UFO$_S$: An Ultra-fast Toolkit for Multiparty Computation on Small Elements

Jiacheng Gao, Moyang Xie, Yuan Zhang, *Member, IEEE*, Sheng Zhong, *Fellow, IEEE*

***Abstract*—In many secure multiparty computation (MPC) applications, the semantic domains of some operands are much smaller than the underlying arithmetic field or ring. The arithmetic domain is typically chosen to meet computation and security requirements (e.g., a 128-bit prime field), whereas the actual data may consist of only a few bits, such as 4-bit values in a 16-category classification task. This mismatch can lead to substantial communication and computation overhead in existing MPC protocols, which typically treat data of different ranges uniformly.**

We introduce UFO$_s$, an ultra-fast toolkit for MPC on small elements, with optimized protocols for one-hot encoding, comparison, and digit decomposition. For $t = 2^k$, our random one-hot vector generator attains, in the fan-in-two arithmetic-circuit model, the lower bounds of $t - \log t - 1$ multiplication gates and $\lceil \log \log t \rceil$ multiplication depth. Our comparison protocol applies to any public small domain, while DigitDecomp avoids binary expansion and supports any public base $t > 1$. Using these primitives, we construct a higher-radix sorting protocol for large field elements. Experiments confirm the resulting gains in random one-hot vector generation, digit decomposition, and sorting. In particular, our sorting protocol achieves a $3.4\times$ online and $1.4\times$ offline speedup over Hamada et al.'s radix-sort baseline when sorting 2^{16} elements among five parties.

***Index Terms*—Multiparty computation, one-hot encoding, secure comparison, digit decomposition, secure sorting.**

I. Introduction

Secure multiparty computation (MPC) enables multiple parties to jointly compute a function over their private inputs while keeping those inputs secret. This powerful cryptographic technique has found applications across various domains, including privacy-preserving data analysis [1], secure voting systems [2], confidential financial computations [3], and federated learning [4].

In a typical MPC setting, the data to be processed are often encoded as elements in a finite field $\mathbb{F}_p$ or ring $\mathbb{Z}_{2^\ell}$, whose modulus is selected according to the arithmetic-range requirements. For example, the MP-SPDZ framework [5] uses a 128-bit prime field as its default configuration, while ABY^3 [6] performs computations over the ring $\mathbb{Z}_{2^{64}}$. Such choices provide sufficient range to prevent unintended wraparound [7, 8, 9].

However, in many practical applications, the actual data domain is far smaller than the underlying encoding space, and this mismatch is pronounced in real-world deployments. For instance, Wang et al. [4] extend Crypten [10] to construct a secure machine learning framework for classification tasks. Their method uses an MPC comparison protocol to encode features into one-hot vectors. However, a mismatch arises: the dataset has at most 15 categories (4 bits), while Crypten operates on a 64-bit integer ring $\mathbb{Z}_{2^{64}}$. Although the surrounding computation may still require this full-sized ring, these comparisons depend only on the 4-bit semantic range and nevertheless process full-width elements. This known small operand range should enable more efficient protocols. Most prior protocols do not exploit it.

In this work, we focus on efficient MPC protocols for small elements. These are elements whose sizes are much smaller than the underlying field or ring. We present UFO$_s$, an ultra-fast toolkit for multiparty computation on small elements. UFO$_s$ contains highly optimized protocols for three fundamental tasks: one-hot encoding, comparison, and digit decomposition. One-hot encoding and comparison are widely used in machine-learning classification tasks [4, 10, 11, 12]. Recent secure Transformer systems expose concrete uses for these building blocks. Ditto [13] uses low-bit quantization and conversions between ring representations, a setting where our base-t digit decomposition fits. BumbleBee [14] converts private token IDs to one-hot vectors for embedding lookup and invokes equality and less-than for token encoding and max/argmax. Mosformer [15] builds maliciously secure nonlinear layers on secure comparison. These examples show the wide applicability of the UFO$_s$ toolkit.

The efficiency of UFO$_s$ stems from several key insights. Prior one-hot encoding constructions rely on random one-hot vectors, whose generation forms the main bottleneck: for example, [16] first generates random bits and then converts them into a one-hot vector. To address this, for $t = 2^k$ we propose an ROHV generation protocol using $t-\log t-1$ multiplications and $O(\log \log t)$ rounds. It matches the multiplication count of Morita et al. while reducing their $O(\log t)$ round complexity. In addition, we propose a constant-round approach to one-hot encoding that avoids random one-hot vectors altogether and instead constructs the encoding using a technique based on Lagrange interpolation. Building on efficient one-hot encodings, we observe that several protocols based on binary representations [17, 18, 19, 20] can instead be implemented using one-hot representations, thereby dispensing with many bit operations. As an example, we design an efficient secure integer- comparison protocol using one-hot representations.

Jiacheng Gao, Moyang Xie, Yuan Zhang, and Sheng Zhong are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: jcgao@smail.nju.edu.cn, xie_moyang@foxmail.com, zhangyuan@nju.edu.cn, zhongsheng@nju.edu.cn).

Jiacheng Gao and Moyang Xie contributed equally to this work.

Some prior digit decomposition protocols, such as [20], lack an efficient integer comparison protocol and therefore rely on bit expansion followed by bitwise comparisons, which leads to a large number of bit operations. By leveraging our integer comparison protocol, this bottleneck can be eliminated, enabling a more efficient digit decomposition.

Using these primitives, we construct a radix sort protocol for large field elements. Operating on base-t digits instead of individual bits reduces the number of radix passes and shuffle invocations by a factor of $\log t$ relative to prior bit-level designs [21, 22, 23, 24]. For 2^{16} elements and five parties, our implementation achieves a $3.4\times$ online and $1.4\times$ offline speedup over Hamada et al.'s radix-sort baseline.

Our contributions are summarized as follows:

1) For $t = 2^k$, we construct an ROHV generator using $t - \log t - 1$ multiplications and $\lceil \log \log t \rceil$ multiplication depth, attaining the corresponding lower bounds for fan-in-two arithmetic circuits [25]. We also give a constant-round one-hot encoding protocol for arbitrary t.
2) We use one-hot representations to realize small-domain comparison and arbitrary-base digit decomposition. Relative to Ning et al. [20], direct digitwise subtraction avoids digit-binary expansion and reduces the number of bit subtractions by a factor of $\lceil \log t \rceil$.
3) We construct IntRadixSort, which reduces radix passes and shuffles by a factor of $\log t$, and give a cost model for selecting t. For 2^{16} elements and five parties, it achieves a $3.4\times$ online speedup and 75.4% lower online communication than Hamada et al.'s radix-sort baseline.

The rest of this paper is organized as follows. Section II reviews related work. Section III introduces the preliminaries. Section IV presents our one-hot encoding protocol. Section V presents our digit decomposition protocol. Section VI describes our radix sort protocol. Section VII reports the experimental evaluation. Finally, Section VIII concludes the paper.

II. Related Work

A. Related Work on One-Hot Encoding

Despite the widespread use of one-hot vectors, only a few works study one-hot encoding in multiparty computation. Attrapadung et al. [26] proposed a two-party protocol based on oblivious transfer (OT). Morita et al. [16] proposed a protocol based on additive and replicated secret sharing. Due to the use of replicated secret sharing, their protocol supports only a small number of parties unless the number of corrupted parties is bounded by a small constant (e.g., ≤ 2).

These works share a common approach. To encode a secret-shared integer $0 \leq x < t$ into a one-hot vector of length t, the parties first generate a secret-shared random one-hot vector (ROHV) $\vec{b} = (b_0, \ldots, b_{t-1})$ with exactly one $b_i = 1$ in the offline phase. In the online phase, they open $d = x - i \bmod t$ and locally shift $\vec{b}$ by d positions so that the 1 moves to position x. To generate such a random one-hot vector, Attrapadung et al. [26] rely on OT, where the sender prepares t random one-hot vectors of length t and the receiver obliviously retrieves one, incurring $O(t^2)$ computation. Morita et al. [16] generate a random one-hot vector of power-of-two length t using $\log t$ random bits, $t - \log t - 1$ multiplications, and $O(\log t)$ rounds. Although the randomness usage is optimal, the round complexity is linear in $\log t$. Furthermore, [26] assume that x and i are shared in $\mathbb{Z}_t$, while [16] require t to be a power of two. These assumptions limit applicability, since in many settings x is shared in a large field $\mathbb{F}_p$ and t may not be a power of two.

To address this, we propose a new protocol for generating ROHVs and new protocols for one-hot encoding with arbitrary lengths. For power-of-two length t, our optimized ROHV generation protocol uses the same $t - \log t - 1$ multiplications but only $O(\log \log t)$ rounds on the m-party linear-sharing backend assumed here. For an arbitrary target length t, our one-hot encoding protocol consumes one ROHV of length $2^{\lceil \log t \rceil}$. We also present a direct construction based on the prefix-multiplication protocol of [8] and Lagrange interpolation, which produces a one-hot vector for a given index with $3t - 4$ multiplication invocations in two rounds and no random bits. Its input-independent first round can be preprocessed, leaving $t - 1$ online invocations. Neither encoding protocol requires x to be shared in $\mathbb{Z}_t$.

B. Related Work on Comparison

Secure comparison commonly relies on binary representations, but obtaining them can be expensive. Several protocols therefore avoid bit decomposition, e.g., [8, 17, 18, 19], but generally require secret-shared random bits and run in $O(\log |\mathbb{F}_p|)$ rounds.

Our one-hot comparison instead exploits the public small domain directly. For secret values in $[t]$, our comparison protocol runs in only $O(t)$ communication and $O(1)$ rounds, avoiding a full-domain comparison.

C. Related Work on Decomposition

The most widely used decomposition method is **bit decomposition**, which decomposes a secret-shared integer into its secret-shared binary representation. It was introduced by Damgård et al. [27], who also applied it to integer comparison and exponentiation. The key idea is to mask the input integer with random bits, open the masked value and decompose it in plaintext. The decomposed bits are unmasked via bitwise addition with the random bits. It was subsequently improved by Toft et al. [28], Reistad et al. [9], and Veugen et al. [29]. For mixed arithmetic-binary computation, edaBits [30] provide correlated arithmetic and binary sharings for conversion, comparison, and truncation.

Bit decomposition is useful for certain non-linear operations [27] and can speed up computations (e.g., radix sort [23]).

In this paper we consider **digit decomposition**, a generalization of bit decomposition, which decomposes an integer into its representation in base t for any integer $t > 1$. Existing works mainly study digit decomposition in the two-party additive secret sharing setting [31, 32, 33], typically assuming that t is a power of 2. In this case, parties can locally truncate their

shares into l-bit chunks (for $t = 2^l$), and then compute the carry bits to correct the chunks.

