

Edabits Generation with Quasi-Linear Communication Complexity

Alexander Bienstock¹, Daniel Escudero², Qijia Fan³, Fengrun Liu⁴, and Yifan Song^{3,5}

¹ JPMorgan AI Research & AlgoCRYPT CoE

² TACEO

³ Tsinghua University

⁴ Carnegie Mellon University

⁵ Shanghai Qi Zhi Institute

Abstract. The work of (Escudero et al., CRYPTO 2020) introduces the so-called *edabits*, which are defined as a secret sharing of an integer over an arithmetic domain, together with its bit decomposition secret shared over a binary domain. In the same work, it is shown how edabits can be used to securely evaluate advanced primitives such as comparison, bit decomposition, and truncation, which are essential building blocks in more complex applications such as privacy-preserving machine learning and beyond. Edabits are general and can be defined over any secret-sharing scheme or corruption model. Such flexibility, coupled with their wide range of applications, makes edabits a powerful target correlation.

Escudero et al. introduced a protocol to generate edabits that works in the arithmetic black-box (ABB) model, that is, it is agnostic to the concrete secret-sharing scheme and even works both for fields $\mathbb{F}_p$ and rings $\mathbb{Z}_{2^\ell}$, as well as both for honest majority $t < n/2$ and dishonest majority $t < n$. Unfortunately, the communication complexity of their approach grows as $\Omega(n^2\ell)$ bits with ℓ being the bit-length of an element in the arithmetic domain. This becomes a bottleneck in larger protocols since most MPC primitives such as multiplications or openings can be done with linear communication of $O(n\ell)$ bits.

In this work we present a novel approach for generating edabits in the ABB model that achieves *quasi-linear* communication complexity of $O(n\ell \log n)$ bits, for the setting where $t < n(1 - \epsilon)$ for a constant $\epsilon > 0$. This includes the honest majority case $t < n/2$, but also dishonest majority with an (arbitrarily small) constant fraction of honest parties. Our result is achieved via novel applications of randomness extraction methods, as well as better edabit aggregation for the binary components. Along the way and of independent interest, we improve the communication complexity of *dabit* generation (dabits are bits shared in both binary and arithmetic domains) from $\Omega(n^2\ell)$ bits achieved in (Rotaru and Wood, INDOCRYPT 2019), to linear $O(n\ell)$ bits.

1 Introduction

Secure multiparty computation (MPC) enables a set of computing parties $P_1, \dots, P_n$ to securely compute an arbitrary function on private inputs without leaking anything beyond the output of the function itself. Different protocols exist depending on the power of the adversary, who corrupts t parties: generally, the larger t , the harder it is to ensure security. Also, passive adversaries follow the protocol while active ones may deviate arbitrarily, the latter being harder to defend against.

Often, computation is modeled as an arithmetic circuit (essentially a series of additions and multiplications) either over a field $\mathbb{F}_p$, or a ring $\mathbb{Z}_{2^\ell}$. Supporting such arithmetic operations is already very useful: in theory circuits over $\mathbb{F}_p$ represent all functions, and in practice a good range of applications is already nicely represented by additions and multiplications, including basic statistics like averages or variances. However, an application often cannot be written solely in terms of additions and multiplications over a single arithmetic domain, and more advanced primitives for evaluating complex building blocks become a pressing need. For instance, in privacy-preserving machine learning (PPML), where the private inputs are data and the output is a model trained on them, one must design MPC subprotocols for non-linear layers, real-valued arithmetic, exponentiations, etc. Most of these blocks reduce to a few “core primitives”, such as secure truncation, secure comparison, or bit decomposition.

MPC subprotocols for these tasks can be tailored to specific aspects such as corruption threshold, passive vs active security, rings vs fields, or the number of parties, among others. However, there are sizable benefits to solutions that work for several combinations of these parameters with little-to-no change, assuming as little as possible from the underlying MPC protocol. For instance, it

is common to assume only that the MPC protocol handles additions and multiplications over different domains (separately), and to construct from these basic operations much more advanced primitives.

This approach, known as the arithmetic black-box (ABB) model, is taken in [21], where the authors introduce an object known as an *edabit*, which can be defined over any secret-sharing scheme or security model. These objects can be used to efficiently evaluate these primitives in the ABB model. They also design a protocol for generating edabit correlations, itself generic enough to work in the ABB model and hence to adapt to any adversarial setting and corruption threshold, over rings or fields.

Unfortunately, the edabit generation of [21] has *quadratic* communication complexity $\Omega(n^2\ell)$ bits, with ℓ the bit-length of an element of the arithmetic domain. This is a factor n larger than the typical cost of simpler MPC tasks, such as evaluating plain arithmetic circuits, which needs only *linear* communication $O(n\ell)$ bits per operation. Several works (cf. [26,6]) have since built on edabits to enable advanced primitives, but all inherit this quadratic overhead. This motivates the following research question:

Can we obtain edabit correlations in the ABB model, with linear communication complexity?

1.1 Our Contribution

We make solid progress towards answering this question by presenting a protocol for edabit generation, set in the ABB model—both for fields and rings—that generates edabits with communication complexity $O(n\ell \log n)$ bits. It scales well with the number of parties and adapts easily to several corruption models, and it gets the claimed benefits when $t < n(1 - \epsilon)$ for an arbitrarily small constant $\epsilon > 0$, which includes the honest majority case $t < n/2$ but also dishonest majority with an arbitrarily small constant fraction of honest parties.

Along the way, we present additional contributions of independent interest:

- We introduce a variant of super-invertible matrices, which we call *block super-invertible*. We show how to construct one whose entries are all 0 or 1, and discuss applications to randomness extraction.
- We show how to aggregate edabits (that is, sum both the arithmetic and the binary parts) more efficiently. Our aggregator needs the binary components shared over a slightly larger domain than $\mathbb{Z}_2$, and we give a protocol for generating and verifying private edabits of this form.
- We introduce a novel method to generate *debabits*, a correlation useful in several applications (including edabit generation itself). We reduce the communication complexity of their generation from $\Omega(n^2\ell)$ bits in [33], to $O(n\ell)$ bits.
- We identify a weakness in [21]: their edabits cannot produce integers random over non-power-of-two ranges, which limits the applications. We show how to generalize edabits to this case.

1.2 Related Work

The task of designing subprotocols for advanced primitives such as secure comparison, truncation, or bit decomposition has been widely studied in the literature. Earlier works such as [14,13] studied this for Shamir secret-sharing, although the core ideas apply to other settings as well. The work of [21] introduces edabits precisely to capture the generality of earlier techniques, defining a concrete correlation flexible enough for many applications. Modern techniques such as [28] address a large number of parties, but introduce other limitations: larger communication complexity (quadratic in the bit-length of an arithmetic element), a dependency on structured primes (Mersenne primes), or lack of active security. In contrast, edabits remain the most efficient and flexible way to construct these primitives, supporting active security and other settings.

For a small number of parties, several ad-hoc protocols can be designed (cf. [30] and the references therein for 2 parties, and [35] and the references therein for 3 parties). However, these techniques are constrained to few parties (in most cases they are tailored to *replicated secret-sharing*, which scales exponentially with n). Furthermore, active security is sometimes out of reach, such as for constructions using function secret-sharing (cf. [24]).

1.3 Technical Overview

We give a high-level overview of our constructions. We write L for the size of the arithmetic domain $\mathbb{Z}_L$ and $\ell := \lceil \log L \rceil$ for the bit-length of one of its elements; L is either a prime (the field case) or a power of 2 (the ring case). For $T \geq 2$, $\llbracket \cdot \rrbracket_T$ denotes a linear secret sharing over $\mathbb{Z}_T$.

An *edabit* of length m such that $2^m \leq L$, first proposed in [21], is a sharing of a random integer $r \in [0, 2^m)$ over an arithmetic domain $\llbracket r \rrbracket_L$, together with its bit decomposition $(r_0, \dots, r_{m-1})$ shared over a binary domain $\llbracket r_0 \rrbracket_2, \dots, \llbracket r_{m-1} \rrbracket_2$, where $r = \sum_{i=0}^{m-1} r_i 2^i$. As for random shares, a natural idea is that each party samples a random value, shares it in both domains, and then the parties aggregate these shared values to obtain a shared edabit. In [21], an edabit with its secret value known to a single party is called a *private edabit*. Once each party has shared a private edabit, the arithmetic parts can be aggregated locally, by linearity of the secret sharing scheme. The binary parts cannot, since the homomorphism there is over $\mathbb{Z}_2$ rather than $\mathbb{Z}_L$.

The original aggregation method of [21] uses $(n-1)$ binary (integer) adder circuits to add these bitwise-shared values. Afterwards, they use random *dabits* [33], random bits shared over both domains $(\llbracket b \rrbracket_L, \llbracket b \rrbracket_2)$, to remove the overflow bits beyond the m -th from the arithmetic part. This costs $O(nm)$ AND gates over $\mathbb{Z}_2$ and $O(\log n)$ dabits, that is $O(n^2m)$ bits plus the dabits, discussed later.

Our global edabit generation builds on two techniques: (1) a new protocol, the *aggregator*, to aggregate private edabits efficiently, and (2) a randomness extraction protocol that reduces the number of private edabits needed, amortized, per shared edabit when the corruption threshold is sub-optimal, i.e., $t = (1 - \epsilon)n$ for a constant ϵ (which includes optimal honest majority). The dependencies among our constructions are summarized in Figure 1.

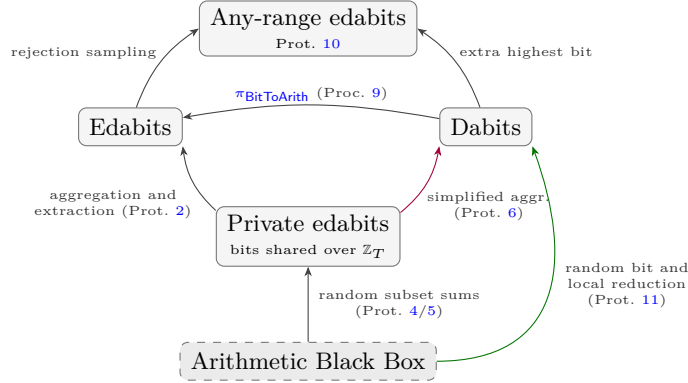


Fig. 1: How the correlations of this paper depend on each other, with the protocol realizing each step. Black arrows apply to both arithmetic domains; the red one is used only over $\mathbb{F}_p$, and the green one only over $\mathbb{Z}_{2^\ell}$.

Using Aggregators Instead of Binary Adders. The original method computes carry bits in the binary adder when adding two private edabits. Our idea is to *delay* that computation: we first locally add up all the bitwise-shared values over a small ring $\mathbb{Z}_T$ with $T = 2^\eta$ and $\eta = O(\log n)$, and then deal with the overflow in each bit position via a secure bit-decomposition procedure. In the end, the overflow bits beyond the m -th bit are removed via dabits following the approach in [21].

To do this, we change the sharing domain of the binary part of private edabits from $\mathbb{Z}_2$ to $\mathbb{Z}_T$, so that these bits can be summed locally with no overflow. This does not affect the final format of edabits, since parties can locally convert shares of a bit over $\mathbb{Z}_T$ to shares of the same bit over $\mathbb{Z}_2$ by reducing modulo 2. This basic implementation costs $O(m\eta)$ multiplication gates over $\mathbb{Z}_T$, that is $O(n \log^2 n \cdot m)$ bits, together with the same $O(\log n)$ dabits as [21]. This is already a significant improvement when the number of parties n is large. In Section 3 we further improve $\pi_{\text{AggEdabit}}$ with a carry-by-block trick, which reduces the number of multiplication gates over $\mathbb{Z}_T$ from $O(m\eta)$ to $O(m)$: we group the m bits into m/A blocks of size $A = O(\log n)$, add them up locally, and bit-decompose the sum one block at a time. Furthermore, the output bits only need to be shared over $\mathbb{Z}_2$, so we adjust the bit-decomposition to improve its concrete efficiency: after opening the

masked block sum, the parties share its bits directly over $\mathbb{Z}_2$ and use a binary adder to remove the mask, whose bit sharings are locally converted to $\mathbb{Z}_2$. This reduces the cost of the adder, which now operates over $\mathbb{Z}_2$ rather than $\mathbb{Z}_T$. Our final protocol requires $O(n \log n \cdot m)$ bits together with $O(\log n)$ dabits.

While our approach reduces the cost of aggregating n private edabits, one per party, it only works when their binary part is shared over the larger domain $\mathbb{Z}_T$. Directly using our approach would lead to a larger communication complexity of $O(n^2(\ell + m \log n))$ bits when all parties share their private edabits, compared with $O(n^2(\ell + m))$ bits in [21]. When $t = (1 - \epsilon)n$ for a constant ϵ , a constant fraction of the private edabits come from honest parties and are random and unknown to the adversary. One may then hope to obtain $\Omega(n)$ random edabits rather than one, amortizing the cost of sharing each private edabit.

Randomness Extraction for Edabits. We start from the idea of [19], which uses an $(n - t) \times n$ Vandermonde matrix to extract $n - t$ shared random values from n private ones over a field. In our case, however, we need to extract randomness over $\mathbb{Z}_{2^m}$, where such matrices are not super-invertible.

Our solution is to let each party provide d inputs and use an $(n - t)d \times nd$ matrix over $\mathbb{Z}_{2^m}$, whose columns are divided into n blocks of d consecutive columns, one for each party. We require that any $n - t$ complete blocks form an invertible square sub-matrix, a property we call *block super-invertibility*. This is weaker than super-invertibility, but suffices for randomness extraction: the uniform inputs of any $n - t$ honest parties ensure that all $(n - t)d$ outputs are uniformly random. Inspired by [12], we construct such a matrix with $d = \lceil \log n \rceil$ in Section 4.1. Letting each party generate d private edabits, we extract $(n - t)d$ edabits from nd private ones, so each output costs only $n/(n - t)$ private edabits on average, which is $O(1)$ when $t = (1 - \epsilon)n$ for a constant ϵ .

One issue we omitted is that our aggregator can only efficiently *add* private edabits, whereas a block super-invertible matrix requires multiplying each by a constant in $\mathbb{Z}_{2^m}$ before adding. To address this, our construction in addition makes every entry of the matrix 0 or 1, so that randomness extraction reduces to adding up private edabits. In summary, our final protocol manages to prepare random edabits with amortized communication $O(n(\ell + m \log n))$ bits (together with the cost of preparing $O(\log n)$ dabits).

Remark 1. One may think [21] could also benefit from randomness extraction. Unfortunately not: while it reduces the number of private edabits per random edabit, each now aggregates $O(nd) = O(n \log n)$ of them, costing $O(n^2 \log^2 n \cdot m)$ bits (plus $O(\log n)$ dabits), which is worse than their original approach.

Generating Partially Arithmetic Private Edabits. We combine random subset sums with our aggregator to generate and verify the partially arithmetic private edabits that our edabit generation requires. The dealer inputs the arithmetic values and their bits directly over $\mathbb{Z}_L$ and $\mathbb{Z}_T$, respectively, and the parties verify their consistency using random subset sums. The basic idea is to select each private edabit independently with probability $1/2$ and compare the sums of the selected arithmetic and binary components. Since the coefficients are only 0 or 1, these sums can be computed using aggregation, without multiplying the binary components by general coefficients. If any input is inconsistent, the check fails with probability at least $1/2$: fixing all other choices, at most one of the two choices for an inconsistent input makes it pass. Repeating independent checks reduces the error further. To avoid revealing the private values, we mask each subset sum before opening it.

Regarding the aggregation, the number of inputs to a subset sum may exceed T , so they cannot all be aggregated locally over $\mathbb{Z}_T$ at once. We therefore aggregate in smaller groups and then aggregate the intermediate results, level by level in a tree. Since all inputs come from the same dealer, we improve this tree aggregation by letting the dealer provide bit decompositions of the intermediate sums, which the parties verify. In Section 5, we also show how to reduce the number of random coins needed for the checks by deriving the subset choices from powers of a single random field element.

Bootstrapping the Generation of Dabits. Dabits play an essential role here: in both our construction and [21], $O(\log n)$ of them are needed to remove the overflow bits after aggregating private edabits.

In the ring case (i.e., $L = 2^\ell$), a random dabit is easy to generate: the parties generate a random bit $\llbracket b \rrbracket_L$ over the arithmetic domain and compute their shares of $\llbracket b \rrbracket_2$ by reducing modulo 2, at a cost of $O(n\ell)$ bits per dabit. In the field case, however, the approach of [33] incurs $\Omega(n^2\ell)$ bits per dabit, significantly degrading edabit generation.

We introduce a more direct method for the field setting. The key idea is to decompose an edabit of length m into m dabits, where m is strictly smaller than the bit length ℓ of the prime. The decomposition is done over the arithmetic domain and costs $O(\ell)$ multiplications over $\mathbb{F}_p$, that is $O(n\ell^2)$ bits. A challenge, however, is that our edabit generation already relies on dabits, creating a circular dependency. To break it, we employ a simplified aggregation: instead of handling carry bits, we aggregate n private edabits of length $m = \ell - 1 - \log nd$, so that the arithmetic part does not overflow, and directly decompose its lowest m bits, avoiding dabits at this stage.

Together with our aggregation and randomness extraction, this achieves $O(n\ell)$ bits of communication per random dabit when $t = (1 - \epsilon)n$ for a constant ϵ (which includes honest majority).⁶ Since each random edabit requires $O(\log n)$ dabits, the total communication per edabit is $O(n \log n \cdot \ell)$ bits.

Remark 2. Unlike our edabit generation, our dabits beat [33] even when $t = n - 1$: the cost per dabit is then $O(n^2 \log n + n\ell)$ bits against their $O(n^2\ell)$.

Any-Range Edabits. Finally, we combine our edabits and dabits to generate edabits whose value is uniform in any range $[0, P)$ with $2 \leq P \leq L$. With $\ell_P = \lceil \log P \rceil$, we append a random dabit as the most significant bit of an edabit of length $\ell_P - 1$ and perform rejection sampling with a binary comparison against P . Taking $P = L$ gives full-range edabits over both rings and fields.

1.4 Outline of the Document

We present in Section 2 the preliminaries, including the necessary functionalities and existing protocols. In Section 3, we show how to aggregate edabits more efficiently, which is one of the two main techniques that enable our quasi-linear communication complexity. In Section 4, we introduce the second technique, randomness extraction, and combine it with our aggregator to generate global edabits from private edabits. We also extend this construction to any-range edabits. Section 5 then presents our method for generating the partially arithmetic private edabits required by our aggregator. Next, in Section 6, we address the challenge of generating dabits with linear communication complexity. Finally, in Section 7, we compare the efficiency of our protocols with prior work.

2 Preliminaries

Let n be the number of parties and t the maximum number of corrupted ones, where $t < n(1 - \epsilon)$ with $\epsilon > 0$ an arbitrary constant, and let κ be the security parameter. Unless otherwise specified, $\log(\cdot)$ denotes the base-2 logarithm. We consider two types of algebraic structures for arithmetic in the MPC: the arithmetic domain $\mathbb{Z}_L$, where L is either a prime p or a power of two. Let $\ell = \lceil \log L \rceil$ denote the bit length of L . In both cases we also use a smaller domain $\mathbb{Z}_T$, where $T = 2^\eta = \Theta(n^2 \log^2 n)$ is a power of two and η is the bit length of T . The concrete choice of η will be specified in Section 4.2. Finally, we also use the binary domain $\mathbb{Z}_2$ in both cases.

Our protocols also use Galois rings. Fix a monic polynomial $h_\kappa(X)$ of degree κ with coefficients in $\{0, 1\}$ whose reduction modulo 2 is irreducible over $\mathbb{Z}_2$. For any $e \geq 1$, define $\text{GR}(2^e, \kappa) := \mathbb{Z}_{2^e}[X]/(h_\kappa(X))$. In particular, $\text{GR}(2, \kappa)$ is the Galois field $\text{GF}(2^\kappa)$. Every $\alpha \in \text{GR}(2^e, \kappa)$ can be written uniquely as

$$\alpha \equiv \sum_{i=0}^{\kappa-1} a_i X^i \pmod{h_\kappa(X)}, \quad a_i \in \mathbb{Z}_{2^e},$$

and we identify α with its coefficient vector $(a_0, \dots, a_{\kappa-1}) \in \mathbb{Z}_{2^e}^\kappa$. Under this representation, addition and multiplication over $\text{GR}(2^e, \kappa)$ are both linear combinations of the coefficients of the two operands, and thus can be implemented using additions and multiplications over the basic domain $\mathbb{Z}_{2^e}$ in the MPC.

A full table of parameters is provided in Appendix A.

⁶ For the field case, this bound also requires $m = \Omega(\ell)$, or equivalently, $\ell - \log(nd) = \Theta(\ell)$. This is a reasonable parameter assumption in practice, where typically $n \ll L$.

2.1 Arithmetic Black Box

As in [21], we model MPC via an Arithmetic Black Box functionality, $\mathcal{F}_{\text{ABB}}$, in the Universal Composability (UC) framework [11]. New to our work, it allows n parties to input private values over three moduli instead of two: $\mathbb{Z}_L, \mathbb{Z}_T, \mathbb{Z}_2$. It is parameterized by L , with T fixed globally as above, and supports addition and multiplication within each domain, opening, generation, public randomness, and abort. We also use a restricted input command, **InputBit**, which lets a party input a value required to be a bit and stores it in the smaller arithmetic domain $\mathbb{Z}_T$. It can also generate and store a random bit in either $\mathbb{Z}_L$ or $\mathbb{Z}_T$. Finally, for $\mathcal{L} \in \{L, T\}$ a power of two, it allows the parties to convert a value x stored in $\mathbb{Z}_{\mathcal{L}}$ to $x \bmod 2 \in \mathbb{Z}_2$.

Functionality 1: $\mathcal{F}_{\text{ABB}}[L]$

Parameters: This functionality is parameterized by a positive integer L , where L is either 2^ℓ or a prime p . There is also a global parameter $T = \Theta(n^2)$ representing the size of the smaller domain.

Input: On input $(\text{Input}, P_i, \mathcal{L}, \text{id}, x)$ from P_i and $(\text{Input}, P_i, \mathcal{L}, \text{id})$ from all other parties, with id a fresh identifier, $\mathcal{L} \in \{L, T, 2\}$, and $x \in \mathbb{Z}_{\mathcal{L}}$, store $(\mathcal{L}, \text{id}, x)$.

Input Bit: On input $(\text{InputBit}, P_i, \text{id}, b)$ from P_i and $(\text{InputBit}, P_i, \text{id})$ from all other parties, with id a fresh identifier, if $b \in \{0, 1\}$, store (T, id, b) ; otherwise, send **abort** to all parties and halt.

Add: On input $(\text{Add}, \mathcal{L}, \text{id}, \text{id}_1, \text{id}_2, c, c_1, c_2)$ from all parties, where both id_1, id_2 are stored in memory with modulus $\mathcal{L}$, $c, c_1, c_2 \in \mathbb{Z}_{\mathcal{L}}$, retrieve $((\mathcal{L}, \text{id}_1, x), (\mathcal{L}, \text{id}_2, y))$, compute $z = c + c_1x + c_2y \in \mathbb{Z}_{\mathcal{L}}$ and store $(\mathcal{L}, \text{id}, z)$.

Mult: On input $(\text{Mult}, \mathcal{L}, \text{id}, \text{id}_1, \text{id}_2)$ from all parties, where both id_1, id_2 are stored in memory with modulus $\mathcal{L}$, retrieve $((\mathcal{L}, \text{id}_1, x), (\mathcal{L}, \text{id}_2, y))$, compute $z = x \cdot y \in \mathbb{Z}_{\mathcal{L}}$ and store $(\mathcal{L}, \text{id}, z)$.

Random Bit: On input $(\text{RandBit}, \mathcal{L}, \text{id})$ from all parties, with $\mathcal{L} \in \{L, T\}$ and a fresh identifier id , sample a uniformly random bit $r \in \{0, 1\}$ and store $(\mathcal{L}, \text{id}, r)$.

Coin: On input $(\text{Coin}, \mathcal{L}, \text{id})$ from all parties, with a fresh identifier id , sample a random coin r , store (L, id, r) , and send r to all parties.

Abort: On input (abort, P_i) from P_i , send **abort** to all other parties and halt.

Local Modular Reduction: On input $(\text{LocalMod}, \mathcal{L}, \text{id}, \text{id}')$ from all parties, where id' is stored in memory with modulus $\mathcal{L} \in \{L, T\}$, retrieve $(\mathcal{L}, \text{id}', x)$ and store $(2, \text{id}, x \bmod 2)$.^a

Open: On input $(\text{Open}, \mathcal{L}, \text{id})$ from all parties, retrieve $(\mathcal{L}, \text{id}, y)$ and send it to the adversary. Wait for an input from the adversary; if this is **deliver** then send y to all parties, otherwise send **abort**.

^a This operation is valid only for the case that $\mathcal{L}$ is a power of 2.

Instantiations and Notations. $\mathcal{F}_{\text{ABB}}$ can be instantiated in several ways: for example, Shamir [34] or replicated [25] secret sharing in the honest majority case, or additive secret sharing in the dishonest majority case. Active security is obtained on top of these schemes by standard techniques [20, 18, 15]. Regardless of the instantiation, we write $\llbracket x \rrbracket_{\mathcal{L}}$ for a secret value $x \in \mathbb{Z}_{\mathcal{L}}$ stored by the functionality. For $x = (x_0, \dots, x_{\kappa-1}) \in \text{GR}(2^e, \kappa)$, we write $\llbracket x \rrbracket_{2^e}^{\kappa} := (\llbracket x_0 \rrbracket_{2^e}, \dots, \llbracket x_{\kappa-1} \rrbracket_{2^e})$ to denote its representation by κ secret values over $\mathbb{Z}_{2^e}$ stored by the functionality, where the superscript κ indicates the extension degree. Instead of explicitly calling $\mathcal{F}_{\text{ABB}}$ on **Add** or **Mult**, we often use the operators $+$ and $\cdot$. For example, we may write $\llbracket z \rrbracket_{\mathcal{L}} \leftarrow \llbracket x \rrbracket_{\mathcal{L}} + \llbracket y \rrbracket_{\mathcal{L}} + c$ to denote that x and y are added together with a public constant c using **Add**.

Usually **Add** is local on the shares, while **Mult** can be instantiated in several ways, including Beaver triples [5] and random double sharings [19]. For local modular reduction over $\mathbb{Z}_{2^e}$, many secret sharing schemes (including all of the above) have the property that reducing the shares of $\llbracket x \rrbracket_{2^e}$ locally modulo 2 yields the sharing $\llbracket x \bmod 2 \rrbracket_2$. To obtain a random bit in $\mathbb{Z}_{\mathcal{L}}$ for $\mathcal{L} \in \{L, T\}$, we use the “square root trick” of [18, Figure 14] for fields and [16, Figure 5] for power-of-two rings. For **InputBit**, a direct approach checks $b(b-1) = 0$ after inputting b in $\mathbb{Z}_T$, at one multiplication and one opening per bit; the batch verification of [23] instead verifies many bits with communication sublinear in their number. We refer the reader to Appendix B for the detailed instantiation of **InputBit**.

Unless stated otherwise, we measure communication as the total number of bits sent by all parties. We use the fact that the existing works can execute inputs, openings, multiplications and

random bit sampling with communication linear in both n and the bit-length $\log \mathcal{L}$, i.e. $\Theta(n \log \mathcal{L})$ bits.

2.2 Building Blocks

Bitwise Sum. Given two bitwise shared integers $\{\llbracket a_i \rrbracket_{\mathcal{L}}\}_{i=0}^{m-1}$ and $\{\llbracket b_i \rrbracket_{\mathcal{L}}\}_{i=0}^{m-1}$, the bitwise sum operation computes $\{\llbracket d_i \rrbracket_{\mathcal{L}}\}_{i=0}^m$ such that $d = a + b$ where $a = \sum_{i=0}^{m-1} a_i 2^i$, $b = \sum_{i=0}^{m-1} b_i 2^i$ and $d = \sum_{i=0}^m d_i 2^i$. The input length m may differ from the bit-length of $\mathcal{L}$. To minimize communication we use a ripple-carry adder, denoted π_{BitAdder} . It computes each carry bit as

$$c_{i+1} = c_i \oplus (a_i \oplus c_i) \wedge (b_i \oplus c_i), \quad i = 0, \dots, m-1.$$

This requires $O(m)$ AND and XOR gates. When the inputs are secret-shared over $\mathbb{Z}_2$, XOR is for free; for a larger domain (i.e. $\mathcal{L} > 2$), XOR gates should be emulated via $x \oplus y = x + y - 2 \cdot x \cdot y$, which requires a product.

Bitwise Less-Than. Given two bitwise shared integers $\{\llbracket a_i \rrbracket_{\mathcal{L}}\}_{i=0}^{m-1}$ and $\{\llbracket b_i \rrbracket_{\mathcal{L}}\}_{i=0}^{m-1}$, the bitwise less-than operation computes $\llbracket c \rrbracket_{\mathcal{L}}$ such that $c = 1$ if $a < b$ and $c = 0$ otherwise, where $a = \sum_{i=0}^{m-1} a_i 2^i$ and $b = \sum_{i=0}^{m-1} b_i 2^i$. We denote this procedure by π_{BitLT} . For any $\mathcal{L} \geq 2$, there are several protocols to instantiate π_{BitLT} using $O(m)$ multiplications and $O(\log m)$ rounds [13, 22].

Bit Decomposition. We require the bit decomposition of a secret-shared value $\llbracket a \rrbracket_p$ for a prime p , and use the standard methods of [17, 29], as detailed in Procedure $\pi_{\text{FieldBitDec}}$ in Appendix C.1. It requires $\log p$ random bits, two calls to π_{BitLT} and one call to the bit adder. Each comparison or addition uses $O(\log p)$ multiplications over $\mathbb{Z}_p$; since each multiplication and random bit costs $O(n \log p)$ bits, the total communication is $O(n \log^2 p)$ bits.

2.3 Private Edabits

We first recall the structure of a standard edabit, whose functionality is defined in Section 2.5. As in [21], an *edabit* of length $m \leq \log L$ consists of correlated sharings across two domains: binary sharings $\llbracket r_0 \rrbracket_2, \dots, \llbracket r_{m-1} \rrbracket_2$ of random bits $r_0, \dots, r_{m-1} \in \mathbb{Z}_2$, and an arithmetic sharing $\llbracket r \rrbracket_L$ of the integer $r = \sum_{i=0}^{m-1} r_i 2^i \in \mathbb{Z}_L$.

A *private edabit* retains this correlation but is generated by an *individual* party, who also learns its value; we model it with a functionality $\mathcal{F}_{\text{PrivEd}}[L, m]$. There the bits $r_0, \dots, r_{m-1}$ are output to the generating party, and are chosen by the adversary if that party is corrupt.

In our variant the bit components are shared over $\mathbb{Z}_T$ rather than over $\mathbb{Z}_2$ as in [21]; we call this a *partially arithmetic edabit*. Unless stated otherwise, private edabits henceforth refer to this partially arithmetic variant.

Functionality 2: $\mathcal{F}_{\text{PrivEd}}[L, m]$

Parameters: The functionality is parameterized by an arithmetic modulus L and length of edabits $m \leq \log L$. L is either a prime or a power of two.

Operations from $\mathcal{F}_{\text{ABB}}$: This functionality inherits *all* the operations from $\mathcal{F}_{\text{ABB}}$.

Generate Private Edabits:

On input $(\text{priv-edabit}, P_{\text{king}}, N, \{(\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1})\}_{i=1}^N)$ from all parties with fresh identifiers, for each $i = 1, \dots, N$: if P_{king} is honest, sample $r_i \leftarrow \mathbb{Z}_{2^m}$ uniformly at random and output r_i to P_{king} ; otherwise, receive $r_i \in \mathbb{Z}_{2^m}$ from P_{king} . Then, for each i , store (L, id_i, r_i) , bit-decompose r_i as $r_i = \sum_{j=0}^{m-1} r_{i,j} 2^j$, and for each j , store $(T, \text{id}_{i,j}, r_{i,j})$.

2.4 Dabits

We now present the functionality for generating dabits, $\mathcal{F}_{\text{Dabits}}[L]$. It inherits all operations from $\mathcal{F}_{\text{PrivEd}}$, including those of $\mathcal{F}_{\text{ABB}}$.

Functionality 3: $\mathcal{F}_{\text{Dabits}}[L]$

Parameters: The functionality is parameterized by an arithmetic modulus L , either a prime or a power of two.

Operations from $\mathcal{F}_{\text{PrivEd}}$: This functionality inherits *all* the operations from $\mathcal{F}_{\text{PrivEd}}$.

Generate Dabit: On input $(\text{dabit}, N, (\text{id}_1, \text{id}'_1), \dots, (\text{id}_N, \text{id}'_N))$ from all parties with fresh $\{(\text{id}_i, \text{id}'_i)\}_{i=1}^N$, sample $b_i \in \{0, 1\}$ uniformly at random and store $(2, \text{id}_i, b_i)$, together with (L, id'_i, b_i) for all $i \in [1, N]$.

Binary to Arithmetic Conversion. An important application of dabits is enabling binary/arithmetic conversions in MPC protocols, as shown in [33, 16, 32]. In our work, we specifically use dabits to convert a binary sharing $\llbracket b \rrbracket_2$ into an arithmetic sharing $\llbracket b \rrbracket_L$. We present the procedure $\pi_{\text{BitToArith}}$ in Appendix C.2.

2.5 Edabits

Having defined private edabits and dabits, we now formally define the functionality $\mathcal{F}_{\text{Edabits}}$ for generating the standard edabits briefly introduced in Section 2.3. In contrast to a private edabit, no individual party knows the underlying value of a standard edabit, and its bit components are shared over $\mathbb{Z}_2$. The functionality $\mathcal{F}_{\text{Edabits}}$ inherits all operations from $\mathcal{F}_{\text{Dabits}}$.

Functionality 4: $\mathcal{F}_{\text{Edabits}}[L, m]$

Parameters: The functionality is parameterized by an arithmetic modulus L and length of edabits $m \leq \log L$. L is either a prime or a power of two.

Operations from $\mathcal{F}_{\text{Dabits}}$: This functionality inherits *all* the operations from $\mathcal{F}_{\text{Dabits}}$.

Generate Edabits: On input $(\text{edabit}, N, \{(\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1})\}_{i=1}^N)$ from all parties, sample $r_i \in \mathbb{Z}_{2^m}$ uniformly at random, bit-decompose it as $r_i = \sum_{j=0}^{m-1} r_{i,j} 2^j$, and store $\{(2, \text{id}_{i,j}, r_{i,j})\}_{j=0}^{m-1}$, together with (L, id_i, r_i) for all $i \in [1, N]$.

3 Aggregating Edabits

A core building block for generating an edabit of length m in [21] is taking E (private) edabits

$$(\llbracket r_1 \rrbracket_L; \llbracket r_{1,0} \rrbracket_2, \dots, \llbracket r_{1,m-1} \rrbracket_2), \dots, (\llbracket r_E \rrbracket_L; \llbracket r_{E,0} \rrbracket_2, \dots, \llbracket r_{E,m-1} \rrbracket_2)$$

of length m , and computing $(\llbracket r \rrbracket_L; \llbracket b_0 \rrbracket_2, \dots, \llbracket b_{m-1} \rrbracket_2)$, where $r = \sum_{i=0}^{m-1} b_i 2^i = \left(\sum_{i=1}^E r_i\right) \bmod 2^m$. We present an alternative aggregation method based on lifted binary sharings that reduces the overall cost of edabit generation compared with [21].

First, we recall the method in [21], which works roughly as follows:

1. Compute locally $\llbracket r' \rrbracket_L = \sum_{i=1}^E \llbracket r_i \rrbracket_L$.
2. Compute an E -input adder circuit that adds $\{(\llbracket r_{i,0} \rrbracket_2, \dots, \llbracket r_{i,m-1} \rrbracket_2)\}_{i=1}^E$ and outputs $(\llbracket b_0 \rrbracket_2, \dots, \llbracket b_{m'-1} \rrbracket_2)$, where $m' = m + \log E$.
3. Use *dabits* to convert $\llbracket b_m \rrbracket_2, \dots, \llbracket b_{m'-1} \rrbracket_2$ to $\llbracket b_m \rrbracket_L, \dots, \llbracket b_{m'-1} \rrbracket_L$.
4. Compute locally $\llbracket r \rrbracket_L = \llbracket r' \rrbracket_L - \sum_{i=m}^{m'-1} 2^i \cdot \llbracket b_i \rrbracket_L$.
5. Output $(\llbracket r \rrbracket_L; \llbracket b_0 \rrbracket_2, \dots, \llbracket b_{m-1} \rrbracket_2)$.

Much of the overhead lies in the E -input adder, due to its multiplication gates. Our idea is to share the binary component over a larger arithmetic modulus $T = 2^n$ instead of $\mathbb{Z}_2$. We call them *partially arithmetic edabits*, and they are of the form $(\llbracket r \rrbracket_L; \llbracket r_0 \rrbracket_T, \dots, \llbracket r_{m-1} \rrbracket_T)$, where $r = \sum_{j=0}^{m-1} 2^j r_j$. Lifting the binary components to this larger domain increases the cost of inputting them and of operating on them, but it allows us to *locally* aggregate several shared bits before invoking bit decomposition to extract the bits of the resulting sum.

Consider E partially arithmetic edabits $\{(\llbracket r_i \rrbracket_L; \llbracket r_{i,0} \rrbracket_T, \dots, \llbracket r_{i,m-1} \rrbracket_T)\}_{i=1}^E$ of length m . Consider first the simpler case $2^m = L$. First, the parties compute the arithmetic part $\llbracket r \rrbracket_L := \sum_{i=1}^E \llbracket r_i \rrbracket_L$. Write $r = \sum_{j=0}^{m-1} 2^j s_j$, where $s_j = \sum_{i=1}^E r_{i,j}$, and note that we get $\llbracket s_j \rrbracket_T$ for free. Let $A \mid m$ be a tweakable parameter; our strategy computes the bits of r , A at a time. We first compute locally the $(A + \log E)$ -bit value $\llbracket a \rrbracket_T = \sum_{j=0}^{A-1} 2^j \llbracket s_j \rrbracket_T$, whose lowest A bits are those of r . Then, we compute the bit-decomposition $\llbracket a_0 \rrbracket_2, \dots, \llbracket a_{\eta-1} \rrbracket_2$ of a directly over $\mathbb{Z}_2$ using the standard mask-and-open approach: The parties generate a bitwise random mask $(\llbracket e_0 \rrbracket_T, \dots, \llbracket e_{\eta-1} \rrbracket_T)$ by calling **RandBit** and compute $\llbracket e \rrbracket_T = \sum_{j=0}^{\eta-1} 2^j \llbracket e_j \rrbracket_T$. After opening $\llbracket \delta \rrbracket_T = \llbracket a \rrbracket_T - \llbracket e \rrbracket_T$, the parties locally reduce each $\llbracket e_i \rrbracket_T$ to $\llbracket e_i \rrbracket_2$ and use a bit adder over $\mathbb{Z}_2$ to remove the mask e and obtain $\llbracket a_0 \rrbracket_2, \dots, \llbracket a_{\eta-1} \rrbracket_2$. Finally we output $\llbracket a_0 \rrbracket_2, \dots, \llbracket a_{A-1} \rrbracket_2$ as the lowest A bits of r , and retain $\llbracket a_A \rrbracket_2, \dots, \llbracket a_{A+\log E-1} \rrbracket_2$ as the overflow to be added to the next block.

