \{1, \dots, \ell\}$ and store $(\text{id}', q, c_0 + \sum_{j=1}^{\ell} c_j x_j \bmod q)$.

Multiplication. On $(\text{Mul}, q, \text{id}_1, \text{id}_2, \text{id}')$, retrieve (id_1, q, x_1) and (id_2, q, x_2) and store $(\text{id}', q, x_1 x_2 \bmod q)$.

Open. On $(\text{Open}, q, \text{id})$, retrieve (id, q, x) and send x to all parties; we call this an *opening* of x .

Divisible Modulus Conversion. On $(\text{DivModConv}, q, q', \text{id}, \text{id}')$ where q' divides q , retrieve (id, q, x) and store $(\text{id}', q', x \bmod q')$.

Random Bit. On $(\text{RandomBit}, q, \text{id})$, sample $b \xleftarrow{\$} \{0, 1\}$ and store (id, q, b) . How to realize this command for every prime power q is discussed in [Section 8.1](#).

Multi Bit. On $(\text{maBit}, (q_1, \dots, q_\ell), (\text{id}_1, \dots, \text{id}_\ell))$, where $\{q_1, \dots, q_\ell\} \subseteq M$, sample $b \xleftarrow{\$} \{0, 1\}$ and store (id_j, q_j, b) for $j \in \{1, \dots, \ell\}$. How to realize this command is the subject of [Appendix B](#).

Abort. On **abort** from the adversary at any point, send **abort** to all parties and halt.

Note that **DivModConv** is a modulus conversion that usually comes for free: for a LSSS over $\mathbb{Z}_q$ and a divisor q' of q , reducing every share modulo q' already yields a sharing of $x \bmod q'$, so the command needs no interaction. Converting to a modulus that does *not* divide q is the subject of this whole work.

Commands That We Realize Ourselves. Two commands of [Functionality 3.1](#) we realize within the ABB model itself rather than leaving them to the underlying MPC protocol, namely **CheckedInput** in [Appendix A](#) and **maBit** in [Appendix B](#). Writing $\mathcal{F}_{\text{ABB}}^M \setminus C$ for the functionality that behaves like $\mathcal{F}_{\text{ABB}}^M$ except that it rejects the commands in C , the two realizations compose: [Theorem A.3](#) realizes $\mathcal{F}_{\text{ABB}}^M \setminus \{\text{maBit}\}$ in the $(\mathcal{F}_{\text{ABB}}^M \setminus \{\text{CheckedInput}, \text{maBit}\}, \mathcal{F}_{\text{Rand}})$ -hybrid model, whereas [Theorem B.6](#) realizes $\mathcal{F}_{\text{ABB}}^M$ in the $(\mathcal{F}_{\text{ABB}}^M \setminus \{\text{maBit}\}, \mathcal{F}_{\text{Rand}})$ -hybrid model and uses **CheckedInput** in doing so. Composing the two yields the full ABB functionality.

Guaranteed Output Delivery. [Functionality 3.1](#) as written captures an ABB that provides security with abort. Three amendments capture an ABB that provides GOD instead: the **abort** command is dropped; **Input** stores the value 0 and tells all parties so whenever the adversary withholds the input of a corrupted party, so that no command can be stalled; and **CheckedInput**, on a statement outside the language, sends **reject** to all parties and stores

nothing, which leaves it to the calling protocol to decide how to go on without that party's input. A party that withholds its input to **CheckedInput** is rejected the same way. Over such an amended realization of $\mathcal{F}_{\text{ABB}}$, all three of our conversion protocols (including their preprocessing) achieve GOD as well. A realization of $\mathcal{F}_{\text{ABB}}$ with GOD needs a strict honest majority [Cle86], which we therefore assume wherever we speak of GOD.

Non-Prime-Power Moduli. Restricting M to prime powers is no loss of generality, and we keep the notation $\llbracket x \rrbracket_q$ for an arbitrary modulus q : writing $q = \prod_i p_i^{e_i}$ with pairwise distinct primes p_i , all of whose factors $p_i^{e_i}$ lie in M , we let $\llbracket x \rrbracket_q$ stand for the tuple of handles $\llbracket x \bmod p_i^{e_i} \rrbracket_{p_i^{e_i}}$. By the Chinese remainder theorem (CRT), every command of **Functionality 3.1** extends to such a tuple by applying it for each factor.

Extension Rings. The constant-round protocols of **Section 8.2** and the fully linear proofs of **Appendix A.1** are carried out over a Galois ring $\mathcal{R} = \text{GR}(q, d) = \mathbb{Z}_q[X]/f(X)$, a degree- d extension of $\mathbb{Z}_q$, where q is a prime power and f is a monic polynomial of degree d that is irreducible modulo the prime dividing q . Here, too, no further functionality is needed, and we write $\llbracket \mathbf{a} \rrbracket_{\mathcal{R}}$ for the tuple of handles $(\llbracket a_0 \rrbracket_q, \dots, \llbracket a_{d-1} \rrbracket_q)$ of the coefficients of $\mathbf{a} = \sum_{c \in [d]} a_c X^c$. Given secure arithmetic modulo q , this suffices for secure arithmetic in $\mathcal{R}$: addition in $\mathcal{R}$ is component-wise addition modulo q and multiplication by a public element of $\mathcal{R}$ is an $\mathbb{Z}_q$ -linear map on the coefficients, so both are **Comb** commands; opening $\mathbf{a}$ amounts to opening its coefficients; and multiplication of two shared elements of $\mathcal{R}$ can be done by computing all d^2 pairwise products of their coefficients, followed by local linear operations, and hence costs d^2 **Mul** commands.

3.2 Comparison via Suffix-AND

Our conversion protocols compare a bitwise shared value against a public threshold: the ABB holds the bits $b_0, \dots, b_{n-1}$ of a value m , the threshold T is public, and what is wanted is a sharing of $1\{m < T\}$. Many existing comparison protocols [DFK⁺06, NO07, Yu11, ON20, MRVW21] are based on the observation that the result $1\{m < T\}$ can be expressed as a linear combination of the suffix-AND of the indicators $1\{b_i = T_i\}$. This is captured by the following lemma. **Figure 3.3** illustrates the lemma on an example and **Protocol 3.4** is the resulting protocol template.

Lemma 3.2. *Let $\mathbf{b}, \mathbf{T} \in \{0, 1\}^n$, let $m := \sum_{i \in [n]} 2^i b_i$ and $T := \sum_{i \in [n]} 2^i T_i$, and let $e_i := 1\{b_i = T_i\}$ for $i \in [n]$. Let $E_n := 1$ and $E_j := E_{j+1} e_j$ for $j \in [n]$, so that $E_j = \prod_{i=j}^{n-1} e_i$, i.e., $\mathbf{E} := (E_j)_{j \in [n]}$ is the suffix-AND of the e_i . Then*

$$E_0 = 1\{m = T\} \quad \text{and} \quad \sum_{j \in [n]: T_j = 1} (E_{j+1} - E_j) = 1\{m < T\}.$$

Proof. E_j is 1 if $b_i = T_i$ for all $i \geq j$, and 0 otherwise. In particular, $E_0 = 1$ if and only if $\mathbf{b} = \mathbf{T}$, i.e., $m = T$. If $m = T$, all E_j are 1, so the sum is 0 = $1\{m < T\}$. Otherwise, let j^* be the largest index with $b_{j^*} \neq T_{j^*}$. Then $E_j = 1$ for $j > j^*$ and $E_j = 0$ for $j \leq j^*$, so $E_{j+1} - E_j$ is 1 for $j = j^*$ and 0 otherwise, and the sum equals T_{j^*} . Since m and T agree above position j^* and differ there, $m < T$ if and only if $T_{j^*} = 1$. $\square$

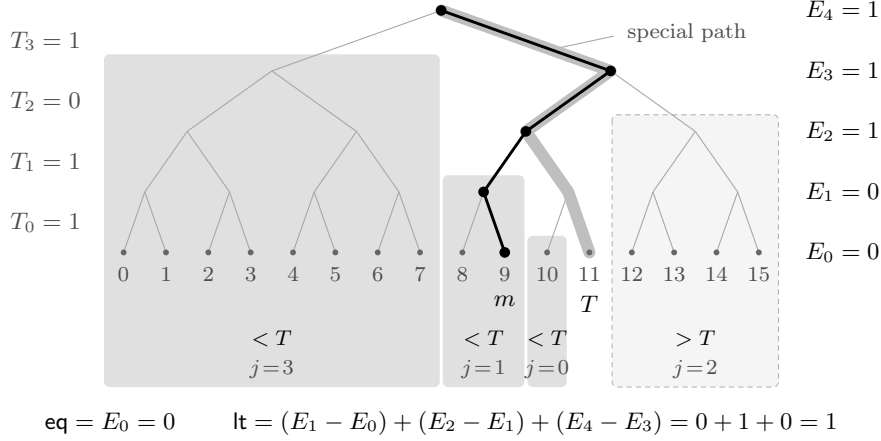


Fig. 3.3. Protocol 3.4 for $n = 4$, comparing a bitwise shared $m = 9 = 1001_2$ against the public threshold $T = 11 = 1011_2$. The public T marks out the *special path*, drawn as the grey band. The black line is the path of m . E_j , read off at depth $n - j$, says whether that path is still on the band at that depth. Every inner node of the band hangs one subtree off it, whose leaves lie below T if the band goes right (i.e., if $T_j = 1$), and above T otherwise. Hence only the j of the subtrees below T enter the sum defining lt , and its only nonzero summand is the j where m leaves the band, here $j = 1$.

Protocol 3.4. $\Pi_{\text{Compare}}^{Q,n}(\llbracket b_0 \rrbracket_Q, \dots, \llbracket b_{n-1} \rrbracket_Q, T)$

Precondition: $b_i \in \{0, 1\}$ for every $i \in [n]$, and $T \in [2^n]$.

Write T_i for the bits of T , so that $T = \sum_{i \in [n]} 2^i T_i$.

1. For $i \in [n]$, set $\llbracket e_i \rrbracket_Q := (1 - T_i) + (2T_i - 1) \cdot \llbracket b_i \rrbracket_Q$.
2. Setting $\llbracket E_n \rrbracket_Q := 1$, compute the suffix products $\llbracket E_j \rrbracket_Q := \prod_{i=j}^{n-1} \llbracket e_i \rrbracket_Q$ for $j \in [n]$ by Mul commands, in any order.
3. Return $\llbracket \text{eq} \rrbracket_Q := \llbracket E_0 \rrbracket_Q$ and $\llbracket \text{lt} \rrbracket_Q := \sum_{j \in [n]: T_j=1} (\llbracket E_{j+1} \rrbracket_Q - \llbracket E_j \rrbracket_Q)$.

We only need to prove correctness, since Protocol 3.4 opens no value.

Theorem 3.5 (Correctness of Π_{Compare}). Let $Q \in M$ and let the precondition of Protocol 3.4 hold. Write $m := \sum_{i \in [n]} 2^i b_i$. Then $\Pi_{\text{Compare}}^{Q,n}$ (Protocol 3.4) returns $\llbracket \text{lt} \rrbracket_Q$ and $\llbracket \text{eq} \rrbracket_Q$ with $\text{lt} = 1\{m < T\}$ and $\text{eq} = 1\{m = T\}$.

Proof. This follows directly from Lemma 3.2. □

We say that Protocol 3.4 is a “template”, because we did not exactly specify how to compute the suffix-AND $(E_j)_j$. Multiplying the e_i sequentially takes $n - 1$ Mul commands and rounds, and computing the products in a tree instead takes $\lceil \log n \rceil$ rounds and $\mathcal{O}(n \log n)$ Mul commands.

4 New Protocols for Modulus Conversion

In this section, we present our conversion protocols. We defer the security proofs of all three protocols to Appendix C.

Notation. The protocols Π_{ModConv1} and Π_{ModConv2} are parameterized by a number β of intervals into which $\mathbb{Z}_q$ is split, where $2 \leq \beta \leq q$. For $i \in [\beta]$, we write

$$I_i := [\lceil iq/\beta \rceil .. \lceil (i+1)q/\beta \rceil) \subseteq \mathbb{Z}_q,$$

so that $I_0, \dots, I_{\beta-1}$ partition $\mathbb{Z}_q$ into intervals of length at most $\lceil q/\beta \rceil$. For $x \in \mathbb{Z}_q$, we let $\alpha(x) := \lfloor \beta x/q \rfloor$ denote the index of the interval containing x , and we abbreviate $I(x) := I_{\alpha(x)}$ for that interval itself. The lengths $|I_i| = \lceil (i+1)q/\beta \rceil - \lceil iq/\beta \rceil$ take only the two values $T := \lceil q/\beta \rceil - 1$ and $T+1$. We denote by $\text{short}_i := \lceil q/\beta \rceil - |I_i|$ the public bit indicating whether I_i is a short interval.

The Functionality. Our conversion protocols realize the conversion command given in [Functionality 4.1](#). The **ModConv** command takes the public bound L as argument. For inputs within that bound, the conversion is correct and secure, whereas for any other input the functionality hands the input to the adversary and lets it choose the output.

Functionality 4.1. Arithmetic Black Box with Modulus Conversion $\mathcal{F}_{\text{ABB}+\text{ModConv}}^M$

This functionality is parameterized by a finite set M of prime powers. It provides all commands of $\mathcal{F}_{\text{ABB}}^M$ ([Functionality 3.1](#)), plus the following one.

Modulus Conversion. On $(\text{ModConv}, q, t, L, \text{id}, \text{id}')$, where $q, t \in M$ and $1 \leq L \leq q$, retrieve (id, q, x) . If $x < L$, store $(\text{id}', t, x \bmod t)$. Otherwise, send $(\text{ModConv}, q, t, L, x)$ to the adversary, wait for $y \in \mathbb{Z}_t$ from it, and store (id', t, y) .

4.1 Modulus Conversion Using a Common Mask

[Protocol 4.2](#) gives our first protocol $\Pi_{\text{ModConv1}}^{q,t,\beta}$, and [Fig. 4.3](#) illustrates its correctness argument for $\beta = 2$ intervals.

Protocol 4.2. $\Pi_{\text{ModConv1}}^{q,t,\beta}(\llbracket x \rrbracket_q)$ for $\beta \geq 2$

Precondition: $x \leq q - \lceil q/\beta \rceil$

Ideal Preprocessing:

1. $m \xleftarrow{\$} \mathbb{Z}_q$.
2. For $i \in [\beta]$, compute $c_i := 1\{\alpha(m) > i\}$ and $u_i := m - c_i q \bmod t$.
3. Store $(\llbracket m \rrbracket_q, \llbracket u_0 \rrbracket_t, \dots, \llbracket u_{\beta-1} \rrbracket_t)$.

Online Phase:

1. Open $\llbracket y \rrbracket_q := \llbracket x \rrbracket_q + \llbracket m \rrbracket_q$.
2. Return $\llbracket x \rrbracket_t := y - \llbracket u_{\alpha(y)} \rrbracket_t$.

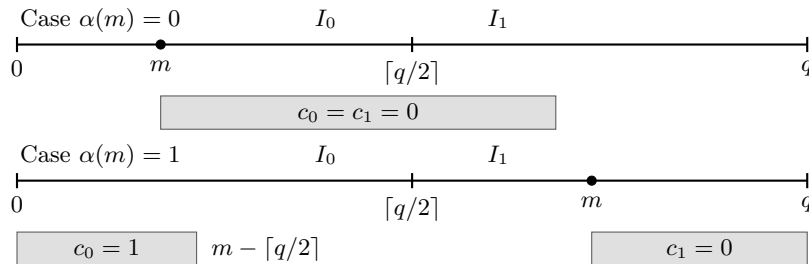


Fig. 4.3. The correctness argument for $\beta = 2$, where the precondition reads $x \leq \lfloor q/2 \rfloor$. The shaded bands cover the values the opened $y = x + m \bmod q$ can take as x ranges over the promised interval. Each band is labelled with the correction stored for the interval where it lies. If $\alpha(m) = 0$, then $x + m$ cannot wrap around $\bmod q$; if $\alpha(m) = 1$, a wrap-around moves y down into I_0 , where $c_0 = 1$ corrects for the wrap-around.

Remark 4.4. When q divides t , the sharing $\llbracket m \rrbracket_q$ need not be stored at all. Every correction satisfies $u_i \equiv m - c_i q \equiv m \pmod{q}$, so a single `DivModConv` command turns any one of $\llbracket u_0 \rrbracket_t, \dots, \llbracket u_{\beta-1} \rrbracket_t$ into $\llbracket m \rrbracket_q$ without interaction, and the material of [Table 1.1](#) drops from $\beta + 1$ sharings to β .

Preprocessing. In [Protocol 4.6](#), we present our preprocessing protocol for $\Pi_{\text{ModConv1}}^{q,t,\beta}$. Instead of sampling the mask m directly, we separately sample the interval index $\alpha(m)$ and the offset of m within the interval: $\alpha(m)$ is assembled from random bits $(\text{idx}_j)_{j \in [n_{\text{idx}}]}$, and the offset of m within that interval is assembled from random bits $(\text{off}_j)_{j \in [n_{\text{off}}]}$. Step 2 turns the bitwise interval index into a one-hot vector δ . This allows us in Step 3 to compute a bit `hot` indicating whether the interval index is valid, and a bit `short` indicating whether $I(m)$ is a short interval (recall that the interval lengths vary by up to 1). Step 5 accepts the mask if and only if both the interval index is valid and the offset does not overshoot the interval; if this step passes, the assembled mask m is uniform over $\mathbb{Z}_q$. Step 4 decides whether the offset fits into the interval, i.e., whether $m_{\text{off}} \leq T - \text{short}$, from a comparison of m_{off} with T , using the following identity.

Lemma 4.5. *Let $m_{\text{off}} \in \mathbb{Z}$ and $\text{short} \in \{0, 1\}$, and let $\text{lt} := 1\{m_{\text{off}} < T\}$ and $\text{eq} := 1\{m_{\text{off}} = T\}$. Then $\text{lt} + (1 - \text{short}) \cdot \text{eq} = 1\{m_{\text{off}} \leq T - \text{short}\}$.*

Proof. If $m_{\text{off}} = T$, both sides equal $1 - \text{short}$. Otherwise, both sides equal lt . $\square$

Protocol 4.6. Secure preprocessing for $\Pi_{\text{ModConv1}}^{q,t,\beta}$ with auxiliary modulus Q

Write $n_{\text{idx}} := \lceil \log \beta \rceil$ and $n_{\text{off}} := \lceil \log(q/\beta) \rceil$. This protocol is parameterized by an auxiliary modulus $Q \in M$, which may or may not coincide with one of q and t .

1. For $j \in [n_{\text{idx}}]$, sample $(\llbracket \text{idx}_j \rrbracket_Q, \llbracket \text{idx}_j \rrbracket_q, \llbracket \text{idx}_j \rrbracket_t) \leftarrow \mathcal{F}_{\text{ABB.mBit}}(Q, q, t)$, and for $j \in [n_{\text{off}}]$, sample $(\llbracket \text{off}_j \rrbracket_Q, \llbracket \text{off}_j \rrbracket_q, \llbracket \text{off}_j \rrbracket_t) \leftarrow \mathcal{F}_{\text{ABB.mBit}}(Q, q, t)$.
2. Over each of the moduli Q, q and t , set $\llbracket \delta^{(0)} \rrbracket := (1)$ and compute, for $j \in [n_{\text{idx}}]$ and $i \in [2^j]$,

$$\llbracket \delta_{i+2^j}^{(j+1)} \rrbracket := \llbracket \delta_i^{(j)} \rrbracket \cdot \llbracket \text{idx}_j \rrbracket, \quad \llbracket \delta_i^{(j+1)} \rrbracket := \llbracket \delta_i^{(j)} \rrbracket - \llbracket \delta_{i+2^j}^{(j+1)} \rrbracket,$$

where the product at $j = 0$ is one with the public constant $\delta_0^{(0)} = 1$ and hence a `Comb` command, and write $\llbracket \delta \rrbracket := \llbracket \delta^{(n_{\text{idx}})} \rrbracket$, of whose $2^{n_{\text{idx}}}$ entries only the first β are used below.

3. Define

$$\begin{aligned} \llbracket m_{\text{off}} \rrbracket &:= \sum_{j \in [n_{\text{off}}]} 2^j \cdot \llbracket \text{off}_j \rrbracket, & \llbracket \text{short} \rrbracket_Q &:= \sum_{i \in [\beta]} \text{short}_i \cdot \llbracket \delta_i \rrbracket_Q, \\ \llbracket m \rrbracket &:= \llbracket m_{\text{off}} \rrbracket + \sum_{i \in [\beta]} \min(I_i) \cdot \llbracket \delta_i \rrbracket, & \llbracket \text{hot} \rrbracket_Q &:= \sum_{i \in [\beta]} \llbracket \delta_i \rrbracket_Q. \end{aligned}$$

4. If q/β is a 2-power: set $\llbracket \text{acc} \rrbracket_Q := \llbracket \text{hot} \rrbracket_Q$; else: compute $(\llbracket \text{lt} \rrbracket_Q, \llbracket \text{eq} \rrbracket_Q) \leftarrow \Pi_{\text{Compare}}^{Q, n_{\text{off}}}(\llbracket \text{off}_0 \rrbracket_Q, \dots, \llbracket \text{off}_{n_{\text{off}}-1} \rrbracket_Q, T)$ ([Protocol 3.4](#)), comparing m_{off} against T , and set $\llbracket \text{acc} \rrbracket_Q := \llbracket \text{hot} \rrbracket_Q \cdot \llbracket \text{lt} \rrbracket_Q + (\llbracket \text{hot} \rrbracket_Q - \llbracket \text{short} \rrbracket_Q) \cdot \llbracket \text{eq} \rrbracket_Q$.
5. If q/β or β is not a 2-power, open $\llbracket \text{acc} \rrbracket_Q$ and, if the opened value is 0, start from the beginning; after 3κ unsuccessful iterations (where κ is the statistical security parameter), abort.
6. For $i \in [\beta]$, $\llbracket u_i \rrbracket_t := \llbracket m \rrbracket_t - q \cdot \sum_{j \in [\beta]: j > i} \llbracket \delta_j \rrbracket_t$.
7. Store $(\llbracket m \rrbracket_q, \llbracket u_0 \rrbracket_t, \dots, \llbracket u_{\beta-1} \rrbracket_t)$.

The reason why we parameterize the preprocessing by an auxiliary modulus Q is that it allows to run the comparison over a modulus whose arithmetic is cheap. When using Π_{Compare}

([Protocol 3.4](#)), there is no restriction on Q , but it is bottlenecked by $n_{\text{off}} - 1$ sequential multiplications over Q . This enables the following options:

- Taking Q to be q or t (whichever has cheaper secure multiplications).
- Taking Q to be a small divisor of q or t : this keeps the maBits over two moduli, since one can convert from q or t to Q via a `DivModConv` command.
- Taking Q unrelated to both, such as $Q = 2$: this makes the maBit generation slightly more expensive, but improves efficiency when q and t have no small divisor. Concretely, it trades $n_{\text{off}} + 1$ multiplications over q or t for $< 2\beta + n_{\text{off}}$ multiplications over Q .

Theorem 4.7 (Security of Π_{ModConv1}). *Let $Q \in M$ and let κ be a statistical security parameter. Consider the protocol that answers each command $(\text{ModConv}, q, t, L, \text{id}, \text{id}')$ by running $\Pi_{\text{ModConv1}}^{q,t,\beta}$ ([Protocol 4.2](#)) for some polynomially bounded β with $2 \leq \beta \leq q$ such that $L - 1 \leq q - \lceil q/\beta \rceil$, with its preprocessing phase run by [Protocol 4.6](#) with auxiliary modulus Q , and passes every other command on unchanged. Provided that every `ModConv` command admits such a β , this protocol statistically realizes $\mathcal{F}_{\text{ABB+ModConv}}^M$ in the $\mathcal{F}_{\text{ABB}}^M$ -hybrid model with simulation error at most $2^{-\kappa}$ per `ModConv` command.*

Guaranteed Output Delivery. Π_{ModConv1} ([Protocol 4.2](#)) and its preprocessing support ABB instantiations with GOD, with nothing to change. The only abort they add is the cutoff of Step 5, which happens with probability at most $2^{-\kappa}$ whatever the adversary does.