The closest related work is by Ning et al. [20], which also studies digit decomposition in the multiparty setting. Given an L-bit integer, their protocol decomposes it in base t with digit length $l = \lceil \log t \rceil$. The approach follows the masking paradigm used in bit decomposition: the parties generate random digits, mask the input, open the masked value, decompose it in plaintext, and then remove the mask. Instead of performing digitwise subtraction directly, their protocol converts the result into a *digit-binary representation*, where each digit is represented in l binary bits, yielding $\lceil L/l \rceil$ digits in total. The mask is then removed via bitwise subtraction on the resulting binary strings of length $\lceil L/l \rceil \cdot l$. This conversion avoids digitwise subtraction, which is difficult to implement securely because it requires handling carries between digits, typically involving secure comparison with the constant t.

In our construction, we directly perform digitwise subtraction to remove the mask. This is enabled by our one-hot encoding protocol, which provides an efficient comparison functionality. In this way, our protocol only requires bitwise subtraction of two ($\lceil L/l \rceil$)-bit strings, avoiding the expansion of each digit into l binary bits. Consequently, the number of required bit subtractions is reduced by a factor of l relative to that in [20].

D. Related Work on MPC Sorting

Sorting, as a key primitive in many protocols, has been extensively studied in secure computation. There are currently three main approaches to sorting in multiparty computation, which we discuss below.

A straightforward approach is to directly implement a **comparison-based sorting** algorithm. Comparison-based sorting protocols are typically built from atomic compare-swap operations, which compare two elements and conditionally swap them. This approach is conceptually simple but generally inefficient. Since comparison outcomes cannot be revealed, efficient data-dependent algorithms such as quicksort and mergesort are not applicable. Existing constructions require either $O(n^2)$ comparisons (e.g., [34, 35]) or $O(n \log^2 n)$ comparisons (e.g., [36, 37]). Constructions using $O(n \log n)$ comparisons either succeed only with some probability (e.g., [38, 39, 40]) or have prohibitively large constants (e.g., [41, 42]).

To enable more efficient data-dependent sorting, Hamada et al. [43] proposed the **shuffle-then-sort** paradigm, which shuffles the input before sorting. This allows comparison outcomes to be safely revealed, as the correspondence between positions and elements is concealed.

Given the high cost of secure comparison, **radix sort** has been widely studied in MPC to avoid comparison operations. Based on the "shuffle-then-reveal" technique, Bogdanov et al. [44] and Laud et al. [22] proposed radix sort protocols that take a bitwise-shared array as input. They process the keys from the least significant bit (LSB) to the most significant bit (MSB), shuffling and revealing one bit per round. The communication cost is $O(nL^2)$, where L is the bit length of the keys. Hamada et al. [23] reduced this to $O(nL)$ by shuffling only constant-length vectors per round, and Chida et al. [24] further optimized the design for passive-secure three-party computation with one corruption. Asharov et al. [45] give passive- and active-secure three-party radix sort based on replicated secret sharing that avoids shuffling the full items after every radix step. Agarwal et al. [35] consider one semi-honest corruption and use input-independent preprocessing and FSS to minimize online communication in two- and three-party protocols based on additive and replicated secret sharing, with most variants taking only two or three online rounds.[1]

Although radix sort improves efficiency in MPC, existing constructions [21, 22, 23, 24] suffer from round complexity linear in the bit length L, as they require bit-decomposed keys and LSB-to-MSB processing. If each shuffle invocation costs c rounds, the total becomes Lc. Moreover, c may not be constant; for example, Keller and Scholl's shuffle [46] has $c = O(m \log n)$ for m parties and n items, yielding an overall round complexity of $O(mL \log n)$. Hence, current MPC radix sort protocols scale poorly with L.

To address this limitation, we apply our digit decomposition protocol to represent the input keys in base t (e.g., $t = 16$ or 256), and use one-hot encoding to perform radix sort directly on the base-t digits. This reduces the number of rounds and shuffles by a factor of $\log t$. Although one-hot encoding introduces additional overhead, the reduced shuffling significantly lowers the communication and computation costs, resulting in improved overall performance.

III. Preliminaries

A. Basic Notations

Let m parties $P_1, \ldots, P_m$ participate in the computation, and suppose there are n integers to be processed (e.g., shuffled or sorted depending on the context), each of bit length L. Let t denote the public small-domain size. Let κ denote the statistical security parameter, and let $\mathbb{F}_p$ be the finite field of prime order p. Throughout the paper, secret values are encoded in $\mathbb{F}_p$ with $p > 2^L$.

We assume

$$p > 2^{L+\lceil \log t \rceil}(m2^\kappa + 2), \tag{III.1}$$

which is sufficient to avoid wrap-around in all masked openings below.

For integers $a \le b$, let $[a, b] := \{a, a+1, \ldots, b\}$ and $[a] := \{0, 1, \ldots, a-1\}$. For a vector $\vec{v}$, denote its i-th entry by v_i. Vector, array, and matrix indices start from zero.

B. Network and Security Model

We assume that every pair of parties is connected by a secure channel providing confidentiality and authentication. No broadcast channel is assumed; thus broadcasting a message requires sending $m - 1$ copies.

[1]These works use fixed two- or three-party settings, and [35, 45] additionally rely on additive or replicated sharing. Our implementation uses m-party Shamir sharing. Because the party, sharing, and threat models differ, raw running-time comparisons would not be matched. Our empirical claims are therefore limited to the implemented baselines.

We consider a static, semi-honest adversary that corrupts at most f parties, where $f < m/2$. Corrupted parties follow the protocol but may collude and attempt to infer additional information from their joint view. The security claims in this paper cover this semi-honest model. A malicious-secure instantiation additionally requires input-validity checks, unbiased generation of the protocol randomness, consistent openings, and abort handling; substituting malicious-secure primitives alone does not provide these guarantees. We leave this extension to future work.

C. Secret Sharing Scheme

Let $[\![x]\!]$ denote the secret sharing of a value x among all parties. For convenience, we sometimes write $[\![x_0, \ldots, x_{n-1}]\!]$ to denote the secret-shared vector $[\![\vec{x}]\!] = ([\![x_0]\!], \ldots, [\![x_{n-1}]\!])$. Our constructions can be instantiated with any linear secret sharing scheme that supports the following operations:

1) $[\![x]\!] \leftarrow \mathrm{Share}(P_i : x)$: party P_i shares a secret x among all parties.
2) $x \leftarrow \mathrm{Open}([\![x]\!])$: reconstruct the secret value x to all parties.
3) $[\![z]\!] \leftarrow \mathrm{Add}([\![x]\!], [\![y]\!])$: compute $[\![z]\!] = [\![x + y]\!]$, where $[\![x]\!]$ and $[\![y]\!]$ are secret-shared values. Since the scheme is linear, addition is performed locally.
4) $[\![z]\!] \leftarrow \mathrm{Mul}([\![x]\!], [\![y]\!])$: compute $[\![z]\!] = [\![xy]\!]$.
5) $[\![r]\!] \leftarrow \mathrm{Rand}()$: sample $r \xleftarrow{\$} \mathbb{F}_p$ and share it among all parties.
6) $[\![b]\!] \leftarrow \mathrm{RandBit}()$: sample a secret random bit b and share it (see [8] for an implementation).

For simplicity, we also write $[\![x]\!]+[\![y]\!]$ and $[\![x]\!]\cdot[\![y]\!]$ to denote $\mathrm{Add}([\![x]\!], [\![y]\!])$ and $\mathrm{Mul}([\![x]\!], [\![y]\!])$, respectively. Besides these basic operations, we require two additional primitives:

RandSum: $[\![r]\!] \leftarrow \mathrm{RandSum}()$. We require a primitive that generates a secret-shared random value equal to the sum of m random values, each sampled from $[2^\kappa]$. This randomness is used in our one-hot encoding protocol to hide the encoded index. The same technique also appears in [8, 9]. Concretely, each party P_i samples $r_i \xleftarrow{\$} [2^\kappa]$, shares $[\![r_i]\!]$ with all parties, and the parties locally compute $[\![r]\!] = \sum_{i=1}^{m} [\![r_i]\!]$.

Shuffle: $[\![\vec{v}_i]\!]_{i\in[n]} \leftarrow \mathrm{Shuffle}([\![\vec{u}_i]\!]_{i\in[n]})$. An MPC shuffle protocol takes as input secret-shared vectors $[\![\vec{u}_i]\!]_{i\in[n]}$ and outputs a randomly permuted version $[\![\vec{u}_{\pi(i)}]\!]_{i\in[n]}$, where π is a secret permutation over $[n]$. Since π is randomly chosen and unknown to all parties, we do not explicitly denote it. Efficient shuffle protocols are well established in MPC [46, 47, 48]. In this paper we instantiate the protocol of [46], which requires $O(m \log n)$ rounds and $O(m^2 n \log n)$ communication.

IV. One-Hot Encoding

A one-hot vector is a binary vector with exactly one entry being 1. In this section, we give our construction for encoding an integer into a one-hot vector, which serves as a key building block for our subsequent constructions.

Given a secret-shared value $[\![i]\!]$ with $i \in [t]$, the one-hot encoding protocol outputs its one-hot representation

$$[\![\vec{s}]\!] = [\![s_0, s_1, \ldots, s_{t-1}]\!], \tag{IV.1}$$

with $s_i = 1$ and $s_j = 0$ for all $j \neq i$. We call i the "index" of the one-hot vector, where $s_i = 1$ is the only "hot" bit.

Let $T := 2^{\lceil \log t \rceil}$. The generation of a one-hot vector can be divided into two phases, an offline one and an online one, which we present in Section IV-A and Section IV-B, respectively. In the offline phase, the parties generate a random one-hot vector for a uniformly random index in $[T]$. In the online phase, the parties use any such random one-hot vector, along with its secret-shared index, to create the index's one-hot vector, with slight overhead. This paradigm is similar to Beaver triples: most computations are performed before the input $[\![i]\!]$ becomes available.

A. Random One-Hot Vector

This protocol generates a one-hot vector whose secret index $[\![r]\!]$ is uniform in $[T]$. The output consists of the vector and its secret-shared index, i.e., the tuple $([\![\vec{s}]\!], [\![r]\!])$ with $s_r = 1$. Note that both the vector and its index are kept secret in the process.

To this end, we first propose a demux protocol that converts random bits into a one-hot vector, which is formally described in Algorithm 1. The protocol works recursively, generating two one-hot vectors $\vec{b}$ and $\vec{c}$ of length $2^{\lfloor k/2 \rfloor}$ and $2^{\lceil k/2 \rceil}$, and combines them into a one-hot vector of length $N = 2^k$ using secure multiplications.