To handle the next set of A bits, we bit-decompose the term $\sum_{j=0}^{A-1} 2^j \llbracket s_{A+j} \rrbracket_T$, and use an additional bit adder to add the overflow from the previous block $\llbracket a_A \rrbracket_2, \dots, \llbracket a_{A+\log E-1} \rrbracket_2$ to get the next A bits of r and the overflow of this block, i.e., the bitwise shares of $a' = \sum_{j=0}^{A-1} 2^j s_{A+j} + \sum_{j=0}^{\log E-1} 2^j a_{A+j}$. We repeat with a' until all blocks are covered. For the general case in which $m < \ell$, we must compute the overflow bits of the addition $\sum_{i=1}^E r_i$, convert them to arithmetic values (using dabits), and then subtract them from $\sum_{i=1}^E r_i$ to obtain $(\sum_{i=1}^E r_i) \bmod 2^m$.

Procedure 1: $\pi_{\text{AggEdabit}}$

Dependencies: This is set in the $\mathcal{F}_{\text{Dabits}}$ -hybrid model. The procedure is parameterized by a block size A . For simplicity we assume that $A \mid (m + \log E)$. Recall $L \in \{2^\ell, p\}$ and $\ell = \lceil \log_2 L \rceil$.

Input: E partially arithmetic edabits $\{\langle r_i \rangle = (\llbracket r_i \rrbracket_L; \llbracket r_{i,0} \rrbracket_T, \dots, \llbracket r_{i,m-1} \rrbracket_T)\}_{i=1}^E$ of length m , with $A + \log E \leq \eta$.

Output: $(\llbracket r \rrbracket_L; \llbracket b_0 \rrbracket_2, \dots, \llbracket b_{m-1} \rrbracket_2)$ where $r = \sum_{i=0}^{m-1} b_i 2^i$ and $r = (\sum_{i=1}^E r_i) \bmod 2^m$.

Procedure: The parties proceed as follows:

1. Locally compute $\llbracket r' \rrbracket_L \leftarrow \sum_{i=1}^E \llbracket r_i \rrbracket_L$. (Note that $r' = S \bmod L$, where $S = r_1 + \dots + r_E$ is an $(m + \log E)$ -bit integer.)
2. For each $j \in \{0, \dots, m-1\}$, locally compute $\llbracket s_j \rrbracket_T \leftarrow \sum_{i=1}^E \llbracket r_{i,j} \rrbracket_T$, and let $\llbracket s_j \rrbracket_T \leftarrow \llbracket 0 \rrbracket_T$ for $j \geq m$.
3. Let $\llbracket c_i \rrbracket_2 \leftarrow \llbracket 0 \rrbracket_2$ for $i = 0, \dots, \eta - A - 1$. For $j = 0, \dots, (m + \log E)/A - 1$: (When $L = 2^\ell$ and $\ell < m + \log E$ we can do this loop for $j = 0, \dots, \ell/A - 1$, and set $\llbracket b_i \rrbracket_2 = \llbracket 0 \rrbracket_2$ for the remaining $i = \ell, \dots, m + \log E - 1$.)
 - (a) Locally compute $\llbracket a \rrbracket_T \leftarrow \sum_{i=0}^{A-1} 2^i \llbracket s_{j \cdot A + i} \rrbracket_T$.
 - (b) For $i = 0, \dots, \eta - 1$, call $\llbracket e_i \rrbracket_T \leftarrow \text{RandBit}()$.
 - (c) $\llbracket e \rrbracket_T \leftarrow \sum_{i=0}^{\eta-1} 2^i \llbracket e_i \rrbracket_T$.
 - (d) $\llbracket \delta \rrbracket_T \leftarrow \llbracket a \rrbracket_T - \llbracket e \rrbracket_T$ and open δ .
 - (e) $\{\llbracket u_i \rrbracket_2\}_{i=0}^{\eta-1} \leftarrow \pi_{\text{BitAdder}}(\delta, \{\llbracket c_i \rrbracket_2\}_{i=0}^{\eta-A-1})$. (This corresponds to adding the overflow from the previous block, to the current block.)
 - (f) Call $\llbracket e_i \rrbracket_2 \leftarrow \text{LocalMod}(\llbracket e_i \rrbracket_T)$ for $i = 0, \dots, \eta - 1$, and compute $\{\llbracket a'_i \rrbracket_2\}_{i=0}^{\eta-1} \leftarrow \pi_{\text{BitAdder}}(\{\llbracket u_i \rrbracket_2\}_{i=0}^{\eta-1}, \{\llbracket e_i \rrbracket_2\}_{i=0}^{\eta-1})$. (This step removes the random mask e , resulting in the bitwise shares of $a + c$.)
 - (g) Let $\llbracket b_{j \cdot A + i} \rrbracket_2 \leftarrow \llbracket a'_i \rrbracket_2$ for $i = 0, \dots, A - 1$.
 - (h) Set $\llbracket c_i \rrbracket_2 \leftarrow \llbracket a'_{i+A} \rrbracket_2$ for $i = 0, \dots, \eta - A - 1$. (It is the overflow of the current block.)
4. For $j = m, \dots, m + \log E - 1$, call $\llbracket b_j \rrbracket_L \leftarrow \pi_{\text{BitToArith}}(\llbracket b_j \rrbracket_2)$.
5. Compute locally $\llbracket r \rrbracket_L \leftarrow \llbracket r' \rrbracket_L - \sum_{i=m}^{(m+\log E)-1} 2^i \cdot \llbracket b_i \rrbracket_L$. (Note that, for $L = 2^\ell$ and $m = \ell$, these two last steps can be omitted.)
6. Output $(\llbracket r \rrbracket_L; \llbracket b_0 \rrbracket_2, \dots, \llbracket b_{m-1} \rrbracket_2)$.

The procedure is correct provided $\eta \geq A + \log E$. To see this, let $S = \sum_{i=1}^E r_i$. Since each input edabit is consistent, we have $S = \sum_{j=0}^{m-1} 2^j s_j$. Consider the loop in Step 3 block by block. At the beginning of each iteration, $c = \sum_i 2^i c_i$ represents the carry from the preceding block. We first show by induction that $c \leq E - 1$. This holds initially since $c = 0$. Assuming that it holds at the

beginning of the current iteration, each $s_{jA+i} \leq E$, and hence

$$a + c \leq E \sum_{i=0}^{A-1} 2^i + E - 1 = E \cdot 2^A - 1 < 2^\eta = T.$$

Thus, adding the carry to the current block sum does not wrap around modulo T . This carry is incorporated during the masked bit decomposition of a : the first binary addition adds c to the opened value $\delta = (a - e) \bmod T$, and the second adds back the bit-decomposed mask e modulo $2^\eta = T$. Therefore, the resulting bits $a'_0, \dots, a'_{\eta-1}$ are exactly the bit decomposition of $a + c$. The lowest A bits are assigned to $b_{jA}, \dots, b_{jA+A-1}$, while the remaining bits represent the new carry, which is at most $\lfloor (a + c)/2^A \rfloor \leq E - 1$. This completes the induction. Applying the same argument block by block gives

$$S = \sum_{j=0}^{m+\log E-1} 2^j b_j.$$

Hence $b_0, \dots, b_{m-1}$ are the lowest m bits of S , so $\sum_{j=0}^{m-1} 2^j b_j = S \bmod 2^m$. Finally, Steps 4–5 remove the contribution of the higher bits from $r' = S \bmod L$, so the arithmetic output r is consistent with these m binary components.

Communication. Taking $A = \log E$, and hence $\eta = 2 \log E$, the procedure costs $O(n(m + \log E) \log E) + \mathcal{C}_{\pi \text{BitToArith}}^{\mathbb{Z}_L} \cdot O(\log E)$ bits, where $\mathcal{C}_{\pi \text{BitToArith}}^{\mathbb{Z}_L}$ is the cost of one binary-to-arithmetic conversion. The aggregation of [21] instead evaluates a tree of $E - 1$ ripple-carry adders over $\mathbb{Z}_2$, at a cost of $\Theta(nEm) + \mathcal{C}_{\pi \text{BitToArith}}^{\mathbb{Z}_L} \cdot O(\log E)$ bits, so our aggregator improves the first summand by a factor of $\frac{E}{\log E}$ whenever $\log E = O(m)$; in [21] one edabit aggregates $E = t + 1 = \Theta(n)$ private edabits, so this is an improvement of $\frac{n}{\log n}$. The trade-off is that the binary components must be shared over the larger domain $\mathbb{Z}_T$ rather than over $\mathbb{Z}_2$. The second summand is handled in Section 6, where we obtain $\mathcal{C}_{\pi \text{BitToArith}}^{\mathbb{Z}_L} = O(n\ell)$, against $O(n^2\ell)$ for the standard dabits of [33] used in [21]. The detailed counts are given in Appendix E.2.

4 Global Edabits from Private Edabits

We now obtain global edabits from private edabits, using the aggregation method of Section 3. We first introduce a randomness extraction method that allows us to consume only $O(1)$ private edabits per global edabit (amortized) when the corruption threshold $t < (1 - \epsilon)n$ with $\epsilon \in (0, 1)$ an arbitrary constant, instead of $\Omega(n)$ as in [21].

4.1 Randomness Extraction

In [21], each party P_i produces a private edabit $(\llbracket r_i \rrbracket_L : \llbracket r_{i,0} \rrbracket_2, \dots, \llbracket r_{i,m-1} \rrbracket_2)$, and these are aggregated into an edabit. We note that if there is more than one honest party, we can extract more than one shared edabit from a given set of private edabits, improving over prior work.

This is the approach of [19] for extracting $n - t$ shared random values from n private ones, using an $(n - t) \times n$ super-invertible matrix $\mathbf{M}$, in which every square $(n - t) \times (n - t)$ submatrix is invertible. Each party shares a random value $\llbracket r_i \rrbracket$ and the parties output $(\llbracket s_1 \rrbracket, \dots, \llbracket s_{n-t} \rrbracket)^\top = \mathbf{M} \cdot (\llbracket r_1 \rrbracket, \dots, \llbracket r_n \rrbracket)^\top$; by super-invertibility, there is a bijection between the randomness of the $n - t$ honest parties and the $n - t$ outputs. Usually $\mathbf{M}$ is a Vandermonde matrix.

We will do the same, but, since the range of edabits is a power of two, 2^m , a Vandermonde matrix is not naively super-invertible. Instead, a common trick [12, 2] is to use the Galois field $\mathbb{F} := \text{GF}(2^d)$, for $d := \lceil \log_2 n \rceil$, over which a super-invertible Vandermonde matrix $\mathbf{M}$ does exist. We use this matrix $\mathbf{M}$ over $\mathbb{F}$ in a special way. Fixing a basis of $\mathbb{F}$ over $\mathbb{Z}_2$ represents each element of $\mathbb{F}$ as a d -dimensional vector over $\mathbb{Z}_2$, and multiplying such a vector by some $x \in \mathbb{F}$ is a $\mathbb{Z}_2$ -linear map, given by a $d \times d$ matrix $\mathbf{M}_x$ over $\mathbb{Z}_2$: there is an injective $\phi : \mathbb{F} \rightarrow \mathbb{Z}_2^{d \times d}$ that is a field isomorphism onto its image [12]. Now, if we have an $(n - t) \times n$ matrix $\mathbf{M}$ over $\mathbb{F}$, then we can represent it as a $(n - t)d \times nd$ matrix $\overline{\mathbf{M}}$ over $\mathbb{Z}_2$, by replacing each entry m_{ij} of $\mathbf{M}$ with the corresponding $d \times d$ matrix $\phi(m_{ij})$. This matrix $\overline{\mathbf{M}}$ will have the following property, which we call *block super-invertibility*:

Definition 1. Let $m \geq 1$ and matrix $\overline{\mathbf{M}} \in \mathbb{Z}_{2^m}^{(n-t)d \times nd}$. Divide the nd columns of $\overline{\mathbf{M}}$ into n non-overlapping blocks of d consecutive columns each. Then, $\overline{\mathbf{M}}$ is block super-invertible with block size d over $\mathbb{Z}_{2^m}$ if for any selection of $n-t$ blocks of columns, the corresponding $(n-t)d \times (n-t)d$ submatrix is invertible.

Lemma 1. Let $\mathbf{M}$ be an $(n-t) \times n$ super-invertible matrix over $\text{GF}(2^d)$. Then, the corresponding $(n-t)d \times nd$ matrix $\overline{\mathbf{M}} := \phi(\mathbf{M})$ over $\mathbb{Z}_2$ is block super-invertible.

The proof is given in Appendix D.1.

Finally, if a matrix $\mathbf{M} \in \mathbb{Z}_2^{(n-t)d \times nd}$ is block super-invertible with block size d , then it is also block super-invertible over $\mathbb{Z}_{2^m}$. Indeed, every relevant square submatrix is invertible over $\mathbb{Z}_2$, so its determinant is odd, hence a unit modulo 2^m , and the submatrix is invertible over $\mathbb{Z}_{2^m}$. A block super-invertible matrix over $\mathbb{Z}_{2^m}$ with entries in $\{0, 1\}$ will be advantageous in the next section.

With this block super-invertible matrix $\mathbf{M} \in \mathbb{Z}_{2^m}^{(n-t)d \times nd}$, we will perform randomness extraction in the next section as follows: each party P_i will share d random values $\llbracket r_i^1 \rrbracket, \dots, \llbracket r_i^d \rrbracket$ and then the parties will compute and output

$$(\llbracket s_1^1 \rrbracket, \dots, \llbracket s_1^d \rrbracket, \dots, \llbracket s_{n-t}^1 \rrbracket, \dots, \llbracket s_{n-t}^d \rrbracket) = \mathbf{M} \cdot (\llbracket r_1^1 \rrbracket, \dots, \llbracket r_1^d \rrbracket, \dots, \llbracket r_n^1 \rrbracket, \dots, \llbracket r_n^d \rrbracket). \quad (1)$$

By block super-invertibility, there is a bijection between the blocks of random values of the $n-t$ honest parties and the $n-t$ blocks of output sharings.

4.2 Global Edabits

With $\pi_{\text{AggEdabit}}$ and the randomness extraction just described, we can present our protocol for generating global edabits from private edabits. Since the block super-invertible matrix from Section 4.1 has entries in $\{0, 1\}$, each output of the extraction is a subset sum of private edabits, which we compute using $\pi_{\text{AggEdabit}}$. Taking $d = \lceil \log_2 n \rceil$, each aggregation has at most nd inputs, so we set $\eta = \lceil 2 \log nd \rceil$ and $T = 2^\eta$ according to the communication analysis in Section 3. Protocol 2 generates N edabits by repeating this extraction of $(n-t)d$ edabits $N/((n-t)d)$ times in parallel.

Protocol 2: $\Pi_{\text{Edabits}}(L, m)$

Dependencies: This protocol is set in the $\mathcal{F}_{\text{Dabits}}[L]$ -hybrid model. The parameter L denotes the arithmetic modulus, and m denotes the length of the edabits to be generated. We require $0 < m \leq \ell$ if $L = 2^\ell$, and $0 < m < \log_2 p$ if $L = p$ is prime. Let $\mathbf{M}$ be an $(n-t)d \times nd$ block super-invertible matrix over $\mathbb{Z}_2$ (and hence also over $\mathbb{Z}_{2^m}$) with block size d .

Procedure: Upon receiving any input instruction related to the operations in $\mathcal{F}_{\text{Dabits}}[L]$, all parties pass them to $\mathcal{F}_{\text{Dabits}}[L]$. Upon receiving the input instruction (edabit, $N, \{(\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1})\}_{i=1}^N$), all parties execute the following steps for $N/((n-t)d)$ times in parallel:

1. The parties call **priv-edabit**, which is an operation in $\mathcal{F}_{\text{Dabits}}[L]$, to obtain nd private edabits

$$\{(\llbracket r_i \rrbracket_L; \llbracket r_{i,0} \rrbracket_T, \dots, \llbracket r_{i,m-1} \rrbracket_T)\}_{i=1}^{nd}$$

of length m , where P_β knows $r_{i,0}, \dots, r_{i,m-1}$ and $r_i = \sum_{k=0}^{m-1} 2^k r_{i,k}$ for $(\beta-1)d < i \leq \beta d$.

2. For each $i \in \{1, \dots, (n-t)d\}$, call

$$\pi_{\text{AggEdabit}}\left(\{(\llbracket r_j \rrbracket_L; \llbracket r_{j,0} \rrbracket_T, \dots, \llbracket r_{j,m-1} \rrbracket_T) \mid j \text{ s.t. } \mathbf{M}_{i,j} = 1\}\right)$$

in parallel to obtain $(\llbracket s_i \rrbracket_L; \llbracket s_{i,0} \rrbracket_2, \dots, \llbracket s_{i,m-1} \rrbracket_2)$.

3. Output $\{(\llbracket s_i \rrbracket_L; \llbracket s_{i,0} \rrbracket_2, \dots, \llbracket s_{i,m-1} \rrbracket_2)\}_{i=1}^{(n-t)d}$.

The following theorem states correctness and security; the proof is in Appendix D.2.

Theorem 1. Protocol $\Pi_{\text{Edabits}}(L, m)$ securely realizes functionality $\mathcal{F}_{\text{Edabits}}[L, m]$ in the $\mathcal{F}_{\text{Dabits}}[L]$ -hybrid model, where $0 < m \leq \ell$ if $L = 2^\ell$ and $0 < m < \log_2 p$ if $L = p$ is prime.

Communication. Thanks to the randomness extraction of Section 4.1, each global edabit consumes only $\frac{n}{n-t} = O(1)$ private edabits and one call to $\pi_{\text{AggEdabit}}$ of fan-in $O(nd)$, for a total of $O(\mathcal{C}_{\text{PrivEd}} + nm \log n + \mathcal{C}_{\pi_{\text{BitToArith}}^{\mathbb{Z}_L}} \log n)$ bits. Using $\mathcal{C}_{\text{PrivEd}} = O(n\ell \log n)$ (Section 5) and $\mathcal{C}_{\pi_{\text{BitToArith}}^{\mathbb{Z}_L}} = O(n\ell)$ (Section 6), this is $O(n\ell \log n)$ bits whenever $t < (1 - \epsilon)n$ for an arbitrary constant $\epsilon \in (0, 1)$. The protocol of [21] instead consumes $E = t + 1$ private edabits and a binary adder tree per global edabit, which costs $\Omega(n^2\ell)$ over rings and $\Omega(n^3\ell)$ over fields, so we improve by at least a factor $\frac{n}{\log n}$ and bring edabit generation from quadratic to quasi-linear in n . The underlying counts are given in Appendix E.3, and the concrete comparison in Section 7.

4.3 Bonus: Any-Range Edabits

Standard edabits sample their underlying value r uniformly from a power-of-two range $[0, 2^m)$. In many applications, such as threshold ML-DSA signing [7], we need r to be uniformly random in the whole ring/field under consideration.⁷ For $L = 2^\ell$ this is achieved by taking $m = \ell$, but for $L = p$ more care is needed. One could take $m = \lfloor \log p \rfloor$, but then r is only *close* to uniform in $\mathbb{Z}_p$, at statistical distance $\frac{p-2^m}{p}$, which may not be negligible for many choices of p .

We address the more general task of generating *any-range edabits*: r is sampled uniformly from a target range $[0, P)$, where $2 \leq P \leq L$, and shared over $\mathbb{Z}_L$ together with its ℓ_P -bit decomposition over $\mathbb{Z}_2$, where $\ell_P = \lceil \log P \rceil$. Setting $P = L$ gives full-range edabits over both rings and fields. We give the corresponding functionality and construction in Appendix C.3.

5 Partially Arithmetic Private Edabits

We show how to realize $\mathcal{F}_{\text{PrivEd}}$ in the $\mathcal{F}_{\text{ABB}}$ -hybrid model. To generate N private edabits provided by a dealer P_{king} , the dealer P_{king} first inputs to $\mathcal{F}_{\text{ABB}}$ the tuples

$$\{(\llbracket r_i \rrbracket_L; \llbracket r_{i,0} \rrbracket_T, \dots, \llbracket r_{i,m-1} \rrbracket_T)\}_{i=1}^N.$$

The binary parts are input through calls to `InputBit`, which guarantees that every $r_{i,j}$ is a bit. It remains to verify that the arithmetic and binary components are consistent, i.e., $r_i = \sum_{j=0}^{m-1} 2^j r_{i,j}$ over the arithmetic domain.

To this end, our idea is to check a random linear combination of all private edabits input by P_{king} . However, such combinations are not efficient for the binary part. A simple fix is to sample each coefficient independently and uniformly from $\{0, 1\}$, which computes a random subset sum of the private edabits. In this way, we can use our efficient aggregation protocol in Section 3.

Turning this idea into a secure and efficient protocol raises three issues.

1. Opening a random subset sum directly would reveal information about the underlying values. We therefore add a fresh random mask, in both arithmetic and binary form, to each subset sum and open their sum for verification.
2. The number of random coins per subset sum scales with N . We instead derive all selection bits from the powers of a single random field element and bound the soundness error with the Schwartz–Zippel lemma, as detailed below for rings and fields.
3. Since each aggregation handles only $O(T)$ inputs, we aggregate each subset sum level by level in a tree, at $O(N/T)$ intermediate aggregations per check, each involving bit decompositions. Unlike in Section 3, all inputs here are known to the same dealer P_{king} , so we let P_{king} provide these bit decompositions and have the parties verify them.

We first present the assisted tree-aggregation procedure, common to both domains (Procedure 3), and then the ring case $L = 2^\ell$ and the field case $L = p$ separately, as their final consistency checks differ.

⁷ In fact, most of the applications of edabits require this, since the arithmetic part is typically used to mask a secret and hence it must be uniform in the whole domain. This is a subtlety that seems to have escaped the authors in [21].

Procedure 3: $\pi_{\text{TreeAggregate}} \left((\llbracket r_{1,j} \rrbracket_T)_{j=0}^{m-1}, \dots, (\llbracket r_{N',j} \rrbracket_T)_{j=0}^{m-1} \right)$

1. Let $E' = 2^{\eta-1} - 1$ and initialize a counter $\text{cnt} = 0$.
2. For each $i = 1, \dots, N'/E'$, from $j = 0$ to $m-1$:
 - (a) $\llbracket s_{i,j} \rrbracket_T \leftarrow \llbracket c_{i,j} \rrbracket_T + \sum_{k=1}^{E'} \llbracket r_{(i-1) \cdot E' + k, j} \rrbracket_T$. ($c_{i,j}$ is considered as 0 if $j = 0$.)
 - (b) P_{king} decomposes $s_{i,j}$ into bits $s_{i,j,0}, \dots, s_{i,j,\eta-1}$ and inputs $\llbracket s_{i,j,0} \rrbracket_T, \dots, \llbracket s_{i,j,\eta-1} \rrbracket_T$ by calling **InputBit**.
 - (c) Increment cnt by 1 and $\llbracket v_{\text{cnt}} \rrbracket_T \leftarrow \llbracket s_{i,j} \rrbracket_T - \sum_{k=0}^{\eta-1} 2^k \cdot \llbracket s_{i,j,k} \rrbracket_T$.
 - (d) Set $\llbracket S_{i,j} \rrbracket_T := \llbracket s_{i,j,0} \rrbracket_T$.
 - (e) $\llbracket c_{i,j+1} \rrbracket_T \leftarrow \sum_{k=1}^{\eta-1} 2^{k-1} \cdot \llbracket s_{i,j,k} \rrbracket_T$.
3. If $N'/E' > 1$, repeat Step 2 with the new aggregated values $(\llbracket S_{1,j} \rrbracket_T)_{j=0}^{m-1}, \dots, (\llbracket S_{N'/E',j} \rrbracket_T)_{j=0}^{m-1}$ and $N' \leftarrow N'/E'$ until only one tuple of sharings is left, denoted by $(\llbracket S_j \rrbracket_T)_{j=0}^{m-1}$.
4. The parties call **Coin** to get κ random elements $\gamma_0, \dots, \gamma_{\kappa-1}$ over $\mathbb{Z}_T$.
5. Let $\gamma = (\gamma_0, \dots, \gamma_{\kappa-1}) \in \text{GR}(T, \kappa)$. All parties locally compute $\gamma^2, \dots, \gamma^{\text{cnt}}$ over $\text{GR}(2^\eta, \kappa)$.
6. $\llbracket o \rrbracket_T^\kappa \leftarrow \sum_{i=1}^{\text{cnt}} \gamma^i \cdot \llbracket v_i \rrbracket_T$ and open o . If $o \neq 0$, abort.
7. Return $(\llbracket S_j \rrbracket_T)_{j=0}^{m-1}$.

Ring Case. When $L = 2^\ell$, we pack the κ masked subset sums into one element of $\text{GR}(2^\ell, \kappa)$ and check consistency modulo 2^m . The parties sample $\beta \in \text{GF}(2^\kappa)$ and compute its powers over this field, viewing each power as its coefficient-wise lift when used over $\text{GR}(2^\ell, \kappa)$. With a random mask $\gamma \in \text{GR}(2^\ell, \kappa)$ supplied by the dealer, they open $R = \gamma + \sum_{i=1}^N \beta^i r_i \in \text{GR}(2^\ell, \kappa)$. For each coordinate, the parties use $\pi_{\text{TreeAggregate}}$ to aggregate the selected binary components and the lowest m bits of the mask coefficient. They collect the resulting bits into $R'_0, \dots, R'_{m-1} \in \text{GR}(T, \kappa)$ and verify

$$R \equiv \sum_{j=0}^{m-1} 2^j R'_j \pmod{2^m}.$$

The check always passes for consistent inputs, and Lemma 2 bounds its soundness error. The protocol is given in Protocol 4, with security stated in Theorem 2. The proofs of Lemma 2 and Theorem 2 are given in Appendices D.3 and D.4, respectively.

Protocol 4: $\Pi_{\text{PrivEd}}(2^\ell, m)$

Dependencies: This is set in the $\mathcal{F}_{\text{ABB}}[2^\ell]$ -hybrid model. The parameter m denotes the length of the private edabits to be generated, where $0 < m \leq \ell$.

Procedure: Upon receiving any input instruction related to the operations in $\mathcal{F}_{\text{ABB}}[2^\ell]$, all parties pass them to $\mathcal{F}_{\text{ABB}}[2^\ell]$. Upon receiving the input instruction $(\text{priv-edabit}, P_{\text{king}}, N, \{\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1}\}_{i=1}^N)$, all parties proceed as follows.

1. For $i = 1, \dots, N$: P_{king} samples $r_i \leftarrow \mathbb{Z}_{2^m}$ and decomposes r_i into $r_{i,0}, \dots, r_{i,m-1} \in \mathbb{Z}_2$ so that $r_i = \sum_{j=0}^{m-1} 2^j r_{i,j}$. Then P_{king} inputs r_i to $\mathcal{F}_{\text{ABB}}[2^\ell]$ as an element in $\mathbb{Z}_{2^\ell}$, and inputs $r_{i,0}, \dots, r_{i,m-1}$ by calling **InputBit**.
2. P_{king} samples $\gamma = (\gamma_0, \dots, \gamma_{\kappa-1}) \leftarrow \text{GR}(2^\ell, \kappa)$ and inputs it to $\mathcal{F}_{\text{ABB}}[2^\ell]$ as κ elements in $\mathbb{Z}_{2^\ell}$, denoted by $\llbracket \gamma \rrbracket_{2^\ell}^\kappa$. For each $k = 0, \dots, \kappa-1$, P_{king} computes the lowest m bits $\gamma_{k,0}, \dots, \gamma_{k,m-1} \in \mathbb{Z}_2$ of γ_k so that $\gamma_k \bmod 2^m = \sum_{j=0}^{m-1} 2^j \gamma_{k,j}$, and inputs all these bits by calling **InputBit**.
3. Get κ random bits $b_1, \dots, b_\kappa$ by calling **Coin**. Let $\beta = (b_1, \dots, b_\kappa) \in \text{GF}(2^\kappa)$. All parties locally compute powers $\beta^2, \dots, \beta^N$ over $\text{GF}(2^\kappa)$. Let $\beta^i = (\beta_0^i, \dots, \beta_{\kappa-1}^i)$ for $i = 1, \dots, N$.
4. $\llbracket R \rrbracket_{2^\ell}^\kappa \leftarrow \llbracket \gamma \rrbracket_{2^\ell}^\kappa + \sum_{i=1}^N \llbracket r_i \rrbracket_{2^\ell} \cdot \beta^i$ over $\text{GR}(2^\ell, \kappa)$, where $\beta^i \in \text{GF}(2^\kappa) \subset \text{GR}(2^\ell, \kappa)$ and $r_i \in \mathbb{Z}_{2^\ell} \subset \text{GR}(2^\ell, \kappa)$. Then open $R \in \text{GR}(2^\ell, \kappa)$.
5. To get the corresponding summation of binary parts, for each $k = 0, \dots, \kappa-1$, all parties compute

$$(\llbracket R'_{j,k} \rrbracket_T)_{j=0}^{m-1} \leftarrow \pi_{\text{TreeAggregate}} \left((\beta_k^1 \llbracket r_{1,j} \rrbracket_T)_{j=0}^{m-1}, \dots, (\beta_k^N \llbracket r_{N,j} \rrbracket_T)_{j=0}^{m-1}, (\llbracket \gamma_{k,j} \rrbracket_T)_{j=0}^{m-1} \right).$$

Set $\llbracket R'_j \rrbracket_T^\kappa = (\llbracket R'_{j,0} \rrbracket_T, \dots, \llbracket R'_{j,\kappa-1} \rrbracket_T)$ for $j = 0, \dots, m-1$.

6. Open $R'_j \in \text{GR}(T, \kappa)$ for $j = 0, \dots, m-1$. If $R \neq \sum_{j=0}^{m-1} 2^j R'_j \pmod{2^m}$, abort.
7. Output $\{(\llbracket r_i \rrbracket_{2^\ell}; \llbracket r_{i,0} \rrbracket_T, \dots, \llbracket r_{i,m-1} \rrbracket_T)\}_{i=1}^N$.

Lemma 2. Let N, m be positive integers, a_0, b_0 be elements in $\text{GR}(2^m, \kappa)$, and $a_1, \dots, a_N, b_1, \dots, b_N$ be $2N$ elements in the ring $\mathbb{Z}_{2^m}$. If there exists $i \in \{0, \dots, N\}$ such that $a_i \neq b_i$, then for a uniformly random $\beta \in \text{GF}(2^\kappa)$ and its powers $\beta^0 = 1, \beta^1, \dots, \beta^N$ over $\text{GF}(2^\kappa)$, the probability that $\sum_{i=0}^N a_i \cdot \beta^i = \sum_{i=0}^N b_i \cdot \beta^i$ over $\text{GR}(2^m, \kappa)$ is at most $N/2^\kappa$.

Theorem 2. Protocol $\Pi_{\text{PrivEd}}(2^\ell, m)$ securely realizes functionality $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ in the $\mathcal{F}_{\text{ABB}}[2^\ell]$ -hybrid model with abort in the presence of a fully malicious adversary controlling t corrupted parties, where ℓ, m are positive integers and $m \leq \ell$.

Field Case. For intuition, assume first that $p > 2^\kappa$ and omit the masks. The parties sample $\beta \in \mathbb{F}_p$ uniformly at random and check a linear combination of the consistency errors with coefficients $\beta, \dots, \beta^N$. The soundness error is at most $N/p < N/2^\kappa$ by the Schwartz–Zippel lemma. To apply these coefficients to the binary components, write $\beta^i = \sum_{b=0}^{\ell-1} 2^b \beta_b^i$ with $\beta_b^i \in \{0, 1\}$ and $\ell = \lceil \log_2 p \rceil$. Writing $\hat{r}_i = \sum_{j=0}^{m-1} 2^j r_{i,j}$, we have

$$\sum_{i=1}^N \beta^i \hat{r}_i = \sum_{b=0}^{\ell-1} 2^b \left(\sum_{i=1}^N \beta_b^i \hat{r}_i \right).$$

Each inner sum can therefore be computed using $\pi_{\text{TreeAggregate}}$ and independently masked as described above.

For a general prime p we work over the extension field $\text{GF}(p^{\kappa_p})$ with $\kappa_p = \lceil \kappa / \log_2 p \rceil$. The parties sample $\beta \in \text{GF}(p^{\kappa_p})$, compute $\beta, \dots, \beta^N$ over this field, and apply the above computation to each of their κ_p coordinates over $\mathbb{F}_p$. Combining the coordinate-wise equalities yields a single polynomial over $\text{GF}(p^{\kappa_p})$. Then the soundness error is bounded by $N/p^{\kappa_p} \leq N/2^\kappa$.

A remaining difficulty is that the arithmetic subset sums are reduced modulo p , whereas $\pi_{\text{TreeAggregate}}$ computes the binary sums over the integers. For each coordinate a and bit position b , let $R_{a,b}$ be the opened masked arithmetic sum. The dealer P_{king} computes $q'_{a,b}$, the number of multiples of p removed from the corresponding integer sum, and could provide it in bits so that the parties subtract $q'_{a,b}p$ from the binary sum. To avoid subtraction, we note that the sum contains at most $N+1$ terms in $\mathbb{F}_p$, so $0 \leq q'_{a,b} \leq N$. The dealer instead inputs the bits of $q_{a,b} = N - q'_{a,b}$, and the parties use $\pi_{\text{TreeAggregate}}$ to add $q_{a,b}p$ to the masked binary sum and check that the result equals $R_{a,b} + Np$. All bitwise sharings are zero-extended to a common length ℓ_{agg} large enough to prevent wraparound, so the comparison is over the integers. The protocol is given in Protocol 5 and its security in Theorem 3, proved in Appendix D.5.

Protocol 5: $\Pi_{\text{PrivEd}}(p, m)$

Dependencies: This is set in the $\mathcal{F}_{\text{ABB}}[p]$ -hybrid model. The parameter m denotes the length of the private edabits to be generated, where $0 < m < \ell = \lceil \log_2 p \rceil$. Let $\kappa_p = \lceil \kappa / \log_2 p \rceil$.

Procedure: Upon receiving any input instruction related to the operations in $\mathcal{F}_{\text{ABB}}[p]$, all parties pass them to $\mathcal{F}_{\text{ABB}}[p]$. Upon receiving the input instruction $(\text{priv-edabit}, P_{\text{king}}, N, \{\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1}\}_{i=1}^N)$, all parties proceed as follows.

1. For $i = 1, \dots, N$: P_{king} samples $r_i \leftarrow \mathbb{Z}_{2^m}$ and decomposes r_i into $r_{i,0}, \dots, r_{i,m-1} \in \mathbb{Z}_2$ so that $r_i = \sum_{j=0}^{m-1} 2^j r_{i,j}$. Then P_{king} inputs r_i to $\mathcal{F}_{\text{ABB}}[p]$ as an element in $\mathbb{Z}_p$, and inputs $r_{i,0}, \dots, r_{i,m-1}$ by calling **InputBit**.
2. For each $a = 0, \dots, \kappa_p - 1$ and $b = 0, \dots, \ell - 1$, P_{king} samples $\gamma_{a,b} \leftarrow \mathbb{F}_p$ and inputs it to $\mathcal{F}_{\text{ABB}}[p]$ as an element in $\mathbb{F}_p$. Then P_{king} decomposes $\gamma_{a,b}$ into $\gamma_{a,b,0}, \dots, \gamma_{a,b,\ell-1} \in \mathbb{Z}_2$ so that $\gamma_{a,b} = \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j}$, and inputs all these bits by calling **InputBit**.
3. Parties get a random element $\beta \leftarrow \text{GF}(p^{\kappa_p})$ by calling **Coin**. Then all parties compute powers $\beta^1, \dots, \beta^N$ over $\text{GF}(p^{\kappa_p})$. Let $\beta^i = (\beta_0^i, \dots, \beta_{\kappa_p-1}^i)$. For each $i = 1, \dots, N$ and $a = 0, \dots, \kappa_p - 1$, parties decompose each β_a^i into bits $\beta_{a,0}^i, \dots, \beta_{a,\ell-1}^i \in \mathbb{Z}_2$ so that $\beta_a^i = \sum_{b=0}^{\ell-1} 2^b \beta_{a,b}^i$.
4. For each $a = 0, \dots, \kappa_p - 1$ and $b = 0, \dots, \ell - 1$:
 - (a) $\llbracket R_{a,b} \rrbracket_p \leftarrow \llbracket \gamma_{a,b} \rrbracket_p + \sum_{i=1}^N \llbracket r_i \rrbracket_p \cdot \beta_a^i$ over $\mathbb{F}_p$ and open $R_{a,b}$.
 - (b) P_{king} computes $q'_{a,b}$ such that $R_{a,b} + q'_{a,b}p = \gamma_{a,b} + \sum_{i=1}^N r_i \cdot \beta_a^i$ over $\mathbb{Z}$ and bit-decomposes $q_{a,b} := N - q'_{a,b}$ into $q_{a,b,0}, \dots, q_{a,b,\lceil \log N \rceil} \in \mathbb{Z}_2$. Then P_{king} inputs $q_{a,b,0}, \dots, q_{a,b,\lceil \log N \rceil}$ to $\mathcal{F}_{\text{ABB}}[p]$ by calling **InputBit**.

- (c) Let $p_0, \dots, p_{\ell-1} \in \{0, 1\}$ be the bit decomposition of p so that $p = \sum_{u=0}^{\ell-1} 2^u p_u$. All parties compute ^a

$$\begin{aligned} (\llbracket R'_{a,b,j} \rrbracket_T)_{j=0}^{\ell_{\text{agg}}-1} \leftarrow \pi_{\text{TreeAggregate}} \left(\right. \\ (\beta_{a,b}^1 \llbracket r_{1,j} \rrbracket_T)_{j=0}^{\ell_{\text{agg}}-1}, \dots, (\beta_{a,b}^N \llbracket r_{N,j} \rrbracket_T)_{j=0}^{\ell_{\text{agg}}-1}, \\ (\llbracket \gamma_{a,b,j} \rrbracket_T)_{j=0}^{\ell_{\text{agg}}-1}, \\ \left. (p_0 \llbracket q_{a,b,j-0} \rrbracket_T)_{j=0}^{\ell_{\text{agg}}-1}, \dots, (p_{\ell-1} \llbracket q_{a,b,j-(\ell-1)} \rrbracket_T)_{j=0}^{\ell_{\text{agg}}-1} \right), \end{aligned}$$

where

$$\ell_{\text{agg}} := \ell + \lceil \log(3N + 1) \rceil,$$

$\llbracket r_{1,j} \rrbracket_T, \dots, \llbracket r_{N,j} \rrbracket_T := 0$ for $j \geq m$, $\llbracket \gamma_{a,b,j} \rrbracket_T := 0$ for $j \geq \ell$, and $\llbracket q_{a,b,j} \rrbracket_T := 0$ for $j < 0$ or $j > \lceil \log N \rceil$. (Note that $(p_u \llbracket q_{a,b,j-u} \rrbracket_T)_{j=0}^{\ell_{\text{agg}}-1}$ is actually the bit-wise sharing of $2^u p_u \cdot q_{a,b}$.)