4.2 Modulus Conversion Using Individual Masks

[Protocol 4.8](#) gives our second protocol Π_{ModConv2} . It uses a mask m that is the sum of N individual masks, one per party. Correspondingly, the number of times that $x + m$ wraps around q is not a bit (as in Π_{ModConv1}) but a number in $\{0, \dots, N\}$. Our observation is that (under the stronger precondition $x \leq q - \lceil Nq/\beta \rceil$) the number of wrap-arounds is determined by the public $\alpha(y)$ together with the sum s of the interval indices of the individual masks, as we show in [Lemma 4.10](#) below. Using individual masks makes the preprocessing cheaper than that of Π_{ModConv1} (amortized over a batch of conversions) when the number N of parties is small.

Protocol 4.8. $\Pi_{\text{ModConv2}}^{q,t,\beta}(\llbracket x \rrbracket_q)$ for $\beta \geq N + 1$

Precondition: $x \leq q - \lceil Nq/\beta \rceil$

Ideal Preprocessing:

1. For $\sigma \in \{1, \dots, N\}$, sample $m^\sigma \xleftarrow{\$} \mathbb{Z}_q$.
2. Compute $m := \sum_{\sigma=1}^N m^\sigma$ over the integers and $s := \sum_{\sigma=1}^N \alpha(m^\sigma)$.
3. For $i \in [\beta]$, compute $c_i := \lceil (s - i)/\beta \rceil$ and $u_i := m - c_i q \bmod t$.
4. Store $(\llbracket m \rrbracket_q, \llbracket u_0 \rrbracket_t, \dots, \llbracket u_{\beta-1} \rrbracket_t)$.

Online Phase: same as in [Protocol 4.2](#).

Remark 4.9. [Remark 4.4](#) applies here unchanged: when q divides t , each of the corrections u_i is again congruent to m modulo q , so $\llbracket m \rrbracket_q$ can be obtained from any $\llbracket u_i \rrbracket_t$ via `DivModConv`.

Lemma 4.10. *Let $\beta \geq N + 1$, let $m^1, \dots, m^N \in \mathbb{Z}_q$, and put $m := \sum_{\sigma} m^\sigma$ and $s := \sum_{\sigma} \alpha(m^\sigma)$. Let $x \leq q - \lceil Nq/\beta \rceil$, let $y := x + m \bmod q$, and let $C := \lfloor (x + m)/q \rfloor$ be the number of wrap-arounds. Then $\lceil (s - \alpha(y))/\beta \rceil = C$.*

Proof. Since $\alpha(z) = \lfloor \beta z / q \rfloor$, we have $\beta m^\sigma / q - 1 < \alpha(m^\sigma) \leq \beta m^\sigma / q$ for every σ , and summing over σ gives $\beta m / q - N < s \leq \beta m / q$. Likewise $\beta y / q - 1 < \alpha(y) \leq \beta y / q$. It follows that

$$\begin{aligned} s - \alpha(y) &< \frac{\beta m}{q} - \left(\frac{\beta y}{q} - 1 \right) = \beta C - \frac{\beta x}{q} + 1 \leq \beta C + 1 \quad \text{and} \\ s - \alpha(y) &> \left(\frac{\beta m}{q} - N \right) - \frac{\beta y}{q} = \beta C - \frac{\beta x}{q} - N \geq \beta C - \beta = \beta(C - 1), \end{aligned}$$

the last step using the precondition $x \leq q - \lceil Nq/\beta \rceil$. As $s - \alpha(y)$ is an integer, the first inequality gives $s - \alpha(y) \leq \beta C$, and together $\lceil (s - \alpha(y)) / \beta \rceil = C$. $\square$

Preprocessing. In [Protocol 4.11](#), we present our preprocessing for Π_{ModConv2} . Each party P_σ samples its mask m^σ locally and inputs the one-hot vector δ^σ of its interval index together with the bits off^σ of its offset within that interval. Steps 3, 4 and 5 certify that the offset fits into the interval that δ^σ claims (via the language $\mathcal{L}_{q,\beta}$ defined after the protocol), that the three sharings encode the same bits, and that δ^σ is indeed a one-hot vector. Since $c_i = \lceil (s - i) / \beta \rceil = \lfloor s / \beta \rfloor + 1 \{s \bmod \beta > i\}$, the corrections need only the quotient and the residue of s modulo β . Step 7 computes both by adding the interval indices in $\mathbb{Z}_\beta$, as a cyclic convolution of the one-hot vectors, and accumulating the carries χ^σ on the side. This convolution is performed over bits, so we run it over Q , which may be as small as 2, and Step 8 moves the resulting bits to t at the cost of one maBit each.

Since each party samples its own mask, a corrupted party may choose m^σ instead of sampling it, and the checks only guarantee $m^\sigma \in \mathbb{Z}_q$. The proof of [Theorem 4.13](#) shows this to be harmless: correctness holds for every tuple of masks (over $\mathbb{Z}_q$) by [Lemma 4.10](#), and privacy relies only on at least one mask being uniform and independent of the adversary's view.

Protocol 4.11. Secure preprocessing for $\Pi_{\text{ModConv2}}^{q,t,\beta}$ with auxiliary modulus Q

Write $n_{\text{off}} := \lceil \log(q/\beta) \rceil$, and let $T_0, \dots, T_{n_{\text{off}}-1}$ be the bits of T . This protocol is parameterized by an auxiliary modulus $Q \in M$, which may or may not coincide with one of q and t .

1. Each party σ locally does the following:
 - Sample $m^\sigma \xleftarrow{\$} \mathbb{Z}_q$.
 - $m_{\text{off}}^\sigma := m^\sigma - \min(I(m^\sigma))$.
 - Let $\text{off}^\sigma := (\text{off}_i^\sigma)_{i \in [n_{\text{off}}]}$ be the bit decomposition of m_{off}^σ .
 - Let $\delta^\sigma := (\delta_i^\sigma)_{i \in [\beta]}$ where $\delta_i^\sigma := 1 \{m^\sigma \in I_i\}$ (i.e., δ^σ is a one-hot vector indicating the interval that m^σ lies in).
 - Let $E^\sigma := (E_j^\sigma)_{j \in [n_{\text{off}}]}$ be the suffix-AND given by $E_j^\sigma := \prod_{i=j}^{n_{\text{off}}-1} 1 \{\text{off}_i^\sigma = T_i\}$.
 - Sample $f^\sigma \xleftarrow{\$} \{0, 1\}^{n'-n}$ and set $v^\sigma := \text{off}^\sigma \parallel \delta^\sigma \parallel f^\sigma \in \{0, 1\}^{n'}$, where n' is the input length that [Protocol B.3](#) requires in order to return $n := n_{\text{off}} + \beta$ maBits.
2. For each party σ and each of the moduli q and t , the parties call $\mathcal{F}_{\text{ABB}}.\text{CheckedInput}$ over that modulus with the language $\{0, 1\}^{n'}$, on which P_σ provides v^σ , resulting in $\llbracket v^\sigma \rrbracket_q$ and $\llbracket v^\sigma \rrbracket_t$.
3. For each party σ , the parties call $\mathcal{F}_{\text{ABB}}.\text{CheckedInput}$ over Q with the language $\mathcal{L}_{q,\beta}$ of [Lemma 4.12](#), on which P_σ provides $v^\sigma \parallel E^\sigma$, resulting in $\llbracket v^\sigma \rrbracket_Q$. Beyond bit-ness, this certifies that the offset off^σ fits into the interval that δ^σ claims.
4. For each party σ , run $\Pi_{\text{maBitsConsCheck}}^{n,(q,t,Q)}(\llbracket v^\sigma \rrbracket_q, \llbracket v^\sigma \rrbracket_t, \llbracket v^\sigma \rrbracket_Q)$, with the moduli ordered so that the largest one plays the role of q_ℓ . This verifies that the three sharings encode the same bit vector and returns the unsacrificed prefixes, of which the protocol uses $\llbracket \text{off}^\sigma \rrbracket_q, \llbracket \delta^\sigma \rrbracket_q, \llbracket \text{off}^\sigma \rrbracket_t, \llbracket \delta^\sigma \rrbracket_t$ and $\llbracket \delta^\sigma \rrbracket_Q$.
5. For each party σ , compute and open $\sum_{i \in [\beta]} \llbracket \delta_i^\sigma \rrbracket_q$, and abort unless the opened value is 1.

6. For each party σ , define $\llbracket m^\sigma \rrbracket := \sum_{i \in [n_{\text{off}}]} 2^i \cdot \llbracket \text{off}_i^\sigma \rrbracket + \sum_{i \in [\beta]} \min(I_i) \cdot \llbracket \delta_i^\sigma \rrbracket$, and $\llbracket m \rrbracket := \sum_{\sigma=1}^N \llbracket m^\sigma \rrbracket$.
7. Setting $\llbracket \Delta^1 \rrbracket_Q := \llbracket \delta^1 \rrbracket_Q$, compute for $\sigma = 2, \dots, N$

$$\begin{aligned} \llbracket \Delta_j^\sigma \rrbracket_Q &:= \sum_{j'+i \equiv_{\beta} j} \llbracket \Delta_{j'}^{\sigma-1} \rrbracket_Q \cdot \llbracket \delta_i^\sigma \rrbracket_Q \quad (j \in [\beta]), \\ \llbracket \chi^\sigma \rrbracket_Q &:= \sum_{j'+i \geq \beta} \llbracket \Delta_{j'}^{\sigma-1} \rrbracket_Q \cdot \llbracket \delta_i^\sigma \rrbracket_Q, \end{aligned}$$

where both sums range over the pairs $(j', i) \in [\beta]^2$ that meet the stated condition and hence involve the same β^2 products, and write $\Delta := \Delta^N$.

8. Move the $\beta + N - 2$ bits $\Delta_1, \dots, \Delta_{\beta-1}$ and $\chi^2, \dots, \chi^N$ from Q to t : for each of them, written $\llbracket z \rrbracket_Q$: sample $(\llbracket r \rrbracket_Q, \llbracket r \rrbracket_t) \leftarrow \mathcal{F}_{\text{ABB.mBit}}(Q, t)$, open $\llbracket d \rrbracket_Q := \llbracket z \rrbracket_Q + \llbracket r \rrbracket_Q - 2 \cdot \llbracket z \rrbracket_Q \cdot \llbracket r \rrbracket_Q$ (for $Q = 2$ this simplifies to $\llbracket d \rrbracket_Q := \llbracket z \rrbracket_Q + \llbracket r \rrbracket_Q$ and no Mul command is needed), and set $\llbracket z \rrbracket_t := d + (1 - 2d) \cdot \llbracket r \rrbracket_t$.
9. Set $\llbracket w \rrbracket_t := \sum_{\sigma=2}^N \llbracket \chi^\sigma \rrbracket_t$.
10. For $i \in [\beta]$, set $\llbracket c_i \rrbracket_t := \llbracket w \rrbracket_t + \sum_{j \in [\beta]: j > i} \llbracket \Delta_j \rrbracket_t$ and $\llbracket u_i \rrbracket_t := \llbracket m \rrbracket_t - q \cdot \llbracket c_i \rrbracket_t$.
11. Store $(\llbracket m \rrbracket_q, \llbracket u_0 \rrbracket_t, \dots, \llbracket u_{\beta-1} \rrbracket_t)$.

The Range Check. Step 3 has to check that $m_{\text{off}}^\sigma \leq T - \text{short}_i$ for the index i that δ^σ claims, which by Lemma 4.5 is a comparison of m_{off}^σ with T . The suffix-AND $\mathbf{E}^\sigma$ is the intermediate result of this comparison in Protocol 3.4, and Lemma 3.2 lets us express the outcome of the comparison by equations of degree two. Writing $v_j = \text{off}_j$ for $j \in [n_{\text{off}}]$ and $v_{n_{\text{off}}+i} = \delta_i$ for $i \in [\beta]$, we let

$$\mathcal{L}_{q,\beta} := \left\{ (\mathbf{v}, \mathbf{E}) \in \{0, 1\}^{n'} \times \mathbb{Z}_Q^{n_{\text{off}}} \left| \begin{array}{l} E_j = E_{j+1}e_j \text{ for all } j \in [n_{\text{off}}], \\ \text{and } \text{lt} + (1 - \text{short})E_0 = 1 \end{array} \right. \right\},$$

where $E_{n_{\text{off}}} := 1$, $e_j := (1 - T_j) + (2T_j - 1)v_j$ as in Step 1 of Protocol 3.4, $\text{short} := \sum_{i \in [\beta]} \text{short}_i v_{n_{\text{off}}+i}$, and $\text{lt} := \sum_{j \in [n_{\text{off}}]: T_j=1} (E_{j+1} - E_j)$.

Lemma 4.12. *Let $\mathbf{v} \in \{0, 1\}^{n'}$ with $\text{short} \in \{0, 1\}$, and let $m_{\text{off}} := \sum_{j \in [n_{\text{off}}]} 2^j v_j$. If $m_{\text{off}} \leq T - \text{short}$, then $(\mathbf{v}, \mathbf{E}) \in \mathcal{L}_{q,\beta}$ for exactly one $\mathbf{E} \in \mathbb{Z}_Q^{n_{\text{off}}}$, and otherwise for none.*

Proof. The first condition of $\mathcal{L}_{q,\beta}$ determines $\mathbf{E}$ as $E_j = \prod_{i=j}^{n_{\text{off}}-1} e_i$, where $e_i = 1\{v_i = T_i\}$ since $v_i \in \{0, 1\}$. Since $T < \lceil q/\beta \rceil \leq 2^{n_{\text{off}}}$, the bits $T_0, \dots, T_{n_{\text{off}}-1}$ represent T , so Lemma 3.2 gives $E_0 = 1\{m_{\text{off}} = T\}$ and $\text{lt} = 1\{m_{\text{off}} < T\}$. By Lemma 4.5, the second condition then reads $1\{m_{\text{off}} \leq T - \text{short}\} = 1$, and as the left-hand side is a bit, this holds in $\mathbb{Z}_Q$ if and only if it holds over the integers. $\square$

The hypothesis $\text{short} \in \{0, 1\}$ is not part of $\mathcal{L}_{q,\beta}$, but follows from Step 5, since $\text{short} = \text{short}_i$ when δ^σ is the one-hot vector of i . Checking the offset against T alone, i.e., against the length of the long intervals, would not suffice: a corrupted party could claim a short interval I_i together with the offset T , so that m^σ lies in I_{i+1} and s falls short of $\sum_{\sigma} \alpha(m^\sigma)$ by one. With up to $N - 1$ parties doing so, the lower bound in the proof of Lemma 4.10 loses up to $N - 1$, and the analysis would need $\beta \geq 2N$ in place of $\beta \geq N + 1$.

Every value occurring in the two conditions of $\mathcal{L}_{q,\beta}$ is a bit, so unlike the one-hot check of Step 5, the range check needs no lower bound on its modulus. We therefore run it over Q , the smallest of the three moduli, and Step 4 carries its guarantee to $\llbracket \mathbf{v}^\sigma \rrbracket_q$ and $\llbracket \mathbf{v}^\sigma \rrbracket_t$. Together with bit-ness of $\mathbf{v}$, the check consists of $n' + n_{\text{off}} + 1$ equations of degree two with $\mathcal{O}(n')$

monomials in total, so by [Corollary A.4](#) it costs $\mathcal{O}(\log n')$ rounds, $n' + n_{\text{off}} + \mathcal{O}(\delta \log n')$ calls to $\mathcal{F}_{\text{ABB}}.\text{Input}$ by P_σ , and $\mathcal{O}(\delta \log n')$ calls to $\mathcal{F}_{\text{ABB}}.\text{Open}$, which is the cost of the bit check of [Appendix A.4](#) up to constants.

If none of q , t and Q satisfies the precondition of [Protocol B.3](#), the parties bring in a further auxiliary modulus $\hat{Q}$ that does: each P_σ additionally provides $\llbracket v^\sigma \rrbracket_{\hat{Q}}$ in [Step 2](#), and [Step 4](#) checks all four sharings against each other. We suggest to choose $\hat{Q}$ as a power of a prime already occurring among q , t and Q , since the sharing over that modulus then follows from $\llbracket v^\sigma \rrbracket_{\hat{Q}}$ by a `DivModConv` command and $\hat{Q}$ replaces it in the check instead of joining as a fourth modulus.

The auxiliary modulus Q serves the same purpose as in [Protocol 4.6](#), and the options for choosing it carry over, with t in the role of the modulus to relate Q to. Taking $Q = t$ leaves [Step 8](#) nothing to do, and taking Q to be a small divisor of t replaces the `maBits` of that step by shared random bits over t (since they can be reduced to Q via `DivModConv`).

Theorem 4.13 (Security of Π_{ModConv2}). *Let $Q \in M$ and let κ be a statistical security parameter. Consider the protocol that answers each command $(\text{ModConv}, q, t, L, \text{id}, \text{id}')$ by running $\Pi_{\text{ModConv2}}^{q,t,\beta}$ ([Protocol 4.8](#)) for some polynomially bounded β with $N + 1 \leq \beta \leq q$ such that $L - 1 \leq q - \lceil Nq/\beta \rceil$, with its preprocessing phase run by [Protocol 4.11](#) with auxiliary modulus Q , and passes every other command on unchanged. Provided that every `ModConv` command admits such a β , this protocol statistically realizes $\mathcal{F}_{\text{ABB}+\text{ModConv}}^M$ in the $(\mathcal{F}_{\text{ABB}}^M, \mathcal{F}_{\text{Rand}})$ -hybrid model with simulation error $\text{negl}(\kappa)$ per `ModConv` command.*

Guaranteed Output Delivery. Π_{ModConv2} ([Protocol 4.8](#)) supports ABB instantiations with GOD as well, but its preprocessing takes inputs from the parties, which can fail the checks. Every check concerns the input of a single party, so a rejection names that party, and an honest party is never rejected. On a rejection of P_σ , or if P_σ supplies no input at all, the parties replace its contribution by the public default $m^\sigma := 0$, i.e., the offset 0 together with $\delta^\sigma := (1, 0, \dots, 0)$, and skip its remaining checks. Privacy is untouched, since one honest mask suffices.

4.3 Modulus Conversion With Unbounded Input

[Protocol 4.14](#) gives our third protocol Π_{ModConv3} , which avoids the bound on x at the cost of one more online round.

Protocol 4.14. $\Pi_{\text{ModConv3}}^{q,t}(\llbracket x \rrbracket_q)$

This protocol performs modulus conversion without precondition, i.e., the input x is unbounded.

Ideal Preprocessing:

1. Sample $m \xleftarrow{\$} \mathbb{Z}_q$ and let $b_0, \dots, b_{n-1}$ be its bits, where $n := \lceil \log q \rceil$.
2. Store $\llbracket m \rrbracket_q$ together with the preprocessed data of $\Pi_{\text{PrepCompare}}^{t,n,g}$ ([Protocol 7.4](#)) for the bits $b_0, \dots, b_{n-1}$ and block size $g := \lceil \log n - \log \log n \rceil$, which contains $\llbracket b_0 \rrbracket_t, \dots, \llbracket b_{n-1} \rrbracket_t$.

Online Phase:

1. Open $\llbracket y \rrbracket_q := \llbracket x \rrbracket_q + \llbracket m \rrbracket_q$.
2. Compute $(\llbracket \text{lt} \rrbracket_t, \llbracket \text{eq} \rrbracket_t) \leftarrow \Pi_{\text{PrepCompare}}^{t,n,g}(y)$ ([Protocol 7.4](#)), and set $\llbracket c \rrbracket_t := 1 - \llbracket \text{lt} \rrbracket_t - \llbracket \text{eq} \rrbracket_t$.
3. Return $\llbracket x \rrbracket_t := y - \llbracket m \rrbracket_t + q \cdot \llbracket c \rrbracket_t$, where $\llbracket m \rrbracket_t := \sum_{i \in [n]} 2^i \cdot \llbracket b_i \rrbracket_t$.

Remark 4.15. When q divides t , the sharing $\llbracket m \rrbracket_q$ need not be stored here either: the parties can convert the bits $\llbracket b_0 \rrbracket_t, \dots, \llbracket b_{n-1} \rrbracket_t$ to modulus q via `DivModConv` and recombine them as $\llbracket m \rrbracket_q = \sum_{i \in [n]} 2^i \cdot \llbracket b_i \rrbracket_q$.

Preprocessing. We produce the preprocessing material via [Protocol 4.16](#), which works similarly to [Protocol 4.6](#), with two changes. First, there are no intervals here, so the mask needs no decomposition into an interval index and an offset. Second, instead of the corrections u_i , it stores the mask bitwise over t .

Protocol 4.16. Secure preprocessing for $\Pi_{\text{ModConv3}}^{q,t}$ with auxiliary modulus Q

This protocol is parameterized by an auxiliary modulus $Q \in M$, which may or may not coincide with one of q and t .

1. For $i \in [n]$, sample $(\llbracket b_i \rrbracket_Q, \llbracket b_i \rrbracket_q, \llbracket b_i \rrbracket_t) \leftarrow \mathcal{F}_{\text{ABB.maBit}}(Q, q, t)$.
2. Define $\llbracket m \rrbracket := \sum_{i \in [n]} 2^i \cdot \llbracket b_i \rrbracket$.
3. If q is not a 2-power:
 - (a) Compute $(\llbracket \text{lt} \rrbracket_Q, \llbracket \text{eq} \rrbracket_Q) \leftarrow \Pi_{\text{Compare}}^{Q,n}(\llbracket b_0 \rrbracket_Q, \dots, \llbracket b_{n-1} \rrbracket_Q, q)$ ([Protocol 3.4](#)), comparing m against q .
 - (b) Open $\llbracket \text{lt} \rrbracket_Q$. If the opened value is 0, start from the beginning; after κ unsuccessful iterations, abort.
4. Run the preprocessing of $\Pi_{\text{PrepCompare}}^{t,n,g}$ ([Protocol 7.4](#)) on $\llbracket b_0 \rrbracket_t, \dots, \llbracket b_{n-1} \rrbracket_t$ and the block size $g = \lceil \log n - \log \log n \rceil$.
5. Store $\llbracket m \rrbracket_q$ together with the preprocessed data of $\Pi_{\text{PrepCompare}}$.

Theorem 4.17 (Security of Π_{ModConv3}). *Let $Q \in M$ and let κ be a statistical security parameter. Consider the protocol that answers each command $(\text{ModConv}, q, t, L, \text{id}, \text{id}')$ by running $\Pi_{\text{ModConv3}}^{q,t}$ ([Protocol 4.14](#)), with its preprocessing phase run by [Protocol 4.16](#) with auxiliary modulus Q , and passes every other command on unchanged. If the symmetric function protocol in Step 2 of $\Pi_{\text{PrepCompare}}$ ([Protocol 7.4](#)) is statistically secure with simulation error at most ϵ , then this protocol statistically realizes $\mathcal{F}_{\text{ABB+ModConv}}^M$ in the $\mathcal{F}_{\text{ABB}}^M$ -hybrid model with simulation error at most $2^{-\kappa} + \lceil \log q \rceil \cdot \epsilon$ per ModConv command.*

Remark. In order for Π_{ModConv3} to take two online rounds, Step 2 of [Protocol 7.4](#) has to use a one-round symmetric function protocol, such as those of [Section 7.1](#), which require a sufficiently large t . Assuming a sufficiently large t is w.l.o.g., since the parties can run Π_{ModConv3} for a sufficiently large power t' of the prime dividing t and turn its output $\llbracket x \rrbracket_{t'}$ into $\llbracket x \rrbracket_t$ by a DivModConv command.