Algorithm 1 $([\![s_0]\!], ..., [\![s_{2^k-1}]\!]) \leftarrow \mathrm{Demux}([\![r_0]\!], ..., [\![r_{k-1}]\!])$

Ensure: A one-hot vector with index $\sum_{i=0}^{k-1} 2^i [\![r_i]\!]$.
1: **if** $k = 1$ **then**
2: Return $(1 - [\![r_0]\!], [\![r_0]\!])$
3: **end if**
4: $\overline{k} \leftarrow \lfloor k/2 \rfloor$
5: $[\![b_0]\!], [\![b_1]\!], \ldots, [\![b_{2^{\overline{k}}-1}]\!] \leftarrow \mathrm{Demux}([\![r_0]\!], \ldots, [\![r_{\overline{k}-1}]\!])$
6: $[\![c_0]\!], [\![c_1]\!], \ldots, [\![c_{2^{k-\overline{k}}-1}]\!] \leftarrow \mathrm{Demux}([\![r_{\overline{k}}]\!], \ldots, [\![r_{k-1}]\!])$
7: **for** $i = 0$ to $2^{\overline{k}} - 2$ **do parallel**
8: **for** $j = 0$ to $2^{k-\overline{k}} - 2$ **do parallel**
9: $[\![s_{i+2^{\overline{k}}\cdot j}]\!] \leftarrow \mathrm{Mul}([\![b_i]\!], [\![c_j]\!])$
10: **end for**
11: **end for**
12: **for** $i = 0$ to $2^{\overline{k}} - 2$ **do**
13: $[\![s_{i+2^{\overline{k}}\cdot(2^{k-\overline{k}}-1)}]\!] \leftarrow [\![b_i]\!] - \sum_{j=0}^{2^{k-\overline{k}}-2} [\![s_{i+2^{\overline{k}}\cdot j}]\!]$
14: **end for**
15: **for** $j = 0$ to $2^{k-\overline{k}} - 2$ **do**
16: $[\![s_{2^{\overline{k}}-1+2^{\overline{k}}\cdot j}]\!] \leftarrow [\![c_j]\!] - \sum_{i=0}^{2^{\overline{k}}-2} [\![s_{i+2^{\overline{k}}\cdot j}]\!]$
17: **end for**
18: $[\![s_{2^k-1}]\!] \leftarrow [\![b_{2^{\overline{k}}-1}]\!] - \sum_{j=0}^{2^{k-\overline{k}}-2} [\![s_{2^{\overline{k}}-1+2^{\overline{k}}\cdot j}]\!]$
19: **return** $[\![s_0, \ldots, s_{2^k-1}]\!]$

Let $\overline{k} = \lfloor k/2 \rfloor$. To understand the last steps of Algorithm 1, note that, since both $\vec{b}$ and $\vec{c}$ are one-hot, for all $i \in [2^{\overline{k}} - 1]$

and $j \in [2^{k-\overline{k}} - 1]$,

$$b_i \cdot c_{2^{k-\overline{k}}-1} = b_i - \sum_{j'=0}^{2^{k-\overline{k}}-2} b_i \cdot c_{j'}, \tag{IV.2}$$

$$b_{2^{\overline{k}}-1} \cdot c_j = c_j - \sum_{i'=0}^{2^{\overline{k}}-2} b_{i'} \cdot c_j. \tag{IV.3}$$

Eq. IV.3 recovers the last row, and the final entry follows from

$$b_{2^{\overline{k}}-1} \cdot c_{2^{k-\overline{k}}-1} = b_{2^{\overline{k}}-1} - \sum_{j=0}^{2^{k-\overline{k}}-2} b_{2^{\overline{k}}-1} \cdot c_j. \tag{IV.4}$$

Our demux protocol requires $2^k - k - 1$ secure multiplications and $\lceil \log k \rceil$ multiplication rounds. The round complexity follows from the recursive structure of the protocol. For the number of multiplications, let $M(k)$ denote the number of secure multiplications. A direct induction gives

$$\begin{aligned} M(k) &= M(\overline{k}) + M(k-\overline{k}) + (2^{\overline{k}} - 1)(2^{k-\overline{k}} - 1) \\ &= 2^k - k - 1. \end{aligned} \tag{IV.5}$$

A comparison with existing ROHV generation protocols is provided in Table III.

For a public target length t, our $\mathrm{RandOneHot}(t)$ protocol uses Demux to generate an internal random one-hot vector of length T as follows:

1) The parties fetch $\log T$ random bits $[\![r_0]\!], \ldots, [\![r_{\log T-1}]\!]$.
2) Each party locally computes $[\![r]\!] = \sum_{i=0}^{\log T-1} 2^i [\![r_i]\!]$.
3) The parties run Demux on these bits to generate the length-T random one-hot vector with index $[\![r]\!]$.

When t is a power of two, $T = t$. Otherwise, the output is a padded internal vector used by Encode, which shifts it and truncates it to the requested length t; $\mathrm{RandOneHot}$ itself does not claim to generate a length-t ROHV in this case.

The protocol requires exactly $T - \log T - 1$ secure multiplications and $\lceil \log \log T \rceil$ multiplication rounds. It incurs $O(Tm) = O(tm)$ computation and communication in the offline phase, where each atomic operation is treated as $O(1)$, since the protocol does not depend on a specific secret-sharing implementation.

Algorithm 2 $([\![\vec{s}]\!], [\![r]\!]) \leftarrow \mathrm{RandOneHot}(t)$

Ensure: $[\![\vec{s}]\!]$ is a one-hot vector of length T, with index $[\![r]\!]$.
1: Let $T \leftarrow 2^{\lceil \log t \rceil}$
2: **for** $i = 0$ to $\log T - 1$ **do**
3: $\quad [\![r_i]\!] \leftarrow \mathrm{RandBit}()$
4: **end for**
5: $[\![r]\!] \leftarrow \sum_{i=0}^{\log T-1} 2^i \cdot [\![r_i]\!]$
6: $[\![\vec{s}]\!] \leftarrow \mathrm{Demux}([\![r_0]\!], \ldots, [\![r_{\log T-1}]\!])$
7: **return** $([\![\vec{s}]\!], [\![r]\!])$

B. One-Hot Encoding

Our target is to generate a one-hot vector for a secret-shared index $[\![k]\!]$, with the help of the random one-hot vector $[\![\vec{s}]\!]$ and its index $[\![r]\!]$. The intuition is that opening $(k - r) \bmod T$ to all parties is safe because the opened value is uniformly distributed over $[T]$. With this value, which is the offset between the random index and the target index, each party can shift locally the hot bit to the correct entry.

Our construction works almost the same as this intuition, except that instead of performing a modulo operation on secret-shared data, we perform it on a public value. A secret-shared modulo operation is costly and requires another $\log T$ fresh random bits [8], whereas reducing a public value has negligible local cost.

Recall that $k - r \in [-(T-1), t-1]$ is not uniformly distributed, and thus directly opening it is unsafe. Hence, we add to it a random "sufficiently large" mask that is divisible by T, and then open it. This mask is produced by the $\mathrm{RandSum}$ protocol. The field-size bound in Eq. III.1 implies

$$p > T + t - 1 + Tm(2^\kappa - 1), \tag{IV.6}$$

so the sum does not overflow modulo p. We will prove later that this is statistically secure with respect to κ.

See Algorithm 3 for the formal description of the protocol.

The $\mathrm{Encode}(t, [\![k]\!])$ protocol requires, in the offline phase, $T - \log T - 1$ secure multiplications and $\lceil \log \log T \rceil$ multiplication rounds, resulting in $O(Tm) = O(tm)$ computational and communication costs. In addition, one communication round with $O(m)$ communication and $O(tm)$ computation is incurred in either the online or offline phase, depending on whether $[\![k]\!]$ is generated online.

Algorithm 3 $([\![\vec{s}]\!]) \leftarrow \mathrm{Encode}(t, [\![k]\!])$

Require: $k \in [t]$.
Ensure: $[\![\vec{s}]\!]$ is a one-hot vector of length t, with index $[\![k]\!]$.
1: Let $T \leftarrow 2^{\lceil \log t \rceil}$
2: $([\![\vec{s}']\!], [\![r]\!]) \leftarrow \mathrm{RandOneHot}(t)$
3: $[\![d]\!] \leftarrow \mathrm{RandSum}()$
4: $[\![\delta']\!] \leftarrow [\![k]\!] - [\![r]\!] + T + T \cdot [\![d]\!]$
5: $\delta' \leftarrow \mathrm{Open}([\![\delta']\!])$
6: $\delta \leftarrow \delta' \bmod T$
7: **for** $j = 0$ to $T - 1$ **do**
8: $\quad$ Each P_i sets $[\![s_j]\!]_i \leftarrow [\![s'_{((j+T-\delta) \bmod T)}]\!]_i$
9: **end for**
10: **return** $[\![\vec{s}]\!] = [\![s_0, \ldots, s_{t-1}]\!]$ ▷ truncate to length t

C. One-Hot Encoding in Constant Rounds

We next present $\mathrm{EncodeC}$, a constant-round one-hot encoding protocol that avoids ROHV generation and random bits. For target length t, it calls $\mathrm{PreMulC}$ on $t - 1$ inputs, using $3t - 4$ multiplication invocations in total. Of these, $2t - 3$ are input-independent and can be preprocessed, leaving $t - 1$ online invocations in one round. $\mathrm{EncodeC}$ targets the setting when ROHV or random bits are unavailable, exhausted, or too expensive.

$\mathrm{EncodeC}$ combines public equality-indicator polynomials with the constant-round prefix-multiplication protocol $\mathrm{PreMulC}$ of Catrina and de Hoogh [8], which builds on the

construction of Bar-Ilan and Beaver [49]. To formalize, for nonzero $x_0, \ldots, x_{q-1} \in \mathbb{F}_p$,

$$(\llbracket p_0 \rrbracket, \ldots, \llbracket p_{q-1} \rrbracket) \leftarrow \mathrm{PreMulC}(\llbracket x_0 \rrbracket, \ldots, \llbracket x_{q-1} \rrbracket),$$

where $p_i = \prod_{j=0}^{i} x_j$ for all $i \in [q]$.

Now for $i \in [1,t]$, let F_i be the polynomial of degree $t-1$, with $F_i(i) = 1$ and $F_i(j) = 0$ for all $j \in [1,t]\backslash\{i\}$. This polynomial is unique and can be computed via Lagrange interpolation. Recall that Eq. III.1 implies $p > t$, so $1, \ldots, t$ are distinct and both the Lagrange denominators and the PreMulC input $k+1$ are nonzero in $\mathbb{F}_p$. For any shared $x \in [1,t]$, by acquiring $\llbracket x^j \rrbracket$ for $j \in [1, t-1]$ via PreMulC, parties can now locally evaluate $F_i(\llbracket x \rrbracket)$ for all $i \in [1,t]$, as the coefficients of F_i are all public. Hence, by evaluating $\llbracket s_{i-1} \rrbracket \leftarrow F_i(\llbracket x \rrbracket)$, we obtain the one-hot vector with index $\llbracket x - 1 \rrbracket$: only the polynomial F_i with $i = x$ will be evaluated to 1. A plain implementation of this idea requires $O(t^2)$ local computation per party.

If $\mathbb{F}_p$ is NTT-friendly, the above interpolations take $O(t \log t)$ local computation. Let the Vandermonde matrix be $V_{i,j} = i^j$ for $i \in [1,t]$ and $j \in [0, t-1]$. All evaluations can be written as one public linear transform $(V^{-1})^{\mathsf{T}}(1, \llbracket x \rrbracket, \ldots, \llbracket x^{t-1} \rrbracket)^{\mathsf{T}}$. Using a number-theoretic transform (NTT), this matrix multiplication takes $O(t \log t)$ local operations after padding.[2]

We denote this protocol by EncodeC. This protocol is formally described in Algorithm 4.