- (d) Open $R'_{a,b,j}$ for $j = 0, \dots, \ell_{\text{agg}} - 1$.
 5. If there exist a, b such that $R_{a,b} + Np \neq \sum_{j=0}^{\ell_{\text{agg}}-1} 2^j R'_{a,b,j}$, then parties **abort**.
 6. Output $\{(\llbracket r_i \rrbracket_p; \llbracket r_{i,0} \rrbracket_T, \dots, \llbracket r_{i,m-1} \rrbracket_T)\}_{i=1}^N$.

^a Within each term, the **red** highlights indicate matching indices.

Theorem 3. Protocol $\Pi_{\text{PrivEd}}(p, m)$ securely realizes functionality $\mathcal{F}_{\text{PrivEd}}[p, m]$ in the $\mathcal{F}_{\text{ABB}}[p]$ -hybrid model with abort in the presence of a fully malicious adversary controlling t corrupted parties, where p is a prime and $0 < m < \log_2 p$.

Cost Analysis. The number of $\mathcal{F}_{\text{ABB}}$ operations required by $\Pi_{\text{PrivEd}}(2^\ell, m)$ and $\Pi_{\text{PrivEd}}(p, m)$ is counted in Appendix E.4. For a large batch, both produce one private edabit of length m at the cost of a single **Input** over $\mathbb{Z}_L$ and mQ calls to **InputBit** over $\mathbb{Z}_T$, where $Q \rightarrow 1$ as the fan-in $T/2 - 1$ of $\pi_{\text{TreeAggregate}}$ grows; see Equations (17) and (18). The entire verification therefore amortizes away.

6 Instantiations of Dabits

Ignoring binary-to-arithmetic conversions, our edabits protocol achieves *quasi-linear* communication complexity in n when $t < (1-\epsilon)n$ for an arbitrary constant $\epsilon \in (0, 1)$, whereas the one from [21] requires quadratic communication. This is a significant gain, especially since most efficient MPC protocols already achieve linear communication complexity. However, binary-to-arithmetic conversions require *dabits*. These can be generated with linear communication over rings, but over fields the approach of [33] requires $\Omega(n^2 \ell)$ bits. We show how to obtain dabits with communication linear in n . Used in the edabit generation of Section 4, they yield $O(n \ell \log n)$ communication per edabit.

6.1 Dabits over Rings

Dabit generation is straightforward when the modulus is a power of two: each party locally converts a random bit sharing in the arithmetic domain to the binary domain. We present the protocol for generating dabits over rings in Protocol 11, as detailed in Section C.4. The total communication is already linear: $O(n \ell)$ bits.

6.2 Dabits over Fields

We improve dabit generation over fields by using our aggregator to emulate a shared random edabit of fixed length and then obtaining arithmetic sharings of its bits by bit decomposition, retaining only the bits within the desired length and discarding the higher ones. We choose private edabits of length $m = \lfloor \log(p/(nd)) \rfloor$, so that aggregating at most nd inputs does not wrap around modulo p . The lowest m bits of the arithmetic sum s therefore agree with those of the corresponding integer sum. Together with the randomness extraction of Section 4.1, these bits define a virtual random edabit with value $s \bmod 2^m$, without explicitly computing an arithmetic sharing of this value.

Specifically, we use a simplified aggregator $\pi_{\text{AggEdabit}}^*$ that omits Steps 4 and 5 of $\pi_{\text{AggEdabit}}$ and returns $\llbracket s \rrbracket_p$ together with the binary sharings $(\llbracket s_0 \rrbracket_2, \dots, \llbracket s_{m-1} \rrbracket_2)$ from Step 3. We then bit-decompose $\llbracket s \rrbracket_p$ over the field and pair the lowest m arithmetic bit sharings with these binary sharings, obtaining m dabits without using dabits to remove the higher bits of s . The complete protocol is given in Protocol 6, with security stated in Theorem 4 and proved in Appendix D.6.

Protocol 6: $\Pi_{\text{FieldDabits}}(p)$

Dependencies: This is set in the $\mathcal{F}_{\text{PrivEd}}[p, m]$ -hybrid model, where $L = p$ is the arithmetic modulus. Let $m = \lfloor \log(p/(nd)) \rfloor < \log p$ and let d be the extension degree of the Galois field $\text{GF}(2^d)$ such that $d \geq \lceil \log n \rceil$.

Procedure: Upon receiving any input instruction related to the operations in $\mathcal{F}_{\text{PrivEd}}[p, m]$, all parties pass it to $\mathcal{F}_{\text{PrivEd}}[p, m]$. Upon receiving the input instruction $(\text{dabit}, N, (\text{id}_1, \text{id}'_1), \dots, (\text{id}_N, \text{id}'_N))$, all parties execute the following steps $N/((n-t)dm)$ times in parallel:

1. For $\beta = 1, \dots, n$, party P_β calls $\mathcal{F}_{\text{PrivEd}}[p, m]$ to obtain d private edabits

$$\{(\llbracket r_i \rrbracket_p; \llbracket r_{i,0} \rrbracket_T, \dots, \llbracket r_{i,m-1} \rrbracket_T)\}_{i=(\beta-1)d+1}^{\beta d}$$

2. For $i = 1, \dots, (n-t)d$, call

$$\pi_{\text{AggEdabit}}^*\left(\{(\llbracket r_j \rrbracket_p; \llbracket r_{j,0} \rrbracket_T, \dots, \llbracket r_{j,m-1} \rrbracket_T) \mid j \text{ s.t. } \mathbf{M}_{i,j} = 1\}\right)$$

in parallel to obtain $(\llbracket s_i \rrbracket_p; \llbracket s_{i,0} \rrbracket_2, \dots, \llbracket s_{i,m-1} \rrbracket_2)$. Here $\mathbf{M} \in \mathbb{Z}_2^{(n-t)d \times nd}$ is a block super-invertible matrix with block size d over $\mathbb{Z}_2^m$.

3. For $i = 1, \dots, (n-t)d$, call $(\llbracket b_{i,0} \rrbracket_p, \dots, \llbracket b_{i,\ell-1} \rrbracket_p) \leftarrow \pi_{\text{FieldBitDec}}(\llbracket s_i \rrbracket_p)$. Recall that $\ell = \lceil \log_2 p \rceil$.
4. Output $(\llbracket b_{i,k} \rrbracket_p; \llbracket s_{i,k} \rrbracket_2)$ for $i = 1, \dots, (n-t)d$ and $k = 0, \dots, m-1$.

Theorem 4. *Protocol $\Pi_{\text{FieldDabits}}(p)$ securely realizes functionality $\mathcal{F}_{\text{Dabits}}[p]$ in the $\mathcal{F}_{\text{PrivEd}}[p, m]$ -hybrid model with abort in the presence of a fully malicious adversary controlling at most t corrupted parties, where p is a prime, $m = \lfloor \log(p/(nd)) \rfloor \geq 1$ and $d = \lceil \log_2 n \rceil$.*

Cost of global dabits over fields. The cost is dominated by the $(n-t)d$ bit decompositions of Step 3, that is $O(n^2 d \ell^2)$ bits per iteration for $(n-t)dm$ dabits. When $t < (1-\epsilon)n$ for a constant $\epsilon \in (0, 1)$ and $\ell - \log(nd) = \Theta(\ell)$, this is $O(n\ell)$ bits per dabit, linear in both n and ℓ , as required. The detailed counts are given in Appendix E.5.

7 Comparison with Prior Work

In this section we compare the efficiency of our techniques with respect to prior work. Appendices E and F express the cost of our edabit generation and that of [21] as numbers of $\mathcal{F}_{\text{ABB}}$ operations, valid at any batch size. Our aim is to find the number of parties beyond which our protocol becomes cheaper, to quantify the gap, and to see how both depend on the corruption threshold and the batch size.

Setting. We consider n parties of which t are corrupted, for different values of the ratio t/n . In both constructions every value is secret shared among all n parties. The protocol of [21] aggregates $E = t + 1$ private edabits per global edabit, the smallest number that guarantees an honest contributor, as discussed in Remark 4; ours consumes $\frac{n}{n-t}$ of them, through the randomness extraction of Section 4.1. We take $\ell = 64$, with full-length edabits $m = \ell$ over $\mathbb{Z}_{2^{64}}$, and $m = \ell - 1$ over a 64-bit prime field, the largest length that [21] supports there. We do not vary m : full length is what applications require, and over rings it is also least favorable to us, since at $m = \ell$ neither construction consumes dabits. For our own parameters we follow Section 4.2: $d = \lceil \log_2 n \rceil$, aggregation fan-in $E = nd$, $\eta = \lceil 2 \log_2 nd \rceil$, $T = 2^\eta$ and block size $A = \log_2 nd$.

Cost of Coin. We do not charge for calls to Coin. In practice one can use a short seed per call and expand it locally with a pseudorandom generator, paying a security parameter's worth of bits per call rather than per bit produced. We avoid PRGs inside our own protocols only because they

complicate the security proofs, not because they are impractical. Prior work makes the same tacit assumption: [21] observes that its coin tossing “can be implemented, for example, with hash-based commitments in the random oracle model”, and likewise does not account for its cost.⁸

Security parameter. The cut-and-choose checks of the prior work fail with probability $2^{-\kappa}$, whereas our subset-sum checks fail with probability $N/2^\kappa$, by Lemma 2 and its field counterpart. The factor N is an artifact of deriving all selection bits from the powers of a single random element, done in Section 5 to keep the proofs statistical; with the PRG approach—which we already presuppose in this section—the bound is $2^{-\kappa}$ (up to *computational* security parameters). However, for the parameter choices we keep our protocols as proven and use $\kappa' = \kappa + \log_2 N$, that is 50, 55 and 60 for the three batches, so that both attain a *provable* 2^{-40} security. This costs us almost nothing: our protocol scales very well with κ' .

Batch sizes. We consider three batch sizes, $N \in \{2^{10}, 2^{15}, 2^{20}\}$, the number of edabits produced in parallel. This is mainly relevant for the edabits of [21] and the dabits of [33], whose cut-and-choose analysis is only sound if the batch is large enough (see Appendix F). In contrast, our constructions have no minimum batch size: larger batches amortize communication better, but soundness does not depend on N . This is an important feature of our approach.

Instantiation of the ABB. Table 1 gives the communication per $\mathcal{F}_{\text{ABB}}$ operation over an x -bit domain, following the SPDZ family of protocols [20,16,31].⁹ A party inputs a value by broadcasting it masked with a preprocessed random sharing that it knows, an opening is done through a designated party, a multiplication is a Beaver evaluation and hence two openings, and RandBit is the square-root trick, one multiplication and one opening over the domain, run modulo 2^{x+2} over rings [16, Fig. 5] and modulo p over fields [18, Fig. 14]. Table 1 charges InputBit as one input over $\mathbb{Z}_T$, its asymptotic cost by Appendix B; our comparison uses the exact cost (3). The binary domain is $\mathbb{Z}_2$ itself, so a binary share is one bit. The MAC-based protocols of the SPDZ family end with a MAC check of constant complexity, which we ignore.

Finally, for simplicity we ignore the preprocessing cost, namely generating triples, input masks and random sharings for RandBit. For this we may assume pseudorandom correlation generators (PCGs) with communication sublinear in the amount of material [31].¹⁰

Input	Open	Mult	RandBit		InputBit
			over $\mathbb{Z}_{2^x}$	over $\mathbb{F}_p$	
bits	nx	$2nx$	$4nx$	$6n(x+2)$	$6nx$
					nx

Table 1: Online communication charged per $\mathcal{F}_{\text{ABB}}$ operation over a domain of x bits, among n parties, in the dishonest majority instantiation.

7.1 Dabits

We begin with our dabit generation of Section 6 (costs in Appendix E.5) against that of [33] (costs in Appendix F.1). This is only relevant for fields, since over rings dabit generation is straightforward. The improvement is of independent interest beyond edabits, since dabits have their own applications

⁸ While our protocol samples a constant number of elements of $\text{GF}(2^\kappa)$, $\text{GF}(p^{\kappa p})$ or $\text{GR}(T, \kappa)$ per check, that of [21,33] shuffles the objects entering its cut-and-choose, and supplying the latter information-theoretically would be much more expensive. Charging both for their coins would therefore increase the cost of prior work and leave ours essentially unchanged; with a PRG it also means that our protocol needs fewer coins, and hence less computation.

⁹ For completeness, we include in Appendix G.3 the comparison, but with Shamir secret-sharing for $t < n/2$.

¹⁰ Over fields this is readily available: the silent vector-OLE and OT of [8,9] were extended to the OLE and multiplication triples we need in [10], and [1] gives the multiparty, authenticated version for SPDZ-style protocols. Over $\mathbb{Z}_{2^k}$ the constructions are more recent: vector-OLE is given in [4], and pseudorandom correlation generators over $\mathbb{Z}/p^k\mathbb{Z}$ in [27]. This convention applies to both sides.

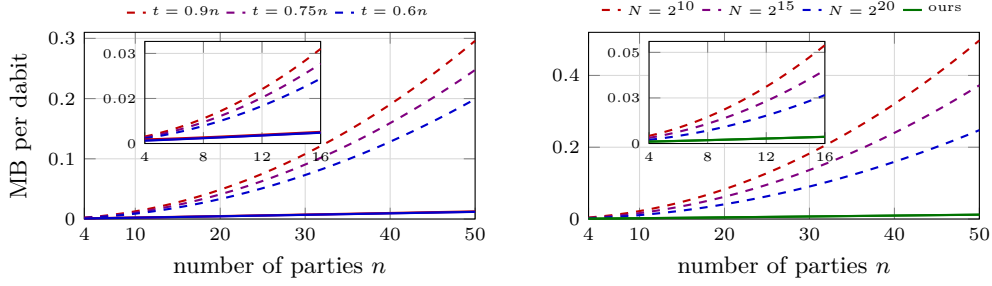


Fig. 2: Communication per field dabit of [33], dashed, against that of Protocol 6, solid, in the same color. Left: three corruption thresholds at $N = 2^{20}$. Right: the three batch sizes at $t = 0.75n$, our two curves (at $N = 2^{10}$ and $N = 2^{20}$) being indistinguishable. The insets repeat the region $n \leq 16$.

$t/n \setminus n$	$N = 2^{10}$							$N = 2^{15}$							$N = 2^{20}$						
	4	8	16	64	128	256	512	4	8	16	64	128	256	512	4	8	16	64	128	256	512
0.5	3.4	5.9	10	35	67	128	246	2.6	4.4	7.8	27	51	97	187	1.6	2.9	5.1	18	34	65	125
0.6	3.7	6.8	12	42	79	152	292	2.8	5.1	9.2	32	60	116	223	1.8	3.3	6.0	21	40	77	149
0.7	3.9	7.5	14	48	91	174	335	3.0	5.7	10	37	70	134	257	1.9	3.7	6.9	24	46	89	172
0.8	×	8.1	15	53	101	194	371	×	6.2	12	41	78	150	288	×	4.0	7.7	27	52	100	192
0.9	×	×	16	56	106	201	383	×	×	12	44	83	159	306	×	×	8.1	29	56	106	204
1 - 1/n	4.0	8.2	15	35	43	46	46	3.0	6.3	12	32	42	49	51	1.9	4.1	8.0	21	29	33	35

Table 2: The improvement factor $\mathcal{C}_{\text{dabit}}^{[33]} / \mathcal{C}_{\text{dabit}}^{\text{ours}}$ for each threshold, batch size and number of parties n . As in Fig. 2, the threshold is $t = \tau n$ as a real number, except in the last row, the maximal threshold $t = n - 1$ of Section G.2. $\times$ marks entries with $\tau n > n - 1$, which leave no honest party. Over $\mathbb{Z}_{2^\ell}$ both constructions produce a dabit with a single RandBit, so there is nothing to compare.

to binary-arithmetic conversions. Asymptotically, and for $t < (1 - \epsilon)n$ with $\epsilon \in (0, 1)$ constant, ours costs $O(n\ell)$ bits per dabit, while [33] costs $\Theta(n^2\ell)$.¹¹

Our benefits go far beyond asymptotics. Fig. 2 shows the cost of our dabit protocol against that of [33], for different batch sizes and thresholds. First, we require no minimum batch size, while achieving $\kappa = 40$ in [33] requires more expensive cut-and-choose parameters for smaller batches (see Appendix F.1). This is visible on the right of Fig. 2, where their cost increases as the batch shrinks. As the threshold decreases, it is their cost that improves faster, being proportional to the number $t + 1$ of contributing parties, while ours is nearly independent of it; on the left of Fig. 2, where we vary the threshold, this is barely perceptible because our costs are already so low. Even at $t = 0.6n$ their curve quickly deviates from ours.

Table 2 gives the improvement factor of our dabits over [33] numerically. By Fig. 9, most of our cost comes from multiplications and random bit generation, not from the aggregation and the private edabits, which are the parts that benefit from a larger honest gap. Hence our improvement grows with the threshold: a bigger threshold means more contributing parties in [33], whose cost grows linearly in that number, whereas in ours only the private edabits depend on it. The factors reach $> 300\times$ in some regimes.

7.2 Edabits

We now compare our edabit protocol with that of [21]. Over fields both constructions need dabits in a black-box way.¹² While [21] uses the dabits of [33], Section 7.1 shows ours to be much more efficient, so we give our dabits to both protocols.¹³

¹¹ At $t = n - 1$ both protocols become quadratic in n , but ours is $\Theta(n^2 \log n)$ against their $\Theta(n^2\ell)$; see Section G.2 and Table 2.

¹² The cost of dabits is only relevant for fields: over rings they are needed only when the edabit length is smaller than the modulus, and even then they are cheap.

¹³ Using [33] in [21], as that paper specifies, would make our improvement factors over fields orders of magnitude larger than those we report.

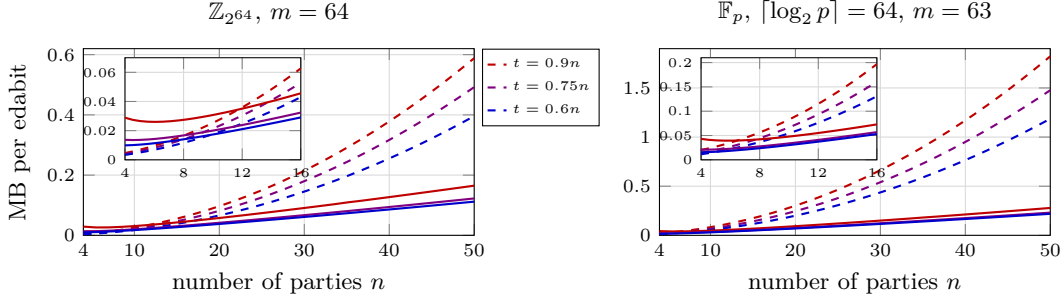


Fig. 3: Communication per global edabit at $N = 2^{20}$, with both sides using our dabits, for three corruption thresholds. Dashed: [21]. Solid: ours, in the same color as the threshold. The inset repeats the region $n \leq 16$, where the two protocols cross.

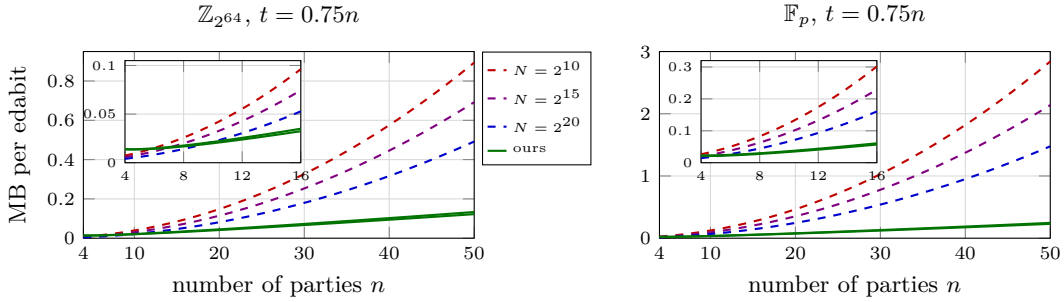


Fig. 4: The same comparison for the three batch sizes, with the bucket size that each batch forces on [21], namely $B = 5, 4$ and 3 for $N = 2^{10}, 2^{15}$ and 2^{20} . Dashed: theirs, one curve per batch. Solid: ours, drawn at $N = 2^{10}$ and $N = 2^{20}$, the two being indistinguishable here. The inset repeats the region $n \leq 16$.

Fig. 3 plots our costs against those of [21], over fields and rings, for the thresholds $t \in \{0.6n, 0.75n, 0.9n\}$.¹⁴ The crossover points in n are very low: our protocol already improves at $n \approx 10$ over rings and $n \approx 6$ over fields. The improvement peaks around $t \approx 0.8n$: as t grows, the cost of [21] grows linearly in the number $t + 1$ of private edabits it aggregates, while ours barely moves, but beyond that point our extraction rate $\frac{n}{n-t}$ degrades fast enough to eat the gain. Even at $t = 0.9n$ we benefit from $n \approx 12$ on, and the curves quickly deviate, with the costs of [21] growing quadratically while ours grow quasi-linearly.

Fig. 4 fixes the threshold to $t = 0.75n$ and varies the batch size $N \in \{2^{10}, 2^{15}, 2^{20}\}$. Like the dabits of [33], the private edabits of [21] need more expensive parameters for smaller batches at a given security level (see Appendix F), whereas ours works for any batch size with the same parameters: smaller batches make their costs grow while ours are unaffected.

Table 3 gives detailed numerical results for varying batch sizes and thresholds. The crossover points are again low, which justifies choosing our protocol in almost all settings. As for dabits, our improvements grow as the batch size decreases: at $N = 2^{10}$ they are almost twice those at $N = 2^{20}$. Unlike the dabit case, however, the factor is not monotone in the threshold: it increases up to $t \approx 0.8n$ and decreases after that.

Finally, our improvements over [21] come from two main contributions (ignoring better dabits): (1) improved private edabit generation, and (2) fewer private edabits per global edabit via randomness extraction.¹⁵ Both contribute to our improvements, as discussed in Appendix G.1.

¹⁴ Just like the dabits, for the maximal adversary case $t = n - 1$ our edabit protocol also outperforms [21] over most of the range, although over $\mathbb{Z}_{2^\ell}$ both are then asymptotically quadratic. See the last row of Table 3, and Section G.2.

¹⁵ The randomness extraction is useless for $t = n - 1$, but (1) allows us to improve over [21] even for the maximal adversary case.

$t/n \setminus n$		$N = 2^{10}$								$N = 2^{15}$								$N = 2^{20}$							
		4	8	16	64	128	256	512	4	8	16	64	128	256	512	4	8	16	64	128	256	512			
$\mathbb{Z}_{2^\ell}$	0.5	.62	1.3	2.3	6.4	11	20	37	.51	1.1	1.8	5.2	9.2	17	30	.34	.76	1.3	3.7	6.6	12	22			
	0.6	.62	1.5	2.6	7.4	13	23	43	.50	1.2	2.1	6.1	11	19	35	.34	.83	1.5	4.3	7.6	14	25			
	0.7	.57	1.5	2.7	8.0	14	25	46	.47	1.2	2.3	6.7	12	21	39	.31	.86	1.6	4.8	8.4	15	28			
	0.8	$\times$	1.4	2.7	8.1	14	26	47	$\times$	1.2	2.3	6.9	12	22	40	$\times$	.81	1.6	4.9	8.7	16	29			
	0.9	$\times$	$\times$	2.2	6.8	12	21	39	$\times$	$\times$	1.9	6.1	11	19	35	$\times$	$\times$	1.4	4.4	7.7	14	25			
$1 - 1/n$		.53	1.1	1.7	2.0	1.9	1.7	1.6	.43	.98	1.6	2.0	1.9	1.8	1.7	.29	.68	1.1	1.4	1.4	1.3	1.2			
$\mathbb{F}_p$	0.5	1.2	2.4	3.9	10	18	31	57	1.0	2.0	3.1	8.0	14	25	44	.73	1.4	2.2	5.6	9.5	17	30			
	0.6	1.3	2.7	4.4	12	21	37	67	1.1	2.2	3.5	9.4	16	29	52	.74	1.5	2.5	6.5	11	20	36			
	0.7	1.2	2.8	4.8	13	23	42	75	1.0	2.3	3.9	11	19	33	60	.71	1.6	2.7	7.4	13	23	41			
	0.8	$\times$	2.7	5.0	14	25	45	81	$\times$	2.3	4.1	12	20	36	65	$\times$	1.6	2.9	8.0	14	25	45			
	0.9	$\times$	$\times$	4.6	14	24	44	80	$\times$	$\times$	3.9	11	20	36	66	$\times$	$\times$	2.7	7.9	14	25	45			
$1 - 1/n$		1.2	2.4	4.0	7.5	11	18	32	.98	2.1	3.4	6.6	9.3	14	24	.68	1.4	2.4	4.6	6.3	9.7	16			

Table 3: The improvement factor $\mathcal{C}_{\text{edabit}}^{[21]} / \mathcal{C}_{\text{edabit}}^{\text{ours}}$ for each domain, threshold, batch size and number of parties n ; a factor below 1 means that we are more expensive. As in Fig. 3, the threshold is $t = \tau n$ as a real number, except in the last row of each block, the maximal threshold $t = n - 1$ of Section G.2. $\times$ marks entries with $\tau n > n - 1$.

Acknowledgments

Disclaimer

This paper was prepared for informational purposes in part by the Artificial Intelligence Research group of JPMorgan Chase & Co. and its affiliates ("JP Morgan") and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

References

1. Abram, D., Scholl, P.: Low-communication multiparty triple generation for SPDZ from ring-LPN. pp. 221–251 (2022). https://doi.org/10.1007/978-3-030-97121-2_9
2. Abspoel, M., Cramer, R., Damgård, I., Escudero, D., Yuan, C.: Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via Galois rings. pp. 471–501 (2019). https://doi.org/10.1007/978-3-030-36030-6_19
3. Aly, A., Orsini, E., Rotaru, D., Smart, N.P., Wood, T.: Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. Cryptology ePrint Archive, Report 2019/974 (2019), <https://eprint.iacr.org/2019/974>
4. Baum, C., Braun, L., Munch-Hansen, A., Scholl, P.: Moz $\mathbb{Z}_{2^k}$ arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. pp. 329–358 (2022). https://doi.org/10.1007/978-3-031-15985-5_12
5. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) *Advances in Cryptology — CRYPTO '91*. pp. 420–432. Springer Berlin Heidelberg, Berlin, Heidelberg (1992)
6. Belorgey, M.G., Carpov, S., Deforth, K., Jetchev, D., Sae-Tang, A., Vuille, M., Gama, N., Katz, J., Leontiadis, I., Mohammadi, M.: Manticore: A framework for efficient multiparty computation supporting real number and boolean arithmetic **36**(3), 31 (Jul 2023). <https://doi.org/10.1007/s00145-023-09464-4>
7. Bienstock, A., de Castro, L., Escudero, D., Polychroniadou, A., Takahashi, A.: Quorus: Efficient, scalable threshold ML-DSA signatures from MPC. In: 35th USENIX Security Symposium (USENIX Security 26). pp. 6405–6424. USENIX Association, Baltimore, MD (Aug 2026), <https://www.usenix.org/conference/usenixsecurity26/presentation/bienstock>
8. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. pp. 896–912 (2018). <https://doi.org/10.1145/3243734.3243868>
9. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. pp. 489–518 (2019). https://doi.org/10.1007/978-3-030-26954-8_16
10. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators from ring-LPN. pp. 387–416 (2020). https://doi.org/10.1007/978-3-030-56880-1_14
11. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. pp. 136–145 (2001). <https://doi.org/10.1109/SFCS.2001.959888>
12. Cascudo, I., Cramer, R., Xing, C., Yuan, C.: Amortized complexity of information-theoretically secure MPC revisited. pp. 395–426 (2018). https://doi.org/10.1007/978-3-319-96878-0_14
13. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. pp. 182–199 (2010). https://doi.org/10.1007/978-3-642-15317-4_13
14. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. pp. 35–50 (2010). https://doi.org/10.1007/978-3-642-14577-3_6
15. Chida, K., Genkin, D., Hamada, K., Ikarashi, D., Kikuchi, R., Lindell, Y., Nof, A.: Fast large-scale honest-majority MPC for malicious adversaries. pp. 34–64 (2018). https://doi.org/10.1007/978-3-319-96878-0_2
16. Damgård, I., Escudero, D., Frederiksen, T.K., Keller, M., Scholl, P., Volgushev, N.: New primitives for actively-secure MPC over rings with applications to private machine learning. pp. 1102–1120 (2019). <https://doi.org/10.1109/SP.2019.00078>
17. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multiparty computation for equality, comparison, bits and exponentiation. pp. 285–304 (2006). https://doi.org/10.1007/11681878_15
18. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. pp. 1–18 (2013). https://doi.org/10.1007/978-3-642-40203-6_1
19. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. pp. 572–590 (2007). https://doi.org/10.1007/978-3-540-74143-5_32
20. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. pp. 643–662 (2012). https://doi.org/10.1007/978-3-642-32009-5_38
21. Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: Improved primitives for MPC over mixed arithmetic-binary circuits. pp. 823–852 (2020). https://doi.org/10.1007/978-3-030-56880-1_29
22. Garay, J.A., Schoenmakers, B., Villegas, J.: Practical and secure solutions for integer comparison. pp. 330–342 (2007). https://doi.org/10.1007/978-3-540-71677-8_22
23. Goyal, V., Song, Y.: Malicious security comes free in honest-majority MPC. Cryptology ePrint Archive, Paper 2020/134 (2020), <https://eprint.iacr.org/2020/134>
24. Gupta, K., Jawalkar, N., Mukherjee, A., Chandran, N., Gupta, D., Panwar, A., Sharma, R.: SIGMA: Secure GPT inference with function secret sharing **2024**(4), 61–79 (Oct 2024). <https://doi.org/10.56553/popets-2024-0107>

25. Ito, M., Saito, A., Nishizeki, T.: Secret sharing schemes realizing general access structures. In: IEEE Global Telecommunication Conference (GLOBECOM). pp. 99–201. IEEE (1987)
26. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. pp. 1575–1590 (2020). <https://doi.org/10.1145/3372297.3417872>
27. Li, Z., Xing, C., Yao, Y., Yuan, C.: Efficient pseudorandom correlation generators over $\mathbb{Z}/p^k\mathbb{Z}$. pp. 207–240 (2025). https://doi.org/10.1007/978-3-032-01884-7_7
28. Liu, F., Xie, X., Yu, Y.: Scalable multi-party computation protocols for machine learning in the honest-majority setting (2024), <https://www.usenix.org/conference/usenixsecurity24/presentation/liu-fengrun>
29. Nishide, T., Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. pp. 343–360 (2007). https://doi.org/10.1007/978-3-540-71677-8_23
30. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2.0: Improved mixed-protocol secure two-party computation. pp. 2165–2182 (2021), <https://www.usenix.org/conference/usenixsecurity21/presentation/patra>
31. Rachuri, R., Scholl, P.: Le mans: Dynamic and fluid MPC for dishonest majority. pp. 719–749 (2022). https://doi.org/10.1007/978-3-031-15802-5_25
32. Rotaru, D., Smart, N.P., Tanguy, T., Vercauteren, F., Wood, T.: Actively secure setup for SPDZ **35**(1), 5 (Jan 2022). <https://doi.org/10.1007/s00145-021-09416-w>
33. Rotaru, D., Wood, T.: MArBled circuits: Mixing arithmetic and Boolean circuits with active security. pp. 227–249 (2019). https://doi.org/10.1007/978-3-030-35423-7_12
34. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (Nov 1979). <https://doi.org/10.1145/359168.359176>, <https://doi.org/10.1145/359168.359176>
35. Zhou, L., Wang, Z., Cui, H., Song, Q., Yu, Y.: Bicaptor: Two-round secure three-party non-linear computation without preprocessing for privacy-preserving machine learning. pp. 534–551 (2023). <https://doi.org/10.1109/SP46215.2023.10179449>

Supplementary Material

A Notations

We summarize the parameters used in this document in Table 4.

Table 4: Summary of parameters.

Notation	Description
n	Number of parties
t	Corruption threshold
κ	Security parameter
L	Arithmetic modulus, either a prime p or a power of two 2^ℓ
ℓ	Bit length of L
T	The size of the small domain, which is a power of two
η	Bit length of T , which is $\Theta(\log n)$
$\mathcal{L}$	The modulus of the arithmetic black-box, either L or T
E	Number of edabits to be aggregated together in $\pi_{\text{AggEdabit}}$
A	Block size of the carry-by-block trick used in $\pi_{\text{AggEdabit}}$
d	Block size for block super-invertible matrices (set to $\lceil \log_2 n \rceil$)
m	Bit length of the edabits to be generated
N	Number of edabits/dabits to be generated

B Instantiation of InputBit

We give a protocol to instantiate **InputBit** in the basic ABB model $\mathcal{F}_{\text{ABBBasic}}[L]$, a functionality containing all the operations of $\mathcal{F}_{\text{ABB}}[L]$ except **InputBit** (see Functionality B). We start with a lemma that turns the check of whether a value is a bit into a multiplication check:

Lemma 3. *For any positive integer η and $x \in \mathbb{Z}_{2^\eta}$, $x(x-1) = 0$ over $\mathbb{Z}_{2^\eta}$ if and only if $x \in \{0, 1\}$.*

Proof. If $x(x-1) = 0$ over $\mathbb{Z}_{2^\eta}$ then $x(x-1) = k \cdot 2^\eta$ over $\mathbb{Z}$ for some integer k . Note that only one of x and $x-1$ is even, so $2^\eta \mid x$ or $2^\eta \mid x-1$. When $2^\eta \mid x$, we have $x \equiv 0 \pmod{2^\eta}$, i.e., $x = 0$ over $\mathbb{Z}_{2^\eta}$. When $2^\eta \mid x-1$, we have $x-1 \equiv 0 \pmod{2^\eta}$, i.e., $x = 1$ over $\mathbb{Z}_{2^\eta}$. It is clear that $x = 0$ or $x = 1$ over $\mathbb{Z}_{2^\eta}$ implies $x(x-1) = 0$ over $\mathbb{Z}_{2^\eta}$. Therefore, $x(x-1) = 0$ over $\mathbb{Z}_{2^\eta}$ if and only if $x \in \{0, 1\}$. $\square$

Functionality 5: $\mathcal{F}_{\text{ABBBasic}}[L]$

Parameters: This functionality is parameterized by a positive integer L , where L is either 2^ℓ or a prime p . There is also a global parameter $T = \Theta(n^2)$ representing the size of the smaller domain.

Input: On input $(\text{Input}, P_i, \mathcal{L}, \text{id}, x)$ from P_i and $(\text{Input}, P_i, \mathcal{L}, \text{id})$ from all other parties, with id a fresh identifier, $\mathcal{L} \in \{L, T, 2\}$, and $x \in \mathbb{Z}_{\mathcal{L}}$, store $(\mathcal{L}, \text{id}, x)$.

Add: On input $(\text{Add}, \mathcal{L}, \text{id}, \text{id}_1, \text{id}_2, c, c_1, c_2)$ from all parties, where both id_1, id_2 are stored in memory with modulus $\mathcal{L}$, $c, c_1, c_2 \in \mathbb{Z}_{\mathcal{L}}$, retrieve $((\mathcal{L}, \text{id}_1, x), (\mathcal{L}, \text{id}_2, y))$, compute $z = c + c_1x + c_2y \in \mathbb{Z}_{\mathcal{L}}$ and store $(\mathcal{L}, \text{id}, z)$.

Mult: On input $(\text{Mult}, \mathcal{L}, \text{id}, \text{id}_1, \text{id}_2)$ from all parties, where both id_1, id_2 are stored in memory with modulus $\mathcal{L}$, retrieve $((\mathcal{L}, \text{id}_1, x), (\mathcal{L}, \text{id}_2, y))$, compute $z = x \cdot y \in \mathbb{Z}_{\mathcal{L}}$ and store $(\mathcal{L}, \text{id}, z)$.

Random Bit: On input $(\text{RandBit}, \mathcal{L}, \text{id})$ from all parties, with $\mathcal{L} \in \{L, T\}$ and a fresh identifier id , sample a uniformly random bit $r \in \{0, 1\}$ and store $(\mathcal{L}, \text{id}, r)$.

Coin: On input $(\text{Coin}, \mathcal{L}, \text{id})$ from all parties, with a fresh identifier id , sample a random coin r , store (L, id, r) , and send r to all parties.

Abort: On input (abort, P_i) from P_i , send **abort** to all other parties and halt.

Local Modular Reduction: On input $(\text{LocalMod}, \mathcal{L}, \text{id}, \text{id}')$ from all parties, where id' is stored in memory with modulus $\mathcal{L} \in \{L, T\}$, retrieve $(\mathcal{L}, \text{id}', x)$ and store $(2, \text{id}, x \bmod 2)$.^a

Open: On input $(\text{Open}, \mathcal{L}, \text{id})$ from all parties, retrieve $(\mathcal{L}, \text{id}, y)$ and send it to the adversary. Wait for an input from the adversary; if this is **deliver** then send y to all parties, otherwise send **abort**.

^a This operation is valid only for the case that $\mathcal{L}$ is a power of 2.

By Lemma 3, verifying that $b_1, \dots, b_C$ are bits reduces to checking the multiplication relations $b_i(b_i - 1) = 0$. To verify these in a batch, our protocol adapts the multiplication-verification technique of [23]. By the Schwartz–Zippel lemma, except with negligible probability, these relations hold if and only if the following single inner-product relation holds for a uniformly random $r \in \text{GR}(T, \kappa)$:

$$\langle (b_1, b_2 r, \dots, b_C r^{C-1}), (b_1 - 1, \dots, b_C - 1) \rangle = 0.$$

Although this combines the C relations into one, directly verifying the resulting C -dimensional inner product would still require $O(C)$ multiplications. To avoid this linear cost, we recursively reduce its dimension as follows. The two vectors are divided into ρ blocks $\{\alpha_i\}_{i=1}^\rho$ and $\{\beta_i\}_{i=1}^\rho$. We interpolate vector polynomials $\mathbf{f}$ and $\mathbf{g}$ from these blocks and a scalar polynomial h from the claimed inner products $\gamma_i = \langle \alpha_i, \beta_i \rangle$. The ρ inner-product relations are then compressed by evaluating $\mathbf{f}$, $\mathbf{g}$, and h at a fresh random point r' , yielding the lower-dimensional relation $\langle \mathbf{f}(r'), \mathbf{g}(r') \rangle = h(r')$. If any claimed relation is incorrect, this equality holds only with small probability. Repeating this compression eventually reduces the check to a single multiplication. In our setting, since P_{king} knows the underlying values, we can let P_{king} compute the required inner products and input them to $\mathcal{F}_{\text{ABBBasic}}[L]$. The full protocol is given in Protocol 7 and its security is stated in Theorem 5.

Protocol 7: $\mathcal{H}_{\text{InputBit}}(L, C)$

Dependencies: This protocol is set in the $\mathcal{F}_{\text{ABBBasic}}[L]$ -hybrid model, where C denotes the number of values input by P_{king} in one batch. Let $\rho \geq 2$ be the compression ratio, with $2\rho - 1 \leq 2^\kappa$. Fix a public exceptional sequence $v_1, \dots, v_{2^\kappa} \in \text{GR}(T, \kappa)$ such that $v_i - v_j$ is a unit for every $i \neq j$.