Guaranteed Output Delivery. Π_{ModConv3} ([Protocol 4.14](#)) and its preprocessing support ABB instantiations with GOD as well, with nothing to change. The cutoff of Step 3b is again the only abort they add.

5 Realization with $\text{SPD}\mathbb{Z}_{2^k}$

Our protocols are stated in the ABB model, where every **Open** command returns the correct value or aborts. In this section, we realize a modulus conversion followed by the opening of a value computed from its result on top of $\text{SPD}\mathbb{Z}_{2^k}$ [[CDE⁺18](#)], i.e., we describe how the openings are checked and what this costs in rounds and in bits. This is the shape that threshold decryption takes: the parties hold $\llbracket x \rrbracket_q$ and run the online phase of a conversion protocol Π on it, which yields $\llbracket x \rrbracket_t$; they then compute $\llbracket v \rrbracket_t$ from $\llbracket x \rrbracket_t$ and from sharings they already hold by **Comb** commands, which is local, and open it. In threshold decryption, v contains the plaintext. The benchmark in [Section 6](#) runs the online phase in this realization. It follows the pipelined authentication that Zyskind et al. [[ZZLP25](#)] propose for their protocol, with two differences that we point out below.

SPD $\mathbb{Z}_{2^k}$ Preliminaries. Let $t = 2^k$ be the target modulus and let κ be the statistical security parameter. We write $[a]$ for an additive sharing of $a \in \mathbb{Z}_{2^{k+\kappa}}$ without a MAC, and $[a]^\sigma$ for the share of P_σ , so that $\sum_\sigma [a]^\sigma \equiv a \pmod{2^{k+\kappa}}$. In SPD $\mathbb{Z}_{2^k}$, a value $x \in \mathbb{Z}_t$ is held as an authenticated sharing $\llbracket x \rrbracket = ([\hat{x}], [\Delta\hat{x}])$, where $\hat{x} \in \mathbb{Z}_{2^{k+\kappa}}$ satisfies $\hat{x} \equiv x \pmod{2^k}$, and the key $\Delta = \sum_\sigma [\Delta]^\sigma$ is the integer sum of key shares $[\Delta]^\sigma \in \mathbb{Z}_{2^\kappa}$ and is never opened. Such an authenticated sharing realizes the ABB handle $\llbracket x \rrbracket_t$. Both kinds of sharing are linear, so every **Comb** command is local. For a 2-power $q = 2^{k'}$ dividing t , reducing the shares and MAC shares of $\llbracket x \rrbracket_t$ modulo $2^{k'+\kappa}$ preserves both congruences and yields $\llbracket x \rrbracket_q$, which is how DivModConv comes for free.

To open $\llbracket x \rrbracket$, every party broadcasts $[\hat{x}]^\sigma$, and everyone computes $\hat{x}$. Here, broadcasting means sending the same message to every other party over authenticated point-to-point channels, which is what the benchmark in Section 6 does. We assume no broadcast channel, so a corrupted party may also send different messages to different honest parties. Nothing about this is authenticated yet: a corrupted party may broadcast $[\hat{x}]^\sigma + \delta$ for any δ of its choice, and the honest parties then reconstruct $\hat{x} + \delta$.

The MAC check of SPD $\mathbb{Z}_{2^k}$ catches this in two further rounds, one to commit to a check share and one to reveal it. Let **Commit** be a non-interactive commitment scheme realizing an ideal commitment functionality, as in [DPSZ12]: a commitment by P_σ to a (a value or a tuple of values) is $c := \text{Commit}(\sigma, a; \rho)$ for fresh randomness ρ , and revealing a means sending (a, ρ) , which the receivers check against c and the sender's index σ , so that no party can open another party's commitment as its own. In the random oracle model (ROM), $\text{Commit}(\sigma, a; \rho) = \mathcal{H}(\sigma \parallel \rho \parallel a)$ is such a scheme. The probability bounds in this section treat **Commit** as ideal.

To check the MAC, every party computes its share $z_\sigma := [\Delta]^\sigma \hat{x} - [\Delta\hat{x}]^\sigma$ of the check, commits to it, and reveals it in the next round, and the parties accept if $\sum_\sigma z_\sigma \equiv 0 \pmod{2^{k'+\kappa}}$ for a value opened modulo $2^{k'}$. For honest parties the sum is $\Delta\hat{x} - \Delta\hat{x} = 0$, whereas an error $\delta \not\equiv 0 \pmod{2^{k'}}$ turns it into $\Delta\delta$ plus whatever offset the corrupted parties add to their own z_σ . The commitment fixes that offset before the honest z_σ are seen, so it has to equal $-\Delta\delta$ for the unknown Δ , which happens with probability at most $2^{-\kappa}$.

In SPD $\mathbb{Z}_{2^k}$ proper, the parties broadcast their shares of $\hat{x} + 2^k r$ for a preprocessed random $\llbracket r \rrbracket$ with $r \in \mathbb{Z}_{2^\kappa}$ in place of $\hat{x}$, since the upper κ bits of $\hat{x}$ may depend on secrets, and the MAC check runs on $\hat{x} + 2^k r$. We call this opening $\llbracket x \rrbracket$ with extra mask $\llbracket r \rrbracket$. Only $r \bmod 2^\kappa$ and $\Delta r \bmod 2^\kappa$ enter such an opening, so an extra mask is stored as a κ -bit share and a κ -bit MAC share per party, i.e., 2κ bits, as SPD $\mathbb{Z}_{2^k}$ does [CDE⁺18]. The same extra mask serves an opening over a divisor $2^{k'}$ of t , where the parties broadcast their shares of $\hat{x} + 2^{k'} r$. Sometimes the extra mask can be omitted, as we discuss after Protocol 5.1.

Two Checks. The online phase of Π opens the masked value y (and, for Π_{ModConv3} , the values of its constant-round subprotocol), and the opening of $\llbracket v \rrbracket_t$ follows. One commit-and-reveal can check several openings at once, with one z_σ per opened value, and we check all openings of Π together. The opening of $\llbracket v \rrbracket_t$ needs a MAC check of its own, because the MAC check on y has to complete before v is revealed. To see why, suppose v were revealed before the MAC check on y has completed. An error δ on y then makes the honest parties read off the interval index $\alpha(y + \delta)$ and subtract the correction for that interval, so what they reveal is v shifted by a public multiple of the difference of two corrections – a quantity that reveals the interval index of the mask m and thereby, since $y = x + m$ is public, information about x . The adversary learns that quantity and is only caught afterwards. The protocol of [ZZLP25] faces the same constraint, for the same reason.

Run one after the other, the two MAC checks and the opening of $\llbracket v \rrbracket_t$ take five rounds beyond those of Π : two for each MAC check and one for the opening. Following [ZZLP25], we pipeline them into three rounds (beyond those of Π), as follows.

1. Run the online phase of Π on $\llbracket x \rrbracket_q$, which opens y , and compute $\llbracket v \rrbracket_t$ from the result.
2. Open $\llbracket v \rrbracket_t + \llbracket w \rrbracket_t$ for a preprocessed random pad w (which leaks nothing even before the MAC check on y) and commit to the shares of the MAC check on y .
3. Reveal the shares of the MAC check on y and commit to those of the MAC check on $v + w$; verify the MAC check on y .
4. Reveal the shares of the MAC check on $v + w$ and open w ; verify the MAC check on $v + w$.

This is not yet complete: the opening of w is not checked. Checking it the same way would take two more rounds. Instead, we let the preprocessing authenticate w a second time, under a one-time key, and open this MAC and the key together with w ; by [Lemma 5.2](#) below, this suffices, provided the shares of these last-round openings were committed to in an earlier round.

Protocol 5.1. Conversion by $\Pi \in \{\Pi_{\text{ModConv1}}, \Pi_{\text{ModConv2}}, \Pi_{\text{ModConv3}}\}$ followed by an opening, in $\text{SPD}\mathbb{Z}_{2^k}$

Precondition: The parties hold $\llbracket x \rrbracket_q$, with x satisfying the precondition of Π , and the sharings over $\mathbb{Z}_t$ from which $\llbracket v \rrbracket_t$ is computed.

Ideal Preprocessing:

1. For $j \in \{0, \dots, \ell\}$, sample $r_j \xleftarrow{\$} \mathbb{Z}_{2^\kappa}$, where ℓ is the number of values that the online phase of Π opens.
2. Sample $\hat{w} \xleftarrow{\$} \mathbb{Z}_{2^{k+\kappa}}$ and $\gamma \xleftarrow{\$} \mathbb{Z}_{2^{k+\kappa}}$, and let $\llbracket w \rrbracket_t := ([\hat{w}], [\Delta \hat{w}])$.
3. Store the preprocessing material of Π and $(\llbracket r_0 \rrbracket, \dots, \llbracket r_\ell \rrbracket, \llbracket w \rrbracket_t, [\gamma], [\gamma \hat{w}])$.

Online Phase: Abort whenever a commitment fails to open or a MAC check fails.

1. Run the online phase of Π on $\llbracket x \rrbracket_q$ without MAC checks, its j -th opening being with extra mask $\llbracket r_j \rrbracket$, obtaining $\llbracket x \rrbracket_t$, and let $y_1, \dots, y_\ell$ be the values it opens.
2. Compute $\llbracket v \rrbracket_t$ from $\llbracket x \rrbracket_t$ and the held sharings by **Comb** commands, and open $\llbracket v \rrbracket_t + \llbracket w \rrbracket_t$ with extra mask $\llbracket r_0 \rrbracket$. Commit to the shares of the MAC check on $y_1, \dots, y_\ell$.
3. Reveal the shares of the MAC check on $y_1, \dots, y_\ell$ and check them. Commit to the share of the MAC check on $v + w$ and to $[\hat{w}]^\sigma$, $[\gamma \hat{w}]^\sigma$ and $[\gamma]^\sigma$.
4. Reveal the share of the MAC check on $v + w$ and check it. Reveal $[\hat{w}]^\sigma$, $[\gamma \hat{w}]^\sigma$ and $[\gamma]^\sigma$, check that $\sum_\sigma [\gamma \hat{w}]^\sigma \equiv (\sum_\sigma [\gamma]^\sigma)(\sum_\sigma [\hat{w}]^\sigma) \pmod{2^{k+\kappa}}$, and output $v = (v + w) - w \bmod t$.

In Step 4, where w is opened, y has already been verified, so the honest parties' shares of $v + w$ are correct. A corrupted party can still send a wrong share of $v + w$, which the MAC check on $v + w$ catches, or a wrong share of w , which [Lemma 5.2](#) below covers. The online phase takes the rounds of Π plus three, i.e., four rounds for Π_{ModConv1} and Π_{ModConv2} . We do not prove the pipelining secure: it interleaves the openings and MAC checks of $\text{SPD}\mathbb{Z}_{2^k}$ in a way that its proof does not cover, so a formal proof cannot rely on $\text{SPD}\mathbb{Z}_{2^k}$ as a black box.

Our realization differs from that of [\[ZZLP25\]](#) in two points. First, we support several modulus conversion protocols, which saves a round with Π_{ModConv1} or Π_{ModConv2} . Second, in [\[ZZLP25\]](#) the revealed shares of w are checked only against the commitments the parties make to them online, whereas we additionally check them against the one-time MAC on w . The commitment alone does not authenticate the opening of w : it binds a party to a value of its choosing, but not to the share that the MAC on w refers to. A corrupted party can commit to $[\hat{w}]^\sigma + \delta$, use $[\hat{w}]^\sigma$ in the opening of $v + w$, and reveal $[\hat{w}]^\sigma + \delta$; the honest parties then output $v - \delta$ and do not abort. The pipelining of [\[ZZLP25\]](#) as described therefore fails correctness with abort, and the one-time MAC on w resolves this issue (cf. [Lemma 5.2](#)).

Omitting Extra Masks. An extra mask serves only to make the upper κ bits of the opened value independent of the secrets, i.e., uniformly random given everything else the adversary sees, so that the opening reveals nothing beyond the value itself.

- The extra mask $\llbracket r_j \rrbracket$ on y_j with $j \geq 1$ can therefore be omitted whenever the structure of Π already grants $\hat{y}_j$ this property. For Π_{ModConv1} and Π_{ModConv2} , this is the case when $\llbracket m \rrbracket_q$ is stored with uniformly random upper bits, since $\hat{y} = \hat{x} + \hat{m}$. It is not the case when $\llbracket m \rrbracket_q$ is instead obtained from a correction $\llbracket u_i \rrbracket_t$ by DivModConv (Remarks 4.4 and 4.9), as in our threshold decryption application (Section 6): the bits k' to $k - 1$ of $\hat{m}$ are then bits of u_i , rather than fresh randomness, so the opening of y needs the extra mask $\llbracket r_1 \rrbracket$.
- The extra mask $\llbracket r_0 \rrbracket$ can be omitted whenever some sharing enters $\llbracket v \rrbracket_t$ with an odd coefficient and has upper κ bits that are uniformly random and hidden by the extra masks of all other openings, since the upper bits of $\hat{v}$ are then uniformly random given everything else the adversary sees. In our threshold decryption application, this holds for the correction $\llbracket u_{\alpha(y)} \rrbracket_t$ of Π_{ModConv1} and Π_{ModConv2} (resp. the bit $\llbracket b_0 \rrbracket_t$ of Π_{ModConv3}), which enters $\llbracket v \rrbracket_t$ with coefficient 1, provided that the corrections $\llbracket u_0 \rrbracket_t, \dots, \llbracket u_{\beta-1} \rrbracket_t$ (resp. the bit $\llbracket b_0 \rrbracket_t$) are stored with uniformly random upper bits.

In both cases, the omission requires that some sharing is stored with uniformly random upper bits. The preprocessing can achieve this by adding an extra mask scaled by the modulus to the sharing before storing it; the corrections $\llbracket u_0 \rrbracket_t, \dots, \llbracket u_{\beta-1} \rrbracket_t$ can all receive the same extra mask, since the online phase uses only one of them.

Batching. Several decryptions can be batched, i.e., run in parallel with the same one-time key γ , so that the preprocessing of $[\gamma]$ and the reveal of $[\gamma]^\sigma$ are paid once per batch. An abort in one decryption aborts the whole batch, and a message that does not arrive by the end of its round counts as an abort.

Lemma 5.2 (One-time MAC). *Consider one or several parallel executions of Protocol 5.1 that use the same $[\gamma]$. Then, in each execution, the parties either abort in Step 4 or the revealed shares of $[\hat{w}]$ sum to $\hat{w}$ modulo 2^k , except with probability at most $2^{-\kappa}$.*

Proof. Fix one execution and let $\delta_{\hat{w}}$, $\delta_{\gamma\hat{w}}$ and δ_γ be the amounts, modulo $2^{k+\kappa}$, by which the revealed shares of $[\hat{w}]$, $[\gamma\hat{w}]$ and $[\gamma]$ sum to more than $\hat{w}$, $\gamma\hat{w}$ and γ ; these are the errors that the corrupted parties introduce, and the honest parties contribute nothing to them. The check of Step 4 passes if and only if $\delta_{\gamma\hat{w}} \equiv \gamma\delta_{\hat{w}} + \delta_\gamma(\hat{w} + \delta_{\hat{w}}) \pmod{2^{k+\kappa}}$. The corrupted parties commit to $\delta_{\hat{w}}$, $\delta_{\gamma\hat{w}}$ and δ_γ in Step 3, when no honest $[\gamma]^\sigma$ or $[\gamma\hat{w}]^\sigma$ has been revealed yet, so at that point γ is uniform in $\mathbb{Z}_{2^{k+\kappa}}$ from their view. If $\delta_{\hat{w}} \equiv 0 \pmod{2^k}$, the reconstructed value is correct whether or not the check passes. Otherwise $\delta_{\hat{w}} = 2^e d$ with $e < k$ and d odd, so $\gamma\delta_{\hat{w}} \pmod{2^{k+\kappa}}$ is uniform over $2^{k+\kappa-e} \geq 2^{\kappa+1}$ values, and the check passes with probability at most $2^{-\kappa}$. $\square$

Communication. Per decryption, a party sends each other party its share and its share of the MAC check for every opened value ($2(k' + \kappa)$ bits for a value opened modulo $2^{k'}$), and $[\hat{w}]^\sigma$ and $[\gamma\hat{w}]^\sigma$ for the opening of w (another $2(k + \kappa)$ bits). Per batch, it sends $[\gamma]^\sigma$ and, per commit round, one commitment and its opening. For Π_{ModConv1} and Π_{ModConv2} with $k = \kappa = 64$ and $q = 2^{63}$, the per-decryption total is $2 \cdot 127 + 2 \cdot 128 + 256 = 766$ bits. Under the same realization, the protocol of [ZZLP25] sends $2 \cdot 127 + 2 \cdot 73 + 2 \cdot 128 + 256 = 912$ bits, for its openings modulo 2^{63} , 2^9 and 2^{64} and the pad.

Preprocessing of the Pads. The one-time MAC $[\gamma\hat{w}]$ is a MAC on $\hat{w}$ under a second key, and the proof of Lemma 5.2 uses that it sums to $\gamma\hat{w}$ modulo $2^{k+\kappa}$ for the same $\hat{w}$ that $[\Delta\hat{w}]$ refers to. A dealer guarantees this by construction. Without a dealer, the parties have to compute $[\gamma\hat{w}]$ from $[\gamma]$ and $[\hat{w}]$, and a product protocol that lets a corrupted party enter $[\hat{w}]^\sigma + \eta$

in place of its share breaks the guarantee: the stored product is $\gamma(\hat{w} + \eta)$, the party reveals $[\hat{w}]^\sigma + \eta$ in Step 4, the check passes, and the honest parties output $v - \eta$. The multiplication of SPDZ_{2^k} on $[[w]]_t$ and an authenticated γ does not suffice either: the preprocessing functionality of SPDZ_{2^k} leaves the upper κ bits of a triple uniformly random [CDE⁺18], so the product is correct only modulo the value modulus 2^k , whereas the check of Step 4 needs it modulo $2^{k+\kappa}$.

We therefore compute the product in a second instance of SPDZ_{2^k} , with value modulus $2^{k+\kappa}$ and statistical security parameter κ , i.e., with shares modulo $2^{k+2\kappa}$, under the same key Δ : the parties sample $\hat{w}$ and γ as random authenticated values of that instance, multiply them using one of its authenticated triples, with the two openings of the multiplication MAC-checked as usual, and reduce the three resulting sharings modulo $2^{k+\kappa}$, which yields $[[w]]_t$, $[\gamma]$ and $[\gamma\hat{w}]$. The product is correct modulo $2^{k+\kappa}$ except with probability at most $2^{-\kappa}$ by the security of SPDZ_{2^k} . Per pad, this takes one triple of the second instance, whose shares are κ bits wider, and two openings, all in the preprocessing; the stored material is that of the reduced sharings, as listed in Protocol 5.1. Since the executions of a batch share γ , one authenticated γ serves all pads of a batch.

6 Benchmark and Comparison to Prior Work

We implemented the online phase of our conversion protocols and of the threshold decryption of Zyskind et al. [ZZLP25], both under the realization of Section 5, in one code base.³ To our knowledge, theirs is the fastest previously published threshold-decryption protocol with a public implementation, and hence the closest point of comparison to our own online phase. Both protocols share the arithmetic, the MAC checks, the commitments and the networking, so that the comparison measures the protocols rather than their implementations. Our implementation serves for both Π_{ModConv1} (Section 4.1) and Π_{ModConv2} (Section 4.2), which differ only in the width β of the correction table. The two differ by less than 3% in every configuration of Tables 6.1 and 6.2, so we report the numbers of Π_{ModConv1} .

Setup. All parties run on one machine with an Intel Xeon Silver 4314 (16 cores, 2.4 GHz), one core per party, and communicate over emulated links with the stated round-trip time and a per-party bandwidth cap in both directions. The preprocessed material comes from a simulated dealer and is in memory before the measurement starts, as in the benchmark of [ZZLP25], so the measurement covers the online phase only. Every run checks its outputs against the plaintexts, and every MAC check and commitment is in place. We report the median of 15 repetitions after a warm-up.

Parameters. We use the LWE parameters of [ZZLP25], namely dimension 1024, ciphertext modulus $t = 2^{64}$ and plaintext modulus 2, with $\kappa = 64$. The plaintext is the most significant bit of the decryption value, so we follow the route for this encoding described in Section 1: the parties add 2^{61} to the shared decryption value and reduce their shares modulo 2^{63} , both locally, and the value to convert is the shifted noise $e + 2^{61}$, so the source modulus for our protocol is $q = 2^{63}$ and the target is the ciphertext modulus t . For Π_{ModConv1} we take $\beta = 2$ and for Π_{ModConv2} the smallest 2-power $\beta \geq 2N$, so that both protocols are correct and secure for any noise e with $|e| < 2^{61}$. For the protocol of [ZZLP25] we keep the parameters of their evaluation, which round the noise away digit by digit with 8-bit digits.

Table 6.1 lists the runtime of a single decryption, without a bandwidth cap, since a single decryption sends very little. The runtime is dominated by the round-trip time, whereas the number of parties has only a mild effect. This matches the number of rounds: a decryption

³ <https://github.com/AarhusCrypto/modulus-operandi>

takes five rounds in the online phase of [ZZLP25] and four in ours, MAC check included, so theirs takes two and a half round trips plus the local computation, and ours two.

Table 6.1. Runtime in ms of the online phase for a single decryption.

Parties	RTT 1 ms		RTT 10 ms		RTT 100 ms	
	[ZZLP25]	Ours	[ZZLP25]	Ours	[ZZLP25]	Ours
2	2.9	2.3	25.5	20.3	250.6	200.6
4	3.2	2.6	25.7	20.6	251.0	200.8
8	3.7	3.0	26.3	21.1	251.6	201.3
16	4.2	3.3	27.0	21.8	252.8	202.2

Table 6.2 gives the throughput of the online phase when decryptions are batched. Each party runs 256 instances of the protocol at once, each on a batch of 256 decryptions (i.e., 65 536 decryptions in flight), with their rounds interleaved, so that one instance computes while the messages of the others are on the wire. At 100 Mbit/s both protocols are bandwidth-bound, and the ratio follows that of the communication: per decryption, a party sends each other party 116 bytes in the protocol of [ZZLP25] and 96 in ours, i.e., the counts of Section 5 with every value rounded up to whole bytes. At 1 Gbit/s the local computation binds for few parties, and the gap narrows to a factor of 1.1.

Table 6.2. Throughput of the online phase in decryptions per second, at a round-trip time of 1 ms and the stated per-party bandwidth, with 256 instances of 256 decryptions in flight.

Parties	100 Mbit/s		1 Gbit/s	
	[ZZLP25]	Ours	[ZZLP25]	Ours
2	85 151	99 247	225 645	250 742
4	33 435	38 941	152 374	175 330
8	14 380	16 694	93 403	109 732
16	6 623	7 755	51 976	62 716

Our implementation of their protocol is faster than their own: [ZZLP25] report a latency of 8.48 ms and a throughput of 64 319 decryptions per second with 4 parties at 1 ms and 1 Gbit/s, whereas ours achieves 3.2 ms and 152 374. We therefore compare against our implementation of their protocol rather than against their published numbers.

7 One-Round Comparison with a Public Threshold

The online phase of Π_{ModConv3} (Section 4.3) compares a bitwise shared value against a public threshold. In this section, we construct one-round protocols for this task. Recall from Section 3.2 that the comparison comes down to the suffix-AND of the agreement bits e_i , computed by the template Π_{Compare} (Protocol 3.4). Observe that each suffix product E_j in Π_{Compare} is a symmetric function of the bits $e_j, \dots, e_{n-1}$. Therefore, when minimizing the round complexity is desired, one can use a constant-round symmetric function protocol. For instance, the interpolation of [DFK⁺06] through the possible values of the sum of the bits takes one round when the sum is opened under an additive mask from the preprocessing.