The $\mathrm{EncodeC}(t, \llbracket k \rrbracket)$ protocol uses $3t-4$ multiplication invocations in total and two communication rounds. Its input-independent first round uses $2t-3$ invocations and can be preprocessed, leaving $t-1$ online invocations in one round. Its computation cost is $O(tm \log t)$ with NTT or $O(t^2 m)$ otherwise, and its communication cost is $O(tm)$. The public transform data can be precomputed locally.

Algorithm 4 $(\llbracket \vec{s} \rrbracket) \leftarrow \mathrm{EncodeC}(t, \llbracket k \rrbracket)$

Require: For each $i \in [1,t]$, F_i is the unique polynomial of degree $t-1$ such that $F_i(i) = 1$ and $F_i(j) = 0$ for $j \in [1,t]\backslash\{i\}$.
Ensure: $\llbracket \vec{s} \rrbracket$ is a one-hot vector of length t, with index $\llbracket k \rrbracket$.
1: $\llbracket \vec{p} \rrbracket \leftarrow \mathrm{PreMulC}(\llbracket k+1 \rrbracket, \ldots, \llbracket k+1 \rrbracket)$ ▷ $t-1$ inputs
2: **for** $i = 1$ to t **do parallel** ▷ Use NTT if feasible.
3: Let the i-th Lagrange polynomial over $[1,t]$ be $F_i(x) = \sum_{j=0}^{t-1} F_{i,j} x^j$
4: $\llbracket s_{i-1} \rrbracket \leftarrow F_{i,0} + \sum_{j=1}^{t-1} F_{i,j} \llbracket p_{j-1} \rrbracket$
5: **end for**
6: **return** $\llbracket s_0, \ldots, s_{t-1} \rrbracket$

Note that the protocol EncodeC can also be used to generate a random one-hot vector of length T with index in $[T]$, with $\lceil \log t \rceil$ random bits and a call to $\mathrm{EncodeC}(T, \sum_i 2^i \llbracket r_i \rrbracket)$. Hence, the protocol to generate a random one-hot vector also has two versions, one in $\log \log t$ rounds and one in constant rounds.

[2]NTT requires padding to some $N \geq t$ with $N \mid p-1$. An NTT-friendly field yields $N = \Theta(t)$ and $O(N \log N) = O(t \log t)$.

D. Security Analysis

By composition, Demux inherits security from secure multiplication; RandOneHot additionally relies on the uniformity of RandBit; and EncodeC inherits security from PreMulC. The remaining case is Encode, whose masked opening requires a statistical argument. We prove its security using a hybrid argument. First, we consider an idealized version of protocol Encode, in which the opened deviation value modulo T is uniformly distributed over $[T]$ and independent of the target index k. We then show that the output distribution of this ideal protocol is statistically indistinguishable from that of the real protocol Encode, with respect to the security parameter κ.

Lemma 1. *Assume that all parties jointly hold a secret random one-hot vector $\llbracket \vec{s}' \rrbracket$ and its index $\llbracket r \rrbracket$, generated by calling protocol* RandOneHot. *Consider the following ideal functionality $\mathcal{F}$:*

1) *Each party P_i sends its shares $\llbracket k \rrbracket_i$ and $\llbracket r \rrbracket_i$ to $\mathcal{F}$.*
2) *$\mathcal{F}$ reconstructs k and r.*
3) *$\mathcal{F}$ samples $d_i \xleftarrow{\$} [2^\kappa]$ independently for each $i \in [1,m]$.*
4) *$\mathcal{F}$ sends*

$$\delta'_{\mathcal{F}} := ((T + k - r) \bmod T) + T \sum_{i \in [1,m]} d_i$$

to all parties.

5) *Each party P_i computes $\delta_{\mathcal{F}} := \delta'_{\mathcal{F}} \bmod T$, and shifts the random one-hot vector in the same manner as in protocol* Encode*, namely,*

$$\llbracket s_j \rrbracket_i \leftarrow \llbracket s'_{((j+T-\delta_{\mathcal{F}}) \bmod T)} \rrbracket_i.$$

Then $\mathcal{F}$ realizes the functionality of protocol Encode *with information-theoretic security.*

Proof: [Sketch] It is immediate that $\mathcal{F}$ implements the same (correct) functionality as protocol Encode.

To see information-theoretic security, observe that

$$\Pr[(T + k - r) \bmod T = j] = \frac{1}{T} \quad \text{for every } j \in [T], \tag{IV.7}$$

independently of k. Indeed, since the index r is uniformly random and never revealed to any party (except the trusted ideal functionality), the resulting value modulo T is uniform from the perspective of any adversarial coalition.

We remark that the ideal functionality could simply send the term

$$(T + k - r) \bmod T$$

to all parties while still achieving information-theoretic security. For direct comparison with the real protocol, $\mathcal{F}$ includes an additional random mask that mirrors Encode.

We now show that $\delta'_{\mathcal{F}}$ is statistically close to the value δ' opened in protocol Encode. Since δ' is the only value revealed during the protocol, statistical indistinguishability of these two distributions implies the statistical security of Encode.

Below we present two lemmas. In the following, Δ denotes statistical distance. The no-wrap condition in Eq. IV.6 ensures that the masked opening does not wrap modulo p.

Lemma 2. *Let X, Y, Z be random variables over $\mathbb{F}_p$, where Z is independent of both X and Y. Then*

$$\Delta(X+Z; Y+Z) \le \Delta(X; Y). \tag{IV.8}$$

Lemma 3. *Let X, Y, Z be random variables over $\mathbb{F}_p$, where Z is independent of both X and Y. Then*

$$\Delta(X+Z, Z; Y+Z, Z) = \Delta(X; Y). \tag{IV.9}$$

The proofs of the above lemmas are omitted due to space constraints. We are now ready to establish the security of the protocol, which follows as a direct corollary of the next theorem.

Theorem 1. *For any adversarial coalition of size at most f, the distributions of $\delta'_{\mathcal{F}}$ and δ' satisfy*

$$\Delta(\delta'_{\mathcal{F}}; \delta') \le \frac{1}{2^{\kappa}}. \tag{IV.10}$$

Proof: Consider an honest party P_h and its random mask $Y_h \leftarrow [2^{\kappa}]$ for the RandSum protocol. Since $r \leftarrow [T]$, the ideal $\delta'_{\mathcal{F}}$ contains the term $((T+k-r) \bmod T) + TY_h$ that is uniform in $[T2^{\kappa}]$, whereas the real δ' contains the term $T+k-r+TY_h$ that is uniform in $[k+1, k+T2^{\kappa}]$. Therefore,

$$\Delta(\delta'_{\mathcal{F}}; \delta') \le \frac{k+1}{T2^{\kappa}} \le 2^{-\kappa}, \tag{IV.11}$$

because $k \in [t] \subseteq [T]$. Adding the remaining independent masks cannot increase the distance by Lemma 2; the same bound holds jointly with the corrupted contributions by Lemma 3.

By Theorem 1, the security of protocol Encode can be formalized as follows.

Theorem 2. *Protocol* Encode *is statistically secure, in the sense that it is statistically indistinguishable from the ideal functionality described in Lemma 1, with respect to the security parameter κ.*

Proof: In the hybrid model, a coalition's view consists of its input shares, the RandOneHot and RandSum preprocessing shares, the opened δ', and its output shares. Given the coalition's prescribed input and output shares, a simulator samples the preprocessing shares from their ideal distributions, replaces δ' by $\delta'_{\mathcal{F}}$, and applies the same public shift. Secret-sharing privacy simulates all unopened values, and Theorem 1 bounds the only distributional change by $2^{-\kappa}$. Security then follows by composition with the underlying primitives.

V. Comparison and Digit Decomposition

In this section, we first construct comparison and subtraction protocols for small-domain values and then use them in a secure digit-decomposition protocol supporting any integer base $t > 1$. The protocol decomposes a secret-shared integer $[\![x]\!]$ into $[\![x_0]\!], [\![x_1]\!], \ldots, [\![x_{k-1}]\!]$ in base t; that is,

$$[\![x]\!] = \sum_{i=0}^{k-1} [\![x_i]\!] \cdot t^i, \tag{V.1}$$

where $0 \le x_i < t$. This is a generalization of bit decomposition [27] and is useful in various applications including data processing and machine learning [31, 32, 33].

As discussed in Section II, our protocol computes the base-t decomposition of $[\![x]\!]$ for public k, where $0 \le x < t^k \le 2^L$, by first generating a random mask $[\![a]\!]$ represented by its base-t digits $[\![a_i]\!]_{i\in[k]}$. We then compute

$$[\![y]\!] = [\![x]\!] + [\![a]\!] + t^k \cdot [\![r]\!]$$

and open it to all parties, with $[\![r]\!] \leftarrow \mathrm{RandSum}()$. After decomposing the public value $(y \bmod t^k)$ into base-t digits $(y_i)_{i\in[k]}$, we obtain the base-t digits of $[\![x]\!]$ by computing

$$([\![x_i]\!])_{i\in[k]} = \mathrm{SubC}(t, (y_i)_{i\in[k]}, ([\![a_i]\!])_{i\in[k]}),$$

where $\mathrm{SubC}(t, \cdot, \cdot)$ performs subtraction in base-t.

To this end, we first construct $\mathrm{RandDigit}(t)$ for sampling random base-t digits and then construct the constant-round subtraction protocol SubC, which uses an efficient comparison subroutine.

A. Efficient Comparison via One-Hot Encoding

In MPC, comparison operations are typically expensive. Existing comparison protocols consider generic comparisons between two secret-shared integers, which are often based on the comparison of their binary representations [17, 18, 19]. In our setting, however, the parties know the ranges of the secret values $[\![x]\!]$ and $[\![y]\!]$, i.e., $x, y \in [t]$ for a small constant t. This enables the following simple yet efficient comparison protocol.

Recall that one-hot encoding represents an integer $x \in [t]$ as a vector $\vec{u}$, where the x-th element is 1 and the rest are 0, i.e.,

$$\vec{u} = (\underbrace{0, 0, \ldots, 0}_{x}, 1, 0, \ldots, 0). \tag{V.2}$$

Clearly, for any integer $c \in [t]$, checking whether $x < c$ is equivalent to checking whether the sum of the first c entries equals 1.

$\mathrm{Cmp}_<(c, t, [\![x]\!])$ checks if a secret-shared integer x is less than a public constant c:

$$\mathrm{Cmp}_<(c, t, [\![x]\!]) = \begin{cases} 1 & \text{if } x < c, \\ 0 & \text{otherwise.} \end{cases} \tag{V.3}$$

The protocol converts x to a one-hot vector using Encode, then sums the first c entries to determine whether $x < c$. Replacing Encode with $\mathrm{EncodeC}$ gives the constant-round variant. See Algorithm 5 for the formal description.