Procedure: Upon receiving an instruction corresponding to any operation supported by $\mathcal{F}_{\text{ABBBasic}}[L]$, all parties forward it to $\mathcal{F}_{\text{ABBBasic}}[L]$. Upon receiving $(\text{InputBit}, \text{P}_{\text{king}}, \text{id}_i, b_i)$ from P_{king} and $(\text{InputBit}, \text{P}_{\text{king}}, \text{id}_i)$ from every other party for each $i \in [C]$, where $\text{id}_1, \dots, \text{id}_C$ are fresh identifiers, the parties execute the following steps:

1. P_{king} inputs $b_1, \dots, b_C$ to $\mathcal{F}_{\text{ABBBasic}}[L]$ as C elements in $\mathbb{Z}_T$, obtaining $\llbracket b_1 \rrbracket_T, \dots, \llbracket b_C \rrbracket_T$.
2. All parties sample a uniformly random element $r \leftarrow \text{GR}(T, \kappa)$ by calling **Coin**.
3. All parties set

$$\llbracket \mathbf{x} \rrbracket_T^\kappa = (\llbracket b_1 \rrbracket_T \cdot 1, \llbracket b_2 \rrbracket_T \cdot r, \dots, \llbracket b_C \rrbracket_T \cdot r^{C-1}) \in (\text{GR}(T, \kappa))^C,$$

$$\llbracket \mathbf{y} \rrbracket_T^\kappa = (\llbracket b_1 \rrbracket_T - 1, \llbracket b_2 \rrbracket_T - 1, \dots, \llbracket b_C \rrbracket_T - 1) \in (\text{GR}(T, \kappa))^C,$$

$$\llbracket z \rrbracket_T^\kappa = 0 \in \text{GR}(T, \kappa).$$

4. Let C' be the dimension of $\mathbf{x}$ and $\mathbf{y}$, which is initially C . All parties repeat the following steps until the dimension of $\mathbf{x}$ and $\mathbf{y}$ is reduced to $C' = 1$:
 - (a) Parse $\llbracket \mathbf{x} \rrbracket_T^\kappa$ and $\llbracket \mathbf{y} \rrbracket_T^\kappa$ into

$$\llbracket \mathbf{x} \rrbracket_T^\kappa = (\llbracket \alpha_1 \rrbracket_T^\kappa, \dots, \llbracket \alpha_\rho \rrbracket_T^\kappa),$$

$$\llbracket \mathbf{y} \rrbracket_T^\kappa = (\llbracket \beta_1 \rrbracket_T^\kappa, \dots, \llbracket \beta_\rho \rrbracket_T^\kappa),$$

where $\alpha_i, \beta_i \in (\text{GR}(T, \kappa))^\nu$ for $1 \leq i \leq \rho$, ρ is the compression ratio, and $\nu = C'/\rho$.

- (b) For $i = 1, \dots, \rho - 1$, P_{king} computes the inner product $\gamma_i = \langle \alpha_i, \beta_i \rangle \in \text{GR}(T, \kappa)$ and inputs γ_i to $\mathcal{F}_{\text{ABBBasic}}[L]$ as κ elements in $\mathbb{Z}_T$, obtaining $\llbracket \gamma_i \rrbracket_T^\kappa$. Then all parties set $\llbracket \gamma_\rho \rrbracket_T^\kappa = \llbracket z \rrbracket_T^\kappa - \sum_{i=1}^{\rho-1} \llbracket \gamma_i \rrbracket_T^\kappa$.
- (c) Let $\mathbf{f}(\cdot), \mathbf{g}(\cdot)$ be vectors of degree- $(\rho - 1)$ polynomials over $\text{GR}(T, \kappa)$ such that $\mathbf{f}(v_i) = \alpha_i$ and $\mathbf{g}(v_i) = \beta_i$ for all $i = 1, \dots, \rho$. All parties compute all the coefficients of $\mathbf{f}(\cdot)$ and $\mathbf{g}(\cdot)$, which are linear combinations of $\{\llbracket \alpha_i \rrbracket_T^\kappa\}_{i \in [\rho]}$ and $\{\llbracket \beta_i \rrbracket_T^\kappa\}_{i \in [\rho]}$ respectively.
- (d) P_{king} locally computes all the coefficients of $\mathbf{f}(\cdot)$ and $\mathbf{g}(\cdot)$. Then P_{king} locally computes $\gamma_i = \langle \mathbf{f}(v_i), \mathbf{g}(v_i) \rangle \in \text{GR}(T, \kappa)$ and inputs γ_i to $\mathcal{F}_{\text{ABBBasic}}[L]$, obtaining $\llbracket \gamma_i \rrbracket_T^\kappa$ for $i = \rho + 1, \dots, 2\rho - 1$.

- (e) Let $h(\cdot)$ be a degree- $2(\rho - 1)$ polynomial over $\text{GR}(T, \kappa)$ such that $h(v_i) = \gamma_i$ for all $i = 1, \dots, 2\rho - 1$. All parties compute all the coefficients of $h(\cdot)$, which are linear combinations of $\{\llbracket \gamma_i \rrbracket_T^\kappa\}_{i \in [2\rho-1]}$.
- (f) All parties sample a uniformly random element r' from the public exceptional sequence by calling **Coin**. If $\exists i \in [\rho] : r' = v_i$, all parties abort.
- (g) All parties locally compute $\llbracket \mathbf{f}(r') \rrbracket_T^\kappa$, $\llbracket \mathbf{g}(r') \rrbracket_T^\kappa$, and $\llbracket h(r') \rrbracket_T^\kappa$, and then reset

$$\llbracket \mathbf{x} \rrbracket_T^\kappa = \llbracket \mathbf{f}(r') \rrbracket_T^\kappa, \quad \llbracket \mathbf{y} \rrbracket_T^\kappa = \llbracket \mathbf{g}(r') \rrbracket_T^\kappa, \quad \llbracket z \rrbracket_T^\kappa = \llbracket h(r') \rrbracket_T^\kappa.$$

Here $\mathbf{x}, \mathbf{y} \in (\text{GR}(T, \kappa))^\nu$ and $z \in \text{GR}(T, \kappa)$.

Let $\llbracket x^* \rrbracket_T^\kappa, \llbracket y^* \rrbracket_T^\kappa, \llbracket z^* \rrbracket_T^\kappa$ be the final values of $\llbracket \mathbf{x} \rrbracket_T^\kappa, \llbracket \mathbf{y} \rrbracket_T^\kappa, \llbracket z \rrbracket_T^\kappa$ after the above steps.

- 5. All parties compute $\llbracket x^* y^* \rrbracket_T^\kappa$ by using $O(\kappa^2)$ calls to **Mult** over $\mathbb{Z}_T$.
- 6. Open $\llbracket x^* y^* \rrbracket_T^\kappa - \llbracket z^* \rrbracket_T^\kappa$, and abort if it is not 0.

Theorem 5. *Protocol $\Pi_{\text{InputBit}}(L, C)$ securely realizes functionality $\mathcal{F}_{\text{ABB}}[L]$ in the $\mathcal{F}_{\text{ABBBasic}}[L]$ -hybrid model.*

See Section D.7 for the proof of Theorem 5.

Cost Analysis. To input C bits and verify that they are bits:

- In Step 1, C elements in $\mathbb{Z}_T$ are input to $\mathcal{F}_{\text{ABBBasic}}[L]$.
- In Step 2, one random element is sampled from $\text{GR}(T, \kappa)$, which requires $\kappa \log_2 T = \kappa \eta$ coins.
- In Step 4, $\lceil \log_\rho C \rceil$ rounds are executed, and in each round, $2\rho - 2$ elements of $\text{GR}(T, \kappa)$, i.e., $\kappa(2\rho - 2)$ elements of $\mathbb{Z}_T$, are input to $\mathcal{F}_{\text{ABBBasic}}[L]$ and one random element is sampled from $\text{GR}(T, \kappa)$, which requires $\kappa \eta$ coins.
- Steps 5–6 cost one multiplication over $\text{GR}(T, \kappa)$, which we charge as the κ^2 multiplication gates over $\mathbb{Z}_T$ of schoolbook multiplication, and κ calls to **Open** over $\mathbb{Z}_T$.

Therefore, the total cost is $C + \kappa(2\rho - 2)\lceil \log_\rho C \rceil$ input gates over $\mathbb{Z}_T$, κ^2 multiplication gates over $\mathbb{Z}_T$, κ openings over $\mathbb{Z}_T$, and $\kappa \eta(1 + \lceil \log_\rho C \rceil)$ random coins. Writing each of these at its $\mathcal{F}_{\text{ABB}}$ price, and not charging the coins, the cost of one batch of C bits supplied by a single dealer is

$$\mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T}(C) = \left(C + \kappa(2\rho - 2)\lceil \log_\rho C \rceil \right) \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_T} + \kappa^2 \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_T} + \kappa \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T}, \quad (2)$$

where the first term collects the C inputs of Step 1 together with the $2(\rho - 1)$ elements of $\text{GR}(T, \kappa)$, that is, $\kappa(2\rho - 2)$ elements of $\mathbb{Z}_T$, which Step 4(b) and Step 4(d) input in each of the $\lceil \log_\rho C \rceil$ compression rounds; the last two terms are the single multiplication over $\text{GR}(T, \kappa)$ of Step 5, charged as κ^2 multiplications over $\mathbb{Z}_T$, and its opening in Step 6, charged as κ of them. Dividing by C , one call to **InputBit** inside a batch of C bits costs on average

$$\mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T} = \left(1 + \frac{\kappa(2\rho - 2)\lceil \log_\rho C \rceil}{C} \right) \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_T} + \frac{\kappa^2}{C} \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_T} + \frac{\kappa}{C} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T}. \quad (3)$$

We drop the argument in (3): this amortized cost, not the batch cost (2), is what the equations of Appendix E refer to. Every term of (3) but the first vanishes as $1/C$, and the correction inside the first one is $O(\kappa \rho \log_\rho C / C)$ at fixed κ and ρ , so that

$$\mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T} = \left(1 + o(1) \right) \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_T}$$

as the batch grows: the whole verification is sublinear in C , and a call to **InputBit** costs, up to that vanishing correction, exactly an **Input** over $\mathbb{Z}_T$.

C Procedures and Protocols

C.1 Field Bit Decomposition

Procedure 8: $\pi_{\text{FieldBitDec}}(\llbracket a \rrbracket_p)$

1. For $i = 0, \dots, \lceil \log_2 p \rceil - 1$: $\llbracket b_i \rrbracket_p \leftarrow \text{RandBit}()$.
2. $\llbracket v \rrbracket_p \leftarrow \pi_{\text{BitLT}}(\{\llbracket b_i \rrbracket_p\}_{i=0}^{\lceil \log_2 p \rceil - 1}, p)$ and open v .
3. If $v = 0$ then start over from Step 1; otherwise continue.
4. $\llbracket b \rrbracket_p \leftarrow \sum_{i=0}^{\lceil \log_2 p \rceil - 1} 2^i \llbracket b_i \rrbracket_p$.
5. $\llbracket c \rrbracket_p \leftarrow \llbracket a \rrbracket_p - \llbracket b \rrbracket_p$ and open $\llbracket c \rrbracket_p$.
6. $\llbracket q \rrbracket_p \leftarrow 1 - \pi_{\text{BitLT}}(\{\llbracket b_i \rrbracket_p\}_{i=0}^{\lceil \log_2 p \rceil - 1}, p - c)$.
7. Let $(f_0, \dots, f_{\lceil \log_2 p \rceil - 1})$ and $(c_0, \dots, c_{\lceil \log_2 p \rceil - 1})$ be the bit-decomposition of $2^{\lceil \log_2 p \rceil} - p + c$ and c , respectively.
8. For $i = 0, \dots, \lceil \log_2 p \rceil - 1$: $\llbracket g_i \rrbracket_p \leftarrow (f_i - c_i) \cdot \llbracket q \rrbracket_p + c_i$.
9. $\{\llbracket h_i \rrbracket_p\}_{i=0}^{\lceil \log_2 p \rceil - 1} \leftarrow \pi_{\text{BitAdder}}(\{\llbracket b_i \rrbracket_p\}_{i=0}^{\lceil \log_2 p \rceil - 1}, \{\llbracket g_i \rrbracket_p\}_{i=0}^{\lceil \log_2 p \rceil - 1})$.
10. Output $(\llbracket h_0 \rrbracket_p, \dots, \llbracket h_{\lceil \log_2 p \rceil - 1} \rrbracket_p)$.

C.2 Binary to Arithmetic Conversion

Procedure 9: $\pi_{\text{BitToArith}}(\llbracket b \rrbracket_2)$

1. Call $\mathcal{F}_{\text{Dabits}}[L]$ to obtain a random dabit $(\llbracket r \rrbracket_L; \llbracket r \rrbracket_2)$.
2. Compute $\llbracket c \rrbracket_2 \leftarrow \llbracket b \rrbracket_2 + \llbracket r \rrbracket_2$ and open c .
3. Output $\llbracket b \rrbracket_L \leftarrow c + \llbracket r \rrbracket_L - 2c \cdot \llbracket r \rrbracket_L$.

C.3 Any-Range Edabits

We formalize any-range edabits using Functionality $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ below and then present a construction based on rejection sampling.

Functionality 6: $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$

Parameters: This functionality is parameterized by an arithmetic modulus L and the range of edabits P ($2 \leq P \leq L$). Let $\ell_P = \lceil \log P \rceil$ be the bit length of P .

Operations from $\mathcal{F}_{\text{Edabits}}$: This functionality inherits all the operations from $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$.

Generate Any-Range Edabits:

On input (any-range-edabit, id, id₀, ..., id_{ℓ_P-1}) from all parties, sample $r \in \mathbb{Z}_P$ uniformly at random, bit-decompose it as $r = \sum_{i=0}^{\ell_P-1} r_i 2^i$, and store $\{(2, \text{id}_i, r_i)\}_{i=0}^{\ell_P-1}$, together with (L, id, r) .

Our key idea is to apply rejection sampling to a global edabit of length ℓ_P . However, when L is a prime, it may be smaller than 2^{ℓ_P} , so an edabit of length ℓ_P cannot always be generated directly over $\mathbb{Z}_L$. On the other hand, $P \leq L$ implies that $L > 2^{\ell_P-1}$. We can therefore obtain the required edabit by generating a global edabit of length $\ell_P - 1$ and appending a global dabit as its most significant bit. This construction also works when L is a power of two, so we use it for both rings and fields in the protocol below. We present the protocol $\Pi_{\text{AnyRangeEdabits}}$ for generating any-range edabits in the $\mathcal{F}_{\text{Edabits}}$ -hybrid model. We state its security in Theorem 6 and prove it in Section D.8. With our improved method to generate global dabits in Section 6, it is easy to see that the cost of generating one any-range edabit is also $O(n\ell_P \log n)$ bits.

Protocol 10: $\Pi_{\text{AnyRangeEdabits}}(L, P)$

Dependencies: This is set in the $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$ -hybrid model where $P \in [2, L]$ is the range and $\ell_P = \lceil \log_2 P \rceil$ is the bit length of the range.

Procedure: Upon receiving any input instruction related to the operations in $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$, all parties pass them to $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$. Upon receiving the input instruction (any-range-edabit, id, id₀, ..., id_{ℓ_P-1}), the parties proceed as follows:

1. The parties call $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$ to obtain an edabit of length $\ell_P - 1$, $(\llbracket r \rrbracket_L; \llbracket r_0 \rrbracket_2, \dots, \llbracket r_{\ell_P - 2} \rrbracket_2)$, where r is uniformly random in $\mathbb{Z}_{2^{\ell_P - 1}}$.
2. The parties call $(\text{dabit}, 1, (\text{id}, \text{id}'))$ to obtain a dabit $(\llbracket s \rrbracket_L, \llbracket s \rrbracket_2)$ where $s \in \{0, 1\}$ is uniformly random.
3. $\llbracket o \rrbracket_2 \leftarrow \pi_{\text{BitLT}}(\llbracket r_0 \rrbracket_2, \dots, \llbracket r_{\ell_P - 2} \rrbracket_2, \llbracket s \rrbracket_2, (P_0, \dots, P_{\ell_P - 1}))$ where $P_i \in \{0, 1\}$ and $P = \sum_{i=0}^{\ell_P - 1} 2^i P_i$.
4. Open o . If $o \neq 1$, restart from Step 1; otherwise, continue.
5. Output $(\llbracket r \rrbracket_L + 2^{\ell_P - 1} \llbracket s \rrbracket_L; \llbracket r_0 \rrbracket_2, \dots, \llbracket r_{\ell_P - 2} \rrbracket_2, \llbracket s \rrbracket_2)$.

Theorem 6. Protocol $\Pi_{\text{AnyRangeEdabits}}(L, P)$ securely realizes the any-range edabit functionality $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ in the $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$ -hybrid model for any target range $2 \leq P \leq L$, where $\ell_P = \lceil \log_2 P \rceil$.

C.4 Ring Dabits

Protocol 11: $\Pi_{\text{RingDabits}}(2^\ell)$

Dependencies: This protocol is set in the $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ -hybrid model, where $0 < m \leq \ell$.

Procedure: Upon receiving any input instruction related to the operations in $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$, all parties pass it to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$. Upon receiving the input instruction $(\text{dabit}, N, (\text{id}_1, \text{id}'_1), \dots, (\text{id}_N, \text{id}'_N))$, all parties execute the following steps in parallel for $i = 1, \dots, N$:

1. $\llbracket r_i \rrbracket_{2^\ell} \leftarrow \text{RandBit}()$.
2. $\llbracket r_i \rrbracket_2 \leftarrow \text{LocalMod}(\llbracket r_i \rrbracket_{2^\ell})$.
3. Output $(\llbracket r_i \rrbracket_{2^\ell}; \llbracket r_i \rrbracket_2)$.

Theorem 7. Protocol $\Pi_{\text{RingDabits}}(2^\ell)$ securely realizes functionality $\mathcal{F}_{\text{Dabits}}[2^\ell]$ in the $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ -hybrid model for any $0 < m \leq \ell$.

See Appendix D.9 for the proof of Theorem 7.

D Missing Proofs

D.1 Proof of Lemma 1

Proof. Let $\overline{\mathbf{H}}$ be some $(n - t)d \times (n - t)d$ submatrix of $\overline{\mathbf{M}}$ that is obtained by selecting $n - t$ non-overlapping blocks of d consecutive columns. Let $\mathbf{H}$ be the corresponding $(n - t) \times (n - t)$ submatrix of $\mathbf{M}$ over $\text{GF}(2^d)$, defined by mapping each $d \times d$ block of $\overline{\mathbf{H}}$ back to the corresponding element of $\text{GF}(2^d)$ via ϕ^{-1} . Since $\mathbf{M}$ is super-invertible, $\mathbf{H}$ is invertible over $\text{GF}(2^d)$. Thus, there exists a matrix $\mathbf{H}^{-1}$ over $\text{GF}(2^d)$ such that $\mathbf{H}^{-1} \cdot \mathbf{H} = \mathbf{I} = \mathbf{H} \cdot \mathbf{H}^{-1}$, where $\mathbf{I}$ is the $(n - t) \times (n - t)$ identity matrix over $\text{GF}(2^d)$. Clearly $\phi(\mathbf{I})$ is the identity over $\mathbb{Z}_2$. Thus $\phi(\mathbf{H}^{-1}) \cdot \phi(\mathbf{H}) = \phi(\mathbf{H}^{-1} \cdot \mathbf{H}) = \phi(\mathbf{I}) = \phi(\mathbf{H} \cdot \mathbf{H}^{-1}) = \phi(\mathbf{H}) \cdot \phi(\mathbf{H}^{-1})$ and therefore $\overline{\mathbf{M}}$ is block super-invertible. $\square$

D.2 Proof of Theorem 1

Proof. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. We further divide $\mathcal{H} = \mathcal{H}_1 \sqcup \mathcal{H}_2$ where $|\mathcal{H}_1| = t - |\mathcal{C}|$. We will construct a simulator $\mathcal{S}$ that runs the code of the adversary $\mathcal{A}$ as a subroutine and has access to the ideal functionality $\mathcal{F}_{\text{Edabits}}[L, m]$. The simulator emulates functionalities $\mathcal{F}_{\text{Dabits}}[L]$ (including the functions inherited from $\mathcal{F}_{\text{PrivEd}}[L, m]$) and keeps track of the ids of all the values stored in them. For a value x stored in $\mathcal{F}_{\text{Dabits}}[L]$, we denote its id in the simulation by id_x . $\mathcal{S}$ also passes all messages between $\mathcal{A}$ and the environment $\mathcal{Z}$. Throughout the simulation, $\mathcal{S}$ forwards an instruction to the ideal functionality on behalf of a corrupted party only after receiving the corresponding instruction from that party while emulating $\mathcal{F}_{\text{Dabits}}[L]$.

Upon receiving $(\text{Input}, P_i, \mathcal{L}, \text{id})$, if P_i is honest, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{Edabits}}[L, m]$ on behalf of each corrupted party. If P_i is corrupted, $\mathcal{S}$ waits to receive P_i 's input value x on behalf of $\mathcal{F}_{\text{Dabits}}[L]$.

Then $\mathcal{S}$ sends $(\text{Input}, P_i, \mathcal{L}, \text{id}, x)$ to $\mathcal{F}_{\text{Edabits}}[L, m]$ on behalf of P_i and sends $(\text{Input}, P_i, \mathcal{L}, \text{id})$ to $\mathcal{F}_{\text{Edabits}}[L, m]$ on behalf of every other corrupted party.

Upon receiving $(\text{Open}, \mathcal{L}, \text{id})$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{Edabits}}[L, m]$ on behalf of each corrupted party and receives $(\mathcal{L}, \text{id}, y)$. Then after $\mathcal{S}$ receives the same instruction when emulating $\mathcal{F}_{\text{Dabits}}[L]$, $\mathcal{S}$ sends $(\mathcal{L}, \text{id}, y)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{Dabits}}[L]$. If $\mathcal{A}$ replies **deliver**, $\mathcal{S}$ sends y to all corrupted parties on behalf of $\mathcal{F}_{\text{Dabits}}[L]$ and sends **deliver** to $\mathcal{F}_{\text{Edabits}}[L, m]$. Otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties on behalf of $\mathcal{F}_{\text{Dabits}}[L]$ and sends **abort** to $\mathcal{F}_{\text{Edabits}}[L, m]$.

Upon receiving $(\text{priv-edabit}, P_{\text{king}}, N, \{(\text{id}_i; \text{id}_{i,0}, \text{id}_{i,1}, \dots, \text{id}_{i,m-1})\}_{i=1}^N)$, $\mathcal{S}$ first passes it to $\mathcal{F}_{\text{Edabits}}[L, m]$ on behalf of each corrupted party. If P_{king} is corrupted, $\mathcal{S}$ waits to receive P_{king} 's input values $r_0, \dots, r_{N-1}$ when emulating $\mathcal{F}_{\text{Dabits}}[L]$. Then $\mathcal{S}$ sends them to $\mathcal{F}_{\text{Edabits}}[L, m]$ on behalf of P_{king} .

Upon receiving an input related to other operations in $\mathcal{F}_{\text{Dabits}}[L]$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{Edabits}}[L, m]$ on behalf of each corrupted party.

In the following, we focus on the simulation when parties receive input $(\text{edabit}, N, \{(\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1})\}_{i=1}^N)$. $\mathcal{S}$ first sends the input of each corrupted party to $\mathcal{F}_{\text{Edabits}}[L, m]$. Then $\mathcal{S}$ simulates the execution of Π_{Edabits} by executing the following steps in parallel for $N/((n-t)d)$ times. For clarity, we focus on one of these executions.

1. In Step 1, when parties call **priv-edabit**, $\mathcal{S}$ receives $\{r_i\}_{i=(\beta-1)d+1}^{\beta d}$ from $\mathcal{A}$ for each corrupted party $P_\beta \in \mathcal{C}$ and then computes the bit decomposition $(r_{i,0}, \dots, r_{i,m-1})$ of each r_i on behalf of $\mathcal{F}_{\text{Dabits}}[L]$.
2. In Step 2, when parties execute $\pi_{\text{AggEdabit}}$ with E private edabits to be aggregated,
 - (a) In Step 3(d) of $\pi_{\text{AggEdabit}}$, for each $j = 0, \dots, (m + \log E)/A - 1$, when parties open δ , $\mathcal{S}$ uniformly samples a random $\delta_j \in \mathbb{Z}_T$ and sends it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{Dabits}}[L]$ and waits for its response. If the response is **deliver**, then $\mathcal{S}$ sends δ_j to all corrupted parties on behalf of $\mathcal{F}_{\text{Dabits}}[L]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and sends **abort** to $\mathcal{F}_{\text{Edabits}}[L, m]$.
 - (b) In Step 4 of $\pi_{\text{AggEdabit}}$, for each $j = m, \dots, m + \log E - 1$, when parties open c in Step 2 of $\pi_{\text{BitToArith}}$, $\mathcal{S}$ uniformly samples a random $c'_j \in \mathbb{Z}_2$ and sends it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{Dabits}}[L]$ and waits for its response. If the response is **deliver**, then $\mathcal{S}$ sends c'_j to all corrupted parties on behalf of $\mathcal{F}_{\text{Dabits}}[L]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and sends **abort** to $\mathcal{F}_{\text{Edabits}}[L, m]$.

We use a hybrid argument to show that the views of $\mathcal{Z}$ in the real and ideal worlds are indistinguishable.

- **Hybrid 0:** This is the real world execution of Π_{Edabits} .
- **Hybrid 1:** In Step 1, for each party P_β , if $P_\beta \in \mathcal{C}$, $\mathcal{S}$ receives $\{(r_i; r_{i,0}, \dots, r_{i,m-1})\}_{i=(\beta-1)d+1}^{\beta d}$ from $\mathcal{A}$; otherwise $\mathcal{S}$ uniformly samples $r_i \in \mathbb{Z}_{2^m}$ and computes its bit decomposition $(r_{i,0}, \dots, r_{i,m-1})$ for $i = (\beta-1)d+1, \dots, \beta d$. It does not change the distribution by the definition of $\mathcal{F}_{\text{PrivEd}}[L, m]$.
- **Hybrid 2:** In Step 2, when executing Step 3(b) of $\pi_{\text{AggEdabit}}$, for $0 \leq j < (m + \log E)/A$, $\mathcal{S}$ first uniformly samples $\delta_j \leftarrow \mathbb{Z}_T$. Then $\mathcal{S}$ computes $e = a - \delta_j$ and its bit decomposition $(e_0, \dots, e_{\eta-1})$ as the result of calling **RandBit** on behalf of $\mathcal{F}_{\text{Dabits}}[L]$. Then in Step 3(d) of $\pi_{\text{AggEdabit}}$, when opening δ , $\mathcal{S}$ sends $\delta = a - e = \delta_j$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{Dabits}}[L]$. It does not change any distribution since $e \in \mathbb{Z}_T$ is a uniformly random value in **Hybrid 1** and the mapping $x \mapsto a - x$ is a bijection over $\mathbb{Z}_T$ for any fixed $a \in \mathbb{Z}_T$.
- **Hybrid 3:** In Step 2, when executing Step 4 of $\pi_{\text{AggEdabit}}$, for $m \leq j < m + \log E$, $\mathcal{S}$ uniformly samples $c'_j \in \mathbb{Z}_2$ and sends it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{Dabits}}[L]$. Then $\mathcal{S}$ just stores $(L, \text{id}_{b_j}, b_j)$ when emulating $\mathcal{F}_{\text{Dabits}}[L]$. In **Hybrid 2**, each call to $\pi_{\text{BitToArith}}$ uses a fresh random dabit with underlying bit r , so the opened value $c = b_j \oplus r$ is uniformly random in $\mathbb{Z}_2$, independently of b_j and the preceding view. Equivalently, one can first sample a uniform c'_j and set $r = b_j \oplus c'_j$, which preserves the joint distribution of the mask and the opened value. Moreover, the arithmetic output of $\pi_{\text{BitToArith}}$ is $c + r - 2cr = b_j$, so directly storing b_j preserves the conversion result. Therefore, the distribution of this hybrid is the same as that of **Hybrid 2**.
- **Hybrid 4:** In Step 2, for each $i = 1, \dots, (n-t)d$, $\mathcal{S}$ samples $\delta_{i,j} \in \mathbb{Z}_T$ for $0 \leq j < (m + \log E_i)/A$ and $c'_{i,j} \in \mathbb{Z}_2$ for $m \leq j < m + \log E_i$, and sends them to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{Dabits}}[L]$, where $E_i = \sum_{j=1}^{nd} M_{i,j}$. Then $\mathcal{S}$ directly computes $(s_1, \dots, s_{(n-t)d}) = M(r_1, \dots, r_{nd})$ in $\mathbb{Z}_{2^m}$ and s_i 's binary representation $(s_{i,0}, \dots, s_{i,m-1})$ when emulating $\mathcal{F}_{\text{Dabits}}[L]$. The simulator no longer computes any other intermediate values in $\pi_{\text{AggEdabit}}$, as they affect neither the output nor the view of $\mathcal{A}$. Thus, the only difference between this hybrid and **Hybrid 3** lies in how the output $\{(\llbracket s_i \rrbracket_L; \llbracket s_{i,0} \rrbracket_2, \dots, \llbracket s_{i,m-1} \rrbracket_2)\}_{i=1}^{(n-t)d}$ is obtained. By $\eta \geq 2 \log nd$ and the correctness

of $\pi_{\text{AggEdabit}}$, which is shown in Section 3, the distribution of this hybrid is the same as that of **Hybrid 3**.

- **Hybrid 5**: In this hybrid we introduce the ideal functionality $\mathcal{F}_{\text{Edabits}}[L, m]$. $\mathcal{S}$ no longer samples r_i or computes $(r_{i,0}, \dots, r_{i,m-1})$ for each honest party $P_\beta \in \mathcal{H}$ in Step 1. $\mathcal{S}$ does not compute $(s_1, \dots, s_{(n-t)d})$ or their bit decompositions either. Note that these values are no longer needed to generate the messages sent to $\mathcal{A}$. The difference between this hybrid and **Hybrid 4** is that $\mathcal{F}_{\text{Edabits}}[L, m]$ samples the output values s_i independently and uniformly from $\mathbb{Z}_{2^m}$ and computes their bit decompositions for $1 \leq i \leq (n-t)d$. In **Hybrid 4**, we have $s_i = \sum_{j=0}^{m-1} 2^j s_{i,j}$ and $(s_1, \dots, s_{(n-t)d}) = \mathbf{M}(r_1, \dots, r_{nd}) \bmod 2^m$. Fix the inputs of the corrupted parties and the values sampled for parties in $\mathcal{H}_1$. Since $|\mathcal{H}_2| = n-t$ and $\mathbf{M}$ is an $(n-t)d \times nd$ block super-invertible matrix over $\mathbb{Z}_{2^m}$, the map from the uniformly random values supplied by $\mathcal{H}_2$ to $(s_1, \dots, s_{(n-t)d})$ is a bijection, even after fixing the contribution of all other parties. Consequently, this output vector is jointly uniform over $\mathbb{Z}_{2^m}^{(n-t)d}$. Moreover, the opened values $\delta_{i,j}$ and $c'_{i,j}$ in **Hybrid 4** are sampled independently of these private values. The output vector therefore remains uniformly distributed given these simulated messages and the corrupted parties' inputs. Replacing it with the values sampled by $\mathcal{F}_{\text{Edabits}}[L, m]$ and their bit decompositions preserves their joint distribution with the simulated messages. Hence, this hybrid has the same distribution as **Hybrid 4**. □

D.3 Proof of Lemma 2

Proof. Let $d_i = a_i - b_i$ for $i \in \{0, \dots, N\}$ and

$$s := \min_{i: d_i \neq 0} \max_{s: 2^s | d_i} \{s\}.$$

Write $d_i = 2^s d'_i$ ($d'_0 \in \text{GR}(2^{m-s}, \kappa)$ and $d'_1, \dots, d'_N \in \mathbb{Z}_{2^{m-s}}$). Since there is at least one i such that $d_i \neq 0$, at least one d'_i is odd. Note that

$$\begin{aligned} \sum_{i=0}^N \beta^i a_i \equiv \sum_{i=0}^N \beta^i b_i \pmod{2^m} &\iff d_0 + \sum_{i=1}^N d_i \beta^i \equiv 0 \pmod{2^m} \\ &\implies d'_0 + \sum_{i=1}^N d'_i \beta^i \equiv 0 \pmod{2^{m-s}} \\ &\implies \sum_{i=0}^N (d'_i \bmod 2) \beta^i = 0 \text{ over } \text{GF}(2^\kappa). \end{aligned}$$

Since at least one d'_i is odd, the polynomial $f(X) = \sum_{i=0}^N (d'_i \bmod 2) X^i$ is a non-zero polynomial of degree at most N over $\text{GF}(2^\kappa)$. Thus it has at most N roots in $\text{GF}(2^\kappa)$. Therefore,

$$\Pr_{\beta \leftarrow \mathbb{F}_{2^\kappa}} \left[\sum_{i=0}^N \beta^i a_i \equiv \sum_{i=0}^N \beta^i b_i \pmod{2^m} \right] \leq \Pr_{\beta \leftarrow \mathbb{F}_{2^\kappa}} [f(\beta) = 0] \leq \frac{N}{2^\kappa}.$$

□

D.4 Proof of Theorem 2

Proof. We construct a simulator $\mathcal{S}$ that runs the code of the adversary $\mathcal{A}$ as a subroutine and has access to the ideal functionality $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$. The simulator emulates the functionality $\mathcal{F}_{\text{ABB}}[2^\ell]$ and keeps track of the ids of all the values stored in it. For a value x stored in $\mathcal{F}_{\text{ABB}}[2^\ell]$, we denote its id in the simulation by id_x . The simulator also passes all messages between $\mathcal{A}$ and the environment $\mathcal{Z}$.

Upon receiving $(\text{Input}, P_i, \mathcal{L}, \text{id})$, if P_i is honest, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ on behalf of each corrupted party. If P_i is corrupted, $\mathcal{S}$ waits to receive P_i 's input value x on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$. Then $\mathcal{S}$ sends $(\text{Input}, P_i, \mathcal{L}, \text{id}, x)$ to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ on behalf of P_i and sends $(\text{Input}, P_i, \mathcal{L}, \text{id})$ to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ on behalf of every other corrupted party.

Upon receiving $(\text{Open}, \mathcal{L}, \text{id})$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ on behalf of each corrupted party and receives $(\mathcal{L}, \text{id}, y)$. Then after $\mathcal{S}$ receives the same instruction when emulating $\mathcal{F}_{\text{ABB}}[2^\ell]$, $\mathcal{S}$ sends $(\mathcal{L}, \text{id}, y)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$. If $\mathcal{A}$ replies **deliver**, $\mathcal{S}$ sends y to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$ and sends **deliver** to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$. Otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$ and sends **abort** to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.

Upon receiving an input related to other operations in $\mathcal{F}_{\text{ABB}}[2^\ell]$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ on behalf of each corrupted party.

In the following, we focus on the simulation when parties receive input $(\text{priv-edabit}, P_{\text{king}}, N, \{\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1}\}_{i=1}^N)$. We divide the proof into two cases depending on whether P_{king} is corrupted or not.

P_{king} is Corrupted. $\mathcal{S}$ executes the following steps:

1. For $i = 1, \dots, N$, $\mathcal{S}$ receives r_i from $\mathcal{A}$ and receives $r_{i,0}, \dots, r_{i,m-1}$ from $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$. If there exist i, j such that $r_{i,j} \notin \{0, 1\}$, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.
2. $\mathcal{S}$ receives $\gamma = (\gamma_0, \dots, \gamma_{\kappa-1})$ from $\mathcal{A}$ and receives $\gamma_{k,0}, \dots, \gamma_{k,m-1}$ from $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$. If there exist k, j such that $\gamma_{k,j} \notin \{0, 1\}$, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.
3. $\mathcal{S}$ samples a random value $\beta \leftarrow \text{GF}(2^\kappa)$ to emulate **Coin** of $\mathcal{F}_{\text{ABB}}[2^\ell]$ and sends it to $\mathcal{A}$.
4. $\mathcal{S}$ computes $R = \gamma + \sum_{i=1}^N \beta^i r_i$ when emulating $\mathcal{F}_{\text{ABB}}[2^\ell]$ and sends it to $\mathcal{A}$. If $\mathcal{S}$ receives **deliver** from $\mathcal{A}$, $\mathcal{S}$ sends R to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.
5. In each execution of $\pi_{\text{TreeAggregate}}$:
 - (a) In Step 2(b), $\mathcal{S}$ receives $s_{i,j,0}, \dots, s_{i,j,\eta-1}$ from $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$. If there exist i, j, k such that $s_{i,j,k} \notin \{0, 1\}$, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.
 - (b) In Step 2(c), $\mathcal{S}$ computes $v_{\text{cnt}} = s_{i,j} - \sum_{k=0}^{\eta-1} 2^k s_{i,j,k}$ when emulating $\mathcal{F}_{\text{ABB}}[2^\ell]$.
 - (c) In Step 4, $\mathcal{S}$ randomly samples $(\gamma_0, \dots, \gamma_{\kappa-1}) \leftarrow \text{GR}(T, \kappa)$ to emulate **Coin** of $\mathcal{F}_{\text{ABB}}[2^\ell]$ and sends it to $\mathcal{A}$.
 - (d) In Step 6, $\mathcal{S}$ computes $\mathbf{o} = \sum_{i=1}^{\text{cnt}} \gamma^i \cdot v_i$ when emulating $\mathcal{F}_{\text{ABB}}[2^\ell]$ and sends it to $\mathcal{A}$. If $\mathcal{S}$ receives **deliver** from $\mathcal{A}$, $\mathcal{S}$ sends $\mathbf{o}$ to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.
 - (e) If there exists i such that $v_i \neq 0$, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.
6. For each $j = 0, \dots, m-1$, $\mathcal{S}$ sends $R'_j \in \text{GR}(T, \kappa)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$ and waits for its response. If $\mathcal{S}$ receives **deliver** from $\mathcal{A}$, $\mathcal{S}$ sends R'_j to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.
7. If $r_i = \sum_{j=0}^{m-1} 2^j r_{i,j}$ for all $i = 1, \dots, N$ and $\gamma_k \bmod 2^m = \sum_{j=0}^{m-1} 2^j \gamma_{k,j}$ for all $k = 0, \dots, \kappa-1$, $\mathcal{S}$ sends $(\text{priv-edabit}, P_{\text{king}}, N, \{\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1}\}_{i=1}^N)$ and $\{r_i\}_{i=1}^N$ to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ on behalf of corrupted parties; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.

We use hybrid arguments to show that the outputs of $\mathcal{Z}$ in the real and ideal worlds are indistinguishable.

- **Hybrid 0:** The simulator $\mathcal{S}$ runs the protocol $\Pi_{\text{PrivEd}}(2^\ell, m)$ honestly and emulates $\mathcal{F}_{\text{ABB}}[2^\ell]$, which corresponds to the real world execution of $\Pi_{\text{PrivEd}}(2^\ell, m)$.
- **Hybrid 1:** In each execution of $\pi_{\text{TreeAggregate}}$, after opening $\mathbf{o}$, instead of checking whether $\mathbf{o} = 0$, $\mathcal{S}$ checks whether $v_1, \dots, v_{\text{cnt}} = 0$ and aborts if any one of them is non-zero. By the Schwartz-Zippel lemma, the probability that $\mathbf{o} = 0$ but some $v_i \neq 0$ is at most

$$\frac{\text{cnt}}{2^\kappa} \leq \frac{m\eta(N+1)}{2^{\eta-1}-2} \cdot \frac{1}{2^\kappa},$$

which is negligible in κ . Therefore **Hybrid 0** and **Hybrid 1** are statistically indistinguishable.