However, the interpolation over $\mathbb{Z}_Q$ requires $n - j < p$, where p is the prime dividing Q , so for small p (in particular for $p = 2$) a one-round protocol is still missing. In [Section 7.1](#), we close this gap with two one-round symmetric function protocols, enabling one-round instantiations of the comparison protocol over powers of arbitrary primes. In [Section 7.2](#), we optimize the comparison protocol for the case where the secret m is fixed by preprocessing and only the public T arrives online.

7.1 One-Round Symmetric Function Evaluation

We present two one-round protocols for evaluating a symmetric function ϕ on shared bits $\llbracket b_0 \rrbracket_Q, \dots, \llbracket b_{u-1} \rrbracket_Q$, i.e., a Boolean function whose output depends only on the number of 1-bits in the input: the first protocol is for Q a 2-power and the second protocol is for Q a power of an odd prime. Both open a masked sum of the input bits and read the result off a lookup table prepared in the preprocessing, so the online phase amounts to one opening and local computation.

2-Power Moduli. For $Q = 2^k$, we present our protocol $\Pi_{\text{SymFuncPow2}}^\phi$ ([Protocol 7.1](#)) below. In the preprocessing, we use the macro **Equal**, which takes a public bit and a shared bit and outputs a shared bit indicating whether they are equal, i.e., $\text{Equal}(a, \llbracket b \rrbracket) := a\llbracket b \rrbracket + (1 - a)(1 - \llbracket b \rrbracket)$.

Protocol 7.1. $\Pi_{\text{SymFuncPow2}}^\phi(\llbracket b_0 \rrbracket_Q, \dots, \llbracket b_{u-1} \rrbracket_Q)$

This protocol securely evaluates the symmetric function ϕ on shared input bits $b_0, \dots, b_{u-1}$ and outputs the result in shared form. Let $\ell := \lceil \log_2(u + 1) \rceil$.

Precondition: $b_j \in \{0, 1\}$ for every $j \in [u]$, and $Q = 2^k > u$.

Preprocessing:

1. For $i \in [\ell]$, sample $\llbracket r_i \rrbracket_Q \leftarrow \mathcal{F}_{\text{ABB}}.\text{RandomBit}()$.
2. For $j \in [2^\ell]$: let $(j_i)_{i \in [\ell]}$ be the bits of j and compute $\llbracket \delta[j] \rrbracket_Q := \prod_{i \in [\ell]} \text{Equal}(j_i, \llbracket r_i \rrbracket_Q)$.^a
3. Store $(\llbracket \delta[j] \rrbracket_Q)_{j \in [2^\ell]}$.

Online Phase:

1. Let $\llbracket B \rrbracket_Q = \sum_{j=0}^{u-1} \llbracket b_j \rrbracket_Q$. Note that B is at most an ℓ -bit number.
2. Compute $\llbracket s \rrbracket_Q := \sum_{j \in [2^\ell]} j \cdot \llbracket \delta[j] \rrbracket_Q + \llbracket B \rrbracket_Q$ and open $s \bmod 2^\ell$.
3. Return (via local computation)

$$\llbracket \phi(b_0, \dots, b_{u-1}) \rrbracket_Q = \llbracket \phi(B) \rrbracket_Q = \sum_{j=0}^u \phi(j) \cdot \llbracket \delta[s - j \bmod 2^\ell] \rrbracket_Q.$$

^a Observe that δ is the one-hot vector of $r := \sum_{i \in [\ell]} 2^i r_i$, i.e., $\delta[j] = 1_{\{j = r\}}$.

Privacy of $\Pi_{\text{SymFuncPow2}}$ follows from the fact that the only opened value is uniform modulo 2^ℓ . For correctness, observe that $s \equiv r + B \pmod{2^\ell}$, hence $\delta[s - j \bmod 2^\ell] = 1 \iff s - j \bmod 2^\ell = r \iff j \equiv B \pmod{2^\ell} \iff j = B$ (since $j, B \leq u < 2^\ell$), so the output is a sharing of $\phi(B)$.

The online phase of $\Pi_{\text{SymFuncPow2}}$ performs only one opening, and the preprocessed data consists of $2^\ell < 2(u + 1)$ sharings modulo Q . The preprocessing can be realized in $\mathcal{O}(\log \log u)$ rounds with $2^\ell - \ell - 1$ multiplications (using a divide-and-conquer approach that first computes the one-hot vector for each half of the bit decomposition of r and then combines them in one more round, where the products with the last entry of either half follow linearly from the others, since each one-hot vector sums to 1). Alternatively, the preprocessing can be realized in constant rounds with $\mathcal{O}((u \log u + \kappa)(\log \log u)^2)$ interactive commands, for a statistical

security parameter κ , by doing the fan-in- ℓ multiplications in Step 2 via our preprocessing-free symmetric function protocol of Section 8.2, with all instances drawing their units from one pool as in Section 8.3.

Odd Prime Powers. For $Q = p^k$ with an odd prime p , the interpolation under an additive mask applies only when $u < p$. For larger fan-in u , we give the following protocol $\Pi_{\text{SymFuncAny}p}$, which is very similar to $\Pi_{\text{SymFuncPow}2}$ (Protocol 7.1), except that we resort to statistical masking of the input because we no longer assume Q to be a 2-power. This comes at the cost of an overhead related to the statistical security parameter. Moreover, since the mask now has to be stored anyway, the preprocessing stores the lookup table itself rather than the one-hot vector, so that the online phase reads the result off the table directly.

Protocol 7.2. $\Pi_{\text{SymFuncAny}p}^\phi(\llbracket b_0 \rrbracket_Q, \dots, \llbracket b_{u-1} \rrbracket_Q)$

This protocol securely evaluates the symmetric function ϕ on shared input bits $b_0, \dots, b_{u-1}$ and outputs the result in shared form. Let $\ell := \lceil \log_2(u+1) \rceil$. We write ϕ as a function of the number of 1-bits, extending it by $\phi(h) := 0$ for $h > u$.

Precondition: $b_j \in \{0, 1\}$ for every $j \in [u]$, and $Q = p^k \geq 2^{\ell+\kappa+1}$, where κ is a statistical security parameter.

Preprocessing:

1. For $i \in [\ell + \kappa]$, sample $\llbracket r_i \rrbracket_Q \leftarrow \mathcal{F}_{\text{ABB}}.\text{RandomBit}()$.
2. Compute $\llbracket R \rrbracket_Q := \sum_{i \in [\ell+\kappa]} 2^i \llbracket r_i \rrbracket_Q$.
3. For $j \in [2^\ell]$: let $(j_i)_{i \in [\ell]}$ be the bits of j and compute $\llbracket \delta[j] \rrbracket_Q := \prod_{i \in [\ell]} \text{Equal}(j_i, \llbracket r_i \rrbracket_Q)$.^a
4. For $z \in [2^\ell]$, compute $\llbracket \phi(z - r \bmod 2^\ell) \rrbracket_Q := \sum_{j=0}^u \phi(j) \cdot \llbracket \delta[z - j \bmod 2^\ell] \rrbracket_Q$.
5. Store $\llbracket R \rrbracket_Q$ and $(\llbracket \phi(z - r \bmod 2^\ell) \rrbracket_Q)_{z \in [2^\ell]}$.

Online Phase:

1. Let $\llbracket B \rrbracket_Q = \sum_{j=0}^{u-1} \llbracket b_j \rrbracket_Q$. Note that B is at most an ℓ -bit number.
2. Compute and open $\llbracket s \rrbracket_Q := \llbracket R \rrbracket_Q + \llbracket B \rrbracket_Q$.
3. Return the stored sharing $\llbracket \phi(b_0, \dots, b_{u-1}) \rrbracket_Q = \llbracket \phi(z - r \bmod 2^\ell) \rrbracket_Q$ for $z = s \bmod 2^\ell$.

^a Again, δ is the one-hot vector of $r := \sum_{i \in [\ell]} 2^i r_i$.

Privacy of $\Pi_{\text{SymFuncAny}p}$ follows from the fact that the only opened value is statistically indistinguishable from a uniform $(\ell + \kappa)$ -bit number. For correctness, observe that $\delta[z - j \bmod 2^\ell] = 1$ if and only if $j \equiv z - r \pmod{2^\ell}$, so the sum in Step 4 is indeed a sharing of $\phi(z - r \bmod 2^\ell)$. Moreover, $r = R \bmod 2^\ell$, hence $s \equiv r + B \pmod{2^\ell}$, and since $B \leq u < 2^\ell$, the output is a sharing of $\phi(s - r \bmod 2^\ell) = \phi(B)$.

As for $\Pi_{\text{SymFuncPow}2}$, the online phase of $\Pi_{\text{SymFuncAny}p}$ performs only one opening, and the preprocessed data consists of $2^\ell + 1 < 2(u+1)$ sharings modulo Q . The preprocessing takes the same $\mathcal{O}(\log \log u)$ rounds and $2^\ell - \ell - 1$ multiplications. The statistical masking comes at the price that we need $Q \geq 2^{\ell+\kappa+1}$. In some cases, this comes for free, if the application already requires Q to be this large, and in general, we can always use an ABB instance where Q is chosen to be large enough, and in the end reduce modulo the power of p that the application wants.

7.2 Optimized Comparison with Preprocessed Secret

In our use case for the one-round comparison ($\Pi_{\text{ModConv}3}$ in Section 4.3), the two sides arrive at different times: the secret bits $b_0, \dots, b_{n-1}$ are produced by the preprocessing, whereas the threshold T is determined online. In this subsection, we exploit this asymmetry by splitting

the bits into blocks and preprocessing the products within each block, which reduces the fan-in of the ANDs that need to be computed online. A single block of all n bits would make the comparison local, but the number of products within a block is exponential in its size, so we keep the blocks small.

For any set $S \subseteq [n]$ of positions, writing $S_1 := \{i \in S : T_i = 1\}$ and $b_U := \prod_{i \in U} b_i$ for $U \subseteq [n]$, we have

$$\prod_{i \in S} e_i = \prod_{i \in S_1} b_i \cdot \prod_{i \in S \setminus S_1} (1 - b_i) = \sum_{S_1 \subseteq U \subseteq S} (-1)^{|U \setminus S_1|} b_U. \quad (7.3)$$

The right-hand side is a linear combination of the products b_U , with coefficients that are public once T is known. Therefore, if the preprocessing stores a sharing of b_U for every $U \subseteq S$, the parties can compute the product of the e_i over S locally in the online phase. Storing every such product is affordable only for small S , since there are $2^{|S|}$ of them. We hence partition $[n]$ into blocks of g positions, store the products within each block, and compute each E_j as an AND of one factor per block, i.e., of fan-in roughly $(n - j)/g$ instead of $n - j$. This makes the evaluation of the AND cheaper by roughly a factor of g . We give the protocol in [Protocol 7.4](#).

Protocol 7.4. $\Pi_{\text{PrepCompare}}^{Q,n,g}(\llbracket b_0 \rrbracket_Q, \dots, \llbracket b_{n-1} \rrbracket_Q; T)$

Precondition: $g \geq 1$, $b_i \in \{0, 1\}$ for every $i \in [n]$, and $T \in [2^n]$. The sharings $\llbracket b_i \rrbracket_Q$ are available in the preprocessing phase.

- Write T_i for the bits of T , so that $T = \sum_{i \in [n]} 2^i T_i$.
- Let $w := \lceil n/g \rceil$ and, for $h \in [w]$, let $S_h := \{n - (w - h)g, \dots, n - (w - h - 1)g - 1\} \cap [n]$, so that $S_0, \dots, S_{w-1}$ partition $[n]$ into blocks of g consecutive positions, the lowest block S_0 possibly smaller.
- For $j \in [n]$, let $h_j \in [w]$ be the index of the block containing j , i.e., $j \in S_{h_j}$, and let $u_j := w - h_j$ be the number of blocks intersecting $\{j, \dots, n - 1\}$.
- Write $b_U := \prod_{i \in U} b_i$ for $U \subseteq [n]$.

Preprocessing:

1. For $h \in [w]$ and every $U \subseteq S_h$ with $|U| \geq 1$, compute and store $\llbracket b_U \rrbracket_Q$.
2. For $j \in [n]$ with $u_j \geq 2$, run the preprocessing phase of a symmetric function protocol for an AND of fan-in u_j (see below).

Online Phase:

1. For $j \in [n]$, let $S := \{i \in S_{h_j} : i \geq j\}$ and $S_1 := \{i \in S : T_i = 1\}$, and compute locally

$$\llbracket a_j \rrbracket_Q := \sum_{S_1 \subseteq U \subseteq S} (-1)^{|U \setminus S_1|} \llbracket b_U \rrbracket_Q.$$

2. Setting $\llbracket E_n \rrbracket_Q := 1$, compute for all $j \in [n]$ in parallel

$$\llbracket E_j \rrbracket_Q := \llbracket a_j \rrbracket_Q \cdot \prod_{h > h_j} \llbracket a_{\min S_h} \rrbracket_Q$$

by the symmetric function protocol prepared in the preprocessing.

3. Return $\llbracket \text{eq} \rrbracket_Q := \llbracket E_0 \rrbracket_Q$ and $\llbracket \text{lt} \rrbracket_Q := \sum_{j \in [n] : T_j = 1} (\llbracket E_{j+1} \rrbracket_Q - \llbracket E_j \rrbracket_Q)$.

[Protocol 7.4](#) opens no value outside of the symmetric function protocol (for computing the ANDs in Step 2 of the online phase), so its security is that of the symmetric function protocol, and it remains to prove correctness.

Theorem 7.5 (Correctness of $\Pi_{\text{PrepCompare}}$). *Let $Q \in M$ and let the precondition of Protocol 7.4 hold. Write $m := \sum_{i \in [n]} 2^i b_i$. If the symmetric function protocol for the ANDs across blocks is correct, then $\Pi_{\text{PrepCompare}}^{Q,n,g}$ (Protocol 7.4) returns $\llbracket \text{lt} \rrbracket_Q$ and $\llbracket \text{eq} \rrbracket_Q$ with $\text{lt} = 1\{m < T\}$ and $\text{eq} = 1\{m = T\}$.*

Proof. By Eq. (7.3), applied to $S = \{i \in S_{h_j} : i \geq j\}$, we have $a_j = \prod_{i \in S_{h_j}, i \geq j} e_i$ for every $j \in [n]$. Since the blocks above S_{h_j} are exactly S_h for $h > h_j$, and $a_{\min S_h}$ is the product of the e_i over all of S_h , the AND across blocks for position j equals $E_j = \prod_{i=j}^{n-1} e_i$, i.e., the same value as in Π_{Compare} (Protocol 3.4). The claim thus follows from Lemma 3.2. $\square$

The AND across Blocks. Each AND across blocks is an AND of $u_j \leq w$ shared bits, i.e., a symmetric function of these bits. Any symmetric function protocol with preprocessing can be used here, and the choice can be made per AND. Let $Q = p^k$.

- For $u_j < p$, the interpolation of [DFK⁺06] under an additive mask takes u_j preprocessed sharings, namely the first u_j powers of the mask.
- For $u_j \geq p$, the one-round protocols of Section 7.1 take $2^{\lceil \log_2(u_j+1) \rceil} < 2(u_j+1)$ preprocessed sharings. (The protocol for $p \geq 3$ additionally requires $Q \geq 2^{\lceil \log_2(u_j+1) \rceil + \kappa + 1}$, where κ is the statistical security parameter. This can always be satisfied by working over a larger power of p .)

Choosing the Block Size. The preprocessing of $\Pi_{\text{PrepCompare}}$ stores $w \cdot (2^g - g - 1)$ products, and the ANDs across blocks have $\sum_{j \in [n]} u_j \approx n^2/(2g)$ inputs in total, each of which takes at most two preprocessed sharings. The sum $n^2/g + (n/g) \cdot 2^g$ is minimized at $g = \log n - \log \log n + \mathcal{O}(1)$, which yields $\mathcal{O}(n^2/\log n)$ preprocessing overall, against $\mathcal{O}(n^2)$ without our blocking optimization. We therefore suggest the block size $g = \lceil \log n - \log \log n \rceil$. Table 7.6 lists the preprocessing for $n = 63$ (i.e., for the source modulus $q = 2^{63}$ that the threshold decryption in Section 6 also uses). There, the suggested $g = 4$ is the best block size, and with it, our blocking optimization reduces the preprocessing by roughly $3\times$.

Table 7.6. Preprocessed sharings of Protocol 7.4 for $p = 2$, $n = 63$ and block size g , counting the stored products and $2^{\lceil \log_2(u_j+1) \rceil}$ sharings per AND across blocks (cf. Section 7.1). Block size 1 is the protocol without our blocking optimization.

Block size g	1	2	3	4	5	6
Products	0	31	84	169	316	574
Lookup table entries	2 728	1 424	1 080	768	648	576
Total	2 728	1 455	1 164	937	964	1 150

8 Constant-Round Protocols without Preprocessing

In this section, we construct constant-round ABB protocols without preprocessing for evaluating symmetric functions and for comparing bitwise shared values modulo $Q = p^k$, where p is an arbitrary prime, at communication quasi-linear in the number of input bits. Throughout, we count the interactive commands of $\mathcal{F}_{\text{ABB}}$ (cf. Section 3.1), each of which costs communication that depends only on its realization and the number of parties, so the communication of a protocol is proportional to its count of interactive commands. We state these counts for constant p .

For a symmetric function of u shared bits, [DFK⁺06] interpolate a polynomial over $\mathbb{Z}_Q$ through the $u + 1$ possible values of the sum of the bits, which requires these values to be units with invertible pairwise differences, i.e., $p > u + 1$. The protocol of [CFIK03] works over any ring in constant rounds, but at communication $\mathcal{O}(u^4)$ for the branching program that counts the ones among the u bits. In Section 7, we constructed one-round protocols for both tasks over any prime power in the preprocessing model, but computing their lookup tables by a multiplication tree takes $\mathcal{O}(\log \log u)$ rounds for u input bits.

This section is structured as follows. In Section 8.1, we generate random shared bits and digits in constant rounds over arbitrary prime-power moduli. In Section 8.2, we give a protocol for symmetric functions that moves the interpolation of [DFK⁺06] to a Galois ring, which contains enough interpolation nodes for any p , at communication quasi-linear in u . In Section 8.3, we open a masked sum of the input bits, as in our SymFunc protocols from Section 7.1, but compute the entries of the lookup table on demand with the protocol of Section 8.2, which reduces the communication of an AND of u bits from quasi-linear to polylogarithmic in u . Since comparing bitwise shared n -bit values amounts to n such ANDs (cf. Section 3.2), we obtain a constant-round comparison at quasi-linear communication as a corollary.

8.1 Random Bits and Digits for Prime Powers

The protocols of this section use $\mathcal{F}_{\text{ABB}}$ as a black box, and the only command they need beyond arithmetic and openings is **RandomBit** (cf. Section 3.1). In this subsection, we show how the standard realizations of $\mathcal{F}_{\text{ABB}}$ provide this command in constant rounds for every prime power $q = p^k$ (the only part of this section that depends on the secret sharing scheme), and then build random digits modulo p from random bits.

Odd Prime Powers. The protocol of [DFK⁺06] draws a random $\llbracket a \rrbracket_q$, opens a^2 , and divides $\llbracket a \rrbracket_q$ by a fixed square root of a^2 , which yields a uniform sign ± 1 and hence a random bit after adding 1 and halving. This works for any odd prime power q , since a unit square modulo q has exactly two square roots, which differ by sign, and 2 is invertible.

2-Powers under Additive Sharing. For $q = 2^k$, neither step applies, and [DEF⁺19] resolve this for SPDZ2k [CDE⁺18], i.e., for additive sharing with dishonest majority, by working modulo $4q$: the parties draw $\llbracket u \rrbracket_{4q}$, set $\llbracket a \rrbracket_{4q} := 2\llbracket u \rrbracket_{4q} + 1$, open $e = a^2$, and let c be the smallest square root of e modulo $4q$. Then $c^{-1}a \equiv \pm 1 \pmod{2q}$ with equal probability, so $w := c^{-1}a + 1$ is 0 or 2 modulo $2q$, and the parties halve their shares of $\llbracket w \rrbracket_{4q}$ and reduce modulo q to obtain $\llbracket b \rrbracket_q$ for a random bit b . Only the halving relies on the sharing being additive: a is held as $2u_1 + 1$ by P_1 and as $2u_i$ by every other party, so after the multiplication by the odd scalar c^{-1} and the addition of 1 by P_1 , the shares $c^{-1}(2u_1 + 1) + 1$ and $2c^{-1}u_i$ are all even.

2-Powers under Shamir Sharing. With an honest majority, the standard realization of $\mathcal{F}_{\text{ABB}}$ modulo a prime power is that of [ACD⁺19], i.e., Shamir sharing over a Galois ring extending $\mathbb{Z}_q$, in which $\mathbb{Z}_q$ is the subring of constant polynomials. We observe that the protocol of [DEF⁺19] runs unchanged over the corresponding extension of $\mathbb{Z}_{4q}$. Here, a share is an evaluation of the sharing polynomial rather than a summand, so the argument for the halving changes: a is shared by the polynomial $2g(X) + 1$, where g shares u , hence w by $c^{-1}(2g(X) + 1) + 1 = 2c^{-1}g(X) + (c^{-1} + 1)$, every coefficient of which is even since c^{-1} is odd, so halving every share yields the sharing of $w/2$ by the polynomial with the halved coefficients. The protocol requires each party's u_i to lie in $\mathbb{Z}_{4q}$ rather than in its extension, which [ACD⁺19] check in constant rounds by masking the input with a random shared element of the subring and opening the sum.

In all three cases, a bit costs one multiplication and one opening, which is how Table 2.1 accounts for shared random bits.⁴ Working modulo $4q$ rather than q makes these two operations slightly more expensive, which is of little significance in practice.

Random Digits. The protocols of Sections 8.2 and 8.3 mask their input with random digits modulo p , and Section 8.3 needs each digit together with its bits. We write $\mathcal{F}_{\text{ABB}}.\text{RandomPit}$ for the command that returns $\llbracket r \rrbracket_Q$ for a uniform $r \in \mathbb{Z}_p$ along with $\llbracket r_0 \rrbracket_Q, \dots, \llbracket r_{v-1} \rrbracket_Q$ for the bits $r = \sum_{h \in [v]} 2^h r_h$, where $v := \lceil \log_2 p \rceil$. For $p = 2$, this is **RandomBit**. For an odd p , we draw v random bits, compute $\llbracket r \rrbracket_Q$ locally, and compare r against the public p by Π_{Compare} (Protocol 3.4) with the result opened, evaluating its ANDs by the interpolation of [DFK⁺06], which applies since their fan-in is at most $v < p - 1$ for $p \geq 5$; for $p = 3$, the comparison is the single product $r_0 r_1$. If $r < p$, we output the digit with its bits; otherwise, the candidate is discarded, which happens with probability $1 - p/2^v < 1/2$.

To obtain K digits without retries, we sample $4K + 24\kappa$ candidates in parallel, where κ is a statistical security parameter, and take the first K accepted ones, which fails with probability at most $2^{-\kappa}$.

8.2 Symmetric Functions via Galois Rings

The interpolation of [DFK⁺06] fails over $\mathbb{Z}_Q$ for $p \leq u$ because two of the values $1, \dots, u + 1$ coincide modulo p , so their difference is not a unit. We resolve this by moving the interpolation to a Galois ring extending $\mathbb{Z}_Q$, which has enough points with invertible pairwise differences for any p .

Galois Rings. Recall from Section 3.1 the Galois ring $\mathcal{R} = \text{GR}(Q, d)$, a degree- d extension of $\mathbb{Z}_Q$, and the notation $\llbracket \mathbf{a} \rrbracket_{\mathcal{R}}$ for a shared element of $\mathcal{R}$, where we write ring elements in boldface to distinguish them from numbers modulo Q . We write $\langle \alpha_0, \dots, \alpha_{d-1} \rangle_{\mathcal{R}}$ for the element of $\mathcal{R}$ with coefficients $\alpha_0, \dots, \alpha_{d-1} \in \mathbb{Z}_Q$. Such an element is a unit if and only if some coefficient is not divisible by p , so the p^d elements with all coefficients in $\{0, \dots, p - 1\}$ have invertible pairwise differences [BBC⁺19b, Claim D.2] and hence serve as nodes for Lagrange interpolation over $\mathcal{R}$. For $c \in [p^\ell]$ with base- p digits $c_0, \dots, c_{\ell-1}$ and $d = \ell + 1$, we write $\alpha_c := \langle 1, c_0, \dots, c_{\ell-1} \rangle_{\mathcal{R}}$, which is one of these nodes.