Algorithm 5 $[\![s]\!] \leftarrow \mathrm{Cmp}_<(c, t, [\![x]\!])$

Require: $x \in [t]$ and $0 < c < t$
Ensure: $[\![s]\!]$ = 1 if $x < c$, 0 otherwise
1: $[\![\vec{u}]\!] \leftarrow \mathrm{Encode}(t, [\![x]\!])$
2: **return** $[\![s]\!] \leftarrow \sum_{i=0}^{c-1} [\![u_i]\!]$

Similarly, $\mathrm{Cmp}_{=}(c, t, [\![x]\!])$ checks if a secret-shared integer equals a public constant:

$$\mathrm{Cmp}_{=}(c, t, [\![x]\!]) = \begin{cases} 1 & \text{if } x = c, \\ 0 & \text{otherwise.} \end{cases} \tag{V.4}$$

Using the same one-hot encoding, the protocol accesses the c-th element of the encoded vector to determine equality. See Algorithm 6 for the protocol's formal description.

Algorithm 6 $[\![s]\!] \leftarrow \mathrm{Cmp}_{=}(c, t, [\![x]\!])$

Require: Secret $x \in [t]$
Ensure: $[\![s]\!]$ = 1 if $x = c$, 0 otherwise
1: $[\![\vec{u}]\!] \leftarrow \mathrm{Encode}(t, [\![x]\!])$
2: **return** $[\![s]\!] \leftarrow [\![u_c]\!]$

The complexities of $\mathrm{Cmp}_{=}$ and $\mathrm{Cmp}_{<}$ equal those of the selected encoding protocol, either Encode or EncodeC.

We note that in practice, when three-way comparison is needed, instead of separately calling $\mathrm{Cmp}_{=}$ and $\mathrm{Cmp}_{<}$, we can reuse the one-hot encoding of $[\![x]\!]$ to perform both comparisons simultaneously.

B. Random Digit Generation

The $\mathrm{RandDigit}(t)$ primitive uses rejection sampling to produce a sharing $[\![r]\!]$ of a uniform $r \in [t]$.

Let $l \coloneqq \lceil \log t \rceil$ and $T \coloneqq 2^l$. Parties first generate l random bits $([\![b_i]\!])_{i \in [l]}$ using the RandBit protocol and combine them by

$$[\![b]\!] \leftarrow \sum_{i=0}^{l-1} 2^i \cdot [\![b_i]\!].$$

Parties then check whether b is less than t by the $\mathrm{Cmp}_{<}(t, T, [\![b]\!])$ protocol. If $b < t$, the value is accepted; otherwise, the process is repeated until a valid number is generated. Since b is uniform in $[T]$, each trial is accepted with probability t/T, so the expected number of trials is $T/t \leq 2$. Since b is independent of all private inputs, the opened acceptance bit and hence the public number of trials depend only on fresh preprocessing randomness and public t, and reveal no private information. The per-trial comparison cost is determined by T. Once $[\![b]\!]$ is accepted, b is uniform in $[t]$.

This protocol is formally described in Algorithm 7. In practice, since the amount of randomness required is typically large, parties can generate multiple random digits in parallel and in advance to amortize the cost of communication rounds.

C. Subtraction in Constant Rounds

We now present an efficient MPC subtraction that computes the difference between two integers represented in base-t. Given $[\![a_i]\!]_{i \in [k]}$ and $[\![b_i]\!]_{i \in [k]}$ with each element in $[t]$, let $a = \sum_{i=0}^{k-1} a_i t^i$ and $b = \sum_{i=0}^{k-1} b_i t^i$. The protocol computes

$$[\![c_i]\!]_{i \in [k]} \leftarrow \mathrm{SubC}(t, [\![a_i]\!]_{i \in [k]}, [\![b_i]\!]_{i \in [k]}),$$

Algorithm 7 $[\![r]\!] \leftarrow \mathrm{RandDigit}(t)$

Ensure: A sharing $[\![r]\!]$ of a uniform $r \in [t]$
1: $l \leftarrow \lceil \log t \rceil$
2: $t' \leftarrow 2^l$
3: **repeat**
4: $\quad [\![b_0, \ldots, b_{l-1}]\!] \leftarrow \mathrm{RandBit}()$
5: $\quad [\![b]\!] \leftarrow \sum_{i=0}^{l-1} [\![b_i]\!] \cdot 2^i$
6: $\quad [\![s]\!] \leftarrow \mathrm{Cmp}_{<}(t, t', [\![b]\!])$
7: $\quad s \leftarrow \mathrm{Open}([\![s]\!])$
8: **until** $s = 1$
9: **return** $[\![r]\!] \leftarrow [\![b]\!]$

where $c_i \in [t]$ and

$$c \equiv a - b \equiv \sum_{i=0}^{k-1} c_i t^i \pmod{t^k}. \tag{V.5}$$

Note that the subtraction is performed modulo t^k, which means in the case of $a < b$, $c = a - b + t^k$.

As a starting point, consider the following naive construction. The parties first compute $[\![s_0]\!]$ by

$$[\![s_0]\!] \leftarrow \mathrm{Cmp}_{<}(t, 2 \cdot t, t + [\![a_0]\!] - [\![b_0]\!]),$$

which is 1 if $a_0 < b_0$ and 0 otherwise. s_0 indicates whether a borrow is needed for the least significant digit. Similarly, for each digit position i, the parties compute $[\![s_i]\!]$ by

$$[\![s_i]\!] \leftarrow \mathrm{Cmp}_{<}(t, 2 \cdot t, t + [\![a_i]\!] - [\![b_i]\!] - [\![s_{i-1}]\!]),$$

which is 1 if $a_i < b_i + s_{i-1}$ and 0 otherwise. After obtaining all borrow bits $[\![s_i]\!]$, the parties can compute each result digit $[\![c_i]\!]$ as

$$[\![c_i]\!] \leftarrow [\![a_i]\!] - [\![b_i]\!] + t \cdot [\![s_i]\!].$$

The problem now is how to compute all borrow bits $[\![s_i]\!]$ efficiently. While the above approach works, it incurs non-constant $O(k)$ communication rounds due to the sequential dependency of borrow bits.

The key observation is that base-t borrow propagation has the same recurrence structure as carry propagation in bitwise addition, which is well studied in MPC. Consider the bitwise-addition case, where two k-bit integers x and y are represented by x_i and y_i respectively. In the propagation of bitwise addition, a carry bit t_i is decided by the following rule:

$$t_i = \begin{cases} 1 & \text{if } x_i + y_i = 2, \\ 0 & \text{if } x_i + y_i = 0, \\ t_{i-1} & \text{if } x_i + y_i = 1. \end{cases} \tag{V.6}$$

In the literature, protocols that compute the carry bits t_i in Eq. V.6 are already well studied [9, 27]. To leverage existing results, we denote the constant-round carry propagation protocol as

$$[\![t_i]\!]_{i \in [k]} \leftarrow \mathrm{CarryC}([\![x_i]\!]_{i \in [k]}, [\![y_i]\!]_{i \in [k]}).$$

In our case of subtraction between a and b, the borrow bit

s_i follows a similar rule:

$$s_i = \begin{cases} 1 & \text{if } a_i < b_i, \\ 0 & \text{if } a_i > b_i, \\ s_{i-1} & \text{if } a_i = b_i. \end{cases} \tag{V.7}$$

Comparing Eqs. V.6 and V.7 shows that base-t borrow propagation reduces to carry propagation. Our constant-round subtraction protocol first computes the comparison bits at every digit position, applies CarryC to obtain all borrow bits, and then computes the result digits $[\![c_i]\!]$.

In Algorithm 8, we present the fully developed constant-round base-t subtraction protocol SubC.

Algorithm 8 $[\![\vec{c}]\!] \leftarrow \mathrm{SubC}(t, [\![\vec{a}]\!], [\![\vec{b}]\!])$

Require: $[\![\vec{a}]\!] = [\![a_0, a_1, \ldots, a_{k-1}]\!]$, $[\![\vec{b}]\!] = [\![b_0, b_1, \ldots, b_{k-1}]\!]$ with $0 \le a_i, b_i < t$
Ensure: $\sum_i c_i \cdot t^i = \sum_i a_i \cdot t^i - \sum_i b_i \cdot t^i \pmod{t^k}$, with $0 \le c_i < t$
for $i = 0$ to $k-1$ **do**
  $[\![d_i]\!] \leftarrow t + [\![a_i]\!] - [\![b_i]\!]$
  $[\![x_i]\!] \leftarrow \mathrm{Cmp}_<(t, 2t, [\![d_i]\!])$
  $[\![y_i]\!] \leftarrow [\![x_i]\!] + \mathrm{Cmp}_=(t, 2t, [\![d_i]\!])$
end for
$[\![\vec{s}]\!] \leftarrow \mathrm{CarryC}([\![\vec{x}]\!], [\![\vec{y}]\!])$
$[\![c_0]\!] \leftarrow t \cdot [\![s_0]\!] + [\![a_0]\!] - [\![b_0]\!]$
for $i = 1$ to $k-1$ **do**
  $[\![c_i]\!] \leftarrow t \cdot [\![s_i]\!] - [\![s_{i-1}]\!] + [\![a_i]\!] - [\![b_i]\!]$
end for
return $[\![\vec{c}]\!] = ([\![c_0]\!], \ldots, [\![c_{k-1}]\!])$

D. Generic-Radix Digit Decomposition

Building upon the previously established primitives for comparison, random digit generation, and constant-round subtraction, we now present a constant-round decomposition protocol $\mathrm{DigitDecomp}(t, k, [\![x]\!])$. Given public t and k and a secret-shared integer $[\![x]\!]$ satisfying $0 \le x < t^k \le 2^L$, the protocol produces k base-t digits satisfying Eq. V.1. As mentioned, parties first generate a random mask $[\![a]\!]$, represented by its base-t digits $[\![a_i]\!]_{i\in[k]}$, using the $\mathrm{RandDigit}(t)$ protocol. The parties sample an additional random $[\![r]\!] \leftarrow \mathrm{RandSum}()$, and open the sum

$$[\![d]\!] \leftarrow [\![x]\!] + [\![a]\!] + t^k \cdot [\![r]\!].$$

Note that since $[\![r]\!]$ is shifted by t^k, the lowest k digits of d in base-t are not affected by $[\![r]\!]$. Applying SubC to the lowest k digits of d and $([\![a_i]\!])_{i\in[k]}$ therefore recovers the base-t digits of $[\![x]\!]$.

The protocol is formally presented in Algorithm 9.

For security, let $B = t^k$ and fix an honest party's RandSum contribution $r_h \leftarrow [2^\kappa]$. Since $a \leftarrow [B]$, $Z = a + Br_h$ is uniform in $[B2^\kappa]$; hence, for $x \in [B]$,

$$\Delta(x + Z; Z) = \frac{x}{B2^\kappa} < 2^{-\kappa}. \tag{V.8}$$

Adding the remaining masks cannot increase the distance in Eq. V.8. The field-size bound in Eq. III.1 implies $p > 2(B-1) + Bm(2^\kappa - 1)$, so the result also holds in $\mathbb{F}_p$. A simulator samples the opened d from the x-independent distribution and invokes the simulators of the underlying primitives for the remaining view.