- **Hybrid 2:** Let $\gamma'_j = (\gamma_{0,j}, \dots, \gamma_{\kappa-1,j}) \in \text{GR}(2^\ell, \kappa)$ for $j = 0, \dots, m-1$, where $\gamma_{0,j}, \dots, \gamma_{\kappa-1,j} \in \{0, 1\}$ are received from $\mathcal{A}$ in Step 2. In Step 6, after opening $R'_0, \dots, R'_{m-1}$, $\mathcal{S}$ checks whether

$$R \equiv \sum_{j=0}^{m-1} 2^j \cdot \gamma'_j + \sum_{i=1}^N \beta^i \cdot \sum_{j=0}^{m-1} 2^j r_{i,j} \pmod{2^m}$$

and aborts if the equality does not hold. Note that in this hybrid, when executing $\pi_{\text{TreeAggregate}}$, $\mathcal{S}$ does not abort only if P_{king} always provides correct bit decompositions of $s_{i,j}$ in Step 2(b)

of $\pi_{\text{TreeAggregate}}$. Then it is routine to verify that the results $S_0, \dots, S_{m-1}$ of $\pi_{\text{TreeAggregate}}$ satisfy $\sum_{j=0}^{m-1} 2^j S_j = \sum_{i=1}^{N'} \sum_{j=0}^{m-1} 2^j r_{i,j} \bmod 2^m$. Therefore in Step 5 of $\Pi_{\text{PrivEd}}(2^\ell, m)$, we have

$$\sum_{j=0}^{m-1} 2^j R'_{j,k} = \left(\sum_{j=0}^{m-1} 2^j \gamma_{k,j} + \sum_{i=1}^N \beta_k^i \cdot \sum_{j=0}^{m-1} 2^j r_{i,j} \right) \bmod 2^m$$

for all $k = 0, \dots, \kappa - 1$. Then in Step 6, we have

$$\sum_{j=0}^{m-1} 2^j \cdot R'_j \equiv \sum_{j=0}^{m-1} 2^j \cdot \gamma'_j + \sum_{i=1}^N \beta^i \cdot \sum_{j=0}^{m-1} 2^j r_{i,j} \pmod{2^m}.$$

Thus the check in **Hybrid 1** is equivalent to the check in this hybrid, and **Hybrid 1** $\equiv$ **Hybrid 2**.

- **Hybrid 3**: In Step 6, after opening $R'_0, \dots, R'_{m-1}$, if there exists $i \in \{1, \dots, N\}$ such that $r_i \neq \sum_{j=0}^{m-1} 2^j r_{i,j}$ or $\gamma \bmod 2^m \neq \sum_{j=0}^{m-1} 2^j \gamma_{i,j}$, $\mathcal{S}$ aborts. Note that $R \equiv \gamma + \sum_{i=1}^N \beta^i \cdot r_i \pmod{2^m}$, and that $R \not\equiv \sum_{j=0}^{m-1} 2^j \cdot \gamma'_j + \sum_{i=1}^N \beta^i \cdot \sum_{j=0}^{m-1} 2^j r_{i,j} \pmod{2^m}$ implies that either $r_i \neq \sum_{j=0}^{m-1} 2^j r_{i,j}$ for some $i \in \{1, \dots, N\}$ or $\gamma \bmod 2^m \neq \sum_{j=0}^{m-1} 2^j \gamma_{i,j}$. On the other hand, if $r_i \neq \sum_{j=0}^{m-1} 2^j r_{i,j}$ for some $i \in \{1, \dots, N\}$ or $\gamma \bmod 2^m \neq \sum_{j=0}^{m-1} 2^j \gamma_{i,j}$, by Lemma 2, the probability that $R \equiv \sum_{j=0}^{m-1} 2^j \cdot \gamma'_j + \sum_{i=1}^N \beta^i \cdot \sum_{j=0}^{m-1} 2^j r_{i,j} \pmod{2^m}$ is at most $N/2^\kappa$, which is negligible in κ . Therefore **Hybrid 2** and **Hybrid 3** are statistically indistinguishable.
- **Hybrid 4**: This is the ideal world execution with the simulator $\mathcal{S}$ constructed above. In this hybrid, the result $\{(\llbracket r_i \rrbracket_{2^\ell}; \llbracket r_{i,0} \rrbracket_T, \dots, \llbracket r_{i,m-1} \rrbracket_T)\}_{i=1}^N$ is generated by the functionality $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$. It should be the same since **Hybrid 3** does not abort only if $r_i = \sum_{j=0}^{m-1} 2^j r_{i,j}$ for all $i = 1, \dots, N$.

P_{king} is *Honest*. The simulator $\mathcal{S}$ proceeds as follows.

1. Upon receiving $(\text{priv-edabit}, P_{\text{king}}, N, \{\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1}\}_{i=1}^N)$ from the corrupted parties, $\mathcal{S}$ forwards the instruction to $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ on behalf of each corrupted party.
2. In Step 3, $\mathcal{S}$ samples a uniformly random $\beta \leftarrow \text{GF}(2^\kappa)$ to emulate Coin of $\mathcal{F}_{\text{ABB}}[2^\ell]$ and sends it to all corrupted parties.
3. In Step 4, $\mathcal{S}$ samples a uniformly random $R = (R_0, \dots, R_{\kappa-1}) \leftarrow \text{GR}(2^\ell, \kappa)$ and sends it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$. If $\mathcal{A}$ replies with *deliver*, $\mathcal{S}$ sends R to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$; otherwise, $\mathcal{S}$ sends *abort* to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ and terminates. For each $k = 0, \dots, \kappa - 1$, $\mathcal{S}$ then computes the bit decomposition

$$R_k \bmod 2^m = \sum_{j=0}^{m-1} 2^j R'_{j,k}$$

and sets $R'_j = (R'_{j,0}, \dots, R'_{j,\kappa-1}) \in \text{GF}(2^\kappa) \subset \text{GR}(T, \kappa)$ for $j = 0, \dots, m - 1$.

4. In each execution of $\pi_{\text{TreeAggregate}}$, $\mathcal{S}$ proceeds as follows:
 - (a) In Step 2(b), $\mathcal{S}$ receives the corresponding *InputBit* instructions from all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$.
 - (b) In Step 4, $\mathcal{S}$ samples a uniformly random element $\gamma \leftarrow \text{GR}(T, \kappa)$ to emulate Coin of $\mathcal{F}_{\text{ABB}}[2^\ell]$ and sends it to all corrupted parties.
 - (c) In Step 6, $\mathcal{S}$ sends $\mathbf{0}$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$ and waits for its response. If $\mathcal{A}$ replies with *deliver*, $\mathcal{S}$ sends $\mathbf{0}$ to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$; otherwise, $\mathcal{S}$ sends *abort* to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.
5. For each $j = 0, \dots, m - 1$, $\mathcal{S}$ sends R'_j to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$ and waits for its response. If $\mathcal{A}$ replies with *deliver*, $\mathcal{S}$ sends R'_j to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[2^\ell]$; otherwise, $\mathcal{S}$ sends *abort* to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.

The simulated view is identically distributed to the real-world view. Indeed, the values generated by $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ have the same distribution as the private edabits sampled by the honest P_{king} . Moreover, the uniform mask $\gamma \leftarrow \text{GR}(2^\ell, \kappa)$ sampled in Step 2 makes $R = \gamma + \sum_{i=1}^N r_i \cdot \beta^i$ uniformly distributed over $\text{GR}(2^\ell, \kappa)$, independently of the private edabits.

It remains to show that the executions of $\pi_{\text{TreeAggregate}}$ produce identically distributed views. Since P_{king} is honest, every bit decomposition supplied in $\pi_{\text{TreeAggregate}}$ is correct. It follows that all decomposition errors $v_1, \dots, v_{\text{cnt}}$ are zero, so the value opened in Step 6 of $\pi_{\text{TreeAggregate}}$ is always $\mathbf{0}$. Furthermore, the bounds used in $\pi_{\text{TreeAggregate}}$ ensure that no intermediate sum wraps around in $\mathbb{Z}_T$. Thus, Step 2 correctly computes the bits of each group sum while propagating its carry to the next bit position. Applying this argument inductively over the tree levels, the returned tuple $(S_j)_{j=0}^{m-1}$ satisfies

$$\sum_{j=0}^{m-1} 2^j S_j \equiv \sum_{i=1}^{N'} \sum_{j=0}^{m-1} 2^j r_{i,j} \pmod{2^m}.$$

In particular, in the execution of $\pi_{\text{TreeAggregate}}$ for coordinate k in Step 5 of $\Pi_{\text{PrivEd}}(2^\ell, m)$, the resulting values $R'_{0,k}, \dots, R'_{m-1,k}$ satisfy

$$\sum_{j=0}^{m-1} 2^j R'_{j,k} \equiv \gamma_k + \sum_{i=1}^N \beta_k^i r_i \equiv R_k \pmod{2^m}.$$

Therefore, the values $R'_0, \dots, R'_{m-1}$ opened in Step 6 of $\Pi_{\text{PrivEd}}(2^\ell, m)$ are exactly those computed by $\mathcal{S}$ from R . Since every internal check in $\pi_{\text{TreeAggregate}}$ opens $\mathbf{0}$, the values observed by the corrupted parties and the output have the same joint distribution in both executions.

D.5 Proof of Theorem 3

Proof. We construct a simulator $\mathcal{S}$ that runs the code of the adversary $\mathcal{A}$ as a subroutine and has access to the ideal functionality $\mathcal{F}_{\text{PrivEd}}[p, m]$. The simulator emulates the functionality $\mathcal{F}_{\text{ABB}}[p]$ and keeps track of the ids of all the values stored in it. For a value x stored in $\mathcal{F}_{\text{ABB}}[p]$, we denote its id in the simulation by id_x . The simulator also passes all messages between $\mathcal{A}$ and the environment $\mathcal{Z}$.

Upon receiving $(\text{Input}, P_i, \mathcal{L}, \text{id})$, if P_i is honest, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{PrivEd}}[p, m]$ on behalf of each corrupted party. If P_i is corrupted, $\mathcal{S}$ waits to receive P_i 's input value x on behalf of $\mathcal{F}_{\text{ABB}}[p]$. Then $\mathcal{S}$ sends $(\text{Input}, P_i, \mathcal{L}, \text{id}, x)$ to $\mathcal{F}_{\text{PrivEd}}[p, m]$ on behalf of P_i and sends $(\text{Input}, P_i, \mathcal{L}, \text{id})$ to $\mathcal{F}_{\text{PrivEd}}[p, m]$ on behalf of every other corrupted party.

Upon receiving $(\text{Open}, \mathcal{L}, \text{id})$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{PrivEd}}[p, m]$ on behalf of each corrupted party and receives $(\mathcal{L}, \text{id}, y)$. After $\mathcal{S}$ receives the same instruction when emulating $\mathcal{F}_{\text{ABB}}[p]$, it sends $(\mathcal{L}, \text{id}, y)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[p]$. If $\mathcal{A}$ replies with **deliver**, $\mathcal{S}$ sends y to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[p]$ and sends **deliver** to $\mathcal{F}_{\text{PrivEd}}[p, m]$. Otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[p]$ and sends **abort** to $\mathcal{F}_{\text{PrivEd}}[p, m]$.

Upon receiving an input related to any other operation in $\mathcal{F}_{\text{ABB}}[p]$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{PrivEd}}[p, m]$ on behalf of each corrupted party.

In the following, we focus on the simulation when the parties receive the private-edabit instruction $(\text{priv-edabit}, P_{\text{king}}, N, \{\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1}\}_{i=1}^N)$. We divide the proof into two cases depending on whether P_{king} is corrupted.

P_{king} is Corrupted. The simulator $\mathcal{S}$ proceeds as follows.

1. For each $i = 1, \dots, N$, $\mathcal{S}$ receives r_i and $r_{i,0}, \dots, r_{i,m-1}$ from $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[p]$. If any $r_{i,j}$ is not a bit, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.
2. For each $a = 0, \dots, \kappa_p - 1$ and $b = 0, \dots, \ell - 1$, $\mathcal{S}$ receives $\gamma_{a,b}$ and $\gamma_{a,b,0}, \dots, \gamma_{a,b,\ell-1}$ from $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[p]$. If any $\gamma_{a,b,j}$ is not a bit, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.
3. $\mathcal{S}$ samples a uniformly random $\beta \leftarrow \text{GF}(p^{\kappa_p})$ to emulate **Coin** of $\mathcal{F}_{\text{ABB}}[p]$ and sends β to $\mathcal{A}$. It then computes $\beta^1, \dots, \beta^N$, their coefficient vectors, and the bit decomposition of each coefficient as in Step 3.
4. For each $a = 0, \dots, \kappa_p - 1$ and $b = 0, \dots, \ell - 1$:
 - (a) $\mathcal{S}$ computes $R_{a,b} = \left(\gamma_{a,b} + \sum_{i=1}^N r_i \beta_{a,b}^i \right) \bmod p$ when emulating $\mathcal{F}_{\text{ABB}}[p]$. It then sends $R_{a,b}$ to $\mathcal{A}$. If $\mathcal{S}$ receives **deliver** from $\mathcal{A}$, it sends **deliver** to $\mathcal{F}_{\text{PrivEd}}[p, m]$; otherwise, it sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.
 - (b) $\mathcal{S}$ receives $q_{a,b,0}, \dots, q_{a,b, \lfloor \log N \rfloor}$ from $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[p]$. If any $q_{a,b,j}$ is not a bit, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.

- (c) In each execution of $\pi_{\text{TreeAggregate}}$:
- i. In Step 2(b), $\mathcal{S}$ receives $s_{i,j,0}, \dots, s_{i,j,\eta-1}$ from $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[p]$. If any $s_{i,j,h}$ is not a bit, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.
 - ii. In Step 4, $\mathcal{S}$ samples a uniformly random $\rho \leftarrow \text{GR}(T, \kappa)$ to emulate **Coin** of $\mathcal{F}_{\text{ABB}}[p]$ and sends ρ to $\mathcal{A}$.
 - iii. In Step 6, $\mathcal{S}$ computes $\mathbf{o} = \sum_{i=1}^{\text{cnt}} \rho^i \cdot v_i$ when emulating $\mathcal{F}_{\text{ABB}}[p]$ and sends $\mathbf{o}$ to $\mathcal{A}$. If $\mathcal{S}$ receives **deliver** from $\mathcal{A}$, it sends $\mathbf{o}$ to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[p]$; otherwise, it sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.
 - iv. If any v_i is nonzero, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.
- (d) For $j = 0, \dots, \ell_{\text{agg}} - 1$, $\mathcal{S}$ sends $R'_{a,b,j}$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[p]$ and waits for its response. If $\mathcal{S}$ receives **deliver** from $\mathcal{A}$, $\mathcal{S}$ sends $R'_{a,b,j}$ to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[p]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.
5. At last, $\mathcal{S}$ sends $(\text{priv-edabit}, \text{P}_{\text{king}}, N, \{\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1}\}_{i=1}^N)$ and $\{r_i\}_{i=1}^N$ to $\mathcal{F}_{\text{PrivEd}}[p, m]$ on behalf of corrupted parties if the following conditions hold:
- $r_i = \sum_{j=0}^{m-1} 2^j r_{i,j}$ for $i = 1, \dots, N$.
 - $\sum_{b=0}^{\ell-1} 2^b \gamma_{a,b} = \sum_{b=0}^{\ell-1} 2^b \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j}$ for all $a = 0, \dots, \kappa_p - 1$.
 - $R_{a,b} + \left(N - \sum_{j=0}^{\lfloor \log N \rfloor} 2^j q_{a,b,j}\right) p = \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j} + \sum_{i=1}^N \beta_{a,b}^i \sum_{j=0}^{m-1} 2^j r_{i,j}$ for all $a = 0, \dots, \kappa_p - 1$ and $b = 0, \dots, \ell - 1$.
- Otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.

We use a sequence of hybrids to show that the output of the environment $\mathcal{Z}$ in the real execution is indistinguishable from its output in the ideal execution.

- **Hybrid 0:** The simulator $\mathcal{S}$ runs the protocol $\Pi_{\text{PrivEd}}(p, m)$ honestly and emulates $\mathcal{F}_{\text{ABB}}[p]$, which corresponds to the real world execution of $\Pi_{\text{PrivEd}}(p, m)$.
- **Hybrid 1:** In each execution of $\pi_{\text{TreeAggregate}}$, after opening $\mathbf{o}$, instead of checking whether $\mathbf{o} = 0$, $\mathcal{S}$ checks whether $v_1, \dots, v_{\text{cnt}} = 0$ and aborts if any one of them is non-zero. By the Schwartz-Zippel lemma, the probability that $\mathbf{o} = 0$ but some $v_i \neq 0$ is at most

$$\frac{\text{cnt}}{2^\kappa} \leq \frac{\ell_{\text{agg}} \cdot \eta(N + 1 + \ell)}{2^{\eta-1} - 2} \cdot \frac{1}{2^\kappa},$$

which is negligible in κ . Therefore **Hybrid 0** and **Hybrid 1** are statistically indistinguishable.

- **Hybrid 2:** In Step 4(d), after opening $R'_{a,b,0}, \dots, R'_{a,b,\ell_{\text{agg}}-1}$, $\mathcal{S}$ checks whether

$$R_{a,b} + Np = \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j} + \sum_{i=1}^N \beta_{a,b}^i \cdot \sum_{j=0}^{m-1} 2^j r_{i,j} + p \cdot \sum_{j=0}^{\lfloor \log N \rfloor} 2^j q_{a,b,j} \quad (4)$$

over $\mathbb{Z}$ and aborts if the equality does not hold. In this hybrid, when executing $\pi_{\text{TreeAggregate}}$, $\mathcal{S}$ does not abort only if P_{king} always provides correct bit decompositions of $s_{i,j}$ in Step 2(b) of $\pi_{\text{TreeAggregate}}$. Then it is routine to verify that the results of $\pi_{\text{TreeAggregate}}$ $S_0, \dots, S_{\ell_{\text{agg}}-1}$ satisfy $\sum_{j=0}^{\ell_{\text{agg}}-1} 2^j S_j = \sum_{i=1}^{N'} \sum_{j=0}^{\ell_{\text{agg}}-1} 2^j r_{i,j} \bmod 2^{\ell_{\text{agg}}}$, which means that in Step 5 of $\Pi_{\text{PrivEd}}(p, m)$, we have

$$\begin{aligned} & \sum_{j=0}^{\ell_{\text{agg}}-1} 2^j R'_{a,b,j} \\ \equiv & \sum_{j=0}^{\ell_{\text{agg}}-1} 2^j \gamma_{a,b,j} + \sum_{i=1}^N \beta_{a,b}^i \sum_{j=0}^{\ell_{\text{agg}}-1} 2^j r_{i,j} + \sum_{u=0}^{\ell-1} p_u \sum_{j=u}^{\ell_{\text{agg}}-1} 2^j q_{a,b,j-u} \\ \equiv & \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j} + \sum_{i=1}^N \beta_{a,b}^i \sum_{j=0}^{m-1} 2^j r_{i,j} + \sum_{u=0}^{\ell-1} 2^u p_u \sum_{j=0}^{\lfloor \log N \rfloor} 2^j q_{a,b,j} \\ \equiv & \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j} + \sum_{i=1}^N \beta_{a,b}^i \sum_{j=0}^{m-1} 2^j r_{i,j} + p \sum_{j=0}^{\lfloor \log N \rfloor} 2^j q_{a,b,j} \pmod{2^{\ell_{\text{agg}}}} \end{aligned}$$

for all $a = 0, \dots, \kappa_p - 1$ and $b = 0, \dots, \ell - 1$. The values of each private edabit $\sum_{j=0}^{m-1} 2^j r_{i,j}$ and each random mask $\sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j}$ are less than 2^ℓ , and $\sum_{j=0}^{\lfloor \log N \rfloor} 2^j q_{a,b,j}$ is less than $2^{\lfloor \log N \rfloor + 1}$. So the sum of all terms on the right-hand side is less than

$$(N+1) \cdot 2^\ell + p \cdot 2^{\lfloor \log N \rfloor + 1} < 2^\ell (N+1+2N) \leq 2^{\ell_{\text{agg}}}.$$

It follows that the equation above holds without modulo reduction, i.e.

$$\sum_{j=0}^{\ell_{\text{agg}}-1} 2^j R'_{a,b,j} = \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j} + \sum_{i=1}^N \beta_{a,b}^i \sum_{j=0}^{m-1} 2^j r_{i,j} + p \sum_{j=0}^{\lfloor \log N \rfloor} 2^j q_{a,b,j}.$$

Thus the check in Step 4(d) of **Hybrid 1** is equivalent to the check in this hybrid, and **Hybrid 1** $\equiv$ **Hybrid 2**.

– **Hybrid 3**: In Step 5, if any of the following conditions are violated, $\mathcal{S}$ aborts:

- $r_i = \sum_{j=0}^{m-1} 2^j r_{i,j}$ for $i = 1, \dots, N$.
- $\sum_{b=0}^{\ell-1} 2^b \gamma_{a,b} = \sum_{b=0}^{\ell-1} 2^b \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j}$ for all $a = 0, \dots, \kappa_p - 1$.
- $R_{a,b} + \left(N - \sum_{j=0}^{\lfloor \log N \rfloor} 2^j q_{a,b,j} \right) p = \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j} + \sum_{i=1}^N \beta_{a,b}^i \sum_{j=0}^{m-1} 2^j r_{i,j}$ for all $a = 0, \dots, \kappa_p - 1$ and $b = 0, \dots, \ell - 1$.

The third condition is exactly the equation that $\mathcal{S}$ checks in Step 4(d) of **Hybrid 2**, so $\mathcal{S}$ will abort in **Hybrid 3** if it aborts in **Hybrid 2**. On the other hand, if $\mathcal{S}$ aborts in **Hybrid 3** but not in **Hybrid 2**, it must be that one of the first two conditions is violated but the third condition (which is equivalent to Equation (4)) holds. By Step 4(a), we have $R_{a,b} \equiv \gamma_{a,b} + \sum_{i=1}^N \beta_{a,b}^i \cdot r_i \pmod{p}$ for all a, b . It implies that

$$\sum_{b=0}^{\ell-1} 2^b \gamma_{a,b} + \sum_{i=1}^N \beta_a^i r_i \equiv \sum_{b=0}^{\ell-1} 2^b \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j} + \sum_{i=1}^N \beta_a^i \sum_{j=0}^{m-1} 2^j r_{i,j} \pmod{p}$$

for all $a = 0, \dots, \kappa_p - 1$, where $\beta_a^i = \sum_{b=0}^{\ell-1} 2^b \beta_{a,b}^i$ is uniformly sampled from $\mathbb{F}_p$. Let

$$\begin{aligned} \Gamma &= \left(\sum_{b=0}^{\ell-1} 2^b \gamma_{0,b}, \dots, \sum_{b=0}^{\ell-1} 2^b \gamma_{\kappa_p-1,b} \right) \in \text{GF}(p^{\kappa_p}), \\ \Gamma' &= \left(\sum_{b=0}^{\ell-1} 2^b \sum_{j=0}^{\ell-1} 2^j \gamma_{0,b,j}, \dots, \sum_{b=0}^{\ell-1} 2^b \sum_{j=0}^{\ell-1} 2^j \gamma_{\kappa_p-1,b,j} \right) \in \text{GF}(p^{\kappa_p}). \end{aligned}$$

Then we get an equation in $\text{GF}(p^{\kappa_p})$:

$$\Gamma \cdot \beta^0 + \sum_{i=1}^N r_i \cdot \beta^i = \Gamma' \cdot \beta^0 + \sum_{i=1}^N \left(\sum_{j=0}^{m-1} 2^j r_{i,j} \right) \cdot \beta^i.$$

By the Schwartz–Zippel lemma in $\text{GF}(p^{\kappa_p})$, the probability that this equation holds for random $\beta \in \text{GF}(p^{\kappa_p})$ is at most $N/p^{\kappa_p} \leq N/2^\kappa$ if there is at least one i such that $r_i \neq \sum_{j=0}^{m-1} 2^j r_{i,j}$ or there is at least one a such that $\sum_{b=0}^{\ell-1} 2^b \gamma_{a,b} \neq \sum_{b=0}^{\ell-1} 2^b \sum_{j=0}^{\ell-1} 2^j \gamma_{a,b,j}$, which is negligible. Therefore the probability that **Hybrid 3** aborts but **Hybrid 2** does not is also negligible, and **Hybrid 2** and **Hybrid 3** are statistically indistinguishable.

– **Hybrid 4**: This is the ideal world execution with the simulator $\mathcal{S}$ constructed above. In this hybrid, the result $\{(\llbracket r_i \rrbracket_p; \llbracket r_{i,0} \rrbracket_T, \dots, \llbracket r_{i,m-1} \rrbracket_T)\}_{i=1}^N$ is generated by the functionality $\mathcal{F}_{\text{PrivEd}}[p, m]$. It should be the same since **Hybrid 3** does not abort only if $r_i = \sum_{j=0}^{m-1} 2^j r_{i,j}$ for all $i = 1, \dots, N$.

P_{king} is *Honest*. The simulator $\mathcal{S}$ proceeds as follows.

1. Upon receiving $(\text{priv-edabit}, \text{P}_{\text{king}}, N, \{\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1}\}_{i=1}^N)$ from the corrupted parties, $\mathcal{S}$ forwards the instruction to $\mathcal{F}_{\text{PrivEd}}[p, m]$ on behalf of each corrupted party.
2. In Step 3, $\mathcal{S}$ samples a uniformly random $\beta \leftarrow \text{GF}(p^{\kappa_p})$ to emulate Coin of $\mathcal{F}_{\text{ABB}}[p]$ and sends it to all corrupted parties.

3. In Step 4(a), for every a, b , $\mathcal{S}$ samples a uniformly random $R_{a,b} \leftarrow \mathbb{F}_p$ and sends it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[p]$. If $\mathcal{A}$ replies with **deliver**, $\mathcal{S}$ sends $R_{a,b}$ to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[p]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.
4. In Step 4(c), for each execution of $\pi_{\text{TreeAggregate}}$, $\mathcal{S}$ proceeds as follows:
 - (a) In Step 2(b), $\mathcal{S}$ receives the corresponding **InputBit** instructions from all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[p]$.
 - (b) In Step 4, $\mathcal{S}$ samples a uniformly random element $\gamma \leftarrow \text{GR}(T, \kappa)$ to emulate **Coin** of $\mathcal{F}_{\text{ABB}}[p]$ and sends it to all corrupted parties.
 - (c) In Step 6, $\mathcal{S}$ sends **0** to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[p]$ and waits for its response. If $\mathcal{A}$ replies with **deliver**, $\mathcal{S}$ sends **0** to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[p]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.
5. In Step 4(d), for every a, b , $\mathcal{S}$ computes the bit decomposition $R'_{a,b,0}, \dots, R'_{a,b,\ell_{\text{agg}}-1}$ of $R_{a,b} + Np$ such that $R_{a,b} + Np = \sum_{j=0}^{\ell_{\text{agg}}-1} 2^j R'_{a,b,j}$. Then, for each $j = 0, \dots, \ell_{\text{agg}}-1$, $\mathcal{S}$ sends $R'_{a,b,j}$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABB}}[p]$ and waits for its response. If $\mathcal{A}$ replies with **deliver**, $\mathcal{S}$ sends $R'_{a,b,j}$ to all corrupted parties on behalf of $\mathcal{F}_{\text{ABB}}[p]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{PrivEd}}[p, m]$.

The simulated view is identically distributed to the real-world view. Indeed, the values generated by $\mathcal{F}_{\text{PrivEd}}[p, m]$ have the same distribution as the private edabits sampled by the honest P_{king} . Moreover, for every a, b , the independently sampled uniform mask $\gamma_{a,b} \leftarrow \mathbb{F}_p$ makes

$$R_{a,b} = \gamma_{a,b} + \sum_{i=1}^N r_i \cdot \beta_{a,b}^i \pmod{p}$$

uniformly distributed over $\mathbb{F}_p$, independently of the private edabits.

It remains to show that the executions of $\pi_{\text{TreeAggregate}}$ produce identically distributed views. To this end, fix a, b and let

$$X_{a,b} := \gamma_{a,b} + \sum_{i=1}^N r_i \cdot \beta_{a,b}^i$$

where the sum is taken over $\mathbb{Z}$. Since $\gamma_{a,b}, r_i \in \{0, \dots, p-1\}$ and $\beta_{a,b}^i \in \{0, 1\}$, we have

$$0 \leq X_{a,b} \leq (N+1)(p-1) < (N+1)p.$$

Since $R_{a,b}$ is the reduction of $X_{a,b}$ modulo p and $0 \leq R_{a,b} < p$, the integer $q'_{a,b}$ satisfying $R_{a,b} + q'_{a,b}p = X_{a,b}$ is exactly $\lfloor X_{a,b}/p \rfloor$ and thus lies in $\{0, \dots, N\}$. Hence, $q_{a,b} = N - q'_{a,b} \geq 0$ admits a bit decomposition of length $\lfloor \log N \rfloor + 1$. Since P_{king} is honest, all bit decompositions supplied in $\pi_{\text{TreeAggregate}}$ are correct. It follows that all decomposition errors $v_1, \dots, v_{\text{cnt}}$ are zero, so the value opened in Step 6 of $\pi_{\text{TreeAggregate}}$ is always **0**.

The bounds used in $\pi_{\text{TreeAggregate}}$ ensure that no intermediate sum wraps around over $\mathbb{Z}_T$. Here, however, we need to recover the full integer sum, rather than only its lowest m bits as in the ring case. By the choice $\ell_{\text{agg}} = \ell + \lceil \log(3N+1) \rceil$ and the fact that $p < 2^\ell$, we have

$$X_{a,b} + pq_{a,b} = R_{a,b} + Np < (N+1)p < (N+1)2^\ell \leq 2^{\ell_{\text{agg}}}.$$

Thus, the final aggregate does not wrap around in $\mathbb{Z}_{2^{\ell_{\text{agg}}}}$. For every a, b , the procedure therefore correctly aggregates the selected private edabits, the mask, and the shifted binary representations of $2^u p u q_{a,b}$, and its output satisfies the following equality over $\mathbb{Z}$:

$$\begin{aligned} \sum_{j=0}^{\ell_{\text{agg}}-1} 2^j R'_{a,b,j} &= \gamma_{a,b} + \sum_{i=1}^N \beta_{a,b}^i r_i + \sum_{u=0}^{\ell-1} 2^u p u q_{a,b} \\ &= X_{a,b} + pq_{a,b} = R_{a,b} + Np. \end{aligned}$$

Consequently, the values opened in Step 4(d) have exactly the same distribution as those generated by $\mathcal{S}$. Since every internal check in $\pi_{\text{TreeAggregate}}$ opens **0**, the values observed by the corrupted parties and the output have the same joint distribution in both executions. $\square$

D.6 Proof of Theorem 4

Proof. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. We further divide $\mathcal{H} = \mathcal{H}_1 \sqcup \mathcal{H}_2$ where $|\mathcal{H}_1| = t - |\mathcal{C}|$. We will construct a simulator $\mathcal{S}$ that runs the code of the adversary $\mathcal{A}$ as a subroutine and has access to the ideal functionality $\mathcal{F}_{\text{Dabits}}[p]$. The simulator emulates the functionality $\mathcal{F}_{\text{PrivEd}}[p, m]$ (including the functions inherited from $\mathcal{F}_{\text{ABB}}[p]$) and keeps track of the ids of all the values stored in it. For a value x stored in $\mathcal{F}_{\text{PrivEd}}[p, m]$, we denote its id in the simulation by id_x . $\mathcal{S}$ also passes all messages between $\mathcal{A}$ and the environment $\mathcal{Z}$. Throughout the simulation, $\mathcal{S}$ forwards an instruction to the ideal functionality on behalf of a corrupted party only after receiving the corresponding instruction from that party while emulating $\mathcal{F}_{\text{PrivEd}}[p, m]$.

Upon receiving $(\text{Input}, P_i, \mathcal{L}, \text{id})$, if P_i is honest, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{Dabits}}[p]$ on behalf of each corrupted party. If P_i is corrupted, $\mathcal{S}$ waits to receive P_i 's input value x when emulating $\mathcal{F}_{\text{PrivEd}}[p, m]$. Then $\mathcal{S}$ sends $(\text{Input}, P_i, \mathcal{L}, \text{id}, x)$ to $\mathcal{F}_{\text{Dabits}}[p]$ on behalf of P_i and sends $(\text{Input}, P_i, \mathcal{L}, \text{id})$ to $\mathcal{F}_{\text{Dabits}}[p]$ on behalf of every other corrupted party.

Upon receiving $(\text{Open}, \mathcal{L}, \text{id})$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{Dabits}}[p]$ on behalf of each corrupted party and receives $(\mathcal{L}, \text{id}, y)$. Then after $\mathcal{S}$ receives the same instruction when emulating $\mathcal{F}_{\text{PrivEd}}[p, m]$, $\mathcal{S}$ sends $(\mathcal{L}, \text{id}, y)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$. If $\mathcal{A}$ replies **deliver**, $\mathcal{S}$ sends y to all corrupted parties on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$ and sends **deliver** to $\mathcal{F}_{\text{Dabits}}[p]$. Otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$ and sends **abort** to $\mathcal{F}_{\text{Dabits}}[p]$.

Upon receiving $(\text{priv-edabit}, P_{\text{king}}, N, \{(\text{id}_i; \text{id}_{i,0}, \text{id}_{i,1}, \dots, \text{id}_{i,m-1})\}_{i=1}^N)$, $\mathcal{S}$ first passes it to $\mathcal{F}_{\text{Dabits}}[p]$ on behalf of each corrupted party. If P_{king} is corrupted, $\mathcal{S}$ waits to receive P_{king} 's input values $r_1, \dots, r_N$ when emulating $\mathcal{F}_{\text{PrivEd}}[p, m]$. Then $\mathcal{S}$ sends them to $\mathcal{F}_{\text{Dabits}}[p]$ on behalf of P_{king} .

Upon receiving an input related to other operations in $\mathcal{F}_{\text{PrivEd}}[p, m]$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{Dabits}}[p]$ on behalf of each corrupted party.

In the following, we focus on the simulation when parties receive input $(\text{dabit}, N, (\text{id}_1, \text{id}'_1), \dots, (\text{id}_N, \text{id}'_N))$. $\mathcal{S}$ first sends the input of each corrupted party to $\mathcal{F}_{\text{Dabits}}[p]$. Then $\mathcal{S}$ simulates the execution of $\Pi_{\text{FieldDabits}}$ by executing the following steps in parallel for $N/((n-t)d)m$ times.

1. In Step 1, when parties call $\mathcal{F}_{\text{PrivEd}}[p, m]$, $\mathcal{S}$ receives $\{(r_i; r_{i,0}, \dots, r_{i,m-1})\}_{i=(\beta-1)d+1}^{\beta d}$ from $\mathcal{A}$ for each corrupted party $P_\beta \in \mathcal{C}$ when emulating $\mathcal{F}_{\text{PrivEd}}[p, m]$.
2. In Step 2, for each $i = 1, \dots, (n-t)d$, when parties execute $\pi_{\text{AggEdabit}}^*$ with E private edabits to be aggregated:
 - In Step 3(d) of $\pi_{\text{AggEdabit}}^*$, for each $j = 0, \dots, (m + \log E)/A - 1$, when parties open δ , $\mathcal{S}$ uniformly samples a random $\delta_{i,j} \in \mathbb{Z}_T$ and sends it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$ and waits for its response. If the response is **deliver**, then $\mathcal{S}$ sends $\delta_{i,j}$ to all corrupted parties on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and sends **abort** to $\mathcal{F}_{\text{Dabits}}[p]$.
3. In Step 3, for each $i = 1, \dots, (n-t)d$, $\mathcal{S}$ emulates as follows when parties execute $\pi_{\text{FieldBitDec}}$:
 - (a) In Step 2 of $\pi_{\text{FieldBitDec}}$, $\mathcal{S}$ uniformly samples $x_i \in \mathbb{Z}_{2^\ell}$ and sets $v_i = [x_i < p]$. Then $\mathcal{S}$ sends v_i to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$ and waits for its response. If the response is **deliver**, then $\mathcal{S}$ sends v_i to all corrupted parties on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and sends **abort** to $\mathcal{F}_{\text{Dabits}}[p]$. $\mathcal{S}$ repeats this step until $v_i = 1$.
 - (b) When parties open c in Step 5 of $\pi_{\text{FieldBitDec}}$, $\mathcal{S}$ uniformly samples a random $c_i \in \mathbb{F}_p$ and sends it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$ and waits for its response. If the response is **deliver**, then $\mathcal{S}$ sends c_i to all corrupted parties on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and sends **abort** to $\mathcal{F}_{\text{Dabits}}[p]$.

We use a hybrid argument to show that the views of $\mathcal{Z}$ in the real and ideal worlds are indistinguishable.

- **Hybrid 0:** $\mathcal{S}$ runs $\Pi_{\text{FieldDabits}}$ and emulates $\mathcal{F}_{\text{PrivEd}}[p, m]$ honestly, which corresponds to the real world execution.
- **Hybrid 1:** In Step 1, for each party P_β , if $P_\beta \in \mathcal{C}$, $\mathcal{S}$ receives $\{(r_i; r_{i,0}, \dots, r_{i,m-1})\}_{i=(\beta-1)d+1}^{\beta d}$ from $\mathcal{A}$; otherwise $\mathcal{S}$ uniformly samples $r_i \in \mathbb{Z}_{2^m}$ and computes its bit decomposition $(r_{i,0}, \dots, r_{i,m-1})$ for $i = (\beta-1)d+1, \dots, \beta d$. It does not change the distribution by the definition of $\mathcal{F}_{\text{PrivEd}}[p, m]$.
- **Hybrid 2:** In Step 2, for each $i = 1, \dots, (n-t)d$, consider the execution of $\pi_{\text{AggEdabit}}^*$ with E private edabits to be aggregated. In Step 3(b) of $\pi_{\text{AggEdabit}}^*$, for $0 \leq j < (m + \log E)/A$, $\mathcal{S}$ first uniformly samples $\delta_{i,j} \leftarrow \mathbb{Z}_T$. Then $\mathcal{S}$ computes $e = a - \delta_{i,j}$ over $\mathbb{Z}_T$ and its bit decomposition $(e_0, \dots, e_{\eta-1})$ as the result of calling **RandBit** on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$. In Step 3(d) of $\pi_{\text{AggEdabit}}^*$,

when opening δ , $\mathcal{S}$ sends $\delta = a - e = \delta_{i,j}$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$. This does not change the distribution, since $e \in \mathbb{Z}_T$ is uniformly random in **Hybrid 1** and the mapping $x \mapsto a - x$ is a bijection over $\mathbb{Z}_T$ for any fixed $a \in \mathbb{Z}_T$. Therefore, the distribution of this hybrid is the same as that of **Hybrid 1**.