Random Units. A random unit of $\mathcal{R}$ together with its inverse is obtained from two random shared elements $\llbracket \mathbf{v} \rrbracket_{\mathcal{R}}, \llbracket \mathbf{w} \rrbracket_{\mathcal{R}}$ by opening $\mathbf{z} = \mathbf{v}\mathbf{w}$: if $\mathbf{z}$ is a unit, then so is $\mathbf{v}$, and $\llbracket \mathbf{v}^{-1} \rrbracket_{\mathcal{R}} = \mathbf{z}^{-1} \llbracket \mathbf{w} \rrbracket_{\mathcal{R}}$ is computed locally. Here, a random shared element of $\mathcal{R}$ is the sum of uniform elements input by all parties. A random element of $\mathcal{R}$ is a unit with probability $1 - p^{-d}$, so a pair fails with probability at most $\varepsilon := 2p^{-d}$, independently of the other pairs.

To obtain K units with their inverses without retries, we sample a pool of $2K + 4\kappa$ pairs in parallel and take the first K units among them, which fails with probability at most $2^{-\kappa}$ whenever $p^d \geq 8$.

Protocol 8.1. $\Pi_{\text{SymFunc}}^\phi(\llbracket b_0 \rrbracket_Q, \dots, \llbracket b_{u-1} \rrbracket_Q)$

This protocol securely evaluates the symmetric function ϕ on shared input bits $b_0, \dots, b_{u-1}$ and outputs the result in shared form. Let $\ell := \lceil \log_p(u + 1) \rceil$, $d := \ell + 1$ and $\mathcal{R} := \text{GR}(Q, d)$.

Precondition: $b_j \in \{0, 1\}$ for every $j \in [u]$ and $u < Q$.

1. For $j \in [\ell]$, sample $\llbracket r_j \rrbracket_Q \leftarrow \mathcal{F}_{\text{ABB}}.\text{RandomPit}()$ (Section 8.1).

⁴ For an odd prime p , the drawn value has to be a unit, which fails with probability $1/p$, so a bit takes $p/(p - 1)$ attempts in expectation; for 2-powers, a is odd by construction.

2. Compute $\llbracket r \rrbracket_Q := \sum_{j \in [\ell]} p^j \llbracket r_j \rrbracket_Q$ and $\llbracket \alpha_r \rrbracket_{\mathcal{R}}$ locally.
3. Compute $\llbracket \alpha_r^i \rrbracket_{\mathcal{R}}$ for $i \in \{2, \dots, u\}$ as in [BIB89], that is:
 - (a) Sample random units $\llbracket \mathbf{v}_0 \rrbracket_{\mathcal{R}}, \dots, \llbracket \mathbf{v}_u \rrbracket_{\mathcal{R}}$ of $\mathcal{R}$ with their inverses.
 - (b) For $i \in \{1, \dots, u\}$, compute $\llbracket \mathbf{z}_i \rrbracket_{\mathcal{R}} := \llbracket \mathbf{v}_{i-1} \rrbracket_{\mathcal{R}} \cdot \llbracket \alpha_r \rrbracket_{\mathcal{R}} \cdot \llbracket \mathbf{v}_i^{-1} \rrbracket_{\mathcal{R}}$ and open $\mathbf{z}_i$.
 - (c) For $i \in \{2, \dots, u\}$, compute $\llbracket \alpha_r^i \rrbracket_{\mathcal{R}} := \mathbf{z}_1 \cdots \mathbf{z}_i \cdot \llbracket \mathbf{v}_0^{-1} \rrbracket_{\mathcal{R}} \cdot \llbracket \mathbf{v}_i \rrbracket_{\mathcal{R}}$.
4. Let $\llbracket B \rrbracket_Q := \sum_{j \in [u]} \llbracket b_j \rrbracket_Q$; note that $B \leq u < p^\ell$.
5. Compute $\llbracket s \rrbracket_Q := \llbracket r \rrbracket_Q + \llbracket B \rrbracket_Q$ and open $s \bmod p^\ell$.
6. For $i \in \{0, \dots, u\}$, let $c^{(i)} := s - i \bmod p^\ell$. Interpolate the polynomial $f_{\phi,s}$ of degree at most u over $\mathcal{R}$ with $f_{\phi,s}(\alpha_{c^{(i)}}) = \langle \phi(i), 0, \dots, 0 \rangle_{\mathcal{R}}$ for every such i , and let $\gamma_0, \dots, \gamma_u$ be its coefficients.
7. Compute $\llbracket \mathbf{y} \rrbracket_{\mathcal{R}} := \gamma_0 + \sum_{i=1}^u \gamma_i \llbracket \alpha_r^i \rrbracket_{\mathcal{R}} = f_{\phi,s}(\llbracket \alpha_r \rrbracket_{\mathcal{R}})$ locally.
8. Return $\llbracket y_0 \rrbracket_Q$, the sharing of the constant coefficient of $\mathbf{y}$.

For correctness, note that $\mathbf{z}_1 \cdots \mathbf{z}_i = \mathbf{v}_0 \alpha_r^i \mathbf{v}_i^{-1}$, so Step 3 yields the powers of α_r . Furthermore, $s - i \bmod p^\ell$ takes $u+1 \leq p^\ell$ distinct values for $i \in \{0, \dots, u\}$, so the nodes $\alpha_{c^{(0)}}, \dots, \alpha_{c^{(u)}}$ are distinct and the interpolation in Step 6 is well defined. Moreover, $r \equiv s - B \pmod{p^\ell}$ with $B \in \{0, \dots, u\}$, so $\alpha_r = \alpha_{c^{(B)}}$ and $\mathbf{y} = \langle \phi(B), 0, \dots, 0 \rangle_{\mathcal{R}}$, where $\phi(B) = \phi(b_0, \dots, b_{u-1})$ since ϕ is symmetric. For privacy, observe that the values opened in Step 3a are products of uniform elements and independent of r , that $\mathbf{z}_1, \dots, \mathbf{z}_u$ are uniform units independent of α_r since $\mathbf{v}_0, \dots, \mathbf{v}_u$ are uniform units, and that s is uniform modulo p^ℓ since r is.

The protocol runs in constant rounds, takes $\mathcal{O}(u + \kappa)$ multiplications, openings and inputs in $\mathcal{R}$, i.e., $\mathcal{O}((u + \kappa) \log^2 u)$ interactive commands, and fails with probability at most $2^{-\kappa}$.

8.3 Symmetric Functions via Equality Tests

The protocols of Section 7.1 open a masked sum s of the input bits and read $\phi(B)$ off a preprocessed lookup table indexed by s . Here, we open the same masked sum, but instead of a table we compute only the entries that matter: since $s \equiv r + B \pmod{p^\ell}$, we have $\phi(B) = \sum_{i: \phi(i)=1} 1\{r \equiv s - i \pmod{p^\ell}\}$, and each summand is an equality test between the mask and a public value, i.e., an AND of $\mathcal{O}(\log u)$ bits derived locally from the bits of the mask digits. We evaluate these ANDs by Π_{SymFunc} (Protocol 8.1), so its fan-in drops from u to $\mathcal{O}(\log u)$.

The instances of Π_{SymFunc} run in parallel and draw their blinding units from one pool rather than one pool each, so that the slack of the pool is paid once: with $|I|$ instances of fan-in f , the pool provides $K = |I|(f + 1)$ units and hence consists of $2|I|(f + 1) + 4\kappa$ pairs.

Protocol 8.2. $\Pi_{\text{SymFuncEq}}^\phi(\llbracket b_0 \rrbracket_Q, \dots, \llbracket b_{u-1} \rrbracket_Q)$

This protocol securely evaluates the symmetric function ϕ on shared input bits $b_0, \dots, b_{u-1}$ and outputs the result in shared form. Let $\ell := \lceil \log_p(u + 1) \rceil$, $v := \lceil \log_2 p \rceil$ and $I := \{i \in \{0, \dots, u\} : \phi(i) = 1\}$.

Precondition: $b_j \in \{0, 1\}$ for every $j \in [u]$ and $u < Q$.

1. For $j \in [\ell]$, sample $\llbracket r_j \rrbracket_Q$ together with its bits $\llbracket r_{j,0} \rrbracket_Q, \dots, \llbracket r_{j,v-1} \rrbracket_Q$ by $\mathcal{F}_{\text{ABB}}.\text{RandomPit}()$ (Section 8.1).
2. Compute $\llbracket r \rrbracket_Q := \sum_{j \in [\ell]} p^j \llbracket r_j \rrbracket_Q$.
3. Let $\llbracket B \rrbracket_Q := \sum_{j \in [u]} \llbracket b_j \rrbracket_Q$; note that $B \leq u < p^\ell$.
4. Compute $\llbracket s \rrbracket_Q := \llbracket r \rrbracket_Q + \llbracket B \rrbracket_Q$ and open $s \bmod p^\ell$.

5. For $i \in I$, let $c^{(i)} := s - i \bmod p^\ell$ with base- p digits $c_j^{(i)}$ and let $c_{j,h}^{(i)}$ be the bits of $c_j^{(i)}$; compute locally

$$\llbracket e_{j,h}^{(i)} \rrbracket_Q := 1 - c_{j,h}^{(i)} - \llbracket r_{j,h} \rrbracket_Q + 2c_{j,h}^{(i)} \llbracket r_{j,h} \rrbracket_Q \quad \text{for } j \in [\ell], h \in [v],$$

and compute $\llbracket E_i \rrbracket_Q := \Pi_{\text{SymFunc}}^{\text{AND}}(\llbracket e_{j,h}^{(i)} \rrbracket_Q)_{j \in [\ell], h \in [v]}$, where **AND** is the symmetric function that is 1 if and only if all inputs are 1, all instances in parallel.

6. Return $\llbracket \phi(b_0, \dots, b_{u-1}) \rrbracket_Q := \sum_{i \in I} \llbracket E_i \rrbracket_Q$.

For correctness, note that $e_{j,h}^{(i)} = 1\{r_{j,h} = c_{j,h}^{(i)}\}$, hence $E_i = 1 \iff r = c^{(i)} \iff i \equiv s - r \equiv B \pmod{p^\ell} \iff i = B$ (since $i, B \leq u < p^\ell$). Hence $\sum_{i \in I} E_i = 1\{B \in I\} = \phi(B) = \phi(b_0, \dots, b_{u-1})$. For privacy, observe that s is uniform modulo p^ℓ since r is, and that the instances of Π_{SymFunc} open nothing else.

For constant p , each instance of Π_{SymFunc} has fan-in $f = \ell v = \mathcal{O}(\log u)$ and hence works over an extension of degree $\mathcal{O}(\log \log u)$, so the protocol runs in constant rounds and takes $\mathcal{O}((|I| \log u + \kappa)(\log \log u)^2)$ interactive commands. For an AND of u bits (where $I = \{u\}$), the cost is $\mathcal{O}((\log u + \kappa)(\log \log u)^2)$, against $\mathcal{O}((u + \kappa) \log^2 u)$ for Π_{SymFunc} alone.

Quasi-Linear Comparison. The suffix products $E_j = \prod_{i=j}^{n-1} e_i$ in Π_{Compare} (Protocol 3.4) are ANDs of fan-in $n - j$, so evaluating all of them by $\Pi_{\text{SymFuncEq}}$ in parallel, again with one pool for all instances, compares a bitwise shared n -bit value against a public threshold in constant rounds at $\mathcal{O}((n \log n + \kappa)(\log \log n)^2)$ interactive commands. For comparing two bitwise shared values, the agreement bits $e_i = 1 - (a_i \oplus b_i)$ cost one multiplication each in one additional round.

9 Impossibility of Non-Interactive Modulus Conversion

In this section we show that our one-round modulus conversion is optimal in the sense that modulus conversion with no interaction at all (i.e., local conversion) is not possible. Ben-David et al. [DNNP25] rule out the perfectly correct case, for two distinct prime fields and a secret promised to be a bit, and pose the error-tolerant version as future work. We answer it, giving a lower bound on the correctness error that holds for arbitrary moduli and showing that bound to be tight. Since our protocols assume a bound on the secret (Protocols 4.2 and 4.8), we prove the impossibility in the presence of such a bound as well: promising the secret to be smaller than L lowers the unavoidable error, but never removes it.

We first need a general model of the type of protocol we want to rule out. We will assume that the goal is to convert from a linear secret-sharing scheme (LSSS) given by a monotone span program (MSP) as detailed below. MSPs are usually defined over fields, but here our modulus q may be composite, which implies a difference to the field case: Over fields, every MSP-based LSSS is perfect, i.e., each set of shareholders is either qualified (can reconstruct the secret) or unqualified (has no information). In our case, this needs to be assumed (though we will only need to assume it for strict subsets of a single qualified set).

Using MSPs means that shares can be vectors rather than single elements, and so we cover, for instance, replicated sharing or schemes constructed from monotone formulae (these are indeed perfect).

Definition 9.1 (Local conversion scheme).

- The source linear secret sharing scheme (LSSS) is given by a monotone span program over $\mathbb{Z}_q$: to share x , the dealer draws a sharing vector $\mathbf{r}$ uniformly from $\{\mathbf{r} \in \mathbb{Z}_q^k : r_1 = x\}$ and hands party j the share $\mathbf{x}_j := M_j \mathbf{r}$ for a fixed matrix M_j . A set A is qualified when it

- reconstructs, i.e., when $x = \sum_{j \in A} \langle \alpha_j^A, \mathbf{x}_j \rangle \bmod q$ for coefficients α_j^A depending only on A . Fix a minimal qualified A with at least two parties, so that every strict subset of A is unqualified, i.e., holds shares whose joint distribution does not depend on x .
- Of the target we ask only that the parties of that same A reconstruct from their converted shares by a linear combination modulo t , with coefficients β_j . It never has to hide anything, so it need not be an LSSS at all.
 - A local conversion scheme from q to t is defined by a set of probabilistic algorithms $(f_1, \dots, f_N)$, one for each party, where each f_i takes party i 's share in the source scheme and outputs its share in the target scheme.
 - Fix a bound L with $1 \leq L \leq q$ on the secret. A local conversion scheme has correctness error ϵ under the bound L if the following experiment succeeds, except with probability ϵ : We choose x uniformly from $\{0, \dots, L-1\}$, sample $\llbracket x \rrbracket_q = (\mathbf{x}_1, \dots, \mathbf{x}_N)$, according to the sharing algorithm, and finally compute $(f_1(\mathbf{x}_1), \dots, f_N(\mathbf{x}_N))$. We have success if the reconstruction from A results in $x \bmod t$.

The case $L = q$ is the one in which nothing is promised about the secret. Any LSSS in which no single party is qualified is of the kind asked for, a minimal qualified set then has at least two parties and strict subsets that are unqualified. A one-party A is ruled out because such a party could convert its share on its own, leaving nothing to prove.

Note that this model includes protocols with preprocessing, since the use of any preprocessed material that parties hold can be seen as part of the specification of the local conversion algorithms.

Lemma 9.2. *If there exists a local conversion scheme from q to t with correctness error ϵ under the bound L , there also exists one among some number $m \geq 2$ of parties, from additive secret sharing modulo q to additive secret sharing modulo t , with at most the same correctness error under the same bound.*

Proof. For a set U of parties write M_U for the stack of the M_j with $j \in U$, so that $M_U \mathbf{r}$ is what U jointly holds. We first show that every unqualified set U admits a vector $\mathbf{d}$ with $M_U \mathbf{d} = 0$ and $d_1 = 1$: the shares U receives for the secret x are distributed as $x \cdot M_U \mathbf{e}_1 + M_U \mathbf{w}$, where $\mathbf{e}_1$ is the first standard basis vector of $\mathbb{Z}_q^k$, and where $W := \{\mathbf{r} \in \mathbb{Z}_q^k : r_1 = 0\}$. Since U is unqualified we have $M_U \mathbf{e}_1 \in M_U W$ (otherwise, for instance, U could tell whether $x = 0$ or $x = 1$). Then, choosing $\mathbf{w} \in W$ with $M_U \mathbf{e}_1 = M_U \mathbf{w}$ and setting $\mathbf{d} := \mathbf{e}_1 - \mathbf{w}$ gives a vector as promised. Moving the randomness along $\mathbf{d}$ thus changes the secret while leaving what U holds untouched.

Let A be a qualified set as in Definition 9.1, with $m := |A| \geq 2$ parties and every strict subset is unqualified. Each $A \setminus \{j\}$ is such a subset, so we obtain $\mathbf{d}_j$ with $M_{A \setminus \{j\}} \mathbf{d}_j = 0$ and $(\mathbf{d}_j)_1 = 1$. Distinct $i, j \in A$ then satisfy $M_i \mathbf{d}_j = 0$, since i lies in $A \setminus \{j\}$: the direction $\mathbf{d}_j$ is invisible to every party of A but j .

For the given conversion functions $\{f_i\}$, let $F_i(M_i \mathbf{r}) := \langle \beta_i, f_i(M_i \mathbf{r}) \rangle \bmod t$ be the contribution of party i to reconstructing from the converted shares. We will say that $\mathbf{r}$ is *correct* if the conversion works when given the shares produced from $\mathbf{r}$. That is, when $\sum_{i \in A} F_i(M_i \mathbf{r}) \equiv r_1 \pmod{t}$.

Recall that our goal is to construct a conversion scheme for additive sharing. Thus, we are given a set of additive shares $\mathbf{u} \in \mathbb{Z}_q^m$ to convert. The idea is to construct locally from $\mathbf{u}$ a set of shares of the same secret in the original LSSS so we can hand these to the given conversion functions.

To this end, let G be the subgroup of $\mathbb{Z}_q^k$ generated by $\{\mathbf{d}_j\}_{j \in A}$, and fix a coset of G . Since G contains each $\mathbf{d}_j$, whose first coordinate is 1, we may take the representative $\mathbf{w}$ of that coset to have $w_1 = 0$. Then, construct the following sharing vector: $\mathbf{r}_w(\mathbf{u}) := \mathbf{w} + \sum_{j \in A} u_j \mathbf{d}_j$. The LSSS share for party i resulting from $\mathbf{r}_w(\mathbf{u})$ is $M_i \mathbf{r}_w(\mathbf{u}) = M_i \mathbf{w} + u_i M_i \mathbf{d}_i$; it depends on

u_i alone, and so can be computed locally. Furthermore, the secret associated with $\mathbf{r}_w(\mathbf{u})$ is $\sum_{j \in A} u_j \bmod q$ as we wanted. Therefore, the maps

$$g_i(u_i) := F_i(M_i \mathbf{w} + u_i M_i \mathbf{d}_i)$$

form an m -party local conversion scheme from additive sharing modulo q to additive sharing modulo t , correct on $\mathbf{u}$ exactly when $\mathbf{r}_w(\mathbf{u})$ is correct.

It remains to choose the coset. Let κ be the number of tuples $\mathbf{u}$ with $\sum_{j \in A} u_j \mathbf{d}_j = 0$, so that $\mathbf{u} \mapsto \sum_j u_j \mathbf{d}_j$ maps $\mathbb{Z}_q^m$ onto G with kernel of size κ and each element of a coset equals $\mathbf{r}_w(\mathbf{u})$ for exactly κ tuples. Of the q^m tuples, Lq^{m-1} have a secret smaller than L , so every coset holds Lq^{m-1}/κ vectors with such a secret, the same number for each. Drawing x uniformly from $\{0, \dots, L-1\}$ and then $\mathbf{r}$ uniformly subject to $r_1 = x$ is thus the same as drawing a coset uniformly and then, among the tuples $\mathbf{u}$ whose secret is smaller than L , one uniformly – which is the experiment the m -party scheme built from that coset is run in. The error of the original scheme is therefore the average, over the cosets, of the errors of the m -party schemes they define, and some coset does at least as well as that average. $\square$

Lemma 9.3. *If there exists a local conversion scheme from additive secret sharing modulo q to additive secret sharing modulo t for $N \geq 2$ parties, with correctness error ϵ under the bound L , there also exists one for two parties from subtractive secret sharing modulo q to subtractive secret sharing modulo t , i.e., where the secret is the difference of the two shares, with the same correctness error under the same bound.*

Proof. Let the original local conversion scheme be given by $(f_1, \dots, f_N)$. We then define the two-party scheme with algorithms g_1, g_2 , as follows: The first party sets $g_1(x_1) := f_1(x_1)$. The second party with input x_2 additively secret shares $-x_2 \bmod q$ among $N-1$ virtual parties to get shares $y_2, \dots, y_N$ and outputs $g_2(x_2) := -\sum_{j=2}^N f_j(y_j) \bmod t$. The tuple $(x_1, y_2, \dots, y_N)$ is an additive sharing of $x_1 - x_2 \bmod q$, and, the target sharing being additive as well, $g_1(x_1) - g_2(x_2) = \sum_{j=1}^N f_j(\cdot) \bmod t$. The new scheme therefore outputs a correct sharing exactly when the old scheme is correct on that tuple. Moreover, conditioned on the secret, the tuple is uniform among all its additive sharings, since x_1 is uniform and $(y_j)_j$ is uniform subject to summing to $-x_2$. The error probability under the bound L is thus preserved. $\square$

Theorem 9.4 (Lower bound on the error of local conversion). *Let $1 \leq L \leq q$. If $q \bmod t \neq 0$, any local conversion scheme from q to t has correctness error at least*

$$\frac{1}{qL} \left\lfloor \frac{L^2}{4} \right\rfloor$$

under the bound L . In particular, without a promise on the secret the error is at least $\lfloor q^2/4 \rfloor / q^2$, which is $1/4$ for even q and $1/4 - 1/(4q^2)$ for odd q , and under a promise $x < q/\beta$ it is still at least about $1/(4\beta)$.

Proof. By Lemmas 9.2 and 9.3, it suffices to show the theorem for conversion from two-party subtractive sharing modulo q to two-party subtractive sharing modulo t .

Note first that it is sufficient to consider only deterministic conversion functions, since any probabilistic scheme can be seen as a convex linear combination of deterministic schemes, which can do no better than the best deterministic scheme.

So, assume we have a two-party local conversion scheme given by deterministic functions f_1, f_2 , and write $c := q \bmod t$, which is nonzero by assumption. If the input is given by shares x_1, x_2 , taken as integers in $\{0, \dots, q-1\}$, then, as $-q < x_1 - x_2 < q$, the corresponding secret is $x_1 - x_2 + q \cdot [x_1 < x_2]$, which modulo t is $x_1 - x_2 + c \cdot [x_1 < x_2]$. We can then

define new functions $A(x_1) := x_1 - f_1(x_1) \bmod t$ and $B(x_2) := x_2 - f_2(x_2) \bmod t$, so that $f_1(x_1) - f_2(x_2) \equiv_t x_1 - x_2 - A(x_1) + B(x_2)$ and hence

$$(x_1, x_2) \text{ is correct} \iff B(x_2) - A(x_1) \equiv_t c \cdot [x_1 < x_2].$$

As f_1, f_2 range over all maps $\mathbb{Z}_q \rightarrow \mathbb{Z}_t$, so do A and B ; nothing is lost.

Finally, let

$$R_L := \{(x_1, x_2) \in \mathbb{Z}_q^2 : (x_1 - x_2 \bmod q) < L\},$$

which has qL elements and on which the experiment of our model samples uniformly. It therefore suffices to show that every scheme is incorrect on at least $\lfloor L^2/4 \rfloor$ pairs of R_L .