Algorithm 9 $[\![\vec{c}]\!] \leftarrow \mathrm{DigitDecomp}(t, k, [\![x]\!])$

Require: $0 \le x < t^k \le 2^L$
Ensure: Base-t digits $([\![c_i]\!])_{i\in[k]}$ satisfying $x = \sum_i c_i t^i$
$[\![a_i]\!] \leftarrow \mathrm{RandDigit}(t)$ for $i \in [k]$
$[\![r]\!] \leftarrow \mathrm{RandSum}()$
$[\![a]\!] \leftarrow t^k \cdot [\![r]\!] + \sum_{i=0}^{k-1} [\![a_i]\!] \cdot t^i$
$d \leftarrow \mathrm{Open}([\![x]\!] + [\![a]\!])$
Let $(d_i)_{i=0}^{k-1}$ be the base-t digit representation of $d \bmod t^k$
$[\![\vec{c}]\!] \leftarrow \mathrm{SubC}(t, (d_i)_{i=0}^{k-1}, [\![a_i]\!]_{i=0}^{k-1})$
return $[\![\vec{c}]\!]$

For efficiency, the $\mathrm{DigitDecomp}(t, k, [\![x]\!])$ protocol incurs $O(ktm)$ online communication. With Encode, online computation is $O(ktm)$, in $O(\log t)$ rounds, with $4k$ secure multiplications; with EncodeC, computation is $O(kt \log t\, m)$, in $O(1)$ rounds, with $(2t+3)k$ online secure multiplications. The offline phase costs $O(ktm)$ communication in $O(1)$ rounds, with (in expectation) $< 6tk$ (Encode) or $(7t-7)k$ (EncodeC) multiplications. See Table I for a summary of the complexities.

VI. Radix Sort

In this section, we consider a protocol among m parties with n inputs. We first focus on vectors of length L, where each entry is a small integer in the range $[t]$, so that one-hot encoding is applicable. We then extend the radix-sort protocol to support general integers. The cases of vectors over general integers and key-value sorting follow straightforwardly from these constructions.

A. Vector Radix Sort

Assume the inputs are n secret-shared vectors $[\![\vec{v}_0]\!], [\![\vec{v}_1]\!], \ldots, [\![\vec{v}_{n-1}]\!]$, with each vector of length L and each entry $v_{i,j}$ in $[t]$. Define $V_i \in \mathbb{Z}$ as the integer represented by $\vec{v}_i$, i.e.,

$$V_i \coloneqq \sum_{j=0}^{L-1} t^j v_{i,j}. \tag{VI.1}$$

Note that V_i may exceed the field modulus p.

The radix sort protocol reorders the input vectors by a permutation $\pi : [n] \to [n]$, such that

$$V_{\pi(i)} \le V_{\pi(i+1)} \quad \text{for all } i \in [0, n-2]. \tag{VI.2}$$

The output of the protocol is the reordered vectors, i.e.,

$$[\![\vec{v}_{\pi(0)}]\!], [\![\vec{v}_{\pi(1)}]\!], \ldots, [\![\vec{v}_{\pi(n-1)}]\!].$$

We denote this protocol by VecRadixSort.

To implement radix sort for vectors, we follow the approach of Hamada et al. [23], which relies on two building blocks: the rank-computation protocol RankComp and the reveal-then-sort protocol RevealSort. RankComp takes n secret-shared keys, implicitly performs a stable sort, and outputs

the corresponding rank vector $\vec{r}$. $\mathrm{RevealSort}$ takes the secret-shared rank vector $\vec{r}$ and values $[\![\vec{v}_i]\!]_{i\in[n]}$, and outputs them reordered according to $\vec{r}$. Together, these two sub-protocols implement the core step of radix sort: in each round, the vectors are ranked by the current digit and then reordered for the next round.

We note that the original rank-computation protocol of Hamada et al. [23] supports only binary inputs, restricting radix sort to base $t = 2$ (see Algorithm 1 in [23]). In this work, we generalize their protocol to support an arbitrary base t. For a suitable choice of t, this reduces the number of calls to $\mathrm{RevealSort}$, typically the main bottleneck, thereby improving communication, computation, and round complexity.

RankComp Protocol The $\mathrm{RankComp}$ protocol takes as input n secret-shared keys $[\![k_0]\!], \ldots, [\![k_{n-1}]\!]$ and outputs a secret-shared rank vector $[\![\vec{r}]\!] = ([\![r_0]\!], \ldots, [\![r_{n-1}]\!])$, which forms a permutation over $[n]$. The protocol is stable: if $k_i = k_j$ and $i < j$, then $r_i < r_j$.

The keys are first transformed into one-hot vectors of length t using Encode or $\mathrm{EncodeC}$, yielding a matrix $[\![A]\!] = ([\![a_{i,j}]\!])_{i\in[n],\, j\in[t]}$, where each row is a one-hot vector.

Parties then compute the prefix sums

$$[\![c_{i,j}]\!] \leftarrow \sum_{\substack{x\in[n]\\ y<j}} [\![a_{x,y}]\!] + \sum_{x\le i} [\![a_{x,j}]\!],$$

for all $i \in [n]$ and $j \in [t]$. For $k_i = j$, the value $[\![c_{i,j}]\!] - 1$ equals the stable rank of k_i. The rank is therefore obtained by selecting the corresponding entry via $[\![r_i]\!] \leftarrow [\![\vec{a}_i]\!] \cdot [\![\vec{c}_i]\!] - 1$.

All computations are performed over secret-shared data, and no intermediate values are opened. Security follows directly from that of the underlying secret-sharing scheme and its primitives.

RevealSort Protocol This protocol is due to Hamada et al. [23]; for completeness, we briefly review it.

The protocol takes as input a list of secret-shared key-value pairs, where each value is a vector of fixed length and, in our setting, the corresponding key is its rank produced by $\mathrm{RankComp}$. It outputs the pairs sorted according to the keys.

The parties first apply a random shuffle to all key-value pairs and then open the shuffled keys and sort accordingly. Since the permutation is randomly chosen and kept secret, the opened keys cannot be linked to the original inputs. Hence, revealing the shuffled keys does not disclose the rank of any specific input vector.

VecRadixSort Protocol Using RankComp and RevealSort as building blocks, we now construct the vector radix sort protocol.

We first extend $\mathrm{RankComp}$ to vectors, obtaining the protocol $\mathrm{VecRankComp}$. The protocol takes public n and t and secret-shared vectors $[\![\vec{v}_i]\!]_{i\in[n]}$, where each entry $v_{i,j}$ lies in $[t]$. It outputs the stable rank of each vector.

Instead of permuting entire vectors in each round, the protocol reorders only the next digit column, proceeding from the LSB to the MSB. It maintains a value-index pair $(\vec{v}', \vec{s})$, where $\vec{v}'$ contains the current digit and $\vec{s}$ records the corresponding original indices. In the j-th round, the protocol computes the rank vector $\vec{r}$ of $\vec{v}'$ using $\mathrm{RankComp}$. It then invokes $\mathrm{RevealSort}$ twice: first to reorder $\vec{r}$ according to the index vector $\vec{s}$, and then to sort the pairs $(v_{i,j+1}, s_i)_{i\in[n]}$ according to the reordered ranks, thereby preparing the next round. After processing all digits, the final ranks are reordered according to the original indices and returned. We denote this procedure by

$$([\![r_i]\!])_{i\in[n]} \leftarrow \mathrm{VecRankComp}\big(t, ([\![\vec{v}_i]\!])_{i\in[n]}\big),$$

where r_i is the stable rank of $\vec{v}_i$.

The vector radix sort protocol $\mathrm{VecRadixSort}$ is obtained directly from $\mathrm{VecRankComp}$. Given the input vectors, it computes their ranks via $\mathrm{VecRankComp}$ and subsequently reorders them using $\mathrm{RevealSort}$:

$$([\![\vec{v}'_i]\!])_{i\in[n]} \leftarrow \mathrm{VecRadixSort}\big(t, ([\![\vec{v}_i]\!])_{i\in[n]}\big).$$

The output is a stably sorted permutation of the input vectors.

B. Integer Radix Sort

To sort integers, one may directly apply $\mathrm{RankComp}$ and $\mathrm{RevealSort}$. However, for large integers, this would require extremely long one-hot vectors.

A more efficient approach is to first decompose each integer into base-t digits using $\mathrm{DigitDecomp}$, and then apply $\mathrm{VecRadixSort}$. This limits the one-hot encoding length to t.

Given public t and k satisfying $0 \le v_i < t^k \le 2^L$ for every input v_i, Algorithm 10 presents the complete integer sorting protocol. The choice of t provides a trade-off: a larger t reduces the number of digits and thus the number of rounds and shuffle invocations, while increasing the per-round communication cost due to longer one-hot vectors. We further discuss the protocol complexity in Section VII-A.

Algorithm 10 $[\![v'_i]\!]_{i\in[n]} \leftarrow \mathrm{IntRadixSort}(t, k, [\![v_i]\!]_{i\in[n]})$

Require: Public t and k, and secrets $0 \le v_i < t^k \le 2^L$ for all $i \in [n]$.
Ensure: $([\![v'_i]\!])_{i\in[n]}$ is a sorted permutation of the inputs.

```
for i = 0 to n − 1 do parallel
    [[d_i]] ← DigitDecomp(t, k, [[v_i]])        ▷ Decompose v_i into k digits in base t
end for
([[d'_i]])_{i∈[n]} ← VecRadixSort(t, ([[d_i]])_{i∈[n]})
for i = 0 to n − 1 do
    [[v'_i]] ← Σ_{j=0}^{k−1} t^j · [[d'_{i,j}]]
end for
return [[v'_i]]_{i∈[n]}
```

VII. Evaluation

We evaluate $\mathrm{UFO_s}$ through concrete and asymptotic cost analysis, followed by matched benchmarks of ROHV generation, digit decomposition, and integer sorting.

A. Theoretical Analysis

In this subsection, we present the complexity of the core protocols of $\mathrm{UFO_s}$ and of the derived secure integer sorting

TABLE I: Secure-operation complexity of the proposed protocols

Protocol	Mult		RandSum	RandBit
	offline	online		
Encode	$t-\log t-1$	0	1	$\log t$
EncodeC	$2t-3$	$t-1$	0	0
Cmp (Encode)	$t-\log t-1$	0	1	$\log t$
Cmp (EncodeC)	$2t-3$	$t-1$	0	0
DigitDecomp (Encode)	$(3t-2\log t-3)\frac{L}{\log t}$	$4\frac{L}{\log t}$	$3\frac{L}{\log t}+1$	$3L+2\frac{L}{\log t}$
DigitDecomp (EncodeC)	$(7t-7)\frac{L}{\log t}$	$(2t+3)\frac{L}{\log t}$	$\frac{L}{\log t}+1$	$L+\frac{L}{\log t}$

· "Cmp" denotes both $\mathrm{Cmp}_<$ and $\mathrm{Cmp}_=$.
· For clarity, the table reports the power-of-two case and suppresses the ceiling in $L/\log t$.
· For generic t, the offline phase of DigitDecomp (Encode) uses fewer than $6t\lceil L/\log t\rceil$ multiplications in expectation.