- **Hybrid 3:** In Step 2, for each $i = 1, \dots, (n-t)d$, $\mathcal{S}$ uniformly samples $\delta_{i,j} \in \mathbb{Z}_T$ for $0 \leq j < (m + \log E_i)/A$ and sends them to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{PrivEd}}[p, m]$, where $E_i = \sum_{j=1}^{nd} M_{i,j}$. Then $\mathcal{S}$ directly computes $(s_1, \dots, s_{(n-t)d}) = \mathbf{M}(r_1, \dots, r_{nd})$ in $\mathbb{Z}$ and the lowest m bits of each s_i 's binary representation $(s_{i,0}, \dots, s_{i,m-1})$ when emulating $\mathcal{F}_{\text{PrivEd}}[p, m]$. The simulator no longer computes any other intermediate values in $\pi_{\text{AggEdabit}}^*$, as they affect neither the output nor the view of $\mathcal{A}$. Thus, the only difference between this hybrid and **Hybrid 2** lies in how the output of $\pi_{\text{AggEdabit}}^*$ is obtained. Note that $m \leq \log(p/(nd))$ and $s_i < nd \cdot 2^m \leq p$, so the arithmetic sum over $\mathbb{F}_p$ agrees with the integer sum. Moreover, $\pi_{\text{AggEdabit}}^*$ omits the conversion and subtraction of the higher bits, so it returns this arithmetic sum together with sharings of its lowest m bits. By $\eta \geq 2 \log nd$ and the correctness of $\pi_{\text{AggEdabit}}$, shown in Section 3, these are exactly the values computed in this hybrid. Therefore, the distribution of this hybrid is the same as that of **Hybrid 2**.

- **Hybrid 4:** In Step 3, for $1 \leq i \leq (n-t)d$,
 - (1) $\mathcal{S}$ uniformly samples $x_i \leftarrow \mathbb{Z}_{2^\ell}$ and sets $v_i = [x_i < p]$. Then $\mathcal{S}$ sends v_i to $\mathcal{A}$. $\mathcal{S}$ repeats this step until $v_i = 1$.
 - (2) $\mathcal{S}$ uniformly samples $c_i \in \mathbb{F}_p$ and sends it to $\mathcal{A}$.
 - (3) $\mathcal{S}$ computes the bit decomposition $(b_{i,0}, \dots, b_{i,\ell-1})$ of s_i .

By the definition of **RandBit**, the correctness of $\pi_{\text{FieldBitDec}}$ and $s_{i,j} \in \{0, 1\}$ for all $1 \leq i \leq (n-t)d$ and $j < m$, the distribution of this hybrid is the same as that of **Hybrid 3**.

- **Hybrid 5:** In this hybrid we introduce the ideal functionality $\mathcal{F}_{\text{Dabits}}[p]$. $\mathcal{S}$ no longer samples r_i or computes $(r_{i,0}, \dots, r_{i,m-1})$ for each honest party $P_\beta \in \mathcal{H}$ in Step 1. $\mathcal{S}$ does not compute $(s_1, \dots, s_{(n-t)d})$ or their bit decompositions either. Note that these values are no longer needed to generate the messages sent to $\mathcal{A}$. The difference between this hybrid and **Hybrid 4** is that the results of the command **dabit**, namely $b_{i,j} = s_{i,j}$ for $1 \leq i \leq (n-t)d$ and $0 \leq j < m$, are sampled as independent uniform bits by $\mathcal{F}_{\text{Dabits}}[p]$. In **Hybrid 4**, we have $b_{i,j} = s_{i,j}$. Let $s'_i = \sum_{j=0}^{m-1} 2^j s_{i,j}$ for $1 \leq i \leq (n-t)d$. Then $(s'_1, \dots, s'_{(n-t)d}) = \mathbf{M}(r_1, \dots, r_{nd}) \bmod 2^m$. Fix the inputs of the corrupted parties and the values sampled for parties in $\mathcal{H}_1$. Since $|\mathcal{H}_2| = n - t$ and $\mathbf{M}$ is an $(n-t)d \times nd$ block super-invertible matrix over $\mathbb{Z}_{2^m}$, the map from the uniformly random values supplied by $\mathcal{H}_2$ to $(s'_1, \dots, s'_{(n-t)d})$ is a bijection, even after fixing the contribution of all other parties. Consequently, this output vector is jointly uniform over $\mathbb{Z}_{2^m}^{(n-t)d}$, and all its bits $s_{i,j}$ are independent and uniform. Moreover, the opened values $\delta_{i,j}$, the rejection-sampling indicators v_i , and the opened values c_i in **Hybrid 4** are sampled independently of these private values. The output vector therefore remains uniformly distributed given these simulated messages and the corrupted parties' inputs. Replacing its bits with those sampled by $\mathcal{F}_{\text{Dabits}}[p]$ preserves their joint distribution with the simulated messages. Hence, this hybrid has the same distribution as **Hybrid 4**. □

D.7 Proof of Theorem 5

Proof. We construct a simulator $\mathcal{S}$ that runs the code of the adversary $\mathcal{A}$ as a subroutine and has access to the ideal functionality $\mathcal{F}_{\text{ABB}}[L]$. The simulator emulates the functionality $\mathcal{F}_{\text{ABBBasic}}[L]$ and keeps track of the ids of all the values stored in it. For a value x stored in $\mathcal{F}_{\text{ABBBasic}}[L]$, we denote its id in the simulation by id_x . $\mathcal{S}$ also passes all messages between $\mathcal{A}$ and the environment $\mathcal{Z}$.

Upon receiving $(\text{Input}, P_i, \mathcal{L}, \text{id})$, if P_i is honest, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{ABB}}[L]$ on behalf of each corrupted party. If P_i is corrupted, $\mathcal{S}$ waits to receive P_i 's input value x while emulating $\mathcal{F}_{\text{ABBBasic}}[L]$. Then $\mathcal{S}$ sends $(\text{Input}, P_i, \mathcal{L}, \text{id}, x)$ to $\mathcal{F}_{\text{ABB}}[L]$ on behalf of P_i and sends $(\text{Input}, P_i, \mathcal{L}, \text{id})$ to $\mathcal{F}_{\text{ABB}}[L]$ on behalf of every other corrupted party.

Upon receiving $(\text{Open}, \mathcal{L}, \text{id})$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{ABB}}[L]$ on behalf of each corrupted party and receives $(\mathcal{L}, \text{id}, y)$. Then, after receiving the same instruction while emulating $\mathcal{F}_{\text{ABBBasic}}[L]$, $\mathcal{S}$ sends $(\mathcal{L}, \text{id}, y)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$. If $\mathcal{A}$ replies with **deliver**, $\mathcal{S}$ sends y to all corrupted parties on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$ and sends **deliver** to $\mathcal{F}_{\text{ABB}}[L]$. Otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$ and sends **abort** to $\mathcal{F}_{\text{ABB}}[L]$.

Upon receiving an input related to any other operation in $\mathcal{F}_{\text{ABBBasic}}[L]$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{ABB}}[L]$ on behalf of each corrupted party.

In the following, we focus on the simulation when parties receive inputs $(\text{InputBit}, P_{\text{king}}, \text{id}_1), \dots, (\text{InputBit}, P_{\text{king}}, \text{id}_C)$. We divide the proof into two cases depending on whether P_{king} is corrupted or not.

P_{king} is Corrupted. $\mathcal{S}$ executes the following steps:

1. $\mathcal{S}$ receives from $\mathcal{A}$ the input bits $b_1, \dots, b_C$ on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$.
2. $\mathcal{S}$ samples a random value $r \leftarrow \text{GR}(T, \kappa)$ to emulate Coin of $\mathcal{F}_{\text{ABBBasic}}[L]$ and sends it to $\mathcal{A}$.
3. In Step 3, $\mathcal{S}$ computes $\mathbf{x}, \mathbf{y}$ and the difference $d^{(1)} := 0 - \langle \mathbf{x}, \mathbf{y} \rangle$.
4. For each execution of Step 4, let $j \geq 1$ be the index of the execution. $\mathcal{S}$ executes the following steps:
 - In Step 4(b), for each $i = 1, \dots, \rho - 1$, $\mathcal{S}$ receives $\gamma_i \in \text{GR}(T, \kappa)$ from $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$ and computes $d_i^{(j)} := \gamma_i - \langle \alpha_i, \beta_i \rangle$. Then $\mathcal{S}$ emulates $\mathcal{F}_{\text{ABBBasic}}[L]$ to compute $\gamma_\rho = z - \sum_{i=1}^{\rho-1} \gamma_i$, and computes $d_\rho^{(j)} := d^{(j)} - \sum_{i=1}^{\rho-1} d_i^{(j)}$.
 - In Step 4(c), $\mathcal{S}$ computes all the coefficients of $\mathbf{f}(\cdot)$ and $\mathbf{g}(\cdot)$ when emulating $\mathcal{F}_{\text{ABBBasic}}[L]$.
 - In Step 4(d), $\mathcal{S}$ receives $\gamma_{\rho+1}, \dots, \gamma_{2\rho-1} \in \text{GR}(T, \kappa)$ from $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$, and then computes $d_i^{(j)} := \gamma_i - \langle \mathbf{f}(v_i), \mathbf{g}(v_i) \rangle$ for $i = \rho + 1, \dots, 2\rho - 1$.
 - In Step 4(e), $\mathcal{S}$ computes all the coefficients of $h(\cdot)$ when emulating $\mathcal{F}_{\text{ABBBasic}}[L]$.
 - In Step 4(f), $\mathcal{S}$ samples a random value $r' \leftarrow \text{GR}(T, \kappa)$ to emulate Coin of $\mathcal{F}_{\text{ABBBasic}}[L]$ and sends it to $\mathcal{A}$. If $\exists i \in [\rho] : r' = v_i$, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{ABB}}[L]$.
 - In Step 4(g), $\mathcal{S}$ computes $\mathbf{f}(r'), \mathbf{g}(r'), h(r')$ and resets $\mathbf{x}, \mathbf{y}, z$ to them when emulating $\mathcal{F}_{\text{ABBBasic}}[L]$. $\mathcal{S}$ computes $d^{(j+1)} := h(r') - \langle \mathbf{f}(r'), \mathbf{g}(r') \rangle$.
5. $\mathcal{S}$ sends $x^*y^* - z^*$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$ and waits for the response from $\mathcal{A}$. If $\mathcal{S}$ receives **deliver** from $\mathcal{A}$, $\mathcal{S}$ sends $x^*y^* - z^*$ to all corrupted parties on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{ABB}}[L]$.
6. If one of the following two conditions holds, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{ABB}}[L]$:
 - There exists $i \in [1, C]$ such that $b_i \notin \{0, 1\}$.
 - There exist $i \in [1, 2\rho - 1] \setminus \{\rho\}$ and $j \geq 1$ such that $d_i^{(j)} \neq 0$.
 Otherwise, $\mathcal{S}$ sends $\{(\text{InputBit}, P_{\text{king}}, \text{id}_i, b_i)\}_{i=1}^C$ to $\mathcal{F}_{\text{ABB}}[L]$ on behalf of P_{king} and sends $\{(\text{InputBit}, P_{\text{king}}, \text{id}_i)\}_{i=1}^C$ to $\mathcal{F}_{\text{ABB}}[L]$ on behalf of all other corrupted parties.

We use hybrid arguments to show that the outputs of $\mathcal{Z}$ in the real and ideal worlds are indistinguishable.

- **Hybrid 0:** The simulator $\mathcal{S}$ runs the protocol $\Pi_{\text{InputBit}}(L, C)$ honestly and emulates $\mathcal{F}_{\text{ABBBasic}}[L]$, which corresponds to the real world execution of $\Pi_{\text{InputBit}}(L, C)$.
- **Hybrid 1:** Let $s = \lceil \log_\rho C \rceil$ be the number of rounds in Step 4. After opening $x^*y^* - z^*$ in the last step, instead of checking whether $x^*y^* - z^* = 0$, $\mathcal{S}$ checks whether

$$d_i^{(s)} = 0, \quad \forall i = 1, \dots, 2\rho - 1, \quad (*)$$

i.e., $\langle \mathbf{f}(v_i), \mathbf{g}(v_i) \rangle = \gamma_i$ for $i = 1, \dots, 2\rho - 1$ in the last round of Step 4. To analyze the difference between **Hybrid 0** and **Hybrid 1**, we only need to consider the case that all parties do not abort until the final check. If Eq. (*) holds, all honest parties will not abort in both hybrids since Eq. (*) implies that the two degree- $(2\rho - 2)$ polynomials $f(\cdot)g(\cdot)$ and $h(\cdot)$ are the same, so $f(r')g(r') = h(r')$, i.e., $x^*y^* - z^* = 0$. If Eq. (*) does not hold, the probability that $x^*y^* - z^* = 0$ is at most $\frac{2\rho-2}{2^\kappa-\rho}$ since there are at most $2\rho - 2$ $r' \in \text{GR}(T, \kappa) \setminus [\rho]$ such that $f(r')g(r') = h(r')$. Therefore **Hybrid 1** is statistically indistinguishable from **Hybrid 0**.

- **Hybrid 2:** $\mathcal{S}$ checks whether $d^{(s)} = 0$ and $d_i^{(s)} = 0$ for all $i \in [1, 2\rho - 1] \setminus \{\rho\}$. By the relationship between $d^{(s)}$ and $d_1^{(s)}, \dots, d_{2\rho-1}^{(s)}$, the new check is equivalent to the previous one. Therefore **Hybrid 2** $\equiv$ **Hybrid 1**.
- **Hybrid 3:** $\mathcal{S}$ checks whether
 - (1) $d_i^{(j)} = 0$ for all $i \in [1, 2\rho - 1] \setminus \{\rho\}$ and $j = 1, \dots, s$.
 - (2) $d^{(1)} = 0$ (i.e., $\langle \mathbf{x}, \mathbf{y} \rangle = 0$ in Step 3).

By the arguments in **Hybrid 1** and **Hybrid 2**, the probability that **Hybrid 3** aborts while **Hybrid 2** does not is at most $(s - 1) \cdot \frac{2\rho-2}{2^\kappa-\rho}$, which is negligible in κ . Therefore **Hybrid 3** is statistically indistinguishable from **Hybrid 2**.

- **Hybrid 4:** We switch to the ideal-world execution and $\mathcal{S}$ emulates all operations in $\mathcal{F}_{\text{ABBBasic}}[L]$ honestly. The only difference is that the second check in **Hybrid 3**, namely $\langle \mathbf{x}, \mathbf{y} \rangle = 0$, is replaced with the check that all of P_{king} 's input values $b_1, \dots, b_C$ are in $\{0, 1\}$. Clearly, $b_1, \dots, b_C \in \{0, 1\}$ implies $\langle \mathbf{x}, \mathbf{y} \rangle = 0$. Conversely, suppose that $b_i \notin \{0, 1\}$ for some $i \in [C]$, and define

$$F(X) = \sum_{i=1}^C b_i(b_i - 1)X^{i-1},$$

so that $\langle \mathbf{x}, \mathbf{y} \rangle = F(r)$. By Lemma 3, F is a nonzero polynomial of degree at most $C - 1$. The root bound over $\text{GR}(T, \kappa)$ gives $\Pr[F(r) = 0] \leq (C - 1)/2^\kappa$. Therefore, the probability that **Hybrid 4** aborts while **Hybrid 3** does not is at most $(C - 1)/2^\kappa$, which is negligible in κ . It follows that **Hybrid 4** is statistically indistinguishable from **Hybrid 3**.

P_{king} is Honest. $\mathcal{S}$ executes the following steps:

1. In Step 1, $\mathcal{S}$ receives $(\text{Input}, \text{P}_{\text{king}}, \text{id}_1), \dots, (\text{Input}, \text{P}_{\text{king}}, \text{id}_C)$ from all corrupted parties on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$.
2. In Step 2, $\mathcal{S}$ randomly samples a random value $r \leftarrow \text{GR}(T, \kappa)$ to emulate Coin of $\mathcal{F}_{\text{ABBBasic}}[L]$ and sends it to all corrupted parties.
3. In Step 4(f), $\mathcal{S}$ randomly samples a random value $r' \leftarrow \text{GR}(T, \kappa)$ to emulate Coin of $\mathcal{F}_{\text{ABBBasic}}[L]$ and sends it to all corrupted parties. If $\exists i \in [\rho] : r' = v_i$, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{ABB}}[L]$.
4. In Step 6, $\mathcal{S}$ sends 0 to all corrupted parties on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$ and waits for the response from $\mathcal{A}$. If $\mathcal{S}$ receives **abort** from $\mathcal{A}$, $\mathcal{S}$ sends **abort** to all corrupted parties and $\mathcal{F}_{\text{ABB}}[L]$; otherwise, $\mathcal{S}$ sends 0 to all corrupted parties on behalf of $\mathcal{F}_{\text{ABBBasic}}[L]$ and sends $\{(\text{InputBit}, \text{P}_{\text{king}}, \text{id}_i)\}_{i=1}^C$ to $\mathcal{F}_{\text{ABB}}[L]$ on behalf of all corrupted parties.

The only difference between the real world and the ideal world is that in the ideal world, $\mathcal{S}$ directly opens 0 to all corrupted parties instead of computing $x^*y^* - z^*$. Since P_{king} is honest, $b_1, \dots, b_C \in \{0, 1\}$, so $b_i(b_i - 1) = 0$ for all $i \in [C]$ and $\langle \mathbf{x}, \mathbf{y} \rangle = 0 = z$ in Step 3. Then by induction, it is routine to verify that in each round of Step 4, $\langle \mathbf{f}(v_i), \mathbf{g}(v_i) \rangle = \gamma_i$ for all $i = 1, \dots, 2\rho - 1$ and then $\langle \mathbf{f}(r'), \mathbf{g}(r') \rangle = h(r')$ in Step 4(g). Therefore, $x^*y^* - z^* = 0$ in Step 6. It follows that the real world is identical to the ideal world. $\square$

D.8 Proof of Theorem 6

Proof. We construct a simulator $\mathcal{S}$ that runs the code of the adversary $\mathcal{A}$ as a subroutine and has access to the ideal functionality $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$. The simulator emulates the functionality $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$ where $\ell_P = \lceil \log_2 P \rceil$ and keeps track of the ids of all the values stored in it. For a value x stored in $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$, we denote its id in the simulation by id_x . The simulator also passes all messages between $\mathcal{A}$ and the environment $\mathcal{Z}$. Throughout the simulation, $\mathcal{S}$ forwards an instruction to the ideal functionality on behalf of a corrupted party only after receiving the corresponding instruction from that party while emulating $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$.

Upon receiving $(\text{Input}, P_i, \mathcal{L}, \text{id})$, if P_i is honest, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ on behalf of each corrupted party. If P_i is corrupted, $\mathcal{S}$ waits to receive P_i 's input value x when emulating $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$. Then $\mathcal{S}$ sends $(\text{Input}, P_i, \mathcal{L}, \text{id}, x)$ to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ on behalf of P_i and sends $(\text{Input}, P_i, \mathcal{L}, \text{id})$ to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ on behalf of every other corrupted party.

Upon receiving $(\text{Open}, \mathcal{L}, \text{id})$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ on behalf of each corrupted party and receives $(\mathcal{L}, \text{id}, y)$. Then after $\mathcal{S}$ receives the same instruction when emulating $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$, $\mathcal{S}$ sends $(\mathcal{L}, \text{id}, y)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$. If $\mathcal{A}$ replies **deliver**, $\mathcal{S}$ sends y to all corrupted parties on behalf of $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$ and sends **deliver** to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$. Otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties on behalf of $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$ and sends **abort** to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$.

Upon receiving $(\text{priv-edabit}, \text{P}_{\text{king}}, N, \{(\text{id}_i; \text{id}_{i,0}, \text{id}_{i,1}, \dots, \text{id}_{i,m-1})\}_{i=0}^{N-1})$, $\mathcal{S}$ first passes it to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ on behalf of each corrupted party. If P_{king} is corrupted, $\mathcal{S}$ waits to receive P_{king} 's input values $r_0, \dots, r_{N-1}$ when emulating $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$. Then $\mathcal{S}$ sends them to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ on behalf of P_{king} .

Upon receiving an input related to other operations in $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ on behalf of each corrupted party.

In the following, we focus on the simulation when parties receive input $(\text{any-range-edabit}, \text{id}, \text{id}_0, \dots, \text{id}_{\ell_P-1})$. $\mathcal{S}$ first sends the input of each corrupted party to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$. Then $\mathcal{S}$ simulates the execution of $\Pi_{\text{AnyRangeEdabits}}(L, P)$ as follows:

- In Step 4, $\mathcal{S}$ uniformly samples $r \in \mathbb{Z}_{2^{\ell_P-1}}$ and $s \in \mathbb{Z}_2$. Then $\mathcal{S}$ sets $o = 1$ if $r + s2^{\ell_P-1} \geq P$ and $o = 0$ otherwise. $\mathcal{S}$ sends o to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$ and waits for its response. If the response is **deliver**, then $\mathcal{S}$ sends o to all corrupted parties on behalf of $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$; otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties on behalf of $\mathcal{F}_{\text{Edabits}}[L, \ell_P - 1]$ and sends **abort** to $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$. $\mathcal{S}$ repeats this step until $o = 0$.

By the definition of $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$ and the correctness of π_{BitLT} , the probability that $o = 1$ is the same in both the real world and the simulation. Besides, note that $r + s2^{\ell_P-1}$ is uniformly random in $\mathbb{Z}_{2^{\ell_P}}$ and it is uniformly random in $\mathbb{Z}_P$ given $o = 0$. Therefore we have that the view of the environment $\mathcal{Z}$ in the real world is the same as that in the ideal world. $\square$

D.9 Proof of Theorem 7

Proof. We construct a simulator $\mathcal{S}$ that runs the code of the adversary $\mathcal{A}$ as a subroutine and has access to the ideal functionality $\mathcal{F}_{\text{Dabits}}[2^\ell]$. The simulator emulates the functionality $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ and keeps track of the ids of all the values stored in it. For a value x stored in $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$, we denote its id in the simulation by id_x . $\mathcal{S}$ also passes all messages between $\mathcal{A}$ and the environment $\mathcal{Z}$. Throughout the simulation, $\mathcal{S}$ forwards an instruction to the ideal functionality on behalf of a corrupted party only after receiving the corresponding instruction from that party while emulating $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$.

Upon receiving $(\text{Input}, P_i, \mathcal{L}, \text{id})$, if P_i is honest, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{Dabits}}[2^\ell]$ on behalf of each corrupted party. If P_i is corrupted, $\mathcal{S}$ waits to receive P_i 's input value x when emulating $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$. Then $\mathcal{S}$ sends $(\text{Input}, P_i, \mathcal{L}, \text{id}, x)$ to $\mathcal{F}_{\text{Dabits}}[2^\ell]$ on behalf of P_i and sends $(\text{Input}, P_i, \mathcal{L}, \text{id})$ to $\mathcal{F}_{\text{Dabits}}[2^\ell]$ on behalf of every other corrupted party.

Upon receiving $(\text{Open}, \mathcal{L}, \text{id})$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{Dabits}}[2^\ell]$ on behalf of each corrupted party and receives $(\mathcal{L}, \text{id}, y)$. Then after $\mathcal{S}$ receives the same instruction when emulating $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$, $\mathcal{S}$ sends $(\mathcal{L}, \text{id}, y)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$. If $\mathcal{A}$ replies **deliver**, $\mathcal{S}$ sends y to all corrupted parties on behalf of $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ and sends **deliver** to $\mathcal{F}_{\text{Dabits}}[2^\ell]$. Otherwise, $\mathcal{S}$ sends **abort** to all corrupted parties on behalf of $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ and sends **abort** to $\mathcal{F}_{\text{Dabits}}[2^\ell]$.

Upon receiving $(\text{priv-edabit}, P_{\text{king}}, N, \{(\text{id}_i; \text{id}_{i,0}, \dots, \text{id}_{i,m-1})\}_{i=1}^N)$, $\mathcal{S}$ forwards it to $\mathcal{F}_{\text{Dabits}}[2^\ell]$ on behalf of each corrupted party. If P_{king} is corrupted, $\mathcal{S}$ waits to receive P_{king} 's input values $r_1, \dots, r_N$ while emulating $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$ and forwards them to $\mathcal{F}_{\text{Dabits}}[2^\ell]$ on behalf of P_{king} .

Upon receiving an input related to other operations in $\mathcal{F}_{\text{PrivEd}}[2^\ell, m]$, $\mathcal{S}$ passes it to $\mathcal{F}_{\text{Dabits}}[2^\ell]$ on behalf of each corrupted party.

Upon receiving $(\text{dabit}, N, (\text{id}_1, \text{id}'_1), \dots, (\text{id}_N, \text{id}'_N))$, $\mathcal{S}$ just sends this input to $\mathcal{F}_{\text{Dabits}}[2^\ell]$ on behalf of each corrupted party.

In the protocol, when the command **dabit** is received, no message is sent to corrupted parties. By the definition of **RandBit** and **LocalMod**, the results set in the protocol are exactly N random bits shared in both domains, which is the same as $\mathcal{F}_{\text{Dabits}}[2^\ell]$. Then we conclude that the view of the environment $\mathcal{Z}$ in the real world is identical to that in the ideal world for the command **dabit**, and likewise for other commands. Therefore, the output of $\mathcal{Z}$ in the real world is identical to that in the ideal world. $\square$

E Cost of Our Edabit Generation

In this appendix we count the cost of our own protocols. We also fix here the conventions that we use for the cost from prior work in Appendix F, so that the two can be compared operation by operation in Section 7.

We express every cost as a *number of $\mathcal{F}_{\text{ABB}}$ operations*, leaving the price of an individual operation unspecified: that price is exactly what changes with the underlying MPC protocol, with the arithmetic domain, and with the corruption setting. We write $C_{\text{Cmd}}^{\mathbb{Z}_{\mathcal{L}}}$ for the cost of one $\mathcal{F}_{\text{ABB}}$ command **Cmd** over the domain $\mathbb{Z}_{\mathcal{L}}$, so that, for instance, $C_{\text{Open}}^{\mathbb{Z}_2}$ is the cost of one opening in the binary domain. The quantities we are after are $C_{\text{dabit}}^{\mathbb{Z}_{\mathcal{L}}}$, $C_{\text{PrivEd}}^{\mathbb{Z}_{\mathcal{L}}}$ and $C_{\text{edabit}}^{\mathbb{Z}_{\mathcal{L}}}$, the costs of producing *one* dabit, one private edabit and one global edabit respectively.

We do not charge for calls to **Coin**, as argued in Section 7.

E.1 Cost of some building blocks

Besides the $\mathcal{F}_{\text{ABB}}$ commands, our protocols and those of Appendix F call the building blocks of Section 2.2, together with the conversion $\pi_{\text{BitToArith}}$ of Section 2.4. These are not $\mathcal{F}_{\text{ABB}}$ commands but procedures built from them, regardless of what realizes $\mathcal{F}_{\text{ABB}}$, so we record their cost once here and charge them by reference below. We write $\mathcal{C}_{\pi_{\text{Proc}}}^{\mathbb{Z}_{\mathcal{L}}}$ for the cost of one call to a procedure π_{Proc} whose inputs are shared over $\mathbb{Z}_{\mathcal{L}}$, adding the bit length of those inputs as an argument when the procedure takes one.

Cost of π_{BitAdder} . The ripple-carry adder computes its carries as $c_0 = 0$ and $c_{i+1} = c_i \oplus (x_i \oplus c_i)(y_i \oplus c_i)$, and its output bits as $x_i \oplus y_i \oplus c_i$. Over $\mathbb{Z}_2$ the XORs are free and each carry costs one Mult, so an addition charged at w bit positions costs

$$\mathcal{C}_{\pi_{\text{BitAdder}}}^{\mathbb{Z}_2}(w) = w \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_2}. \quad (5)$$

Over a larger domain the XORs must be emulated as $x \oplus y = x + y - 2xy$ and are no longer free. Writing the carry instead as $c_{i+1} = x_i y_i + c_i(x_i \oplus y_i)$, the two products $x_i y_i$ and $c_i(x_i \oplus y_i)$ determine both the next carry and the output bit by linear operations, so each position costs two multiplications and

$$\mathcal{C}_{\pi_{\text{BitAdder}}}^{\mathbb{Z}_{\mathcal{L}}}(w) = 2w \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_{\mathcal{L}}} \quad (\mathcal{L} > 2). \quad (6)$$

Both cases are used in our paper: the aggregator of Appendix E.2 adds over $\mathbb{Z}_2$, whereas the bit decomposition $\pi_{\text{FieldBitDec}}$ adds over $\mathbb{F}_p$.

It does not matter if one addend is public, as in some of our calls: the incoming carry c_i is secret, so the position still costs one multiplication. Also, if the sum is taken modulo 2^w , the top carry is not needed and one multiplication is saved; we ignore this minor optimization. Finally, we use the ripple-carry adder rather than a shallower one because it minimizes multiplications, and hence communication, at the price of depth; it is also the adder for which [21] establishes the *weak additive tamper resilience* that its cut-and-choose analysis requires.

Cost of π_{BitLT} . The comparison reduces to the carry chain we have just costed. Writing $\bar{b}_i = 1 - b_i$, which is local, we have $a + \bar{b} + 1 = a - b + 2^w$, so the carry out of adding a and $\bar{b}$ with carry-in 1 is 1 exactly when $a \geq b$. We therefore run one ripple-carry pass and keep only its carries,

$$c_0 = 1, \quad c_{i+1} = a_i \bar{b}_i + c_i(a_i \oplus \bar{b}_i),$$

outputting $\llbracket c \rrbracket_{\mathcal{L}} \leftarrow 1 - \llbracket c_w \rrbracket_{\mathcal{L}}$; no sum bit is needed. In all our calls the second operand is public, being the modulus p , a shift $p - c$ of it, or the bound P of the any-range edabits. This leaves $a_i \bar{b}_i$ and $a_i \oplus \bar{b}_i$ linear and charges only the product $c_i(a_i \oplus \bar{b}_i)$, so that in either domain

$$\mathcal{C}_{\pi_{\text{BitLT}}}^{\mathbb{Z}_{\mathcal{L}}}(w) = w \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_{\mathcal{L}}}, \quad (7)$$

and $2w \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_{\mathcal{L}}}$ when both operands are secret and $\mathcal{L} > 2$. The protocols of [13,22] evaluate the same chain as a prefix computation, trading more multiplications for $O(\log w)$ depth; we keep the sequential version, as we optimize for communication.

Cost of $\pi_{\text{FieldBitDec}}$. The bit decomposition of Section 2.2 is used only over fields, in the dabit generation of Appendix E.5. Reading Procedure 8, Step 1 makes ℓ calls to RandBit over $\mathbb{F}_p$, Step 2 one call to π_{BitLT} and one opening, Step 5 one further opening, Step 6 a second call to π_{BitLT} , and Step 9 one call to π_{BitAdder} ; every other step is local. Steps 1–3 are repeated until the random value falls below p , which happens after $2^\ell/p < 2$ attempts in expectation. Hence

$$\mathcal{C}_{\pi_{\text{FieldBitDec}}}^{\mathbb{F}_p} = \frac{2^\ell}{p} \left(\ell \cdot \mathcal{C}_{\text{RandBit}}^{\mathbb{F}_p} + \mathcal{C}_{\pi_{\text{BitLT}}}^{\mathbb{F}_p}(\ell) + \mathcal{C}_{\text{Open}}^{\mathbb{F}_p} \right) + \mathcal{C}_{\pi_{\text{BitLT}}}^{\mathbb{F}_p}(\ell) + \mathcal{C}_{\pi_{\text{BitAdder}}}^{\mathbb{F}_p}(\ell) + \mathcal{C}_{\text{Open}}^{\mathbb{F}_p}. \quad (8)$$

Cost of $\pi_{\text{BitToArith}}$. Finally, the conversion $\pi_{\text{BitToArith}}$ of Section 2.4 takes a bit shared over $\mathbb{Z}_2$ and returns the same bit shared over $\mathbb{Z}_L$. Reading Procedure 9, it consumes one dabit over $\mathbb{Z}_L$ and opens one value over $\mathbb{Z}_2$, the final reconstruction being local, so

$$\mathcal{C}_{\pi_{\text{BitToArith}}}^{\mathbb{Z}_L} = \mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L} + \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2}. \quad (9)$$

Here $\mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L}$ measures the cost of generating a dabit over $\mathbb{Z}_L$, which we discuss later in Appendix E.5.

E.2 Aggregation

We count the operations required by one call to $\pi_{\text{AggEdabit}}$ (Procedure 1), which turns E partially arithmetic edabits of length m into a single one. Recall that the procedure is parameterized by a block size A , that $L \in \{2^\ell, p\}$, and that correctness requires $\eta \geq A + \log E$, where $T = 2^\eta$. Since operations over $\mathbb{Z}_T$ cost an amount proportional to η , we take $\eta = A + \log E$, the smallest admissible value, and fix A at the end of this subsection.

Steps 3(a), 3(c), 3(g) and 3(h) are local, as are Steps 1, 2, 5 and 6. The number of bit positions of the aggregated sum that the loop of Step 3 must process depends on the modulus and on the edabit length m . The sum of E values of m bits has $m + \log E$ bits, but when $L = 2^\ell$ the positions from ℓ upwards vanish modulo 2^ℓ and need not be computed, so the loop runs over

$$\mu := \begin{cases} \min\{\ell, m + \log E\} & \text{if } L = 2^\ell, \\ m + \log E & \text{if } L = p, \end{cases}$$

positions, that is, over μ/A blocks. Each of the μ/A iterations of Step 3 then costs:

- η calls to RandBit over $\mathbb{Z}_T$ in Step 3(b), which produce the mask e , and one call to Open over $\mathbb{Z}_T$ in Step 3(d), which reveals the masked block δ . This costs $\eta \cdot \mathcal{C}_{\text{RandBit}}^{\mathbb{Z}_T} + \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T}$.
- $\mathcal{C}_{\pi_{\text{BitAdder}}}^{\mathbb{Z}_2}(\log E)$ in Step 3(e). This call to π_{BitAdder} adds the public value δ to the carry of the previous block, which is secret but only $\eta - A = \log E$ bits long; above that position both addends are public and the chain propagates locally, so only $\log E$ positions are charged.
- $\mathcal{C}_{\pi_{\text{BitAdder}}}^{\mathbb{Z}_2}(\eta)$ in Step 3(f), for the adder on the two secret η -bit values that removes the mask. By (5) the two adders together cost $\mathcal{C}_{\pi_{\text{BitAdder}}}^{\mathbb{Z}_2}(\eta) + \mathcal{C}_{\pi_{\text{BitAdder}}}^{\mathbb{Z}_2}(\log E) = (\eta + \log E) \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_2}$.

The number ν of overflow bits that Step 4 converts back to the arithmetic domain likewise depends on L and m . These occupy the positions $m, \dots, m + \log E - 1$, of which again only those below ℓ survive when $L = 2^\ell$, so

$$\nu := \begin{cases} \min\{\log E, \ell - m\} & \text{if } L = 2^\ell, \\ \log E & \text{if } L = p. \end{cases}$$

In particular $\mu = \ell$ and $\nu = 0$ when $L = 2^\ell$ and $m = \ell$, in which case no conversion is performed at all. Hence, Step 4 costs $\nu \cdot \mathcal{C}_{\pi_{\text{BitToArith}}}^{\mathbb{Z}_L}$.

Writing $\mathcal{C}_{\text{Agg}}^{\mathbb{Z}_L}$ for the cost of a single call to $\pi_{\text{AggEdabit}}$, we obtain

$$\mathcal{C}_{\text{Agg}}^{\mathbb{Z}_L} = \frac{\mu}{A} \cdot \left(\eta \cdot \mathcal{C}_{\text{RandBit}}^{\mathbb{Z}_T} + \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T} + (\eta + \log E) \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_2} \right) + \nu \cdot \mathcal{C}_{\pi_{\text{BitToArith}}}^{\mathbb{Z}_L}. \quad (10)$$

It remains to fix the block size A , the only free parameter left. Neither μ nor ν depends on it, so using $\eta = A + \log E$, this amounts to minimizing

$$\frac{1}{A} \left((A + \log E) \cdot \mathcal{C}_{\text{RandBit}}^{\mathbb{Z}_T} + \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T} + (A + 2 \log E) \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_2} \right).$$

The first term dominates, since a block draws η random bits against a single opening and $A + 2 \log E$ binary multiplications. Recalling the assumption above that an operation over $\mathbb{Z}_T$ costs an amount proportional to η , that term is proportional to $(A + \log E)^2/A$, which is minimized at

$$A = \log E,$$

the value we adopt throughout. It gives $\eta = 2 \log E$, in agreement with the choice $T = 2^{\lceil 2 \log nd \rceil}$ of Section 4.2. Figure 5 shows how (10) then divides among its terms in a concrete instantiation.¹⁶

E.3 Global Edabits

We now count the operations required by $\Pi_{\text{Edabits}}(L, m)$ (Protocol 2) per global edabit produced. Each iteration of its loop consumes nd private edabits in Step 1 and makes $(n-t)d$ calls to $\pi_{\text{AggEdabit}}$ in Step 2, one for every row of the block super-invertible matrix $\mathbf{M}$, and then outputs $(n-t)d$

¹⁶ One can be more fine-grained with the choice of A , but the gains are minimal and therefore we favor clarity by taking a simpler expression $A = \log E$.

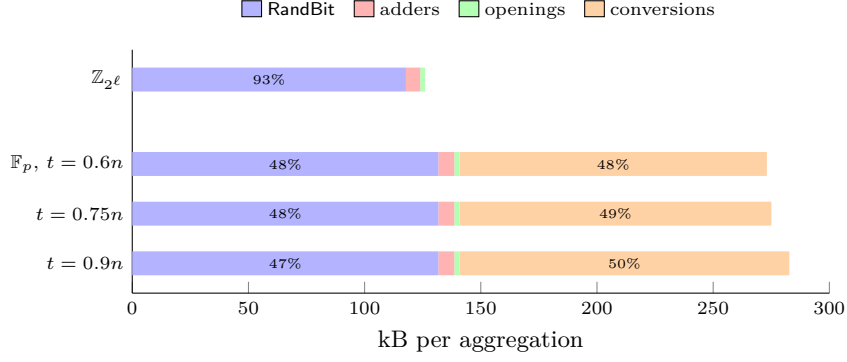


Fig. 5: Where the cost (10) of one aggregation goes, priced as in Table 1 at $n = 64$ and $N = 2^{20}$. The bars are drawn to scale, so that their lengths may be compared across rows, and each segment is labeled with its share of its own bar. The η random bits that each block needs for its bit decomposition dominate everything else; over $\mathbb{F}_p$ the $\nu = \log nd$ conversions, and with them the dabits they consume, take about half of the total. Only those dabits depend on the corruption threshold, which is why a single bar suffices over $\mathbb{Z}_{2^\ell}$.

global edabits in Step 3, which is local. Amortized over that output, one global edabit costs $\frac{n}{n-t}$ private edabits and exactly one call to $\pi_{\text{AggEdabit}}$, whose fan-in is the weight $E_i = \sum_j M_{i,j} \leq nd$ of the corresponding row. We charge every aggregation at the largest fan-in $E = nd$, which is also the value from which $\eta = \lceil 2 \log nd \rceil$ is fixed in Section 4.2. We therefore obtain

$$\mathcal{C}_{\text{edabit}}^{\mathbb{Z}_L} = \frac{n}{n-t} \cdot \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L} + \mathcal{C}_{\text{Agg}}^{\mathbb{Z}_L}, \quad (11)$$

with $\mathcal{C}_{\text{Agg}}^{\mathbb{Z}_L}$ given by (10) at $E = nd$, and $\mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L}$ by (17) or (18) below, depending on the domain.