A sliding-window reformulation. Write $i := x_2$ and $d := (x_1 - x_2) \bmod q$, so that $x_1 = (i + d) \bmod q$. This indexes the pairs of R_L bijectively by $(i, d) \in [q] \times [L]$, and the wrap-around condition $x_1 < x_2$ can be equivalently stated as $i + d \geq q$. We will say that (i, d) is *correct* if the underlying conversion scheme is correct on the corresponding input (x_1, x_2) .

For convenience in the following we will modify the functions A, B into $\tilde{A}, \tilde{B}$ that take arbitrary integers as input, and are further tweaked such that (i, d) is correct exactly if $\tilde{A}, \tilde{B}$ output the same value modulo t on inputs $i + d, i$.

Accordingly we define, for all $z \in \mathbb{Z}$,

$$\tilde{A}(z) := A(z \bmod q) + c \lfloor z/q \rfloor, \quad \tilde{B}(z) := B(z \bmod q) + c \lfloor z/q \rfloor.$$

Every such pair of functions arises from a unique pair (A, B) . It is easy to see that we have $\tilde{A}(z + q) \equiv_t \tilde{A}(z) + c$, and similarly for $\tilde{B}$.

Since $0 \leq i + d < 2q$ we have $\lfloor (i + d)/q \rfloor = [i + d \geq q]$, so

$$(i, d) \text{ is correct} \iff \tilde{B}(i) \equiv_t \tilde{A}(i + d).$$

We define *the window at i* to be the set $\{(i + d, i) \mid d = 0, 1, \dots, L - 1\}$. In order to have as many correct pairs in a window as possible, $\tilde{B}$ should be chosen such that on input i , it outputs a value that occurs a maximal number of times among $\tilde{A}(i), \dots, \tilde{A}(i + L - 1)$. Hence, if we define, for $i \in \mathbb{Z}$ and $s \in \mathbb{Z}_t$

$$n_s(i) := \#\{d \in [L] : \tilde{A}(i + d) \equiv_t s\}, \quad \text{def}(i) := L - \max_s n_s(i),$$

there must be at least $\text{def}(i)$ incorrect pairs in each window, and so we obtain

$$\sum_{i \in [q]} \text{def}(i) \leq \#\{(i, d) \mid (i, d) \text{ is incorrect}\}$$

In the following, we call $\text{def}(i)$ the *cost* of the window at i .

As we saw above $\tilde{A}(z + q) \equiv_t \tilde{A}(z) + c$ which implies $n_s(i + q) = n_{s-c}(i)$. From this we get by straightforward calculation that $\text{def}(i) = \text{def}(i + q)$ and so, to lower bound the sum above we may take it over any q consecutive indices.

In order to bound the sum, we need three observations:

- (a) $|n_s(i + 1) - n_s(i)| \leq 1$, since the window slides by one position.
- (b) $n_s(i + q) = n_{s-c}(i)$, as noted above.
- (c) $\text{def}(i) \geq \min(n_u(i), L - n_u(i))$ for every value u : if $n_u(i)$ is maximal, then $\text{def}(i) = L - n_u(i)$; otherwise $\max_s n_s(i) \leq L - n_u(i)$, because the $n_s(i)$ sum to L , and so $\text{def}(i) \geq n_u(i)$.

Let $m := \max_{i,s} n_s(i)$ and $\varepsilon := L - m = \min_i \text{def}(i)$, so ε is a global lower bound on the cost of any window.

If $\varepsilon > L/2$, then each of the q windows costs more than $L/2$, so the total exceeds $qL/2 \geq L^2/2 \geq \lfloor L^2/4 \rfloor$, and we are done.

Otherwise $\varepsilon \leq L/2$. Note that this means the interval $[\varepsilon + 1..L - \varepsilon] = [\varepsilon + 1..m]$ is non-empty, which will come in handy later.

Choose i_0 and u with $n_u(i_0) = m$. Since $c \neq 0$, we have $u - c \neq u$ and hence $n_u(i_0) + n_{u-c}(i_0) \leq \sum_s n_s(i_0) = L$, so that by (b)

$$n_u(i_0 + q) = n_{u-c}(i_0) \leq L - m = \varepsilon.$$

By (a), the integer sequence $i \mapsto n_u(i)$ moves from m to at most ε over $[i_0, i_0 + q]$ in steps of at most one, so it attains every value $j \in \{\varepsilon + 1, \dots, m\}$, and does so at a distinct index of $[i_0, i_0 + q]$. By (c), the window at that index costs at least $\min(j, L - j)$.

We have $q - (m - \varepsilon)$ remaining windows and, using $\varepsilon = L - m$, this number is bounded as

$$q - (m - \varepsilon) \geq L - (m - \varepsilon) = L - (L - 2\varepsilon) = 2\varepsilon.$$

Each of these remaining windows cost at least ε each.

Using $\sum_{j=0}^L \min(j, L - j) = \lfloor L^2/4 \rfloor$ and $\sum_{j \leq \varepsilon} j + \sum_{j < \varepsilon} j = \varepsilon^2$, the number of incorrect pairs is therefore at least

$$\sum_{i=i_0}^{i_0+q-1} \text{def}(i) \geq \sum_{j=\varepsilon+1}^{L-\varepsilon} \min(j, L - j) + 2\varepsilon \cdot \varepsilon = \left(\left\lfloor \frac{L^2}{4} \right\rfloor - \varepsilon^2 \right) + 2\varepsilon^2 \geq \left\lfloor \frac{L^2}{4} \right\rfloor. \quad \square$$

Tightness. The bound of [Theorem 9.4](#) cannot be improved in general, as it is attained in the two-party case. It is attained by the familiar lifting trick for subtractive sharing [\[BKS19\]](#), i.e., when the secret is $x = x_1 - x_2 \bmod q$; this is an LSSS with reconstruction coefficients 1 and -1 . Let the parties simply reduce their shares modulo t , except that the first one subtracts the offset $h := \lfloor L/2 \rfloor$ beforehand and adds it back afterwards:

$$f_1(x_1) := ((x_1 - h \bmod q) \bmod t) + h \bmod t, \quad f_2(x_2) := x_2 \bmod t.$$

Subtracting h centers the range $[0, L)$ of the secret around 0. Since $(x_1 - h \bmod q) + h$ equals x_1 if $x_1 \geq h$ and $x_1 + q$ otherwise, we have $f_1(x_1) - f_2(x_2) \equiv_t x_1 - x_2 + c \cdot [x_1 < h]$, whereas $x \equiv_t x_1 - x_2 + c \cdot [x_1 < x_2]$. As $c \neq 0$, the scheme is thus correct if and only if $[x_1 < h] = [x_1 < x_2]$: the first party predicts the wrap-around from its own share alone, and a wrap-around occurs precisely when $x_1 < x_2$. The pairs of R_L on which the prediction fails are those with $x_1 < h$ that do not wrap around, of which there are $\binom{h+1}{2}$, and those with $x_1 \geq h$ that do, of which there are $\binom{L-h}{2}$. Their sum equals exactly $\lfloor L^2/4 \rfloor$.

The trick is normally used in the regime $L \ll q$, where the offset h is tiny compared to the modulus and its error is about $L/(4q)$. By [Theorem 9.4](#), no local conversion scheme does better.

Acknowledgments. This research was supported by the European Research Council (ERC) under the European Union's Horizon Europe research and innovation programme (grant agreement No 101124977, DECRYPSYS), and by the Next-Generation Cybersecurity programme of the Digital Research Centre Denmark (DIREC), the National Defence Technology Centre (NFC) and the Danish Industry Foundation (project 2025-0766).

Declaration of the use of AI. We have used Claude to write the software implementation and the (trivial) simulation proofs in the appendix; they have all been verified by the authors. Regarding the impossibility result, the authors obtained the result that, assuming each party holds only one value as its share and no bound is promised on the secret, then the error probability of non-interactive modulus conversion from q to t is at least $\approx 1/4$, unless $t \mid q$. The generalization to several values per party and the error probability assuming a promise on the secret were found in dialog with Claude. The proofs thus generated were verified and substantially rewritten by the authors. The authors retain full responsibility for the entire paper.

References

- ACD⁺19. Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via Galois rings. In *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 471–501. Springer, Cham, December 2019.
- AOR⁺19. Abdelrahman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. Cryptology ePrint Archive, Report 2019/974, 2019.
- BBC⁺19a. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Cham, August 2019.
- BBC⁺19b. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. Cryptology ePrint Archive, Report 2019/188, 2019.
- BCD⁺25. Carl Bootland, Kelong Cong, Daniel Demmler, Tore Kasper Frederiksen, Benoit Libert, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Samuel Tap, and Michael Walter. Threshold (fully) homomorphic encryption. Cryptology ePrint Archive, Report 2025/699, 2025.
- BD10. Rikke Bendlin and Ivan Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In *TCC 2010*, volume 5978 of *LNCS*, pages 201–218. Springer, Berlin, Heidelberg, February 2010.
- BGG⁺18. Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 565–596. Springer, Cham, August 2018.
- BGIN21. Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Sublinear GMW-style compiler for MPC with preprocessing. In *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 457–485, Virtual Event, August 2021. Springer, Cham.
- BIB89. Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *8th ACM PODC*, pages 201–209. ACM, August 1989.
- BKS19. Elette Boyle, Lisa Kohl, and Peter Scholl. Homomorphic secret sharing from lattices without FHE. In *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 3–33. Springer, Cham, May 2019.
- BS23. Katharina Boudgoust and Peter Scholl. Simple threshold (fully homomorphic) encryption from LWE with polynomial modulus. In *ASIACRYPT 2023, Part I*, volume 14438 of *LNCS*, pages 371–404. Springer, Singapore, December 2023.
- Can00. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000.
- Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
- CDE⁺18. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Cham, August 2018.
- CdH10. Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Berlin, Heidelberg, September 2010.
- CDI05. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Berlin, Heidelberg, February 2005.
- CFIK03. Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. Efficient multi-party computation over rings. In *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 596–613. Springer, Berlin, Heidelberg, May 2003.
- CKL24. Wonhee Cho, Jiseung Kim, and Changmin Lee. (In)security of threshold fully homomorphic encryption based on Shamir secret sharing. Cryptology ePrint Archive, Report 2024/1858, 2024.
- Cle86. Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*, pages 364–369. ACM Press, May 1986.
- DEF⁺19. Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120. IEEE Computer Society Press, May 2019.
- DFK⁺06. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, Berlin, Heidelberg, March 2006.
- DKOP26. Ivan Damgård, Sebastian Kolby, Claudio Orlandi, and Stanislas Pawlak. Efficient MPC-based modulus conversion for threshold FHE decryption. Cryptology ePrint Archive, Report 2026/1058, 2026.

- DN03. Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *CRYPTO 2003*, volume 2729 of *LNCS*, pages 247–264. Springer, Berlin, Heidelberg, August 2003.
- DNNP25. Tamar Ben David, Varun Narayanan, Olga Nissenbaum, and Anat Paskin-Cherniavsky. New results in share conversion, with applications to evolving access structures. In *ITC 2025*, volume 343 of *LIPICs*, pages 11:1–11:23. Schloss Dagstuhl, August 2025.
- DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Berlin, Heidelberg, August 2012.
- dVPS26. François Colin de Verdière, Alain Passelègue, and Damien Stehlé. On threshold fully homomorphic encryption with synchronized decryptors. Cryptology ePrint Archive, Report 2026/031, 2026.
- HLW⁺25. Zhenkai Hu, Haofei Liang, Xiao Wang, Xiang Xie, Kang Yang, Yu Yu, and Wenhao Zhang. Ajax: Fast threshold fully homomorphic encryption without noise flooding. Cryptology ePrint Archive, Report 2025/1834, 2025.
- KLL⁺26. Won Kim, Changmin Lee, JeongHwan Lee, Alain Passelègue, and Damien Stehlé. Asynchronous Lagrange-based threshold FHE with smaller modulus overhead. Cryptology ePrint Archive, Report 2026/973, 2026.
- MRVW21. Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In *FC 2021, Part I*, volume 12674 of *LNCS*, pages 249–270. Springer, Berlin, Heidelberg, March 2021.
- MUdV23. Wicher Malten, Mehmet Ugurbil, and Miguel de Vega. More efficient comparison protocols for MPC. Cryptology ePrint Archive, Report 2023/1934, 2023.
- NO07. Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC 2007*, volume 4450 of *LNCS*, pages 343–360. Springer, Berlin, Heidelberg, April 2007.
- ON20. Satsuya Ohata and Koji Nuida. Communication-efficient (client-aided) secure two-party protocols and its application. In *FC 2020*, volume 12059 of *LNCS*, pages 369–385. Springer, Cham, February 2020.
- OT25. Hiroki Okada and Tsuyoshi Takagi. Low communication threshold FHE from standard (module-)LWE. In *ASIACRYPT 2025, Part VII*, volume 16251 of *LNCS*, pages 165–198. Springer, Singapore, December 2025.
- PS24. Alain Passelègue and Damien Stehlé. Low communication threshold fully homomorphic encryption. In *ASIACRYPT 2024, Part I*, volume 15484 of *LNCS*, pages 297–329. Springer, Singapore, December 2024.
- Rei09. Tord Ingolf Reistad. Multiparty comparison: An improved multiparty protocol for comparison of secret-shared values. In *SECRYPT 2009 - Proceedings of the International Conference on Security and Cryptography*, pages 325–330. SciTePress, 2009.
- RST⁺22. Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. *Journal of Cryptology*, 35(1):5, January 2022.
- RST⁺26. Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Correction to: Actively secure setup for SPDZ. *Journal of Cryptology*, 39(1):22, 2026.
- RW19. Dragos Rotaru and Tim Wood. MARBled circuits: Mixing arithmetic and Boolean circuits with active security. In *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Cham, December 2019.
- WGC19. Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, July 2019.
- WTB⁺21. Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETs*, 2021(1):188–208, January 2021.
- YLKM23. Albert Yu, Donghang Lu, Aniket Kate, and Hemanta K. Maji. SIM: Secure interval membership testing and applications to secure comparison. In *2023 IEEE European Symposium on Security and Privacy*, pages 757–772. IEEE Computer Society Press, July 2023.
- Yu11. Ching-Hua Yu. Sign modules in secure arithmetic circuits. Cryptology ePrint Archive, Report 2011/539, 2011.
- ZCZC26. Qinghui Zhang, Xiaojun Chen, Yansong Zhang, and Xudong Chen. SC-DT: Scalable constant round secure comparison and its application to privacy decision tree evaluation. Cryptology ePrint Archive, Report 2026/1389, 2026.
- ZZLP25. Guy Zyskind, Doron Zarchy, Max Leibovich, and Chris Peikert. High-throughput universally composable threshold FHE decryption. In *ACM CCS 2025*, pages 2339–2353. ACM Press, October 2025.

A Checked Input via Fully Linear Proofs

In this appendix, we realize the `CheckedInput` command of [Functionality 3.1](#). We first recall the proof systems that it rests on and then compile them onto the ABB.

A.1 Fully Linear Proof Systems

The checks that we build are zero-knowledge proofs on secret-shared data in the sense of Boneh et al. [\[BBC⁺19a\]](#), whose relevant notions we recall here. The setting is that one party, the prover, knows a vector $\mathbf{x}$ in its entirety, while all parties jointly hold $[\![\mathbf{x}]\!]$ and jointly act as the verifier. No single verifier holds $\mathbf{x}$, so the verifier can access $\mathbf{x}$ only through the sharing. What makes verification cheap is that it accesses $\mathbf{x}$ only through *linear* queries: each such query is answered by local computation followed by a single opening, with no multiplications on secret-shared values. Note that this constrains how the verifier accesses $\mathbf{x}$, not what it computes: the decision predicate itself may be an arbitrary function of the query answers, which are public once opened.

The proof systems we use are interactive; such a system is called a fully linear interactive oracle proof (FLIOP). We only need its public-coin variant, in which every challenge is a uniformly random value made public as soon as it is drawn.

Definition A.1 (Public-coin zk-FLIOP [\[BBC⁺19b, Defs. 3.9 and 3.11, adapted\]](#)).

Let $\mathcal{R}$ be a finite ring and $\mathcal{L} \subseteq \mathcal{R}^n$ a language. A ρ -round ℓ -query public-coin zk-FLIOP for $\mathcal{L}$ with soundness error ϵ is an interactive protocol between a prover P and a verifier V that proceeds as follows. Let $S_1, \dots, S_\rho$ be a sequence of finite challenge sets, given as part of the protocol description. P 's initial input is $\mathbf{x}$ and V receives no input. In round $i \in \{1, \dots, \rho\}$:

- P outputs a proof $\pi_i \in \mathcal{R}^{m_i}$;
- V , after π_i is fixed, samples a uniformly random challenge $r_i \xleftarrow{\$} S_i$, computes the query vectors $\mathbf{q}_{i,1}, \dots, \mathbf{q}_{i,\ell_i} \in \mathcal{R}^{n+m_i}$ as a function only of $(r_1, \dots, r_i)$, and sends r_i to P .

In the end, V computes the decision (accept or reject) as a function only of the challenges and the answers $a_{i,j} := \langle \mathbf{x} \parallel \pi_i, \mathbf{q}_{i,j} \rangle \in \mathcal{R}$. We call $\sum_{i=1}^\rho m_i$ the proof length and $\ell := \sum_{i=1}^\rho \ell_i$ the query complexity, and we require the following three properties.

- **Completeness.** V accepts whenever $\mathbf{x} \in \mathcal{L}$ and P is honest.
- **Soundness.** For every $\mathbf{x} \notin \mathcal{L}$ and every computationally unbounded malicious prover, V accepts with probability at most ϵ .
- **Strong honest-verifier zero-knowledge (HVZK).** There is a simulator Sim that takes no input such that, for every $\mathbf{x} \in \mathcal{L}$, the output of Sim is distributed identically to the honest verifier's view $((r_1, \dots, r_\rho), (a_{i,j})_{i,j})$ in an execution with an honest prover on $\mathbf{x}$.

The last of these properties is what keeps $\mathbf{x}$ hidden from a verifier who does not access $\mathbf{x}$ directly, but only through the prescribed linear queries. Note that nothing binds a *malicious* verifier to the prescribed queries. That suffices for us: in our application ([Theorem A.3](#) below) the queries are determined by public coins that no party controls, and only the query answers $a_{i,j}$ are opened.

The statements we prove are systems of equations of low degree, for which [\[BBC⁺19a\]](#) construct a proof system whose length is logarithmic in the size of the system.

Theorem A.2 (Public-coin zk-FLIOP for low-degree languages [\[BBC⁺19b, Corollary E.3, adapted\]](#)). Let $S \subseteq \mathcal{R}$ be a set whose pairwise differences are units, let $D \geq 2$, and let $\mathcal{L} := \{\mathbf{x} \in \mathcal{R}^n \mid C_1(\mathbf{x}) = \dots = C_m(\mathbf{x}) = 0\}$ for polynomials $C_1, \dots, C_m$ over $\mathcal{R}$ in n variables of degree at most D , in which μ monomials occur in total. Then $\mathcal{L}$ admits a

public-coin zk-FLIOP, drawing all its challenges from S , that has $\mathcal{O}(\log \mu)$ rounds, proof length $\mathcal{O}(D \log \mu)$, query complexity $D + \mathcal{O}(\log \mu)$, and soundness error $\mathcal{O}((m + D \log \mu)/|S|)$.⁵

The challenge set S appears in the theorem because $\mathcal{R}$ need not be a field, as is the case for the rings $\mathbb{Z}_q$ that we work with. The soundness bounds of [BBC⁺19a] come from bounding the number of zeros of a nonzero polynomial, which over a ring exceeds its degree in general; restricting the challenges to a set whose pairwise differences are units restores the bound, in that a nonzero polynomial has at most as many zeros in S as its degree, which is why it is $|S|$ rather than $|\mathcal{R}|$ that determines the soundness error. For a field one may take $S = \mathcal{R}$. For $\mathcal{R} = \mathbb{Z}_q$ one may take $S = \{0, \dots, p-1\}$, where p is the smallest prime factor of q , since the difference of two such elements is nonzero modulo every prime factor of q and hence a unit; no larger set will do, as any $p+1$ elements contain two that coincide modulo p . When p is too small for the desired soundness error, as it is for $\mathbb{Z}_{2^k}$, one replaces $\mathcal{R}$ by an extension $\mathbb{Z}_q[x]/f(x)$ of degree d with f irreducible modulo every prime factor of q , which raises $|S|$ to p^d [BBC⁺19b, Claim D.2]. Besides the choice of S , the constructions underlying Theorem A.2 have to be adapted to the ring setting. This is carried out in [BBC⁺19b, Appendices D and E], and [BBC⁺19b, Corollary E.3] states the resulting parameters for extensions of $\mathbb{Z}_w$ with an arbitrary modulus w , which covers all moduli we use.

A.2 From Proof Systems to Checked Input

The CheckedInput command of Functionality 3.1 lets a party input values together with a proof that they lie in a given language, revealing only whether they do. Theorem A.3 realizes it in the $\mathcal{F}_{\text{ABB}}$ -hybrid model from a fully linear proof system for that language. It follows the compiler of [BBC⁺19b, Theorem 6.6] in the variant of [BBC⁺19b, Section 6.2.1], where the statement is the secret-shared value itself. Note that [BBC⁺19b, Theorem 6.6] tolerates a corrupted prover or corrupted verifiers, but not both, and [BBC⁺19b, Theorem 6.10] tolerates both only if at most half of the verifiers are corrupted. As Boyle et al. [BGIN21] point out, soundness against a prover colluding with verifiers needs the statement to be held robustly. In our setting, $\mathcal{F}_{\text{ABB}}$ holds the statement and opens every query answer itself, so soundness holds against a corrupted prover colluding with any number of corrupted parties. The public coins of Definition A.1 are drawn from $\mathcal{F}_{\text{Rand}}$.

Theorem A.3 (Checked input from fully linear proofs). *For every prime power $q \in M$, let $\mathfrak{L}_q$ be a set of languages $\mathcal{L} \subseteq \mathbb{Z}_q^n$, each of which comes with an extension ring $\mathcal{R}_{\mathcal{L}} = \text{GR}(q, d_{\mathcal{L}})$ of $\mathbb{Z}_q$, a language $\hat{\mathcal{L}} \subseteq \mathcal{R}_{\mathcal{L}}^n$ with $\hat{\mathcal{L}} \cap \mathbb{Z}_q^n = \mathcal{L}$, and a public-coin zk-FLIOP $\Pi_{\mathcal{L}}$ for $\hat{\mathcal{L}}$ with soundness error at most ϵ . Then there is a protocol that statistically realizes $\mathcal{F}_{\text{ABB}}^M \setminus \{\text{maBit}\}$ in the $(\mathcal{F}_{\text{ABB}}^M \setminus \{\text{CheckedInput}, \text{maBit}\}, \mathcal{F}_{\text{Rand}})$ -hybrid model with simulation error ϵ per CheckedInput command, provided that every such command over a modulus q uses a language in $\mathfrak{L}_q$. A command of P_{σ} for $\mathcal{L} \subseteq \mathbb{Z}_q^n$ inherits the parameters of $\Pi_{\mathcal{L}}$: if $\Pi_{\mathcal{L}}$ has ρ rounds, proof length m and query complexity ℓ , the command takes ρ rounds, each consisting of Input, $\mathcal{F}_{\text{Rand}}$ and Open commands, and costs $n + d_{\mathcal{L}}m$ calls to $\mathcal{F}_{\text{ABB}}$.Input by P_{σ} , ρ calls to $\mathcal{F}_{\text{Rand}}$ and $d_{\mathcal{L}}\ell$ calls to $\mathcal{F}_{\text{ABB}}$.Open in total.*

Proof. The protocol. Every command other than CheckedInput is passed on unchanged, so we only consider a CheckedInput command of P_{σ} over q for $\mathcal{L} \in \mathfrak{L}_q$, and we write $\mathcal{R} := \mathcal{R}_{\mathcal{L}}$. P_{σ} inputs $\llbracket \mathbf{x} \rrbracket_q$ via $\mathcal{F}_{\text{ABB}}$.Input, which is also a sharing $\llbracket \mathbf{x} \rrbracket_{\mathcal{R}}$ since $\mathbb{Z}_q \subseteq \mathcal{R}$. The parties then run $\Pi_{\mathcal{L}}$ on it, with P_{σ} as the prover and all parties jointly as the verifier. In round i , P_{σ} inputs

⁵ The two summands have different origins: the term $\mathcal{O}(m/|S|)$ is the price of the random linear combination by which the system is reduced to a single equation, whereas the term $\mathcal{O}(D \log \mu/|S|)$ is the error of the proof system for that equation [BBC⁺19b, Theorem 5.1].