TABLE II: Asymptotic complexity of the proposed integer-sorting protocols

Protocol	Computation	Communication	Rounds
IntRadixSort (Encode)	$O\left(\frac{L(\mathcal{S}+tmn)}{\log t}\right)$	$O\left(\frac{L(\mathcal{S}+tmn)}{\log t}\right)$	$O\left(\frac{L}{\log t}\mathcal{S}+\log\log t+c\right)$
IntRadixSort (EncodeC)	$O\left(\frac{L}{\log t}\mathcal{S}+Ltmn\right)$	$O\left(\frac{L(\mathcal{S}+tmn)}{\log t}\right)$	$O\left(\frac{L}{\log t}\mathcal{S}+c\right)$

· Parentheses identify the encoding instantiation.
· If $\mathbb{F}_p$ is not NTT-friendly, the computation of IntRadixSort (EncodeC) increases to $O\left(\frac{L}{\log t}\mathcal{S}+\frac{t^2L}{\log t}mn\right)$.

protocols. Recall that we denote the bit length of each input integer by L, the number of parties by m, and the number of elements to be sorted by n. We use $[t]$ to denote both the domain of one-hot encoding/secure comparison and the base for digit decomposition.

ROHV generation. All three constructions generate an ROHV of length $t=2^k$ from k secret-shared random bits [16, 50]. Table III compares their multiplication costs after these random bits are available. Let $M_{\mathrm{L}}(k)$ denote the multiplication count of the recursive ROHV construction of Launchbury et al. [50], where

$$\begin{aligned} &M_{\mathrm{L}}(1)=0,\\ &M_{\mathrm{L}}(k)=M_{\mathrm{L}}(\lfloor k/2\rfloor)+M_{\mathrm{L}}(\lceil k/2\rceil)+2^k\\ &\qquad=2^k+O(2^{\lfloor k/2\rfloor}) \text{ for } k>1. \end{aligned} \tag{VII.1}$$

As shown in Eq. IV.5, our ROHV generation uses fewer multiplications than [50] while preserving its round complexity, and fewer rounds than [16] with the same number of multiplications.[3]

TABLE III: Comparison of protocols for generating an ROHV of length $t=2^k$

Protocol	Mult	Rounds
Launchbury et al. [50]	$2^k+O(2^{\lfloor k/2\rfloor})$	$\lceil\log k\rceil R_M$
Morita et al. [16]	2^k-k-1	$(k-1)R_M$
Ours	2^k-k-1	$\lceil\log k\rceil R_M$

· R_M denotes the round complexity of a secure multiplication.

[3]The 2^k-k-1 gates and $\log k$ depth are optimal in the arithmetic-circuit model with fan-in-two multiplication gates and free linear operations, given k random input bits [25].

Core protocols of UFO$_{\mathrm{s}}$. We report the concrete counts of the main secure operations for the two encoding protocols (Encode and EncodeC) as well as the comparison ($\mathrm{Cmp}_<$, $\mathrm{Cmp}_=$) and digit-decomposition (DigitDecomp) protocols derived from them. For the L-based expressions below, we use the full-range setting with the minimum k such that $t^k\geq 2^L$. Table I details these counts in terms of the core primitives. The dominant online cost of DigitDecomp with Encode is only $4L/\log t$ secure multiplications. Reistad et al.'s bit decomposition [9] requires $9.5L$ multiplications in total; at $t=8$, our online phase is approximately 2.6 times faster (see Section VII-B). Ning et al. [20] require more than $300L$ secure multiplications for arbitrary-base decomposition, making the online-cost reduction still larger.

Sorting. For integer sorting, Table II reports the asymptotic complexity of our IntRadixSort protocol in terms of computation, communication, and rounds. Its complexity depends on the Shuffle protocol, denoted by $\mathcal{S}$.

Parameter t introduces a trade-off between the number of digits and the cost of processing each digit. To tune it, suppose we fix the input size n and network settings. We model the sorting time as

$$F(t)=F_{\mathrm{dec}}(t)+k(t)\big(F_{\mathrm{digit}}(t)+F_{\mathrm{shuffle}}\big), \tag{VII.2}$$

where $k(t)$ is the number of base-t digits, and the three F terms denote digit decomposition, processing one digit, and one Shuffle. Ignoring ceilings and lower-order variations in the round count, we have

$$\begin{aligned} &k(t)\approx\frac{L}{\log t},\quad F_{\mathrm{dec}}(t)\approx a_0\frac{Lt}{\log t}+c_0,\\ &F_{\mathrm{digit}}(t)+F_{\mathrm{shuffle}}\approx a_1t+c_1. \end{aligned} \tag{VII.3}$$

TABLE IV: Performance of ROHV generation for a batch of 65,536 vectors

Length	Protocol	Run time (s)			Comm (MB)	Mult	Rounds
		RTT 20 ms	RTT 200 ms	RTT 2,000 ms			
4	Ours	1.259	1.620	5.221	15.10	1	4
	Launchbury et al. [50]	2.000	2.361	5.964	35.23	4	4
	MAESTRO [16]	1.264	1.624	5.228	15.10	1	4
8	Ours	2.398	2.939	8.340	39.43	4	6
	Launchbury et al. [50]	4.621	5.160	10.561	86.40	12	6
	MAESTRO [16]	2.430	2.971	8.372	39.43	4	6
16	Ours	4.798	5.335	10.739	90.60	11	6
	Launchbury et al. [50]	8.476	9.016	14.420	171.13	24	6
	MAESTRO [16]	4.848	5.572	12.776	90.60	11	8
32	Ours	9.607	10.326	17.536	195.45	26	8
	Launchbury et al. [50]	15.609	16.319	23.532	322.96	48	8
	MAESTRO [16]	9.628	10.534	19.544	195.45	26	10
64	Ours	19.522	20.229	27.431	407.69	57	8
	Launchbury et al. [50]	27.566	28.296	35.492	588.88	88	8
	MAESTRO [16]	19.168	20.249	31.047	407.69	57	12
128	Ours	37.530	38.245	45.437	834.67	120	8
	Launchbury et al. [50]	49.620	50.334	57.525	1082.97	164	8
	MAESTRO [16]	38.052	39.290	51.899	834.67	120	14
256	Ours	75.530	76.206	83.438	1691.14	247	8
	Launchbury et al. [50]	91.109	91.776	99.006	2026.69	304	8
	MAESTRO [16]	76.077	77.464	91.897	1691.14	247	16

· Simulated bandwidth is 50 MB/s.
· "Comm" denotes the communication volume per party.
· "Mult" denotes the number of multiplication gates per generated ROHV.

TABLE V: Party-count scalability of ROHV generation for 65,536 length-64 vectors

Protocol	$m = 3,\ f = 1$		$m = 4,\ f = 1$		$m = 5,\ f = 2$	
	Run time (s)	Comm (MB)	Run time (s)	Comm (MB)	Run time (s)	Comm (MB)
Ours	12.465	255.85	12.123	271.32	18.782	407.69
Launchbury et al. [50]	17.916	385.88	17.782	409.47	27.744	615.72
MAESTRO [16]	12.814	255.85	12.349	271.32	18.894	407.69

· Run time includes simulated network overhead at 50 MB/s and 20 ms RTT.
· "Comm" denotes the communication volume per party.

The constants c_0 and c_1 mainly capture round latency, while c_1 also absorbs F_{shuffle}. All constants can be estimated from measurements of local computation, communication, and round latency. Substituting Eq. VII.3 into Eq. VII.2 gives

$$F(t) \approx \frac{L}{\log t}(\alpha t + \beta) + \gamma = \frac{L}{\ln t}(\alpha' t + \beta') + \gamma, \quad \text{(VII.4)}$$

where $\alpha = a_0 + a_1$, $\beta = c_1$, $\gamma = c_0$, $\alpha' = (\ln 2)\alpha$, and $\beta' = (\ln 2)\beta$. The minimizer of Eq. VII.4 is characterized by $t(\ln t - 1) = \beta'/\alpha'$, which gives a direct initial choice. To obtain a practically optimal t, we suggest rounding this initial choice and testing the performance of neighboring choices, until the performance degenerates at some point.

B. Experimental Evaluation

We evaluate all protocols on a server equipped with Ubuntu 24.04.4 LTS and two Intel Xeon Gold 6326 CPUs, providing 32 physical cores and 64 hardware threads, and 188 GiB of memory. Unless otherwise stated, we use $m = 5$ parties with corruption threshold $f = 2$. Each party is executed as a separate single-threaded process, and communication uses the Message Passing Interface (MPI). We model WAN communication between each pair of parties with 50 MB/s bandwidth and a default RTT of 20 ms.

We implement the protocols in C++ over a semi-honest, honest-majority Shamir secret sharing backend. All protocols use a 127-bit prime field with 16 bytes per field element and the standard statistical security parameter $\kappa = 40$. Each

TABLE VI: Performance of 2^d-decomposition on 32-bit integers

Protocol	Base	Run time (ms)		Comm (KB)		Rounds	
		online	offline	online	offline	online	offline
[9]	2	0.440	1.502	6.586	26.989	10	3
Our DigitDecomp	2	0.457	2.096	7.200	35.590	11	4
	8	0.172	1.455	2.362	25.882	11	5
	32	0.123	2.528	1.440	50.995	11	6
	128	0.102	6.223	0.979	134.797	11	6

· "Comm" denotes the communication volume per party.

TABLE VII: Performance of digit decomposition on 32-bit integers

Base	Protocol	Run time (ms)		Comm (KB)		Rounds	
		online	offline	online	offline	online	offline
5	[20]	133.341	119.987	604.090	1731.046	105	17
	Our DigitDecomp	0.420	6.049	3.053	111.072	11	8
20	[20]	155.894	129.788	698.791	1860.755	127	17
	Our DigitDecomp	0.343	11.015	1.670	223.955	11	10
80	[20]	176.394	122.573	780.576	1821.747	149	17
	Our DigitDecomp	0.311	26.031	1.210	573.459	11	10
320	[20]	185.580	127.170	822.355	1965.619	149	17
	Our DigitDecomp	0.330	61.295	0.749	1386.528	11	12

· "Comm" denotes the communication volume per party.

configuration is repeated 5 times and the averages are reported. We report online and offline phases separately. The offline phase consists of input-independent preprocessing that may run before private inputs arrive, and the online phase consists of input-dependent computation. The source code of the experiments is publicly available[4].

ROHV generation. We benchmark our protocol, the recursive construction of Launchbury et al. [50], and MAESTRO [16], ported to the same Shamir backend for a controlled comparison. Each run generates a batch of 65,536 ROHVs.

Averaging over the tested lengths in Table IV, we find that our protocol reduces generation time by 34.71%, 32.17%, and 20.68% relative to Launchbury et al. at RTTs of 20, 200, and 2,000 ms, respectively. Relative to MAESTRO, the corresponding reductions are 0.46%, 1.69%, and 8.73%. Our protocol matches the communication of MAESTRO and uses less communication than Launchbury et al. for every tested length; its round count also matches Launchbury et al. and is lower than MAESTRO for lengths of at least 16.