The factor $\frac{n}{n-t}$ in the first term is what the randomness extraction of Section 4.1 buys: one loop turns nd private edabits into $(n-t)d$ global ones, so the number of private edabits consumed per output is $\frac{n}{n-t}$, which is a constant as soon as $t < (1-\epsilon)n$ for an arbitrary constant $\epsilon \in (0, 1)$, rather than a quantity growing with n . It is also the only place where the threshold enters (11) explicitly; over $\mathbb{F}_p$ it enters once more, and for the same reason, through the dabits that the conversions inside $\mathcal{C}_{\text{Agg}}^{\mathbb{F}_p}$ consume, see (20).

Substituting the concrete counts of Section 7 for each command in (11) gives the figures reported there; the asymptotic form of the same bound is stated in Section 4.2. Figure 6 shows how the cost divides among the private edabits, the aggregator and, over $\mathbb{F}_p$, the conversions.

E.4 Private Edabits via Random Subset Sum

We count the basic operations required to generate one private edabit of length m , separately for $\Pi_{\text{PrivEd}}(2^\ell, m)$ and $\Pi_{\text{PrivEd}}(p, m)$ of Section 5. Recall that $T = 2^\eta$. Both protocols delegate the aggregation of the binary components to $\pi_{\text{TreeAggregate}}$, so we write $\mathcal{C}_{\pi_{\text{TreeAggregate}}}^{\mathbb{Z}_T}(N', w)$ for the cost of one call to it on N' inputs of length w , and postpone its value to the end of this subsection.

For the ring case, the costs of generating a batch of N private edabits with $\Pi_{\text{PrivEd}}(2^\ell, m)$ are as follows:

- In Steps 1–2, $N + \kappa$ elements over $\mathbb{Z}_{2^\ell}$ are input, and $(N + \kappa)m$ bits are input through **InputBit**.
- In Step 4, opening $R \in \text{GR}(2^\ell, \kappa)$ requires κ calls to **Open** over $\mathbb{Z}_{2^\ell}$.
- In Step 5, κ calls to $\pi_{\text{TreeAggregate}}$ are executed, each on $N + 1$ inputs of length m .
- In Step 6, opening the aggregated bit components requires κm calls to **Open** over $\mathbb{Z}_T$.

Step 3 only samples public randomness, and every remaining operation is local. Dividing by N , the cost of one private edabit over $\mathbb{Z}_{2^\ell}$ is therefore

$$\begin{aligned} \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_{2^\ell}} &= \left(1 + \frac{\kappa}{N}\right) \cdot \left(\mathcal{C}_{\text{Input}}^{\mathbb{Z}_{2^\ell}} + m \cdot \mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_{2^\ell}}\right) \\ &\quad + \frac{\kappa}{N} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_{2^\ell}} + \frac{\kappa m}{N} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T} + \frac{\kappa}{N} \cdot \mathcal{C}_{\pi_{\text{TreeAggregate}}}^{\mathbb{Z}_T}(N + 1, m). \end{aligned} \quad (12)$$

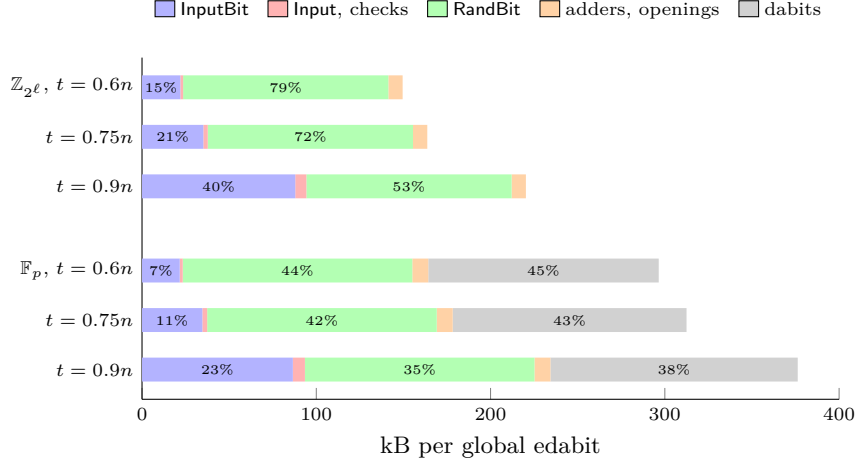


Fig. 6: Where the cost (11) of one global edabit goes, priced as in Table 1 at $n = 64$ and $N = 2^{20}$, with the bars drawn to scale and each segment labeled with its share of its own bar. The first two segments are the private edabits the extraction consumes, the next two the aggregator, and the last the conversions with the dabits they use. The threshold enters through the factor $\frac{n}{n-t}$ on the private edabits and, over $\mathbb{F}_p$, through the dabits that the conversions consume, so the aggregator proper, which dominates as long as t stays well below n , loses ground as t grows: over $\mathbb{Z}_{2^\ell}$ its RandBit falls from 79% at $t = 0.6n$ to 54% at $t = 0.9n$.

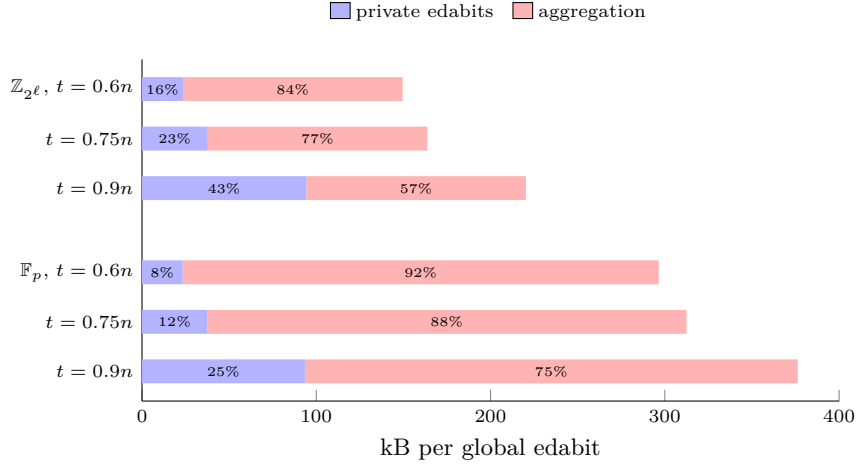


Fig. 7: The cost of Figure 6, at the same parameters and on the same scale, grouped into the two ingredients that Section G.1 compares against [21]: the $\frac{n}{n-t}$ private edabits that the randomness extraction consumes, and the aggregation, the latter including over $\mathbb{F}_p$ the conversions and the dabits they use. The aggregation is the larger of the two at every threshold, and it is what our protocol has to pay in order to consume so few private edabits; the balance moves towards the private edabits as t approaches n and the extraction rate deteriorates. The same split in [21] is the opposite, roughly 88% private edabits against 12% aggregation over $\mathbb{Z}_{2^\ell}$ and 92% against 8% over $\mathbb{F}_p$, and it barely moves with the threshold.

Now we analyze the field case $\mathbb{F}_p$. Recall that $\ell = \lceil \log_2 p \rceil$ and $\kappa_p = \lceil \kappa / \log_2 p \rceil$, and write $\hat{\kappa} := \kappa_p \ell$ for the product that multiplies almost every term below. It is best read as the soundness the check attains: β is drawn from $\text{GF}(p^{\kappa_p})$, so an inconsistent batch is accepted with probability at most N/p^{κ_p} , and $\hat{\kappa}$ is the resulting $\kappa_p \log_2 p \geq \kappa$ bits rounded up to whole bit positions. It overshoots κ because κ_p must be an integer: over a large prime a single coordinate already buys more than κ bits and there is no way to buy fewer, so $\hat{\kappa} = \ell$ whenever $\log_2 p \geq \kappa$, whereas for a small prime $\kappa_p \approx \kappa/\ell$ gives $\hat{\kappa} \approx \kappa$; in short, $\hat{\kappa} \approx \max\{\ell, \kappa\}$. Accordingly $\Pi_{\text{PrivEd}}(p, m)$ runs one subset-sum check per bit position of the representation of β , that is $\hat{\kappa}$ of them, whereas the ring case runs κ . The costs of generating a batch of N private edabits are as follows:

- In Steps 1–2, $N + \hat{\kappa}$ elements over $\mathbb{F}_p$ are input, and $Nm + \hat{\kappa}\ell$ bits are input through **InputBit**.
- In Steps 4(a)–4(b), $\hat{\kappa}$ elements over $\mathbb{F}_p$ are opened, and the bit decompositions of the values $q_{a,b}$ require $\hat{\kappa}(\lfloor \log N \rfloor + 1)$ calls to **InputBit**.
- In Step 4(c), $\hat{\kappa}$ calls to $\pi_{\text{TreeAggregate}}$ are executed, each on $N + 1 + \ell$ inputs of length $\ell_{\text{agg}} = \ell + \lceil \log(3N + 1) \rceil$.
- In Step 4(d), opening the aggregated bit components requires $\hat{\kappa}\ell_{\text{agg}}$ calls to **Open** over $\mathbb{Z}_T$.

In Step 3 the parties only sample $\beta \in \text{GF}(p^{\kappa_p})$; its powers and the bit decompositions of their coefficients are public and computed locally, as are the zero padding and the shifts used to represent multiples of p in Step 4(c). Dividing by N , the cost of one private edabit over $\mathbb{F}_p$ is

$$\begin{aligned} \mathcal{C}_{\text{PrivEd}}^{\mathbb{F}_p} &= \left(1 + \frac{\hat{\kappa}}{N}\right) \cdot \mathcal{C}_{\text{Input}}^{\mathbb{F}_p} + \left(m + \frac{\hat{\kappa}(\ell + \lfloor \log N \rfloor + 1)}{N}\right) \cdot \mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T} \\ &\quad + \frac{\hat{\kappa}}{N} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{F}_p} + \frac{\hat{\kappa}\ell_{\text{agg}}}{N} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T} + \frac{\hat{\kappa}}{N} \cdot \mathcal{C}_{\pi_{\text{TreeAggregate}}}^{\mathbb{Z}_T}(N + 1 + \ell, \ell_{\text{agg}}). \end{aligned} \quad (13)$$

$\mathcal{C}_{\pi_{\text{TreeAggregate}}}^{\mathbb{Z}_T}(N', w)$: Cost of $\pi_{\text{TreeAggregate}}$. It remains to price the tree. In a call to $\pi_{\text{TreeAggregate}}$ on N' inputs of length w , N'/E' blocks of size $E' = 2^{\eta-1} - 1 = T/2 - 1$ are aggregated together, and then the resulting N'/E' sums are, once again, split into blocks of size E' and aggregated, until a single output is produced. The number of blocks in the first level is N'/E' , in the second $N'/(E')^2$, and so on, so the total number of blocks is $N'/E' + N'/(E')^2 + \dots \leq N'/(E' - 1) = N'/(T/2 - 2)$. For each block, each of the w bits is summed locally, giving an η -bit value whose bit decomposition the dealer re-inputs in Step 2(b), that is η calls to **InputBit**. This costs $\eta w \cdot \mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T}$ per block.

The batched check of Step 6 then opens one element of $\text{GR}(T, \kappa)$, that is κ calls to **Open** over $\mathbb{Z}_T$, and every remaining operation in the tree is local.¹⁷ Hence

$$\mathcal{C}_{\pi_{\text{TreeAggregate}}}^{\mathbb{Z}_T}(N', w) = \frac{N' w \eta}{T/2 - 2} \cdot \mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T} + \kappa \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T}. \quad (14)$$

$\mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L}$: Final cost of $\Pi_{\text{PrivEd}}(2^\ell, m)$ and $\Pi_{\text{PrivEd}}(p, m)$. Substituting (14) into (12) and collecting the calls to **InputBit** gives

$$\begin{aligned} \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_{2^\ell}} &= \left(1 + \frac{\kappa}{N}\right) \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_{2^\ell}} + m \left(1 + \frac{\kappa}{N} + \frac{\kappa(N + 1)\eta}{N(T/2 - 2)}\right) \cdot \mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T} \\ &\quad + \frac{\kappa}{N} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_{2^\ell}} + \frac{\kappa(m + \kappa)}{N} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T}, \end{aligned} \quad (15)$$

and the same substitution in (13) gives

$$\begin{aligned} \mathcal{C}_{\text{PrivEd}}^{\mathbb{F}_p} &= \left(1 + \frac{\hat{\kappa}}{N}\right) \cdot \mathcal{C}_{\text{Input}}^{\mathbb{F}_p} \\ &\quad + m \left(1 + \frac{\hat{\kappa}(\ell + \lfloor \log N \rfloor + 1)}{Nm} + \frac{\hat{\kappa}(N + 1 + \ell)\ell_{\text{agg}}\eta}{Nm(T/2 - 2)}\right) \cdot \mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T} \\ &\quad + \frac{\hat{\kappa}}{N} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{F}_p} + \frac{\hat{\kappa}(\ell_{\text{agg}} + \kappa)}{N} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T}. \end{aligned} \quad (16)$$

Dropping the terms that vanish as $1/N$, but keeping ℓ_{agg} , which grows with the batch, we obtain

$$\mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_{2^\ell}} = \mathcal{C}_{\text{Input}}^{\mathbb{Z}_{2^\ell}} + mQ_{2^\ell} \cdot \mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T}, \quad Q_{2^\ell} = 1 + \frac{\kappa\eta}{T/2 - 2}, \quad (17)$$

and

$$\mathcal{C}_{\text{PrivEd}}^{\mathbb{F}_p} = \mathcal{C}_{\text{Input}}^{\mathbb{F}_p} + mQ_p \cdot \mathcal{C}_{\text{InputBit}}^{\mathbb{Z}_T}, \quad Q_p = 1 + \frac{\hat{\kappa}\ell_{\text{agg}}\eta}{m(T/2 - 2)}. \quad (18)$$

Both factors have the same shape, namely the number of subset-sum checks times the width of each tree times the η bits that the dealer re-inputs per position, divided by the m bits of the edabit and by the fan-in $E' - 1 = T/2 - 2$ of a tree node:

$$Q - 1 = \frac{(\text{checks}) \cdot (\text{width})}{m} \cdot \frac{\eta}{T/2 - 2},$$

¹⁷ Step 4 also consumes κ public elements of $\mathbb{Z}_T$, that is $\kappa\eta$ random bits, which we do not charge; see Section 7.

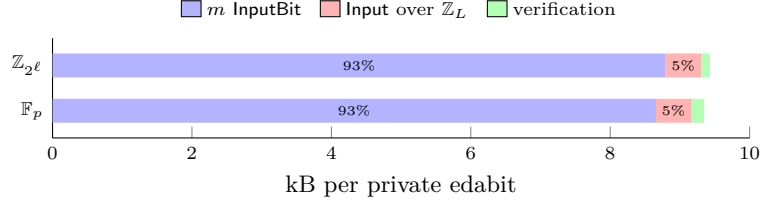


Fig. 8: Where the cost of one private edabit goes, namely (15) over rings and (16) over fields, priced as in Table 1 at $n = 64$ and $N = 2^{20}$, where it amounts to 9.4kB over $\mathbb{Z}_{2^{64}}$ and 9.3kB over a 64-bit field. Neither the cost nor the split depends on the corruption threshold, since the dealer inputs its own private edabit and the verification is over the whole batch, so one bar per domain suffices. The subset-sum verification is what the Q of (17) and (18) measures, and at this batch size it has amortized away to a few percent: a private edabit is, for all practical purposes, the m bits the dealer inputs.

with (κ, m) over rings and $(\hat{\kappa}, \ell_{\text{agg}})$ over fields, so that $Q_p - 1 = \frac{\hat{\kappa}}{\kappa} \cdot \frac{\ell_{\text{agg}}}{m} \cdot (Q_{2^\ell} - 1)$.

Asymptotically the two behave alike. With $T = 2^{\lceil 2 \log nd \rceil}$ and $\eta = \lceil 2 \log nd \rceil$ from Section 4.2, where $d = \lceil \log_2 n \rceil$ is the extension degree of the randomness extraction of Section 4.1, we have $T = \Theta(n^2 \log^2 n)$ and $\eta = \Theta(\log n)$, so

$$Q_{2^\ell} - 1 = \Theta\left(\frac{\kappa}{n^2 \log n}\right), \quad Q_p - 1 = \Theta\left(\frac{\hat{\kappa} \ell_{\text{agg}}}{m n^2 \log n}\right).$$

Both tend to 0 at the same rate, and differ only in the constant: the verification weighs most when there are few parties, and the field pays for it by the factor $\frac{\hat{\kappa} \ell_{\text{agg}}}{\kappa m}$ identified above. Figure 8 shows what remains of the verification at a realistic batch size.

E.5 Dabits

It remains to settle $\mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L}$, which enters (10) through the ν conversions. Over rings this is immediate: by Section 6.1 a dabit over $\mathbb{Z}_{2^\ell}$ is obtained from a single RandBit followed by a local reduction modulo 2, so

$$\mathcal{C}_{\text{dabit}}^{\mathbb{Z}_{2^\ell}} = \mathcal{C}_{\text{RandBit}}^{\mathbb{Z}_{2^\ell}}. \quad (19)$$

No such local map exists from $\mathbb{F}_p$ to $\mathbb{Z}_2$, and the rest of this subsection counts $\Pi_{\text{FieldDabits}}(p)$ of Section 6.2, which supplies $\mathcal{C}_{\text{dabit}}^{\mathbb{F}_p}$. It suffices to consider a single iteration of its loop, which produces $(n - t)dm$ dabits. Recall that $\ell = \lceil \log_2 p \rceil$ and that the private edabits consumed have length $m = \lfloor \log(p/(nd)) \rfloor$, the largest for which aggregating nd of them does not wrap around modulo p .

- Step 1 generates nd private edabits of length m over $\mathbb{F}_p$.
- Step 2 makes $(n - t)d$ calls to $\pi_{\text{AggEdabit}}^*$, the variant of $\pi_{\text{AggEdabit}}$ that keeps only the least significant m bits of the aggregate and discards the carries. In the notation of Appendix E.2 this is $\mu = m$ and $\nu = 0$, so its cost $\mathcal{C}_{\text{Agg}}^*$ is (10) with those two values; in particular it consumes no dabits, which is what keeps this count from being circular.
- Step 3 makes $(n - t)d$ calls to $\pi_{\text{FieldBitDec}}$, one bit decomposition per aggregate.
- Step 4 is local.

Dividing by the $(n - t)dm$ dabits produced,

$$\mathcal{C}_{\text{dabit}}^{\mathbb{F}_p} = \frac{n}{(n - t)m} \cdot \mathcal{C}_{\text{PrivEd}}^{\mathbb{F}_p} + \frac{1}{m} (\mathcal{C}_{\text{Agg}}^* + \mathcal{C}_{\pi_{\text{FieldBitDec}}}^{\mathbb{F}_p}), \quad (20)$$

where $\mathcal{C}_{\pi_{\text{FieldBitDec}}}^{\mathbb{F}_p}$ is given by (8). Equation (20) holds at any batch size, $\mathcal{C}_{\text{PrivEd}}^{\mathbb{F}_p}$ being either (16) or its large-batch form (18). On the other hand, $\mathcal{C}_{\text{Agg}}^*$ is (10) at $\mu = m$ and $\nu = 0$, namely

$$\mathcal{C}_{\text{Agg}}^* = \frac{m}{A} \cdot \left(\eta \cdot \mathcal{C}_{\text{RandBit}}^{\mathbb{Z}_T} + \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T} + \mathcal{C}_{\pi_{\text{BitAdder}}^{\mathbb{Z}_2}}^{\mathbb{Z}_2}(\log E) + \mathcal{C}_{\pi_{\text{BitAdder}}^{\mathbb{Z}_2}}^{\mathbb{Z}_2}(\eta) \right), \quad (21)$$

which, at the fan-in $E = nd$ of Step 2 and the choices $A = \log nd$ and $\eta = 2 \log nd$ of Section 4.2, becomes

$$\mathcal{C}_{\text{Agg}}^* = 2m \cdot \mathcal{C}_{\text{RandBit}}^{\mathbb{Z}_T} + \frac{m}{\log nd} \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_T} + 3m \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_2}. \quad (22)$$

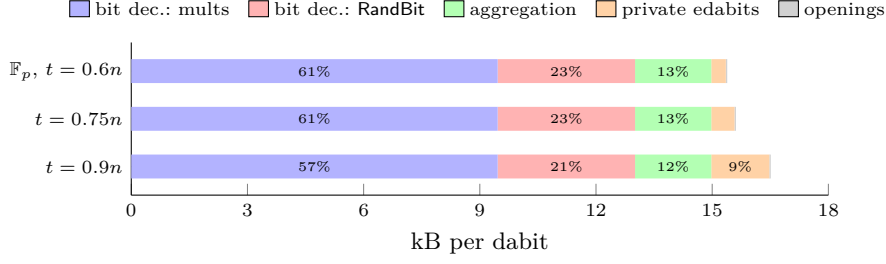


Fig. 9: Where the cost (20) of one field dabit goes, priced as in Table 1 at $n = 64$ and $N = 2^{20}$, where it amounts to 15.4, 15.6 and 16.5 kB for $t = 0.6n$, $0.75n$ and $0.9n$ respectively. The bit decomposition of (8) is the bulk of the cost, and the private edabits, which the randomness extraction of Section 4.1 makes $\frac{n}{(n-t)m}$ per dabit, go from 2% to 9% of the total as t grows; everything else is independent of t .

E.6 Asymptotics

The equations above are counts of $\mathcal{F}_{\text{ABB}}$ operations, whereas the main body states two asymptotic claims: that one global edabit costs $O(n\ell \log n)$ bits (Sections 4.2 and 3), and that one dabit over a field costs $O(n\ell)$ bits (Section 6). This subsection derives both.

We use throughout the pricing that every instantiation of Section 7 satisfies: one $\mathcal{F}_{\text{ABB}}$ operation over a domain of x bits costs $O(nx)$ bits, including RandBit and InputBit. Since a binary share is at most $\lceil \log_2 n \rceil$ bits wide, an operation over $\mathbb{Z}_2$ costs $O(n \log n)$ bits, and one over $\mathbb{Z}_T$ costs $O(n\eta)$ bits. Recall from Section 4.2 that $d = \lceil \log_2 n \rceil$, that the aggregator has fan-in $E = nd$, and that $A = \log E$ and $\eta = 2 \log E$, so that $\eta = \Theta(\log n)$ and $T = 2^\eta = \Theta(n^2 \log^2 n)$. We assume $t < (1-\epsilon)n$ for an arbitrary constant $\epsilon \in (0, 1)$, so that $\frac{n}{n-t} = O(1)$.

Private edabits. Both (17) and (18) consist of one input over the arithmetic domain and mQ calls to InputBit, and $Q = O(1)$ by the discussion following them. Hence, using $m \leq \ell$,

$$\mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L} = O(n\ell) + O(m) \cdot O(n\eta) = O(n\ell + nm \log n) = O(n\ell \log n).$$

Aggregation. In (10) a block costs η random bits and one opening over $\mathbb{Z}_T$, together with $\eta + \log E$ multiplications over $\mathbb{Z}_2$ by (5), so at $A = \log E$ and $\eta = 2 \log E$ each block is $O(n \log^2 n)$ bits and there are μ/A of them. With $\mu \leq m + \log E$ and $\nu \leq \log E$,

$$\mathcal{C}_{\text{Agg}}^{\mathbb{Z}_L} = O(n(m + \log n) \log n) + O(\log n) \cdot \mathcal{C}_{\pi_{\text{BitToArith}}^{\mathbb{Z}_L}},$$

which is the bound quoted in Section 3. By (9) the conversion costs $\mathcal{C}_{\pi_{\text{BitToArith}}^{\mathbb{Z}_L}} = \mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L} + O(n \log n)$, so everything now turns on the dabits.

Dabits. Over rings (19) gives $\mathcal{C}_{\text{dabit}}^{\mathbb{Z}_{2^\ell}} = \mathcal{C}_{\text{RandBit}}^{\mathbb{Z}_{2^\ell}} = O(n\ell)$ at once. Over fields we read (20) term by term, assuming in addition that $\ell - \log(nd) = \Theta(\ell)$, so that $m = \Theta(\ell)$. The first term is $\frac{n}{(n-t)m} \mathcal{C}_{\text{PrivEd}}^{\mathbb{F}_p} = O(n \log n)$ by the bound above. The second is $\frac{1}{m} \mathcal{C}_{\text{Agg}}^* = O(n \log n)$, since (22) is $O(m)$ operations over $\mathbb{Z}_T$ and over $\mathbb{Z}_2$. The third dominates: by (8) a bit decomposition consists of $O(\ell)$ random bits over $\mathbb{F}_p$, two comparisons and one addition of width ℓ , and a constant number of openings, and by (7) and (6) each comparison and the addition are $O(\ell)$ multiplications over $\mathbb{F}_p$; hence $\mathcal{C}_{\pi_{\text{FieldBitDec}}^{\mathbb{F}_p}} = O(n\ell^2)$ and $\frac{1}{m} \mathcal{C}_{\pi_{\text{FieldBitDec}}^{\mathbb{F}_p}} = O(n\ell)$. Adding the three, and absorbing the first two because $\ell > \log(nd)$, which follows from $m \geq 1$,

$$\mathcal{C}_{\text{dabit}}^{\mathbb{F}_p} = O(n\ell) \quad \text{bits per dabit,}$$

linear in both the number of parties and the bit length. This is the claim of Section 6, and it is what separates our dabits from those of Appendix F.1, whose cost per dabit is $\Theta(E)$ operations over $\mathbb{F}_p$ by (23) and hence $\Theta(n^2 \ell)$ bits.

Global edabits. Substituting into (11), the first term is $\frac{n}{n-t} \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L} = O(n\ell \log n)$ and the second is

$$\mathcal{C}_{\text{Agg}}^{\mathbb{Z}_L} = O(n(m + \log n) \log n) + O(\log n) \cdot O(n\ell) = O(n\ell \log n),$$

using $\mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L} = O(n\ell)$ in both domains and $m \leq \ell$. Therefore

$$\mathcal{C}_{\text{edabit}}^{\mathbb{Z}_L} = O(n\ell \log n) \quad \text{bits per global edabit}$$

whenever $t < (1 - \epsilon)n$, which is the quasi-linear bound claimed in Section 4.2. Each of the three ingredients is needed for it: without the randomness extraction of Section 4.1 the first term would carry a factor $\Theta(n)$, without the aggregator of Section 3 the μ/A blocks would be $\Theta(E)$ binary adders instead, and without the dabits of Section 6 the term $O(\log n) \cdot \mathcal{C}_{\text{dabit}}^{\mathbb{F}_p}$ would be $\Theta(n^2 \ell \log n)$ over fields.

F The Edabit Generation of [21]

In this appendix we recall the edabit generation protocol of [21], together with the dabit generation of [33] on which it relies. Our goal is not to restate these protocols formally, but to describe them in enough detail to count their communication. We follow the conventions of Appendix E throughout: every cost is a number of $\mathcal{F}_{\text{ABB}}$ operations, written as a linear combination of the quantities $\mathcal{C}_{\text{Cmd}}^{\mathbb{Z}_L}$ and reported per unit produced. We do not charge for calls to **Coin**, as argued in Section 7.

Throughout this appendix, N denotes the size of the batch being produced, m the length of the edabits, and E the number of private edabits that are aggregated to obtain one global edabit. The protocol of [21] is described with $E = n$, that is, every party contributes. However, for security it suffices that *one* of the aggregated private edabits comes from an honest party, so $E = t + 1$ is enough for any corruption threshold $t < n$. We use $E = t + 1$ throughout, which makes the comparison fair; see Remark 4.

The construction has three layers: dabits are produced first (Appendix F.1), consumed by the generation of *private* edabits (Appendix F.2), which in turn are aggregated into global edabits (Appendix F.3).

F.1 Dabits [33]

Recall that a dabit is a pair $(\llbracket b \rrbracket_L, \llbracket b \rrbracket_2)$ for a random $b \in \{0, 1\}$. When $L = 2^\ell$, a dabit costs a constant number of $\mathcal{F}_{\text{ABB}}$ operations: one **RandBit** over $\mathbb{Z}_{2^\ell}$, followed by a *local* reduction modulo 2, which yields a valid $\mathbb{Z}_2$ -sharing of the same bit (see Section 6). The cost of the **RandBit** itself is left as a symbol, exactly as in our own counts. No such local map exists from $\mathbb{F}_p$ to $\mathbb{Z}_2$, so over fields one must resort to the protocol below, which requires $\Theta(E)$ multiplications instead. It is therefore only relevant when $L = p$ is prime, and that is precisely where dabits become the bottleneck of [21].

The protocol $\Pi_{\text{daBits+MPC}}$ of [33, Fig. 6] produces a batch of N_d dabits at once, where we write N_d rather than N because the dabit batch is separate from the batch of N private edabits that consumes it. It is parameterized by two cut-and-choose parameters B_d and C_d , which must satisfy $C_d^{B_d} \cdot \binom{B_d N_d}{B_d} > 2^\kappa$ for the cut-and-choose analysis to go through. We describe the dabit protocol with binary domain $\mathbb{Z}_2$, which is the instantiation required by [21]; the protocol is stated in [33] over $\mathbb{F}_{2^k}$ for their garbling application, but the steps are identical and the analysis only simplifies. It proceeds in four stages.

- **Generate.** Each of the E contributing parties samples $C_d B_d N_d$ random bits and calls **Input** on each of them *twice*, once over $\mathbb{F}_p$ and once over $\mathbb{Z}_2$.
- **Cut and choose.** A single random subset of $(C_d - 1) B_d N_d$ positions is sampled with **Coin** and used for all parties, and every party's bits at these positions are opened in both domains. The parties abort unless every opened value lies in $\{0, 1\}$ and the two domains agree, which leaves $B_d N_d$ candidates per party.
- **Combine.** For each of the $B_d N_d$ surviving positions, the E parties' bits are XORed together. Over $\mathbb{Z}_2$ this is local, but over $\mathbb{F}_p$ each XOR must be emulated as $b + b' - 2bb'$, and therefore costs one **Mult**. Combining E bits thus costs $E - 1$ multiplications over $\mathbb{F}_p$ for each of the $B_d N_d$ positions, and this is the step that makes the protocol expensive.

- **Consistency check.** The $B_d N_d$ combined dabits are placed, using **Coin**, into N_d buckets of size B_d . In each bucket the first dabit is XORed with each of the other $B_d - 1$ dabits in both domains, and the two results are opened; again the parties abort unless the opened values are bits and agree across the two domains. Each such check costs one **Mult** and one **Open** over $\mathbb{F}_p$, and one **Open** over $\mathbb{Z}_2$. The first dabit of each bucket is output and the rest are discarded, yielding N_d dabits in total.

Dividing the total by N_d we obtain the cost per dabit, namely

$$\begin{aligned} \mathcal{C}_{\text{dabit}}^{\mathbb{F}_p} &= E C_d B_d \cdot \left(\mathcal{C}_{\text{Input}}^{\mathbb{F}_p} + \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2} \right) \\ &\quad + \left(B_d(E - 1) + B_d - 1 \right) \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{F}_p} \\ &\quad + \left(E(C_d - 1)B_d + B_d - 1 \right) \cdot \left(\mathcal{C}_{\text{Open}}^{\mathbb{F}_p} + \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} \right). \end{aligned} \quad (23)$$

Every term of (23) grows with B_d and with C_d , so both should be as small as the constraint allows, at the expense of a larger batch. Since [33] requires $B_d, C_d > 1$, the cheapest admissible choice is $B_d = C_d = 2$, which for $\kappa = 40$ forces $N_d \geq 370\,728 \approx 2^{18.5}$.¹⁸

By using $B_d = C_d = 2$, Equation (23) becomes

$$\begin{aligned} \mathcal{C}_{\text{dabit}}^{\mathbb{F}_p} &= 4E \cdot \left(\mathcal{C}_{\text{Input}}^{\mathbb{F}_p} + \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2} \right) \\ &\quad + (2E - 1) \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{F}_p} \\ &\quad + (2E + 1) \cdot \left(\mathcal{C}_{\text{Open}}^{\mathbb{F}_p} + \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} \right), \end{aligned} \quad (24)$$

which is the concrete cost of one dabit over fields. For comparison, the ring counterpart discussed above is simply

$$\mathcal{C}_{\text{dabit}}^{\mathbb{Z}_{2^\ell}} = \mathcal{C}_{\text{RandBit}}^{\mathbb{Z}_{2^\ell}}, \quad (25)$$

the local reduction modulo 2 being free.

Remark 3. The protocol of [33] is set in the $\mathcal{F}_{\text{MPC}}$ -hybrid model and makes no assumption on the access structure; as the authors observe, the access structure of the protocol is inherited entirely from the instantiation of $\mathcal{F}_{\text{MPC}}$. It is therefore applicable verbatim in every setting we consider, honest or dishonest majority and over rings or fields alike, which is what makes it the appropriate baseline for a comparison in the $\mathcal{F}_{\text{ABB}}$ model. We note that the later work of [3] gives a dabit generation protocol with a reported constant-factor improvement of up to $4\times$ over [33]. That protocol is not generic: it reasons directly about the least significant bits of the *shares* held by pairs of parties, and consequently requires a dedicated adaptation for Q_2 access structures. Since its cost per dabit remains $\Theta(n^2 \ell)$ bits, using it as the baseline would not change the asymptotics below.

F.2 Private Edabits

A private edabit is an edabit whose underlying secret is known to the party that produced it. The protocol $\Pi_{\text{edaBitsPriv}}$ of [21, Fig. 4] lets a dealer P_i produce a batch of N of them, and is parameterized by a bucket size B and two cut-and-choose parameters C, C' . The length is constrained by $m \leq \ell$ when $L = 2^\ell$, and by $m < \log_2 p$ when $L = p$. The latter is needed because the adder used in the check performs plain integer addition, so that the sum $r + s$ of two candidates must not wrap around modulo p ; since $r, s < 2^m$ this asks for $2^{m+1} \leq p$. By [21, Theorem 2] it is secure except with probability $2^{-\kappa}$ provided $N \geq 2^{\kappa/(B-1)}$ and $C = C' = B$ with $B \in \{3, 4, 5\}$. The same reasoning as for the dabits applies to the choice of B : every term of the cost below grows with it, so one takes the smallest admissible value $B = 3$ and accepts the resulting requirement $N \geq 2^{\kappa/2}$, which is 2^{20} for $\kappa = 40$.

The dealer first commits to $NB + C$ candidates, by inputting the m bits of each candidate over $\mathbb{Z}_2$ and the corresponding integer over $\mathbb{Z}_L$. It additionally inputs $(N(B - 1) + C')m$ multiplication

¹⁸ Larger parameters buy much smaller batches, for instance $B_d = 3$ and $C_d = 2$ needs only $N_d \geq 3127$, but we accept large batch sizes to make the complexity in the prior work as low as possible. Note that our work does not need these large batch sizes.

triples over $\mathbb{Z}_2$, which will be used to evaluate the binary adders below. That the dealer supplies these triples itself is an optimization already present in [21], and it is safe because the dealer knows all the underlying values anyway. The triples are not verified individually: the adders below are evaluated with a **FaultyMult** command that offers no correctness guarantee. Instead the cut-and-choose is analyzed so as to tolerate incorrect triples, which is also why [21] insists on a ripple-carry adder, that circuit being *weakly additively tamper resilient*.

The candidates are then checked by the $\Pi_{\text{CutNChoose}}$ procedure of [21, Fig. 5]:

1. The candidates and the triples are shuffled using two random permutations sampled with **Coin**, one for the $NB + C$ candidates and one for the $(N(B - 1) + C')m$ triples.
2. The first C edabit candidates are opened in both $\mathbb{Z}_2$ and $\mathbb{Z}_L$ domains, and the first $C'm$ triples are also opened; any inconsistency causes an abort. This step costs C openings over $\mathbb{Z}_L$, and $Cm + 3C'm$ openings over $\mathbb{Z}_2$, the latter because opening a triple means opening its three components.
3. The remaining NB candidates are placed, using **Coin**, into N buckets of size B , and the remaining triples into matching buckets of size $(B - 1)m$, one for every bucket of candidates.
4. In each bucket, the first candidate $(\llbracket r \rrbracket_L; \llbracket r_0 \rrbracket_2, \dots, \llbracket r_{m-1} \rrbracket_2)$ is checked against each of the other $B - 1$ candidates $(\llbracket s \rrbracket_L; \llbracket s_0 \rrbracket_2, \dots, \llbracket s_{m-1} \rrbracket_2)$ as follows.
 - (a) The parties add the two arithmetic parts $\llbracket r \rrbracket_L + \llbracket s \rrbracket_L$ locally, and add the two binary parts by evaluating a ripple-carry adder π_{BitAdder} , at a cost of $\mathcal{C}_{\pi_{\text{BitAdder}}}^{\mathbb{Z}_2}(m)$ by (5), that is m AND gates: both addends are secret and m bits long, and the carry out of the top position is the overflow bit c_m used in the next step. These gates are evaluated with m of the $N(B - 1)m$ available triples, at two openings over $\mathbb{Z}_2$ each.
 - (b) Writing $\llbracket c_m \rrbracket_2$ for the carry bit out of the adder, the parties obtain $\llbracket c_m \rrbracket_L$ from $\llbracket c_m \rrbracket_2$ by calling the procedure $\pi_{\text{BitToArith}}$ of Section 2 (which [21] calls **ConvertB2A**). This procedure consumes one dabits and one opening over $\mathbb{Z}_2$. If $L = 2^\ell$ and $\ell = m$ we skip it and set $\llbracket c_m \rrbracket_L := \llbracket 0 \rrbracket_L$, since $2^m \equiv 0 \pmod L$ makes the term $2^m \cdot \llbracket c_m \rrbracket_L$ vanish in the next step.
 - (c) The parties then open $\llbracket r + s \rrbracket_L - 2^m \cdot \llbracket c_m \rrbracket_L$, and also open the low m bits of the binary sum, and check they are the same. This costs one opening over $\mathbb{Z}_L$, and m openings over $\mathbb{Z}_2$.
5. There are N buckets and $B - 1$ checks in each of them, so the checks of Step 4 are performed $N(B - 1)$ times in total, for a total cost of $N(B - 1) \cdot \left((3m + 1) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} + \mathcal{C}_{\text{Open}}^{\mathbb{Z}_L} + \mathcal{C}_{\text{dabits}}^{\mathbb{Z}_L} \right)$ when Step 4(b) is performed, and $N(B - 1) \cdot \left(3m \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} + \mathcal{C}_{\text{Open}}^{\mathbb{Z}_L} \right)$ when it is skipped. If all of them pass, the first candidate of every bucket is output and the remaining ones are discarded, yielding the N verified private edabits.