$\llbracket \pi_i \rrbracket_{\mathcal{R}}$ via $\mathcal{F}_{\text{ABB}}.\text{Input}$, the parties draw $r_i \leftarrow \mathcal{F}_{\text{Rand}}(S_i)$, and for each of the ℓ_i queries they compute $\llbracket a_{i,j} \rrbracket_{\mathcal{R}}$ by **Comb** commands and open it, which is possible since $a_{i,j} = \langle \mathbf{x} \parallel \pi_i, \mathbf{q}_{i,j} \rangle$ is a linear combination of values held by $\mathcal{F}_{\text{ABB}}$ with public coefficients. After the last round, every party evaluates the decision predicate of $\Pi_{\mathcal{L}}$ on the public values $(r_1, \dots, r_\rho)$ and $(a_{i,j})_{i,j}$, so all parties reach the same decision, and they output **accept** or **abort** accordingly. The stated cost follows by counting.

Simulation. The simulator emulates $\mathcal{F}_{\text{ABB}}$ and $\mathcal{F}_{\text{Rand}}$ towards the adversary. If P_σ is honest, the view of the corrupted parties consists of the challenges and the opened answers, which is the honest verifier's view in $\Pi_{\mathcal{L}}$, since $\mathcal{F}_{\text{Rand}}$ draws the challenges honestly whichever parties are corrupted. The simulator of strong HVZK produces this view from no input, so this case is simulated perfectly.

If P_σ is corrupted, the simulator sees $\mathbf{x}$ and the proofs π_i , since it emulates $\mathcal{F}_{\text{ABB}}$, so it can run the real execution with the adversary. It sends $\mathbf{x}$ to the ideal functionality, and if the real verifier rejects, it issues **abort**. The two executions then differ only if $\mathbf{x} \notin \mathcal{L}$ and the real verifier accepts, since the ideal functionality accepts exactly the $\mathbf{x} \in \mathcal{L}$. In the real execution, $\mathcal{F}_{\text{ABB}}$ opens the correct answers for the values it holds, and $\mathcal{F}_{\text{Rand}}$ draws each r_i independently of π_i , which is fixed before, so the corrupted parties influence the execution only through $\mathbf{x}$ and the π_i , i.e., as a malicious prover in $\Pi_{\mathcal{L}}$. As $\mathbf{x} \in \mathbb{Z}_q^n \setminus \mathcal{L}$ lies outside $\hat{\mathcal{L}}$, the real verifier accepts with probability at most ϵ , which is the claimed error per command. $\square$

Guaranteed Output Delivery. **Theorem A.3** provides security with abort, matching the abort command of **Functionality 3.1**. Over an ABB instantiation with GOD, the same protocol provides the amended **CheckedInput** command of **Section 3.1**: the decision is a function of values that $\mathcal{F}_{\text{ABB}}$ and $\mathcal{F}_{\text{Rand}}$ make public, so only the inputs of P_σ can make it a rejection, and an honest P_σ is never rejected by completeness. If P_σ withholds one of its inputs, the amended **Input** tells all parties so, and they reject P_σ as well.

A.3 Checked Input of Degree-Two Languages

Every language we check is defined by equations of degree two, so **Theorem A.2** supplies the proof systems that **Theorem A.3** needs. It remains to choose the extension ring and the challenge set S , whose pairwise differences have to be units and whose size determines the soundness error.

Corollary A.4 (Checked input of degree-two languages). *Let κ be a statistical security parameter and, for every prime power $q \in M$, let $\mathfrak{L}_q$ be a set of languages $\mathcal{L} \subseteq \mathbb{Z}_q^n$, each defined by a system of equations of degree two over $\mathbb{Z}_q$. Then there is a protocol that statistically realizes $\mathcal{F}_{\text{ABB}}^M \setminus \{\text{maBit}\}$ in the $(\mathcal{F}_{\text{ABB}}^M \setminus \{\text{CheckedInput}, \text{maBit}\}, \mathcal{F}_{\text{Rand}})$ -hybrid model with simulation error $2^{-\kappa}$ per **CheckedInput** command, provided that every such command over a modulus q uses a language in $\mathfrak{L}_q$. If μ monomials occur in the system defining $\mathcal{L}$, a command of P_σ for $\mathcal{L}$ costs $\mathcal{O}(\log \mu)$ rounds, $n + \mathcal{O}(d \log \mu)$ calls to $\mathcal{F}_{\text{ABB}}.\text{Input}$ by P_σ and $\mathcal{O}(d \log \mu)$ calls to $\mathcal{F}_{\text{ABB}}.\text{Open}$, where $d = \mathcal{O}((\kappa + \log \mu) / \log p)$ for the prime p dividing q .*

Proof. Let $\mathcal{R}_{\mathcal{L}} := \text{GR}(q, d)$, and let S consist of the p^d elements of $\mathcal{R}_{\mathcal{L}}$ with all coefficients in $\{0, \dots, p-1\}$, whose pairwise differences are units by [BBC⁺19b, Claim D.2]. Reading the equations that define $\mathcal{L}$ over $\mathcal{R}_{\mathcal{L}}$ defines a language $\hat{\mathcal{L}} \subseteq \mathcal{R}_{\mathcal{L}}^n$ with $\hat{\mathcal{L}} \cap \mathbb{Z}_q^n = \mathcal{L}$. **Theorem A.2** gives a public-coin zk-FLIOP for $\hat{\mathcal{L}}$ with $\mathcal{O}(\log \mu)$ rounds, proof length and query complexity, and with soundness error $\mathcal{O}((m + 2 \log \mu) / p^d) = \mathcal{O}(\mu / p^d)$, where $m \leq \mu$ is the number of equations, which the stated d makes at most $2^{-\kappa}$. Now **Theorem A.3** gives the claim. $\square$

A.4 Checked Input of Bit Vectors

We now instantiate [Corollary A.4](#) for the language $\{0, 1\}^n$ of bit vectors, which the protocols of [Section 4.2](#) and [Appendix B](#) use.

Corollary A.5 (Checked input of bit vectors). *Let κ be a statistical security parameter. There is a protocol that statistically realizes $\mathcal{F}_{\text{ABB}}^M \setminus \{\text{maBit}\}$ in the $(\mathcal{F}_{\text{ABB}}^M \setminus \{\text{CheckedInput}, \text{maBit}\}, \mathcal{F}_{\text{Rand}})$ -hybrid model with simulation error $2^{-\kappa}$ per `CheckedInput` command, provided that every such command uses a language $\{0, 1\}^n$. A command of P_σ for $\{0, 1\}^n$ over the modulus q costs $\mathcal{O}(\log n)$ rounds, $n + \mathcal{O}(d \log n)$ calls to $\mathcal{F}_{\text{ABB}}.\text{Input}$ by P_σ and $\mathcal{O}(d \log n)$ calls to $\mathcal{F}_{\text{ABB}}.\text{Open}$, with d as in [Corollary A.4](#).*

Proof. The language $\{0, 1\}^n$ is defined by the n equations $x_k(x_k - 1) = 0$ of degree two, in which $\mu = 2n$ monomials occur, so [Corollary A.4](#) applies. $\square$

Beyond the n inputs themselves, the check costs P_σ only $\mathcal{O}(\log n)$ inputs of elements of the extension ring, and the parties open as many, which for prime q amounts to $\mathcal{O}(\kappa \log n + \log^2 n)$ bits. Since the check runs once per batch of n bits in the preprocessing, both this communication and its $\mathcal{O}(\log n)$ rounds are amortized over the batch.

B maBits

A *multiply authenticated bit*, or *maBit* for short [\[RST⁺26\]](#), is a single bit, known to no party, that is secret-shared over several moduli at once. That is, for moduli $q_1, \dots, q_\ell$, a $(q_1, \dots, q_\ell)$ -maBit is a tuple $(\llbracket b \rrbracket_{q_1}, \dots, \llbracket b \rrbracket_{q_\ell})$ of sharings of one and the same $b \in \{0, 1\}$. It is authenticated in the sense that the sharings are held by $\mathcal{F}_{\text{ABB}}$, so that no corrupted party can make an opening deviate from the shared value. For $\ell = 2$ and one of the two moduli equal to 2, maBits coincide with the daBits from [\[RW19, AOR⁺19\]](#). The work [\[RST⁺26\]](#) also gives a protocol generating maBits.⁶ However, that protocol supports only additive (i.e., full threshold) secret sharing schemes. In this section, we give an alternative maBit protocol that works under every instantiation of the ABB, rather than only full-threshold ones.

The restriction to additive sharing comes from the way the protocol of [\[RST⁺26\]](#) generates the bits: the bits are first produced over a single auxiliary modulus and are then moved to the target moduli, and moving them rests on their Lemma 4.1, a rounding trick that works only for additive secret sharing. Our protocol below moves nothing between moduli at all: the parties draw local randomness once and the sharing is formed inside each modulus separately.

We also construct a new consistency check (in [Appendix B.2](#)) to certify that the sharings produced over the ℓ moduli all encode one and the same bit vector. This prevents corrupted parties from undetectedly inputting different bits over different moduli. Our check differs from the check of [\[RST⁺26\]](#) in that all its inputs are guaranteed to be bits, which reduces the number of maBits it sacrifices whenever a modulus is a proper prime power. For example, for $q = 2^{64}$, $\kappa = 64$ and a batch of n maBits, our check sacrifices $\lceil \log n \rceil + 128$ maBits instead of $64(\lceil \log n \rceil + 65)$ (when the check of [\[RST⁺26\]](#), which is stated for primes, is carried over to 2-powers).

B.1 Generation

Generating maBits over prime powers of pairwise distinct primes is all we need to consider. A sharing over a composite modulus is by the convention of [Section 3.1](#) nothing but the tuple of sharings over its prime-power factors, so maBits over such a modulus are obtained

⁶ That version corrects [\[RST⁺22\]](#), whose privacy rested on a subset-sum assumption that was attacked. The correction replaces the assumption by statistical masking.

by generating them over each factor. And if two of the resulting prime powers were powers $p^{e'} \leq p^e$ of the same prime, it would suffice to generate the maBits over p^e , since $p^{e'}$ divides p^e and the DivModConv command of [Functionality 3.1](#) therefore turns $\llbracket b \rrbracket_{p^e}$ into $\llbracket b \rrbracket_{p^{e'}}$.

Protocol B.1. $\Pi_{\text{maBits}}^{n, (q_1, \dots, q_\ell)}$

This protocol generates n independent $(q_1, \dots, q_\ell)$ -maBits. As explained above, we assume w.l.o.g. that the q_j are prime powers of pairwise distinct primes. Furthermore, we assume w.l.o.g. that q_ℓ is the largest modulus and satisfies the precondition of [Protocol B.3](#).

1. Each party P_σ samples $\mathbf{x}_\sigma \xleftarrow{\$} \{0, 1\}^{n'}$, where n' is the input length that [Protocol B.3](#) requires in order to return n maBits.
2. For $j \in \{1, \dots, \ell\}$ (in parallel): $\llbracket \mathbf{b} \rrbracket_{q_j} \leftarrow \Pi_{\text{xorBits}}^{(q_j)}(\mathbf{x}_1, \dots, \mathbf{x}_N)$ ([Protocol B.2](#)).
3. Return $\Pi_{\text{maBitsConsCheck}}^{n, (q_1, \dots, q_\ell)}(\llbracket \mathbf{b} \rrbracket_{q_1}, \dots, \llbracket \mathbf{b} \rrbracket_{q_\ell})$ ([Protocol B.3](#)).

Remark. If none of $q_1, \dots, q_\ell$ is large enough to satisfy the precondition of [Protocol B.3](#), the parties run [Protocol B.1](#) for the moduli $(q_1, \dots, q_\ell, \hat{Q})$, where $\hat{Q}$ is an auxiliary prime power that is large enough, and discard $\llbracket \mathbf{b} \rrbracket_{\hat{Q}}$ afterwards. Consistency of the remaining ℓ sharings then follows from each of them being checked against $\llbracket \mathbf{b} \rrbracket_{\hat{Q}}$.

It remains to supply the XOR subprotocol that [Protocol B.1](#) invokes for each modulus:

Protocol B.2. $\Pi_{\text{xorBits}}^{(q)}(\mathbf{x}_1, \dots, \mathbf{x}_N)$ for prime-power q

On input $\mathbf{x}_\sigma \in \{0, 1\}^n$ from each party P_σ , this protocol computes $\llbracket \bigoplus_\sigma \mathbf{x}_\sigma \rrbracket_q$. Here, q must be a prime power.

If q is a 2-power:

1. Each party P_σ samples $\mathbf{m}_\sigma \xleftarrow{\$} \mathbb{Z}_q^n$ and inputs $\llbracket \mathbf{x}_\sigma + 2\mathbf{m}_\sigma \rrbracket_q$.
2. Open $\llbracket \mathbf{y} \rrbracket_q := \sum_\sigma \llbracket \mathbf{x}_\sigma + 2\mathbf{m}_\sigma \rrbracket_q + \llbracket \mathbf{b} \rrbracket_q$ where $\llbracket \mathbf{b} \rrbracket_q \leftarrow \mathcal{F}_{\text{ABB}}.\text{RandomBit}()$.
3. Return the vector whose k -th entry is $\llbracket b_k \rrbracket_q$ if y_k is even and $1 - \llbracket b_k \rrbracket_q$ otherwise.

Else:

1. For each party P_σ , the parties call $\mathcal{F}_{\text{ABB}}.\text{CheckedInput}(P_\sigma, \{0, 1\}^n, \cdot)$, on which P_σ provides $\mathbf{x}_\sigma$, resulting in $\llbracket \mathbf{x}_\sigma \rrbracket_q$.
2. Return $\llbracket \mathbf{b} \rrbracket_q := \bigoplus_\sigma \llbracket \mathbf{x}_\sigma \rrbracket_q$,
where $\oplus$ is defined by $\llbracket u \rrbracket_q \oplus \llbracket v \rrbracket_q := \llbracket u \rrbracket_q + \llbracket v \rrbracket_q - 2 \cdot \llbracket u \rrbracket_q \cdot \llbracket v \rrbracket_q$.

For a 2-power modulus, [Protocol B.2](#) takes the cheaper route of summing the inputs and extracting the least-significant bit of the sum. The restriction of this route to 2-powers has two sources. First, the protocol needs q to be even, since otherwise the doubled mask $2 \sum_\sigma \mathbf{m}_\sigma$ could wrap around modulo q and thereby flip the parity of the opened value $\mathbf{y}$. Second, $\mathcal{F}_{\text{ABB}}$ is restricted to prime powers, which leaves the 2-powers.

For $q = 2$, no masking is needed at all, since $\Pi_{\text{xorBits}}^{(2)}$ may return the sum of the inputs directly. A sharing over $\mathbb{Z}_2$ therefore costs a maBit no multiplication. In contrast, over a modulus that is not a 2-power, the XOR costs $N - 1$ multiplications per bit, whereas the maBit protocol of [\[RST⁺26\]](#) spends one multiplication per bit in its auxiliary modulus and none in the targets; this is the price of supporting every instantiation of the ABB, instead of just additive secret sharing as in [\[RST⁺26\]](#).

B.2 Consistency Check

Here, we give the check that [Protocol B.1](#) runs in its last step. It takes ℓ secret-shared bit vectors of length n' , held over pairwise distinct moduli $q_1, \dots, q_\ell$, and verifies that they all encode the same bit vector, without revealing anything else about it. Throughout, we assume that each input $\llbracket \mathbf{b}^{(j)} \rrbracket_{q_j}$ is already known to be a sharing of a bit vector, e.g., because the calling protocol obtained it through the `CheckedInput` command of [Corollary A.5](#).

The idea is as follows. The parties draw a random integer vector $\mathbf{w}$ with entries in $\{0, \dots, W-1\}$ and open $\langle \mathbf{w}, \llbracket \mathbf{b}^{(j)} \rrbracket_{q_j} \rangle$ for every j . Over the largest modulus q_ℓ , the combination does not wrap around, so this opening reveals the integer $\langle \mathbf{w}, \mathbf{b}^{(\ell)} \rangle$, which the parties compare modulo q_i with the opening over q_i for every $i < \ell$. If $\mathbf{b}^{(i)} \neq \mathbf{b}^{(\ell)}$, this comparison passes with probability at most $1/W$, and it is repeated R times to bring this down to $2^{-\kappa}$. Since the openings reveal linear equations in the bits, the protocol *sacrifices* part of its input: a block of $\tau := \lceil \log(nW) \rceil + \kappa$ input bits with the coefficients $1, 2, \dots, 2^{\tau-1}$ serves as a statistical mask of the opened value, and each repetition uses a fresh block. A caller who needs n maBits therefore generates $n' := n + R\tau$ of them and receives the unsacrificed prefixes of length n .

Bitness Assumptions. The assumption that the inputs are bit vectors serves two purposes. Over q_ℓ , it bounds $\langle \mathbf{w}, \mathbf{b}^{(\ell)} \rangle$ so that the opening does not wrap around; without this bound the openings could not be compared at all, and the check of [\[RST⁺26\]](#) rests on the same guarantee. Over q_i with $i < \ell$, it makes a nonzero difference of two entries a unit in $\mathbb{Z}_{q_i}$, so that at most one coefficient in $\{0, \dots, W-1\}$ leaves the difference undetected. Without the second assumption, the number of repetitions would have to be governed by the smallest prime dividing a modulus instead of by $q_{\min}$, as in the check of [\[RST⁺26\]](#), whose bits are only known to be bits over the auxiliary modulus.

Protocol B.3. $\Pi_{\text{maBitsConsCheck}}^{n, (q_1, \dots, q_\ell)}(\llbracket \mathbf{b}^{(1)} \rrbracket_{q_1}, \dots, \llbracket \mathbf{b}^{(\ell)} \rrbracket_{q_\ell})$

W.l.o.g. $q_\ell = \max_{j \in \{1, \dots, \ell\}} q_j$. Let κ be a statistical security parameter. Let $q_{\min} := \min_{i \in \{1, \dots, \ell-1\}} q_i$, $R := \lceil \kappa / \log q_{\min} \rceil$, $W := \lceil 2^{\kappa/R} \rceil$ and $\tau := \lceil \log(nW) \rceil + \kappa$.

Precondition: $q_\ell \geq 2^{\tau+1}$, and each input vector has length $n' = n + R\tau$.

The choice of n' lets the sacrificed indices $\{n, \dots, n' - 1\}$ be partitioned into blocks $K^{(\rho)} = (k_0^{(\rho)}, \dots, k_{\tau-1}^{(\rho)})$, one per iteration $\rho \in \{1, \dots, R\}$.

For each $\rho \in \{1, \dots, R\}$ (in parallel):

1. $\mathbf{r}^{(\rho)} := (r_k^{(\rho)})_{k \in [n]} \leftarrow \mathcal{F}_{\text{Rand}}(\{0, \dots, W-1\}^n)$.
2. Let $\mathbf{w}^{(\rho)} \in \mathbb{Z}^{n'}$ be the public vector whose entry at $k \in [n]$ is $r_k^{(\rho)}$, whose entry at $k_c^{(\rho)}$ is 2^c for $c \in [\tau]$, and whose remaining entries are 0.
3. Compute and open $\llbracket \tilde{z}^{(\rho)} \rrbracket_{q_\ell} := \langle \mathbf{w}^{(\rho)}, \llbracket \mathbf{b}^{(\ell)} \rrbracket_{q_\ell} \rangle$.
4. For each $i \in \{1, \dots, \ell-1\}$: compute and open $\llbracket z^{(i, \rho)} \rrbracket_{q_i} := \langle \mathbf{w}^{(\rho)}, \llbracket \mathbf{b}^{(i)} \rrbracket_{q_i} \rangle$.
5. If $z^{(i, \rho)} \neq \tilde{z}^{(\rho)} \bmod q_i$ for some i , abort.

If no invocation of Step 5 aborted, return the unsacrificed prefixes $(\llbracket b_k^{(j)} \rrbracket_{q_j})_{k \in [n]}$ for $j \in \{1, \dots, \ell\}$.

The repetition count R and the coefficient bound W of [Protocol B.3](#) are the best possible. A corrupted party that deviates in one position at the modulus $q_{\min}$ passes a repetition with probability at least $1/q_{\min}$, whatever the coefficient bound is, so at least $\kappa / \log q_{\min}$ repetitions are needed. For a bound $W \leq q_{\min}$, this probability is exactly $1/W$, so W is the smallest bound for which R repetitions suffice. For example, $q_{\min} = 2$ gives $R = \kappa$ and $W = 2$, and for

$\kappa = 64$, $q_{\min} = 2^{32}$ gives $R = 2$ and $W = 2^{32}$; for $q_{\min} \geq 2^\kappa$, a single repetition with $W = 2^\kappa$ suffices, which sacrifices $\tau = \lceil \log n \rceil + 2\kappa$ maBits.

Theorem B.4 (Security of $\Pi_{\text{maBitsConsCheck}}$). *Let κ be a statistical security parameter, and consider an execution of [Protocol B.3](#) whose precondition holds and whose inputs are sharings of bit vectors. If $\mathbf{b}^{(1)} = \dots = \mathbf{b}^{(\ell)}$, then the protocol does not abort and returns their prefixes of length n . For arbitrary inputs, it returns prefixes that are not all equal with probability at most $2^{-\kappa}$. If, given the adversary's view, the sacrificed bits are uniform and independent of each other and of the remaining bits, then the opened values and whether the protocol aborts can be simulated from the differences $\mathbf{b}^{(i)} - \mathbf{b}^{(\ell)}$ alone, within statistical distance $2^{-\kappa}R$.*

For equal inputs the simulator is given nothing, so the check reveals nothing about the returned bits, and whether it aborts does not depend on them either. The hypothesis on the sacrificed bits fails only if the adversary knows them, which happens in [Step 4](#) of [Protocol 4.11](#) when the party supplying $\mathbf{v}^\sigma$ is corrupted; in that case, everything the check opens is a function of that party's inputs, and the simulator computes it directly.

Proof. Fix $\rho \in \{1, \dots, R\}$ and write $\mathbf{w} := \mathbf{w}^{(\rho)}$. The entries of $\mathbf{w}$ on $[n]$ are at most $W - 1$, those on $K^{(\rho)}$ sum to $2^\tau - 1$, and the remaining ones are 0. Hence, for every bit vector $\mathbf{b}$ of length n' ,

$$\langle \mathbf{w}, \mathbf{b} \rangle < nW + 2^\tau \leq 2^{\tau-\kappa} + 2^\tau \leq 2^{\tau+1} \leq q_\ell,$$

using $nW \leq 2^{\lceil \log(nW) \rceil} = 2^{\tau-\kappa}$ and the precondition. So [Step 3](#) does not wrap around modulo q_ℓ , and $\tilde{z}^{(\rho)} = \langle \mathbf{w}, \mathbf{b}^{(\ell)} \rangle$ holds over the integers. For $i < \ell$, reducing this identity modulo q_i and subtracting it from $z^{(i,\rho)}$ gives

$$z^{(i,\rho)} - (\tilde{z}^{(\rho)} \bmod q_i) \equiv \langle \mathbf{w}, \mathbf{b}^{(i)} - \mathbf{b}^{(\ell)} \rangle \pmod{q_i}. \quad (\text{B.5})$$

[Step 5](#) thus aborts if and only if the right-hand side of [Eq. \(B.5\)](#) is nonzero for some $i < \ell$.