Dividing by N , the cost of one private edabit for $\mathbb{Z}_L$, with either $L = p$, or $L = 2^\ell$ with $m < \ell$, is

$$\begin{aligned} \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L} = & \left(B + \frac{C}{N} \right) \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_L} + \left((4B - 3)m + \frac{(C + 3C')m}{N} \right) \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2} \\ & + \left(B - 1 + \frac{C}{N} \right) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_L} + \left((B - 1)(3m + 1) + \frac{(C + 3C')m}{N} \right) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} \\ & + (B - 1) \cdot \mathcal{C}_{\text{dabits}}^{\mathbb{Z}_L}, \end{aligned} \quad (26)$$

where the $1/N$ terms are the C candidates and $C'm$ triples that the dealer supplies over and above the NB and $N(B - 1)m$ actually consumed, together with the openings of Step 2 that dispose of them. They are a fixed cost per batch, so letting the batch grow they vanish and we obtain

$$\begin{aligned} \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L} = & B \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_L} + (4B - 3)m \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2} \\ & + (B - 1) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_L} + (B - 1)(3m + 1) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} \\ & + (B - 1) \cdot \mathcal{C}_{\text{dabits}}^{\mathbb{Z}_L}. \end{aligned} \quad (27)$$

Note that the dabits in the final term are *global* dabits, and they are consumed at a rate of $B - 1$ per private edabit. This is the single most expensive ingredient of the whole construction whenever L is prime: each such dabits costs $\Theta(E)$ operations over $\mathbb{F}_p$ by (23), whereas the remaining terms amount to only $\Theta(1)$ operations over $\mathbb{Z}_L$ and $\Theta(m)$ over $\mathbb{Z}_2$. In contrast, when $L = 2^\ell$ and $m = \ell$ no dabits is consumed, and the single opening over $\mathbb{Z}_2$ that masked the conversion disappears with

it, leaving 2ℓ openings from the AND gates and ℓ from Step 4(c) in each check. The cost of one private edabit is then, again for a large batch and with the same $1/N$ terms as in (26) otherwise,

$$\begin{aligned} \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_2^\ell} &= B \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2^\ell} + (4B - 3)\ell \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2} \\ &\quad + (B - 1) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2^\ell} + 3(B - 1)\ell \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2}. \end{aligned} \quad (28)$$

Instantiating (27) with the parameters $B = C = C' = 3$ fixed above gives, for $L = p$ or $L = 2^\ell$ with $m < \ell$:

$$\begin{aligned} \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L} &= 3 \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_L} + 9m \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2} \\ &\quad + 2 \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_L} + (6m + 2) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} \\ &\quad + 2 \cdot \mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L}. \end{aligned} \quad (29)$$

For $\mathbb{Z}_L$ with $L = 2^\ell$ and $\ell = m$ we have

$$\mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_2^\ell} = 3 \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2^\ell} + 9\ell \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2} + 2 \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2^\ell} + 6\ell \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2}. \quad (30)$$

At these parameters the $1/N$ terms dropped from (26) are $\frac{3}{N}$ extra inputs and openings over $\mathbb{Z}_L$, and $\frac{12m}{N}$ extra inputs and openings over $\mathbb{Z}_2$, per private edabit.

F.3 From Private Edabits to Global Edabits

Finally, Π_{edaBits} of [21, Fig. 3] turns E private edabits, at least one of which is guaranteed to come from an honest party, into a single global edabit whose secret is unknown to everyone. Each dealer contributes one private edabit to each global edabit, so producing a batch of N global edabits means running the protocol of Appendix F.2 once for each of the E dealers; this is the factor E multiplying $\mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L}$ below. The arithmetic parts are summed locally. The binary parts are summed by a tree of $E - 1$ two-input ripple-carry adders, producing $m + \log E$ bits. At level i there are $E/2^i$ adders, each acting on inputs of $m + i - 1$ bits and hence, by (5), using that many AND gates, for a total of

$$\sum_{i=1}^{\log E} \frac{E}{2^i} (m + i - 1) = (E - 1)m + E \sum_{i=1}^{\log E} \frac{i - 1}{2^i} = (E - 1)m + E - \log E - 1,$$

where the first step uses $E \sum_{i=1}^{\log E} 2^{-i} = E - 1$ and the second the identity $\sum_{i=1}^K (i - 1)2^{-i} = 1 - (K + 1)2^{-K}$ at $K = \log E$. We use the bound $(E - 1)m + E$ below, looser by only $\log E + 1$ gates, which keeps the expressions readable. It is still sharper than the bound $(m + \log E)(E - 1)$ of [21], which charges every adder the full width of the final output.

The top $\log E$ bits of the result, those in positions $m, \dots, m + \log E - 1$, are overflow bits and must be removed from the arithmetic part. This is done by converting each bit with $\pi_{\text{BitToArith}}$, consuming one dabit and one opening over $\mathbb{Z}_2$, and then subtracting $2^m \sum_{j=0}^{\log E - 1} 2^j \cdot \llbracket b_{j+m} \rrbracket_L$ from the arithmetic sum. Note however that, for $\mathbb{Z}_2^\ell$ and $\ell \leq m + \log E - 1$, only the bits in positions $m, m + 1, \dots, \ell - 1$ need to be converted, since those in positions $\ell, \dots, m + \log E - 1$ disappear modulo 2^ℓ . For $L = p$ no such optimization is available, since no power of two vanishes modulo a prime.

For $\mathbb{Z}_L$ with $L = p$, or $L = 2^\ell$ with $m + \log E - 1 < \ell$, the cost of one global edabit of length m is therefore

$$\mathcal{C}_{\text{edabit}}^{\mathbb{Z}_L} = E \cdot \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L} + \left((E - 1)m + E \right) \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_2} + \log E \cdot \left(\mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} + \mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L} \right), \quad (31)$$

where the three terms are, in order, the E private edabits that the dealers must supply, the AND gates of the adder tree, and the $\log E$ conversions of the overflow bits. The Mult's here are those of the adder tree only.

In contrast, for $\mathbb{Z}_L$ with $L = 2^\ell$ and $\ell \leq m + \log E - 1$, only the bits in positions $m, \dots, \ell - 1$ have to be converted, so the last term shrinks from $\log E$ to $\ell - m$ and the expression becomes

$$\mathcal{C}_{\text{edabit}}^{\mathbb{Z}_2^\ell} = E \cdot \mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_2^\ell} + \left((E - 1)m + E \right) \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_2} + (\ell - m) \cdot \left(\mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} + \mathcal{C}_{\text{dabit}}^{\mathbb{Z}_2^\ell} \right). \quad (32)$$

Note that $\mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_2^\ell}$ is here (27) when $m < \ell$ and (28) when $m = \ell$, the two differing by exactly the conversion performed inside $\Pi_{\text{CutNChoose}}$.

Equations (31) and (32) hold for any batch size, since $\mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_L}$ enters them as a symbol; substituting (26) into them therefore gives the cost at any N . Substituting instead the large-batch form of a private edabit gives the expanded forms below. It is convenient to regroup the cases here, since what distinguishes the two expressions is not the number of conversions in the aggregation, which only changes a coefficient, but whether a conversion is performed *at all*, which is what removes the dabits. We therefore write ν for the number of overflow bits that must be converted, so that $\nu = \log E$ in (31) and $\nu = \ell - m$ in (32), and split on the condition $2^m \equiv 0 \pmod L$. When $2^m \not\equiv 0 \pmod L$, that is for $L = p$ or for $L = 2^\ell$ with $m < \ell$, the conversion inside $\Pi_{\text{CutNChoose}}$ is performed and

$$\begin{aligned} \mathcal{C}_{\text{edabit}}^{\mathbb{Z}_L} &= EB \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_L} + E(4B - 3)m \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2} \\ &\quad + \left((E - 1)m + E\right) \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_2} \\ &\quad + E(B - 1) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_L} + \left(E(B - 1)(3m + 1) + \nu\right) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2} \\ &\quad + \left(E(B - 1) + \nu\right) \cdot \mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L}, \end{aligned} \tag{33}$$

where $\nu = \log E$ when $L = p$ and $\nu = \min\{\log E, \ell - m\}$ when $L = 2^\ell$. When instead $2^m \equiv 0 \pmod L$, which for $m \leq \ell$ happens exactly when $L = 2^\ell$ and $m = \ell$, we have $\nu = 0$ and no conversion is performed anywhere, neither inside $\Pi_{\text{CutNChoose}}$ nor in the aggregation, so that

$$\begin{aligned} \mathcal{C}_{\text{edabit}}^{\mathbb{Z}_2^\ell} &= EB \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2^\ell} + E(4B - 3)\ell \cdot \mathcal{C}_{\text{Input}}^{\mathbb{Z}_2} \\ &\quad + \left((E - 1)\ell + E\right) \cdot \mathcal{C}_{\text{Mult}}^{\mathbb{Z}_2} \\ &\quad + E(B - 1) \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2^\ell} + 3E(B - 1)\ell \cdot \mathcal{C}_{\text{Open}}^{\mathbb{Z}_2}. \end{aligned} \tag{34}$$

Comparing (33) with (34) already explains the two regimes. In both, the operations over the arithmetic domain number $\Theta(E) = \Theta(n)$, and every such operation costs $\Omega(n\ell)$ bits in any instantiation, so all terms other than $\mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L}$ contribute $\Omega(n^2\ell)$ bits per edabit. In the case (34) there are no dabits at all and this is the whole story, so the cost is at least quadratic in n . In the case (33), however, the $\mathcal{C}_{\text{dabit}}^{\mathbb{Z}_L}$ term requires $\Theta(n)$ dabits per edabit. Over a field each of these costs $\Omega(n^2\ell)$ bits by (23), so the total is at least cubic in n ; over a ring they are the cheap dabits of (25) and the total stays quadratic. In every case the cost is at least quadratic in n , which is what our protocol improves upon.

Remark 4. Taking $E = t + 1$ rather than $E = n$ is a strict improvement to the protocol of [21], and it is the version we compare against. Every term of (33) and of (34) is proportional to E , apart from the two contributions of the overflow conversions, which are smaller still. Using all n parties instead would therefore multiply the cost by roughly $n/(t + 1)$, which is about 2 in the honest majority setting $t < n/2$ and decreases towards 1 as t approaches n .

Remark 5. The number E of aggregated private edabits should not be confused with the number of parties that actually *execute* the protocol, which is what determines the price of each individual $\mathcal{F}_{\text{ABB}}$ operation. In principle the correlations could be produced by a committee of fewer than n parties and reshared to all n at the end; the committee must still tolerate t corruptions among its own members, so it needs at least $2t + 1$ parties to use honest majority tools and at least $t + 1$ otherwise. We do not consider this variant, for two reasons. First, it applies verbatim to our own protocols, so it offers no insight for the comparison. Second, in the settings we consider the corruption threshold is never below roughly $n/2$, so the committee would in any case comprise most of the parties. We therefore charge every operation at n parties for both constructions throughout.

F.4 Asymptotics

The equations above count $\mathcal{F}_{\text{ABB}}$ operations, whereas the claims made in the introduction and in Section 7 are in bits. This subsection turns the one into the other, in the same way and under the same conventions as Appendix E.6 does for our own protocols, so that the two are compared on equal terms.

One $\mathcal{F}_{\text{ABB}}$ operation over a domain of x bits costs $O(nx)$ bits in the instantiation of Table 1, and $\Omega(nx)$ bits in every instantiation we consider, since each of the n parties holds a share of the value. An operation over $\mathbb{Z}_2$ accordingly costs $\Omega(n)$ bits, and $O(n \log n)$ when a binary share is $\lceil \log_2 n \rceil$ bits wide. We take the bucket parameters B, C, C' of Appendix F.2 and B_d, C_d of Appendix F.1 to be constants, which is what a large batch buys, and we let $t = \Theta(n)$, so that the number $E = t + 1$ of private edabits aggregated into one global edabit is $\Theta(n)$.

Dabits. Over $\mathbb{Z}_{2^\ell}$ a dabit is a single RandBit by (25), so $\mathcal{C}_{\text{dabit}}^{\mathbb{Z}_{2^\ell}} = O(n\ell)$ bits, the same as ours. Over $\mathbb{F}_p$ every term of (23) is $\Theta(E)$, amounting to $\Theta(E) = \Theta(n)$ operations over $\mathbb{F}_p$ and as many over $\mathbb{Z}_2$, the latter being no more expensive than the former as long as $\log_2 n = O(\ell)$, so that

$$\mathcal{C}_{\text{dabit}}^{\mathbb{F}_p} = \Theta(n^2\ell) \quad \text{bits per dabit.}$$

This is structural rather than an artifact of the parameters: the protocol combines one contribution per contributing party, and over a prime field each of the $\Theta(n)$ XORs of that combination must be emulated by a multiplication, as noted in the *Combine* step of Appendix F.1. It is the quantity that Section 6 improves to $O(n\ell)$.

Private edabits. Both (27) and (28) consist of $\Theta(1)$ operations over $\mathbb{Z}_L$ and $\Theta(m)$ over $\mathbb{Z}_2$, the first of them also consuming $B - 1 = \Theta(1)$ dabits. The operations alone are between $\Omega(n\ell)$ and $O(n\ell \log n)$ bits, using $m \leq \ell$, so over $\mathbb{Z}_{2^\ell}$ with $m = \ell$, where (28) applies and no dabit is consumed,

$$\mathcal{C}_{\text{PrivEd}}^{\mathbb{Z}_{2^\ell}} = \Omega(n\ell) \quad \text{and} \quad O(n\ell \log n).$$

Over $\mathbb{F}_p$ the dabits dominate everything else by a factor $\Theta(n)$, and

$$\mathcal{C}_{\text{PrivEd}}^{\mathbb{F}_p} = \Theta(n^2\ell) \quad \text{bits per private edabit.}$$

Global edabits. Equations (33) and (34) add three contributions: the E private edabits, the $\Theta(Em) = \Theta(nm)$ AND gates of the adder tree, and the $\nu = O(\log n)$ conversions of the overflow bits. The second is $\Omega(n^2m)$ bits on its own, and the third, which is $O(n\ell \log n)$ over rings and $\Theta(n^2\ell \log n)$ over fields, stays below the first in both domains. Over $\mathbb{Z}_{2^\ell}$ the first is $\Theta(n) \cdot \Omega(n\ell)$, so

$$\mathcal{C}_{\text{edabit}}^{\mathbb{Z}_{2^\ell}} = \Omega(n^2\ell) \quad \text{bits per global edabit,}$$

and $O(n^2\ell \log n)$ in the other direction. Over $\mathbb{F}_p$ the first is $\Theta(n) \cdot \Theta(n^2\ell)$ and swallows the rest, so that

$$\mathcal{C}_{\text{edabit}}^{\mathbb{F}_p} = \Theta(n^3\ell) \quad \text{bits per global edabit.}$$

Two of these three contributions are quadratic for reasons that have nothing to do with the dabits: producing the $\Theta(n)$ private edabits that one global edabit consumes costs $\Omega(n^2\ell)$ bits whatever a private edabit is made of, and the adder tree costs $\Omega(n^2m)$ bits whatever is added. Supplying the protocol of [21] with the cheap dabits of Section 6 therefore removes the factor n that separates the field case from the ring case, but leaves a cost that is still quadratic in the number of parties; this is the comparison reported in Section 7.2. Against the $O(n\ell \log n)$ bits per global edabit of Appendix E.6, the separation is thus $\Omega(n/\log n)$ over $\mathbb{Z}_{2^\ell}$, and $\Omega(n^2/\log n)$ over $\mathbb{F}_p$ when [21] is left with the dabits of [33].

G Appendix to Section 7

G.1 Fine-grained edabit comparison

Our edabit generation improves on [21] in two places, and this subsection measures each separately. The first is the generation of the private edabits of Section 5. The second is the aggregation of Section 3 which, combined with the randomness extraction of Section 4.1, also reduces the number of private edabits per global edabit. Throughout this subsection both protocols are given *our* dabits, so that the improvement of Section 7.1 is not counted twice; over $\mathbb{Z}_{2^\ell}$ this makes no difference, since neither construction consumes dabits there.

The two private edabits are different objects. A private edabit of [21] carries its bits over $\mathbb{Z}_2$, whereas ours carries them over $\mathbb{Z}_T$, so neither generation protocol can be plugged into the other as it stands. The mismatch is not symmetric, however. Since $T = 2^\eta$, reducing a sharing over $\mathbb{Z}_T$ modulo 2 is the same local map that produces a dabait from a random ring element in Section 6, and the verification of $\Pi_{\text{PrivEd}}(2^\ell, m)$ and $\Pi_{\text{PrivEd}}(p, m)$ already guarantees that the shared values are bits and that they decompose the arithmetic part. Our private edabits therefore become theirs at no cost.¹⁹ Our private edabits are thus a drop-in replacement inside the protocol of [21], and the comparison below is between two ways of producing the same object.

Private edabits. Figure 10 plots the two private edabit generation protocols against each other. The two costs are (27) and (28) of Appendix F.2 against (17) and (18) of Appendix E.4, and they differ in how they check for incorrect edabits, which is what such a protocol really pays for.

The protocol of [21] verifies by cut and choose: the dealer supplies B candidates for every private edabit that survives, and in each bucket the first candidate is checked against the other $B - 1$ by adding the two binary parts with a ripple-carry adder and opening the result against the arithmetic sum. Every term of (27) therefore carries a factor B , and B cannot be made small: by [21, Theorem 2] the analysis needs $N \geq 2^{\kappa/(B-1)}$, so the three batch sizes of Section 7 force $B = 5, 4$ and 3 , and no batch that one would run in practice brings it to 2 . This is the gap between the dashed curves of Figure 10, one per batch size, and it is the same mechanism that makes their global edabits sensitive to the batch size N in Figure 4. Our protocols instead verify a whole batch at once, with κ subset-sum checks over $\text{GR}(2^\ell, \kappa)$ in $\Pi_{\text{PrivEd}}(2^\ell, m)$ and $\hat{\kappa}$ of them over $\text{GF}(p^{\kappa_p})$ in $\Pi_{\text{PrivEd}}(p, m)$. Fig. 8 shows that producing a private edabit then costs essentially what inputting it costs.

Asymptotically the private edabits from [21] cost $\Theta(n\ell)$ bits and ours $O(n\ell \log n)$. However, the crossing point is far away: their private edabits become cheaper only at $n \approx 1100$, are 6% cheaper at $n = 2^{11}$, and even at $n = 2^{16}$ are cheaper by less than a factor 1.5. With very few parties it is the verification, not the bits, that makes ours expensive, which is why the solid curves fall before they rise.

Over $\mathbb{F}_p$ the improvement is even larger (note the y -axis scale in Fig. 10). The reason is that their private edabits consume $B - 1$ dabits each, whereas ours consume *none*, in either domain. This is because their check has to *compute* a shared value before opening it, and ours does not. Even given our dabits, those two are 31 of the 45 kB of one of their private edabits at $n = 64$ and $N = 2^{20}$, the whole of the factor 4 to 5 separating the two families of curves in the right panel.

Aggregation. Figure 11 runs both protocols end to end on *our* private edabits, so that what is left between the two curves is the aggregation together with the extraction it enables. The aggregation itself, per global edabit produced, is generally more expensive for us: the binary adder tree of [21], which is the $(E - 1)m + E$ AND gates of (31) plus, over $\mathbb{F}_p$, the $\log E$ conversions of the overflow bits, against our (10). Asymptotically theirs is $\Theta(n^2\ell)$ bits per output and ours is $O(n\ell \log n)$, but with few parties ours is the more expensive: an adder tree is cheap when there is little to add, whereas each of our μ/A blocks needs η random bits over $\mathbb{Z}_T$ however few parties there are. Their aggregation is in fact cheaper than ours until $n \approx 85$ over rings and $n \approx 100$ over fields; at $n = 16$ it is 4 times cheaper, and at $n = 64$ still 1.3 times.

Our main advantage is not the aggregation itself but the rate: prior work consumes $t + 1$ private edabits per global one, while we require $\frac{n}{n-t}$; at $n = 64$ and $t = 0.75n$ this is 49 against 4.

G.2 Maximal adversary case: $t = n - 1$

All of our protocols are secure for any $t < n$, and in particular for the maximal threshold $t = n - 1$, where a single honest party remains. The quasi-linear claim of this work requires a constant honest fraction: the randomness extraction of Section 4.1 turns nd private edabits into $(n - t)d$ global ones, so a global edabit consumes $\frac{n}{n-t}$ of them, and this is $O(1)$ only while $t < (1 - \epsilon)n$. At $t = n - 1$ it degenerates to n , exactly the number $E = t + 1 = n$ that [21] aggregates, leading to quadratic communication. The extraction therefore buys nothing in this regime, and this section measures the improvements in the other parts of our protocol.

¹⁹ In the other direction each bit would have to be lifted from $\mathbb{Z}_2$ to $\mathbb{Z}_T$, which costs a conversion per bit and is more expensive than either generation protocol.

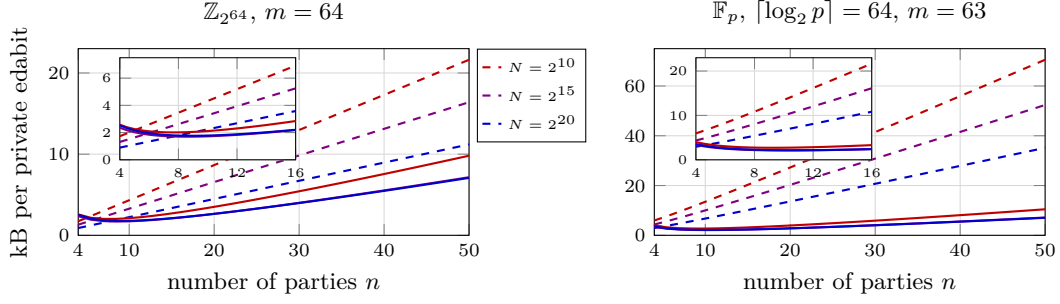


Fig. 10: Communication per *private* edabit of [21], dashed, against $\Pi_{\text{PrivEd}}(2^\ell, m)$ and $\Pi_{\text{PrivEd}}(p, m)$, solid, for the three batch sizes and the bucket size $B = 5, 4, 3$ that each of them forces on [21]. Neither protocol depends on the corruption threshold over $\mathbb{Z}_{2^\ell}$; over $\mathbb{F}_p$ theirs does, through the $B - 1$ dabits it consumes, and we fix $t = 0.75n$ and our dabits for those. The insets repeat the region $n \leq 16$, where our subset-sum verification is still expensive because the domain $\mathbb{Z}_T$ of the tree is small.

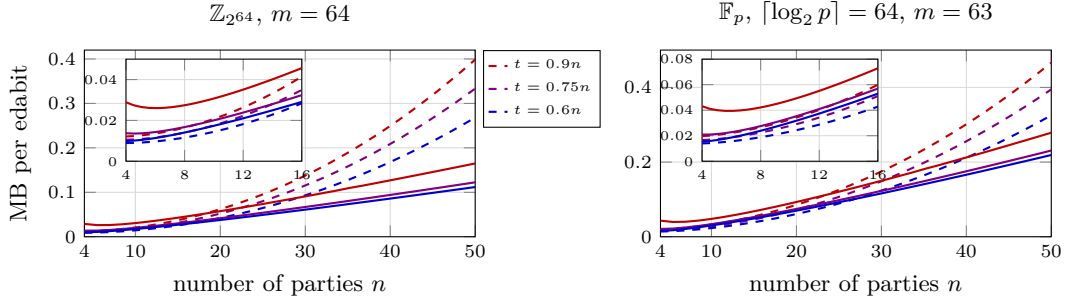


Fig. 11: Communication per global edabit at $N = 2^{20}$ when *both* protocols are given our private edabits and our dabits, so that only the aggregation differs. Dashed: the aggregation of [21], which consumes $t + 1$ of them per output. Solid: ours, which consumes $\frac{n}{n-t}$. The insets repeat the region $n \leq 16$: with the private edabits equalized, our aggregation only becomes the cheaper of the two between $n = 14$ and $n = 27$, against $n \approx 6$ – 12 for the protocols as a whole in Figure 3.

Dabits. Our dabits remain substantially cheaper. Both constructions become quadratic in the number of parties: the extraction rate of (20) degrades to $\frac{n}{m_{\text{db}}}$ private edabits per dabbit, where $m_{\text{db}} = \lceil \log_2(p/nd) \rceil$ is the length of the private edabits that the dabbit protocol consumes, written m in Appendix E.5, so our first term takes over and grows as $\Theta(n^2 \log n)$ bits, against the $\Theta(n^2 \ell)$ of [33]. The quadratic term is therefore present on both sides, but ours is paid over $\mathbb{Z}_T$, a domain of $\eta = \lceil 2 \log_2 nd \rceil$ bits, and theirs over $\mathbb{F}_p$, which leaves a factor of order $\ell / \log n$. Concretely, at $N = 2^{20}$ one dabbit costs us 25 kB against their 527 kB at $n = 64$, and 972 kB against 33.8 MB at $n = 512$, that is factors of 21 and 35; at $N = 2^{10}$, where their cut-and-choose parameters are worse, the same factors are 35 and 46. Unlike the factors of Table 2, these stop growing: they peak around $n \approx 500$, at 46 for $N = 2^{10}$ and 35 for $N = 2^{20}$, and then decrease as slowly as $\ell / \log n$ does, so that the improvement is a large constant rather than a linear gain.

Global edabits. The answer depends on the domain; Figure 12 shows both. Over $\mathbb{Z}_{2^\ell}$ we are essentially at parity. Both constructions now consume n private edabits per global edabit, so each is n private edabits plus one aggregation, and the comparison collapses onto the private-edabit comparison of Section G.1, up to the two aggregations, which are a similar fraction of each total. We are accordingly behind for very few parties, ahead by at most a factor 2 in the middle of the range, and behind again for very large n : at $N = 2^{20}$ the factor is 0.29 at $n = 4$, 1.11 at $n = 16$, 1.44 at $n = 64$, 1.23 at $n = 512$, and it falls back below 1 around $n = 4000$; at $N = 2^{10}$, where their bucket size is $B = 5$, it is above 1 from $n = 8$ on and stays there over the whole range we computed, reaching 1.01 at $n = 2^{16}$. The explanation is short: at $t = n - 1$ our cost is $\Theta(n^2 \ell \log n)$ bits per global edabit against their $\Theta(n^2 \ell)$, the extra $\log n$ being the width of $\mathbb{Z}_T$ that we already paid for in Section G.1, and the constants favor us between $n \approx 8$ and $n \approx 4000$.

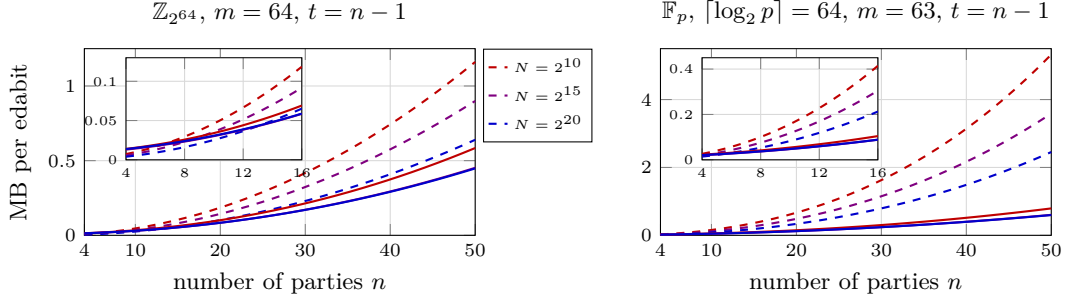


Fig. 12: Communication per global edabit at the maximal threshold $t = n - 1$, for the three batch sizes, with both sides using our dabits. Dashed: [21], with the bucket size $B = 5, 4, 3$ that each batch forces on it. Solid: ours, in the same color. Over $\mathbb{Z}_{2^\ell}$, where the extraction no longer buys anything, the two constructions stay within a factor 2 of each other from $n \approx 8$ on; over $\mathbb{F}_p$ the dabits that their private edabits consume keep the two apart. The insets repeat the region $n \leq 16$.

Over $\mathbb{F}_p$ we keep a growing advantage, for the reason identified in Section G.1 rather than any reason of rate: their private edabits consume $B - 1$ dabits each and ours none, and at $t = n - 1$ dabits are expensive for everybody. Even after giving them our dabits, their cost per global edabit is $\Theta(n^3 \log n)$ bits against our $\Theta(n^2 \ell \log n)$, a factor $\Theta(n/\ell)$; at $N = 2^{20}$ this is $2.4\times$ at $n = 16$, $4.6\times$ at $n = 64$ and $16\times$ at $n = 512$. With the dabits of [33] that [21] actually specifies, the same numbers are $14\times$, $75\times$ and $532\times$.

Summary. At maximal corruption our improvements come from the ingredients and not from the rate: dabits that avoid $\Theta(n)$ field multiplications, and private edabits verified in a single batched check that consume no dabits. The aggregation and the extraction, which make the honest-fraction case quasi-linear, contribute nothing here, and over $\mathbb{Z}_{2^\ell}$ the price of the domain $\mathbb{Z}_T$ they require makes us asymptotically a factor $\log n$ more expensive than [21], though not within any practical number of parties. A user in the maximal-corruption regime should therefore take our dabits and our private edabits, and may aggregate them with either aggregator.

G.3 Honest majority without PCGs

Everything so far was priced in the dishonest majority instantiation of Table 1, where sharings are additive and authenticated and where, following Section 7, the preprocessing is assumed to come from pseudorandom correlation generators. We now redo the comparison in the opposite corner of the design space: honest majority $t < n/2$, Shamir secret sharing, and no PCGs, so every random and double sharing is produced by the parties themselves and charged. Calls to Coin remain free, as argued in Section 7.

Extensions. Shamir sharing needs $n + 1$ distinct evaluation points, which a small domain lacks, so every domain that is not a large field must be extended. Over $\mathbb{Z}_2$ the parties share in $\text{GF}(2^k)$, and over the two power-of-two rings in the Galois rings $\text{GR}(T, k)$ and $\text{GR}(2^\ell, k)$ [2], in both cases with

$$k = \lceil \log_2(n + 1) \rceil,$$

so that one share of an x -bit value costs kx bits rather than the x bits of an additive sharing. The prime field needs no extension, since $p > 2^{63} > n$ throughout. This asymmetry is the structural difference with Table 1. Over $\mathbb{Z}_{2^\ell}$ it is neutral: every operation of both constructions lives in a power-of-two domain, so k cancels. Over $\mathbb{F}_p$ it is not: all of our operations carry it, with the exception of the bit decompositions inside our dabits, whereas the $\mathbb{F}_p$ operations of [21] and nearly all of [33] escape it.²⁰

²⁰ We make no use of reverse multiplication friendly embeddings [12] or of any other technique that amortizes the cost of these extensions.

bits	Input	Open	Mult	RandBit		InputBit
	nkx	$2nkx$	$6nkx$	over $\mathbb{Z}_{2^x}$	over $\mathbb{F}_p$	over $\mathbb{Z}_T$
				$10nk(x+2)$	$10nx$	(3)

Table 5: Communication charged per $\mathcal{F}_{\text{ABB}}$ operation over a domain of x bits, among n parties, in the honest majority instantiation with Shamir sharing and information-theoretic security. The factor $k = \lceil \log_2(n+1) \rceil$ is the degree of the extension that $\mathbb{Z}_2$, $\mathbb{Z}_T$ and $\mathbb{Z}_{2^t}$ require, and is absent over $\mathbb{F}_p$. Unlike Table 1, these prices include the preprocessing: a random sharing costs $2nkx$ bits and the double sharing of a multiplication $4nkx$.

Instantiation. Table 5 gives the resulting prices. A party inputs a value by sending one share to each other party, and an opening goes through a designated party who collects the shares and sends back the result. A multiplication is the protocol of [19]: the parties consume a double sharing $(\llbracket r \rrbracket_t, \llbracket r \rrbracket_{2t})$, reconstruct the masked product through the designated party and send it back, which is two elements per party, and the double sharing itself is produced from the parties' own contributions with a Vandermonde extraction, at an amortized four elements per party. A plain random sharing costs two elements per party by the same extraction, and RandBit is then one random sharing, one multiplication and one opening, run modulo 2^{x+2} over rings as in Table 1. Consistency of the dealt shares and correctness of the multiplications are verified in one batched check, sublinear in the number of operations, which we ignore as Table 1 ignores the MAC check. These constants are standard and deliberately conservative; sharper protocols exist [23] and would scale both sides alike. The cost of InputBit is again (3), now evaluated at these prices.

The comparison. Everything becomes more expensive in absolute terms: one global edabit over $\mathbb{Z}_{2^{64}}$ at $n = 64$, $t = n/2$ and $N = 2^{20}$ costs us 1.4 MB here against 145 kB in the dishonest majority instantiation, a factor 9, of which the extension accounts for $k \approx 6$ and the rest is the preprocessing that is now charged. Figure 13 redraws Figure 4 in this instantiation, and the improvement factors are collected in Table 6. They are smaller than those of Tables 2 and 3 but of the same nature: at $N = 2^{20}$ we improve on [21] by $2.5\times$ at $n = 64$ and $15\times$ at $n = 512$ over rings, and by $3.1\times$ and $17\times$ over fields, against $3.7\times$, $22\times$, $5.6\times$ and $30\times$ in Table 3. The crossover moves to the right by a few parties: at $N = 2^{20}$ from $n \approx 12$ to $n \approx 19$ over rings and from $n \approx 6$ to $n \approx 11$ over fields, and at $N = 2^{10}$ from $n \approx 6$ to $n \approx 9$ and from $n \approx 4$ to $n \approx 5$. Our dabits improve on [33] by $8.4\times$ at $n = 64$ and $43\times$ at $n = 512$, against $18\times$ and $125\times$ in Table 2. We evaluate at $t = n/2$ throughout, as in the first row of Tables 2 and 3; taking the largest admissible threshold $t = \lfloor (n-1)/2 \rfloor$ instead lowers every factor slightly, by about 10% at $n = 16$ and by less than 3% from $n = 64$ on.

Two effects work against us, and they are worth telling apart. The first is that the preprocessing is now charged, which raises RandBit from 6 to 10 units and Mult from 4 to 6 while leaving inputs and openings where they were. This hits our aggregator, between 57% and 92% of our cost by Figure 7 and dominated by RandBit, harder than the cut-and-choose of [21], which is made of inputs and openings. Over $\mathbb{Z}_{2^t}$ this is the whole story, the extension cancelling as explained above: at $n = 64$ our cost grows by $9.4\times$ and theirs by $6.4\times$, and the factor drops from 3.7 to 2.5. The second is that over $\mathbb{F}_p$ the extension does not cancel, for the reason given above: their cost grows by only $3.6\times$ against our $6.4\times$, and the factor drops from 5.6 to 3.1. Neither effect touches the rate, and the honest majority remains the best case for our randomness extraction, one global edabit consuming $\frac{n}{n-t} = 2$ private edabits where [21] consumes $t+1$, a ratio of about $n/4$; this is why the factors are roughly halved but keep growing linearly with n . The same factor k enters the asymptotics: the bounds of Appendix E.6 assume that an operation over a domain of x bits costs $O(nx)$ bits, so in this instantiation they acquire a factor $k = \Theta(\log n)$ and read $O(n\ell \log^2 n)$ bits per global edabit; the operations of [21] over $\mathbb{Z}_2$ and $\mathbb{Z}_{2^t}$, which dominate its cost, acquire the same factor, so the separation between the two remains $\Omega(n/\log n)$. An instantiation that amortizes the binary domain, for instance through reverse multiplication friendly embeddings, would recover part of the second effect over $\mathbb{F}_p$, removing the factor k from our operations while their $\mathbb{F}_p$ operations have none to remove; we have not pursued it.

Functionalities

Functionality 1: $\mathcal{F}_{\text{ABB}}[L]$ 6

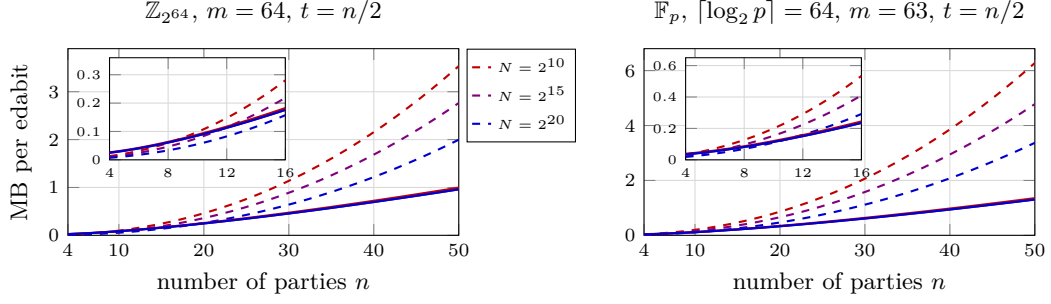


Fig. 13: Communication per global edabit at $t = n/2$ in the honest majority instantiation of Table 5, for the three batch sizes, with both sides using our dabits. Dashed: [21], with the bucket size $B = 5, 4, 3$ that each batch forces on it. Solid: ours, in the same color. The picture is the one of Figure 4, an order of magnitude more expensive and with the curves closer together: the extension that Shamir sharing needs over $\mathbb{Z}_2$, $\mathbb{Z}_T$ and $\mathbb{Z}_{2^\ell}$, and the preprocessing that is now charged, weigh more on us than on [21]. The insets repeat the region $n \leq 16$.

n	$N = 2^{10}$			$N = 2^{20}$		
	dabit	edabit $\mathbb{Z}_{2^\ell}$	edabit $\mathbb{F}_p$	dabit	edabit $\mathbb{Z}_{2^\ell}$	edabit $\mathbb{F}_p$
4	2.3	.49	.88	1.1	.28	.53
8	3.8	.95	1.5	1.9	.55	.85
16	6.1	1.5	2.2	3.0	.89	1.2
64	16	4.3	5.6	8.4	2.5	3.1
128	28	7.6	9.5	14	4.5	5.3
256	48	14	17	25	8.0	9.3
512	83	25	30	43	15	17

Table 6: The factors by which our constructions improve on [33] and [21] at $t = n/2$, in the honest majority instantiation of Table 5. As in Section 7.2, both sides use our dabits in the two edabit columns. A factor below 1 means that we are more expensive.

Functionality 2: $\mathcal{F}_{\text{PrivEd}}[L, m]$	7
Functionality 3: $\mathcal{F}_{\text{Dabits}}[L]$	8
Functionality 4: $\mathcal{F}_{\text{Edabits}}[L, m]$	8
Functionality 5: $\mathcal{F}_{\text{ABBBasic}}[L]$	23
Functionality 6: $\mathcal{F}_{\text{AnyRangeEdabits}}[L, P]$	26

Procedures

Procedure 1: $\pi_{\text{AggEdabit}}$	9
Procedure 3: $\pi_{\text{TreeAggregate}}((\llbracket r_{1,j} \rrbracket_T)_{j=0}^{m-1}, \dots, (\llbracket r_{N',j} \rrbracket_T)_{j=0}^{m-1})$	13
Procedure 8: $\pi_{\text{FieldBitDec}}(\llbracket a \rrbracket_p)$	26
Procedure 9: $\pi_{\text{BitToArith}}(\llbracket b \rrbracket_2)$	26

Protocols

Protocol 2: $\Pi_{\text{Edabits}}(L, m)$	11
Protocol 4: $\Pi_{\text{PrivEd}}(2^\ell, m)$	13
Protocol 5: $\Pi_{\text{PrivEd}}(p, m)$	14
Protocol 6: $\Pi_{\text{FieldDabits}}(p)$	16
Protocol 7: $\Pi_{\text{InputBit}}(L, C)$	24
Protocol 10: $\Pi_{\text{AnyRangeEdabits}}(L, P)$	26
Protocol 11: $\Pi_{\text{RingDabits}}(2^\ell)$	27