Completeness. If $\mathbf{b}^{(1)} = \dots = \mathbf{b}^{(\ell)}$, then all right-hand sides of [Eq. \(B.5\)](#) are zero, so no iteration aborts and the protocol returns the prefixes.

Soundness. Suppose the prefixes are not all equal, and fix $i < \ell$ and $k_0 \in [n]$ with $b_{k_0}^{(i)} \neq b_{k_0}^{(\ell)}$. Both entries are bits, so $d := b_{k_0}^{(i)} - b_{k_0}^{(\ell)} \in \{-1, 1\}$ is a unit in $\mathbb{Z}_{q_i}$. For each ρ , the right-hand side of [Eq. \(B.5\)](#) is $r_{k_0}^{(\rho)}d$ plus a term that does not involve $r_{k_0}^{(\rho)}$. Since d is a unit and $W \leq q_{\min} \leq q_i$, where $W \leq q_{\min}$ follows from $R \geq \kappa / \log q_{\min}$, at most one $r \in \{0, \dots, W - 1\}$ makes rd equal to a given residue modulo q_i . As $r_{k_0}^{(\rho)}$ is uniform in $\{0, \dots, W - 1\}$ and independent of everything else, the comparison at q_i passes with probability at most $1/W$, independently for each ρ . All R comparisons at q_i pass with probability at most $W^{-R} \leq 2^{-\kappa}$, since $W \geq 2^{\kappa/R}$, so the protocol returns with at most this probability.

Privacy. The protocol takes no input from any party, so the adversary's view consists of the coefficients $\mathbf{r}^{(\rho)}$, the openings $\tilde{z}^{(\rho)}$ and $z^{(i,\rho)}$, and whether [Step 5](#) aborts. The simulator samples the $\mathbf{r}^{(\rho)}$ as $\mathcal{F}_{\text{Rand}}$ does. Let $D^{(i,\rho)}$ be the right-hand side of [Eq. \(B.5\)](#). The simulator computes it from the differences it is given, and from these values it computes whether the protocol aborts. It then samples $\tilde{z}^{(\rho)} \xleftarrow{\$} \{0, \dots, 2^\tau - 1\}$ and sets $z^{(i,\rho)} := (\tilde{z}^{(\rho)} + D^{(i,\rho)}) \bmod q_i$ for $i < \ell$, which is the same function of $\tilde{z}^{(\rho)}$ as in the real execution by [Eq. \(B.5\)](#). Since no two blocks $K^{(\rho)}$ share an index, the iterations use independent masks, so it suffices to bound the distance for one iteration and to multiply by the number R of iterations. Write the real opening as $\tilde{z}^{(\rho)} = V + M$ with $V := \sum_{k \in [n]} r_k^{(\rho)} b_k^{(\ell)}$ and $M := \sum_{c \in [\tau]} 2^c b_{k_c^{(\rho)}}^{(\ell)}$. By assumption, M is uniform in $\{0, \dots, 2^\tau - 1\}$ and independent of V . Since $V < nW \leq 2^{\tau-\kappa}$, the statistical distance between $V + M$ and M is $V/2^\tau < 2^{-\kappa}$ for every value of V , which is the claimed distance per iteration. $\square$

B.3 Putting It Together

Theorem B.6 (Security of Π_{maBits}). *Let $q_1, \dots, q_\ell \in M$ be prime powers of pairwise distinct primes satisfying the precondition of [Protocol B.3](#), let κ be a statistical security parameter, and write $\mathcal{F}' := \mathcal{F}_{\text{ABB}}^M \setminus \{\text{maBit}\}$. Consider the protocol that answers every batch of n **maBit** commands for $(q_1, \dots, q_\ell)$ by one execution of [Protocol B.1](#) and passes every other command on unchanged. Then this protocol statistically realizes $\mathcal{F}_{\text{ABB}}^M$ in the $(\mathcal{F}', \mathcal{F}_{\text{Rand}})$ -hybrid model with simulation error $\text{negl}(\kappa)$ per batch.*

Proof. Every command other than **maBit** is passed on unchanged, so we only consider one batch.

Correctness. For a modulus q_j that is not a 2-power, **CheckedInput** guarantees $\mathbf{x}_\sigma \in \{0, 1\}^{n'}$, and $u + v - 2uv = u \oplus v$ for bits u, v , so $\Pi_{\text{XorBits}}^{(q_j)}$ stores $\bigoplus_\sigma \mathbf{x}_\sigma$. For a 2-power q_j , the inputs are not checked, but no check is needed: fix one entry, write v_σ for the value that P_σ inputs in [Step 1](#), which is $x_\sigma + 2m_\sigma$ for an honest party, and write b and y for the entries of $\mathbf{b}$ and $\mathbf{y}$. The value opened in [Step 2](#) is $y = \sum_\sigma v_\sigma + b \bmod q_j$, and since q_j is even, $y \equiv \sum_\sigma v_\sigma + b \pmod{2}$, so [Step 3](#) returns $\llbracket \sum_\sigma v_\sigma \bmod 2 \rrbracket_{q_j}$. This is a bit whatever the corrupted parties input, and it is $\bigoplus_\sigma x_\sigma$ if all parties are honest. By [Theorem B.4](#), the ℓ returned vectors encode the same bit vector except with probability $2^{-\kappa}$. At least one honest party contributes a uniform $\mathbf{x}_\sigma$ over every modulus, so all n' bits are uniform and independent given the adversary's view, which is the hypothesis of that theorem. In particular, the returned bits are uniform and independent, as the **maBit** command requires.

Simulation. The bits never leave $\mathcal{F}'$, so the simulator only has to produce the opened values: the vectors $\mathbf{y}$ of [Step 2](#) of [Protocol B.2](#), the outcomes of the **CheckedInput** commands, and the openings of [Protocol B.3](#). Each entry of $\mathbf{y}$ is uniform in $\mathbb{Z}_{q_j}$, since an honest party's $\mathbf{m}_\sigma$ makes $2 \sum_\sigma \mathbf{m}_\sigma$ uniform on the even residues and $\mathbf{b}$ consists of uniform bits, so the simulator samples $\mathbf{y}$ uniformly. A **CheckedInput** command reveals only whether the input lies in $\{0, 1\}^{n'}$, which holds for an honest party and which the simulator knows for a corrupted one. For the consistency check, [Theorem B.4](#) needs the differences between the ℓ vectors, and the simulator has them: every honest party inputs the same $\mathbf{x}_\sigma$ over every modulus, so a difference can only come from a corrupted party, whose inputs the simulator sees. The simulation distance $2^{-\kappa}R$ and the soundness error $2^{-\kappa}$ of that theorem add up to the claimed $\text{negl}(\kappa)$. $\square$

Guaranteed Output Delivery. Over an ABB instantiation with GOD, [Protocol B.1](#) has to deliver output as well. A rejected **CheckedInput** command of [Protocol B.2](#) names the party, and as in [Section 4.2](#) the parties replace its input by $\mathbf{0}$, which leaves the XOR of the remaining vectors. An abort of the consistency check, however, says that some party was inconsistent across the moduli without saying which, since the check runs on $\llbracket \mathbf{b} \rrbracket_{q_j} = \bigoplus_\sigma \llbracket \mathbf{x}_\sigma \rrbracket_{q_j}$. There are two ways around this.

- Swap the last two steps of [Protocol B.1](#): the parties first run the check on $\llbracket \mathbf{x}_\sigma \rrbracket_{q_1}, \dots, \llbracket \mathbf{x}_\sigma \rrbracket_{q_\ell}$ for each σ and XOR the surviving vectors afterwards, so a rejection names a party. The N checks can use the same sacrificed positions, since each masks its openings with the block bits of its own party, so n' is unchanged and the extra cost consists of the openings of the $N - 1$ further checks. The XOR only has to be computed on the prefixes, which saves $(N - 1)(n' - n)$ multiplications per modulus that is not a 2-power. Over a 2-power modulus, [Step 1](#) of [Protocol B.2](#) has to be split, each party inputting $\mathbf{x}_\sigma$ through **CheckedInput** and $\mathbf{m}_\sigma$ through **Input**, since the check runs on $\llbracket \mathbf{x}_\sigma \rrbracket_{q_j}$ and requires it to be a bit vector.
- Keep the order and identify the cheater afterwards: a failed batch is discarded anyway, so the parties open all $\llbracket \mathbf{x}_\sigma \rrbracket_{q_j}$. By completeness of [Theorem B.4](#), at least one party turns out to be inconsistent across the moduli, and the batch is regenerated without that party.

Each failure removes a corrupted party, so with an honest majority at most $\lceil N/2 \rceil - 1$ failures occur, and no extra cost arises when nobody cheats.

C Security Proofs for the Conversion Protocols

Here, we prove the security theorems of our three conversion protocols, restating each of them for convenience. Each of the protocols passes every command other than **ModConv** on to $\mathcal{F}_{\text{ABB}}^M$ unchanged, and the simulator forwards these commands to $\mathcal{F}_{\text{ABB}+\text{ModConv}}^M$, so the proofs only consider the **ModConv** command.

Theorem 4.7 (Security of $\Pi_{\text{ModConv}1}$). *Let $Q \in M$ and let κ be a statistical security parameter. Consider the protocol that answers each command $(\text{ModConv}, q, t, L, \text{id}, \text{id}')$ by running $\Pi_{\text{ModConv}1}^{q,t,\beta}$ (Protocol 4.2) for some polynomially bounded β with $2 \leq \beta \leq q$ such that $L - 1 \leq q - \lceil q/\beta \rceil$, with its preprocessing phase run by Protocol 4.6 with auxiliary modulus Q , and passes every other command on unchanged. Provided that every **ModConv** command admits such a β , this protocol statistically realizes $\mathcal{F}_{\text{ABB}+\text{ModConv}}^M$ in the $\mathcal{F}_{\text{ABB}}^M$ -hybrid model with simulation error at most $2^{-\kappa}$ per **ModConv** command.*

Proof. Correctness. We first show that Protocol 4.6 stores a uniform mask $m \in \mathbb{Z}_q$ together with the corrections $u_i = m - c_i q \bmod t$ of Protocol 4.2, unless it aborts. The bits of Step 1 are uniform and independent, so $\text{idx} := \sum_{j \in [n_{\text{idx}}]} 2^j \text{idx}_j$ is uniform in $[2^{n_{\text{idx}}}]$, m_{off} is uniform in $[2^{n_{\text{off}}}]$, and the two are independent. By induction on j , the vector $\delta^{(j)}$ of Step 2 is the one-hot vector of $\text{idx} \bmod 2^j$: the recursion moves the single 1 of $\delta^{(j)}$ from position $i = \text{idx} \bmod 2^j$ to position $i + 2^j \text{idx}_j = \text{idx} \bmod 2^{j+1}$. Hence δ is the one-hot vector of idx , so Step 3 yields $\text{hot} = 1\{\text{idx} < \beta\}$, and $\text{short} = \text{short}_{\text{idx}}$ if $\text{idx} < \beta$ while $\text{short} = 0$ otherwise. By Theorem 3.5, Step 4 computes $\text{lt} = 1\{m_{\text{off}} < T\}$ and $\text{eq} = 1\{m_{\text{off}} = T\}$. Therefore $\text{acc} = 0$ if $\text{idx} \geq \beta$, and otherwise $\text{acc} = \text{lt} + (1 - \text{short}_{\text{idx}}) \cdot \text{eq} = 1\{m_{\text{off}} \leq T - \text{short}_{\text{idx}}\}$ by Lemma 4.5. Since $T - \text{short}_{\text{idx}} = |I_{\text{idx}}| - 1$, an iteration is accepted if and only if $\text{idx} < \beta$ and $m_{\text{off}} < |I_{\text{idx}}|$. If q/β is a 2-power, all intervals have length $2^{n_{\text{off}}}$, so the second condition always holds and $\text{acc} = \text{hot}$ as the protocol sets it; if moreover β is a 2-power, then $2^{n_{\text{idx}}} = \beta$ and every iteration is accepted.

In an accepted iteration, $m = m_{\text{off}} + \min(I_{\text{idx}})$ lies in I_{idx} , so $\alpha(m) = \text{idx}$ and $\sum_{j>i} \delta_j = 1\{\alpha(m) > i\} = c_i$. Hence Steps 3 and 6 store $\llbracket m \rrbracket_q$ and $\llbracket u_i \rrbracket_t$ with $u_i = m - c_i q \bmod t$, as in Protocol 4.2. Conversely, every $m \in \mathbb{Z}_q$ arises from exactly one accepted pair $(\text{idx}, m_{\text{off}})$, namely $\text{idx} = \alpha(m)$ and $m_{\text{off}} = m - \min(I(m))$. Since every pair has probability $2^{-(n_{\text{idx}}+n_{\text{off}})}$, the accepted m is uniform in $\mathbb{Z}_q$, and an iteration is accepted with probability

$$\theta = \frac{q}{2^{n_{\text{idx}}+n_{\text{off}}}} > \frac{q}{2\beta \cdot 2q/\beta} = \frac{1}{4},$$

using $2^{n_{\text{idx}}} < 2\beta$ and $2^{n_{\text{off}}} < 2q/\beta$.

Now let $x < L$. Treating x , m and y as integers in $[0..q)$ and putting $c := 1\{x + m \geq q\}$, the opened value is $y = x + m - cq$. The protocol therefore outputs

$$y - u_{\alpha(y)} \equiv y - (m - c_{\alpha(y)}q) \equiv x + (c_{\alpha(y)} - c)q \pmod{t},$$

and it remains to show $c_{\alpha(y)} = c$, i.e., $1\{\alpha(m) > \alpha(y)\} = c$. If $c = 0$, then $y = x + m \geq m$, hence $\alpha(y) \geq \alpha(m)$ because α is non-decreasing. If $c = 1$, then $y = m - (q - x)$ with $q - x \geq \lceil q/\beta \rceil \geq q/\beta$ since $x \leq L - 1 \leq q - \lceil q/\beta \rceil$, hence

$$\alpha(y) = \left\lfloor \frac{\beta m}{q} - \frac{\beta(q-x)}{q} \right\rfloor \leq \left\lfloor \frac{\beta m}{q} - 1 \right\rfloor = \alpha(m) - 1.$$

Simulation. The simulator emulates $\mathcal{F}_{\text{ABB}}^M$ towards the adversary. In the preprocessing, no step takes an input from a party, so a corrupted party can only abort, and the only values opened are the bits **acc** of Step 5, one per iteration, since Protocol 3.4 opens nothing. Since every iteration draws fresh maBits, the opened values **acc** are independent across iterations, and each equals 1 with the probability θ computed above, which does not depend on any secret. Moreover, the accepted m is uniform in $\mathbb{Z}_q$ regardless of the number of iterations. The simulator therefore samples the number ℓ of iterations from the geometric distribution with parameter θ and sends $\ell - 1$ zeros followed by a 1 to the corrupted parties, or 3κ zeros if $\ell > 3\kappa$.

In the online phase, if $x \geq L$, the functionality sends x to the simulator, which runs the online phase on x exactly as the honest parties would and returns the result to the functionality. If $x < L$, the only value opened is y , since the other step is a **Comb** command. Since m is uniform in $\mathbb{Z}_q$ and independent of x , so is y , and the simulator samples $y^* \xleftarrow{\$} \mathbb{Z}_q$ and sends it to the corrupted parties in place of y . The value stored in the ideal execution is $x \bmod t$, which by correctness is also the value the protocol stores. The real and ideal executions thus differ only if the preprocessing aborts, which happens with probability $(1 - \theta)^{3\kappa} < (3/4)^{3\kappa} < 2^{-\kappa}$. $\square$

Theorem 4.13 (Security of Π_{ModConv2}). *Let $Q \in M$ and let κ be a statistical security parameter. Consider the protocol that answers each command $(\text{ModConv}, q, t, L, \text{id}, \text{id}')$ by running $\Pi_{\text{ModConv2}}^{q, t, \beta}$ (Protocol 4.8) for some polynomially bounded β with $N + 1 \leq \beta \leq q$ such that $L - 1 \leq q - \lceil Nq/\beta \rceil$, with its preprocessing phase run by Protocol 4.11 with auxiliary modulus Q , and passes every other command on unchanged. Provided that every **ModConv** command admits such a β , this protocol statistically realizes $\mathcal{F}_{\text{ABB}+\text{ModConv}}^M$ in the $(\mathcal{F}_{\text{ABB}}^M, \mathcal{F}_{\text{Rand}})$ -hybrid model with simulation error $\text{negl}(\kappa)$ per **ModConv** command.*

Proof. Correctness. Fix a party P_σ . Steps 2 and 3 guarantee $\mathbf{v}^\sigma \in \{0, 1\}^{n'}$, and Step 4 guarantees that the three sharings of $\mathbf{v}^\sigma$ encode the same vector, except with probability $2^{-\kappa}$ by Theorem B.4. Since the δ_i^σ are bits and $\beta \leq q$, their sum is congruent to 1 modulo q only if it equals 1, so Step 5 passes if and only if δ^σ is the one-hot vector of some index i_σ . Then $\text{short} = \text{short}_{i_\sigma} \in \{0, 1\}$, and Lemma 4.12 turns Step 3 into the condition $m_{\text{off}}^\sigma \leq T - \text{short}_{i_\sigma} = |I_{i_\sigma}| - 1$. The mask $m^\sigma = m_{\text{off}}^\sigma + \min(I_{i_\sigma})$ of Step 6 therefore lies in I_{i_σ} , so $\alpha(m^\sigma) = i_\sigma$ and $s = \sum_\sigma i_\sigma$. For an honest P_σ , the mask is uniform in $\mathbb{Z}_q$, and for a corrupted P_σ it is an element of $\mathbb{Z}_q$ of the adversary's choice.

Let $s_\sigma := i_1 + \dots + i_\sigma$. By induction on σ , the vector Δ^σ of Step 7 is the one-hot vector of $s_\sigma \bmod \beta$, and $\chi^2 + \dots + \chi^\sigma = \lfloor s_\sigma / \beta \rfloor$: of the β^2 products formed for $\sigma \geq 2$, only the one with $j' = s_{\sigma-1} \bmod \beta$ and $i = i_\sigma$ is nonzero. It contributes to Δ_j^σ for $j \equiv_\beta s_\sigma$, and to χ^σ if and only if $j' + i \geq \beta$, i.e., if and only if $\lfloor s_\sigma / \beta \rfloor = \lfloor s_{\sigma-1} / \beta \rfloor + 1$. Hence $\Delta_j = 1\{j = s \bmod \beta\}$ and $w = \lfloor s / \beta \rfloor$, so Step 10 computes $c_i = \lfloor s / \beta \rfloor + 1\{s \bmod \beta > i\} = \lceil (s - i) / \beta \rceil$ and $u_i = m - c_i q \bmod t$, as in Protocol 4.8. Here we used that Step 8 moves each bit z unchanged from Q to t : the opened value is $d = z \oplus r$, and $d + (1 - 2d)r = d \oplus r = z$.

Now let $x < L$. Since $x \leq L - 1 \leq q - \lceil Nq/\beta \rceil$, Lemma 4.10 applies to x . With C as in the lemma, the opened value is $y = x + m - Cq$, so as in the proof of Theorem 4.7 the protocol outputs $x + (c_{\alpha(y)} - C)q \bmod t$. By the lemma, $c_{\alpha(y)} = \lceil (s - \alpha(y)) / \beta \rceil = C$. The lemma only requires $m^1, \dots, m^N \in \mathbb{Z}_q$, so correctness holds for every choice of the corrupted parties' masks.

Simulation. The simulator emulates $\mathcal{F}_{\text{ABB}}^M$ towards the adversary. In particular, it sees the vector $\mathbf{v}^\sigma$ that a corrupted P_σ inputs. The **CheckedInput** commands of Steps 2 and 3 reveal only whether the input lies in the language, which the simulator knows. The check of Step 4 is simulated by Theorem B.4: for an honest P_σ the three sharings agree and the sacrificed bits are uniform, and for a corrupted P_σ the simulator knows all its inputs. The value opened in Step 5 is 1 for an honest P_σ and known to the simulator for a corrupted one.

Each value d opened in Step 8 is masked by a fresh maBit r , which is uniform and independent of everything else, so d is a uniform bit and the simulator samples it as such. The remaining steps of the preprocessing consist of **Comb** and **Mul** commands and reveal nothing.

In the online phase, at least one party is honest, and its mask is uniform in $\mathbb{Z}_q$ and independent of the masks that the adversary supplies, so $y = x + m \bmod q$ is uniform in $\mathbb{Z}_q$ and independent of x , and the simulator proceeds as in the proof of Theorem 4.7. Summing the errors of Theorem B.4 over the N parties gives the claimed bound. $\square$

Theorem 4.17 (Security of Π_{ModConv3}). *Let $Q \in M$ and let κ be a statistical security parameter. Consider the protocol that answers each command $(\text{ModConv}, q, t, L, \text{id}, \text{id}')$ by running $\Pi_{\text{ModConv3}}^{q,t}$ (Protocol 4.14), with its preprocessing phase run by Protocol 4.16 with auxiliary modulus Q , and passes every other command on unchanged. If the symmetric function protocol in Step 2 of $\Pi_{\text{PrepCompare}}$ (Protocol 7.4) is statistically secure with simulation error at most ϵ , then this protocol statistically realizes $\mathcal{F}_{\text{ABB}+\text{ModConv}}^M$ in the $\mathcal{F}_{\text{ABB}}^M$ -hybrid model with simulation error at most $2^{-\kappa} + \lceil \log q \rceil \cdot \epsilon$ per **ModConv** command.*

Proof. Correctness. The bits of Step 1 are uniform and independent, so m is uniform in $[2^n]$. If q is a 2-power, then $m < 2^n = q$ always, and every iteration is accepted. Otherwise $q < 2^n$, so q has n bits, and by Theorem 3.5 the value opened in Step 3b is $\text{lt} = 1\{m < q\}$. In both cases, an iteration is accepted if and only if $m \in \mathbb{Z}_q$, hence with probability $\theta := q/2^n > 1/2$, and the accepted m is uniform in $\mathbb{Z}_q$. Since the bits $\llbracket b_i \rrbracket_t$ are those of m , the stored data is that of the preprocessing phase of Protocol 4.14.

In the online phase, treating x and m as integers in $[0..q)$ and putting $c := 1\{x + m \geq q\}$, the opened value is $y = x + m - cq$, and $c = 1\{y < m\}$: if $c = 1$, then $y - m = x - q < 0$, and if $c = 0$, then $y = x + m \geq m$. Since $y < q \leq 2^n$, the threshold y of Step 2 has n bits, so by Theorem 7.5 the step computes $1 - \text{lt} - \text{eq} = 1 - 1\{m < y\} - 1\{m = y\} = 1\{y < m\} = c$. Step 3 then outputs $y - m + qc = x$ modulo t . The protocol is thus correct for every $x \in \mathbb{Z}_q$, whatever the bound L is.

Simulation. The simulator emulates $\mathcal{F}_{\text{ABB}}^M$ towards the adversary. The preprocessing is simulated as in the proof of Theorem 4.7 with θ in place of the acceptance probability there and κ in place of 3κ iterations.

The online phase opens y and the values opened by the symmetric function protocol, since $\Pi_{\text{PrepCompare}}$ (Protocol 7.4) opens nothing else. The value y is uniform in $\mathbb{Z}_q$ and independent of x because m is. As in the proof of Theorem 4.7, the simulator sends $y^* \xleftarrow{\$} \mathbb{Z}_q$ to the corrupted parties in place of y . It does the same for $x \geq L$, where it additionally returns $x \bmod t$ to the functionality, since the protocol stores $x \bmod t$ in this case as well. It simulates the $n - g$ runs of the symmetric function protocol, in the preprocessing and the online phase alike, by the statistical security of that protocol, with simulation error at most ϵ each. The real and ideal executions thus differ with probability at most $(1 - \theta)^\kappa + (n - g)\epsilon < 2^{-\kappa} + n\epsilon$. $\square$