To assess party-count scalability, we repeat the length-64 benchmark for $(m, f) \in \{(3, 1), (4, 1), (5, 2)\}$ under the default network setting.

Across all three party counts in Table V, our protocol remains the fastest. It reduces run time by 30.43–32.30% relative to Launchbury et al. and by 0.59–2.73% relative to MAESTRO. Its per-party communication matches MAESTRO and is 33.70–33.79% lower than Launchbury et al. Thus, the relative ordering in Table IV is not specific to the five-party setting.

2^d-Decomposition. We compare our DigitDecomp protocol with the representative bit-decomposition protocol by Reistad et al. [9], which can be extended to a 2^d-decomposition (i.e., representing a number in base 2^d) by combining consecutive bits at low cost. The experimental results are presented in Table VI. DigitDecomp sharply reduces online cost for bases larger than 2; at larger bases, this comes with increased offline cost. At base 32, its online runtime is $0.123\,\mathrm{ms}$ and communication is $1.440\,\mathrm{KB}$, improving runtime and communication by factors of approximately 3.6 and 4.6, respectively, over the baseline. These gains directly benefit latency-sensitive online execution.

Digit Decomposition. Table VII demonstrates that, compared to the digit decomposition scheme by Ning et al. [20], our protocol reduces online costs: runtime decreases from 133.341–185.580 ms to 0.311–0.420 ms, a 318–567× speedup, while communication decreases by 99.5–99.9% (0.749–3.053 KB versus 604.090–822.355 KB). The protocol uses 11 online and 8–12 offline rounds, compared with 105–149 online and 17 offline rounds for the baseline. Our total runtime is lower for every tested t, by factors of 5.1–39.2. The baseline remains expensive across all tested bases and varies less with t than the offline cost of our protocol.

Choice of radix. Figure 1 reports the measured running time plus the simulated network overhead. For all tested bandwidths, $t = 16$ is optimal. As the RTT increases from

[4]https://anonymous.4open.science/r/SelectionVectorMPC-A1E8

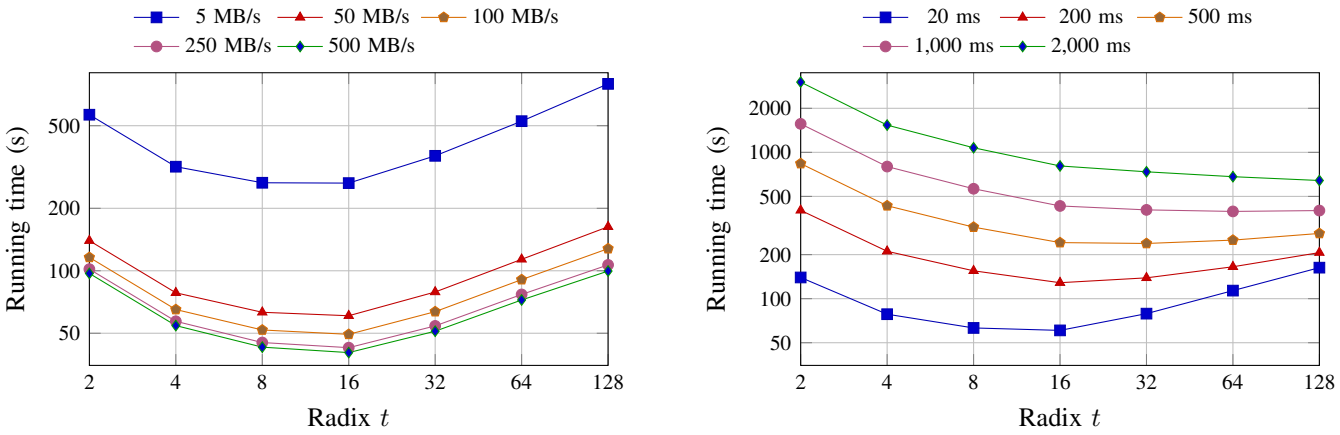

Fig. 1: Effect of radix t on the total running time for sorting 16,384 32-bit integers. Left: bandwidth sweep at 20 ms RTT. Right: RTT sweep at 50 MB/s. Both vertical axes use a logarithmic scale.

TABLE VIII: Continuous prediction and measured optimum for sorting 16,384 32-bit integers under different network settings.

Setting	Bandwidth (MB/s)				
	5	50	100	250	500
$\tilde{t}$	10.98	13.46	14.47	15.49	15.95
$t^\star$	16	16	16	16	16
Setting	**RTT (ms)**				
	20	200	500	1,000	2,000
$\tilde{t}$	13.46	31.19	49.51	67.84	86.89
$t^\star$	16	16	32	64	128

· $\tilde{t}$ is the predicted optimum.
· $t^\star$ is the measured optimum.

200 to 2,000 ms, the optimum shifts from $t = 16$ to 32, 64, and 128, since reducing the number of digits increasingly dominates the higher per-digit cost.

To verify our theory, we also estimate the predicted optimum $\tilde{t}$ using Eq. VII.4. To estimate the slope β'/α', we sample the component costs at $t \in \{2, 8, 32\}$, fit their slopes and intercepts by least squares, and substitute them into Eq. VII.4. Minimizing the fitted expression gives $\tilde{t}$ in Table VIII; $t^\star$ is the measured optimum over all seven plotted radices.

As can be observed from Table VIII, the continuous predictions capture the shift toward larger radices as latency increases. Except at an RTT of 200 ms, the predicted $\tilde{t}$ can always be rounded up or down to the measured optimum $t^\star$. The discrepancy at 200 ms RTT is likely due to the omission of the ceiling in the analytical approximation of Eq. VII.4.

Integer Sorting. We implemented three representative secure sorting methods that were previously analyzed in Section II: Odd-even merge sort [37] (oblivious, $O(n \log^2 n)$), Quicksort [43] (shuffle-then-sort, $O(n \log n)$), and Hamada et al.'s Radix sort [23], a representative shuffle-and-reveal radix-sort baseline. We use the shuffle protocol of [46] to implement the shuffle operation required by [23, 43] and our

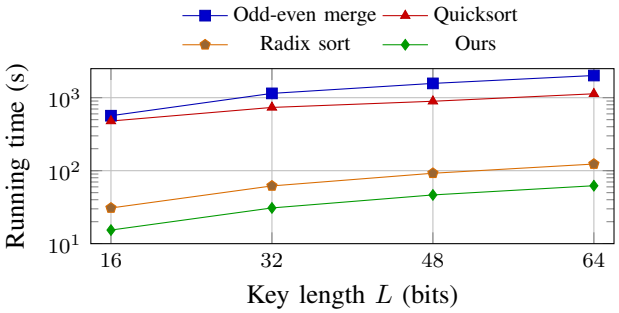

Fig. 2: Total simulated running time for sorting 8,192 integers with 16–64-bit keys at 50 MB/s and 20 ms RTT.

radix sort. The experimental results are shown in Table IX. For $65,536$ elements, the online runtime is $39.18\,\mathrm{s}$, compared with $134.64\,\mathrm{s}$ for the radix-sort baseline, a $3.4\times$ improvement. Online communication is $727.08\,\mathrm{MB}$, or 24.6% of that baseline's. Our protocol uses 371 online rounds and 6 offline rounds. Odd-even merge sort and Quicksort results are unavailable at the largest size.

Key-length scalability. Figure 2 reports total (online plus offline) running time as the key length L varies. The tuner selects $t = 16$ for all tested lengths. Increasing L raises the number of radix passes and the cost of bitwise comparison. Both radix-sort protocols therefore scale nearly linearly; across 16–64 bits, ours takes 15.36–62.22 s, approximately half the time of Hamada et al.'s Radix sort.

VIII. Conclusion

We presented UFO$_\mathrm{s}$, a toolkit that exploits known small operand domains in MPC. For power-of-two length t, our ROHV generator attains, in the fan-in-two arithmetic-circuit model, the lower bounds of $t - \log t - 1$ multiplication gates and $\lceil \log \log t \rceil$ multiplication depth. The resulting one-hot comparison enables direct arbitrary-base digit decomposition without expanding every digit into bits. IntRadixSort applies these primitives to reduce radix passes and shuffles by a factor of $\log t$. In our experiments, DigitDecomp achieves a

TABLE IX: Performance of sorting on 32-bit integers

Items	Protocol	Run time (s)		Comm (MB)		Rounds	
		online	offline	online	offline	online	offline
4096	Odd-even merge sort [37]	231.59	283.96	1430.75	5188.72	1102	3
	Quicksort [43]	206.30	118.09	1380.97	2048.57	858	3
	Radix sort [23]	21.85	17.53	184.73	380.76	1418	3
	Our IntRadixSort	9.35	11.86	45.44	241.30	371	6
16384	Odd-even merge sort [37]	1187.51	1471.75	7787.40	28223.00	1480	3
	Quicksort [43]	1038.62	554.05	7701.84	10113.70	1148	3
	Radix sort [23]	43.60	65.31	738.92	1523.06	1418	3
	Our IntRadixSort	15.22	46.83	181.77	965.21	371	6
65536	Odd-even merge sort [37]	–	–	–	–	–	–
	Quicksort [43]	–	–	–	–	–	–
	Radix sort [23]	134.64	266.03	2955.72	6092.23	1418	3
	Our IntRadixSort	39.18	188.44	727.08	3860.86	371	6

· "Comm" denotes the communication volume per party.
· "–" denotes that the protocol did not complete within one hour.
· For our protocol, $t = 16$ for all three item counts.

318–567× online speedup over Ning et al., and IntRadixSort achieves a 3.4× online speedup while using 24.6% of the online communication of Hamada et al.'s radix-sort baseline for 2^{16} elements and five parties.

References

[1] K. Sahinbas and F. O. Catak, "Secure multi-party computation-based privacy-preserving data analysis in healthcare IoT systems," in *Interpretable cognitive Internet of things for healthcare*. Springer, 2023, pp. 57–72.

[2] P. S. Naidu, R. Kharat, R. Tekade, P. Mendhe, and V. Magade, "E-voting system using visual cryptography & secure multi-party computation," in *Proc. IEEE ICCUBEA*, 2016, pp. 1–4.

[3] S. S. Cao, L. W. Cong, and B. Yang, "Distributed ledgers and secure multiparty computation for financial reporting and auditing," *Management Science*, vol. 71, no. 5, pp. 3852–3872, 2025.

[4] L. Wang, H. Guo, W. Wu, and L. Zhou, "Efficient and privacy-preserving feature selection based on multiparty computation," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 3505–3518, 2025.

[5] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *Proc. ACM CCS*, 2020.

[6] P. Mohassel and P. Rindal, "ABY3: A mixed protocol framework for machine learning," in *Proc. ACM CCS*, 2018, pp. 35–52.

[7] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *Proc. CRYPTO*, 2007, pp. 572–590.

[8] O. Catrina and S. De Hoogh, "Improved primitives for secure multiparty integer computation," in *Proc. Security and Cryptography for Networks*, 2010, pp. 182–199.

[9] T. Reistad and T. Toft, "Linear, constant-rounds bit-decomposition," in *Proc. International Conference on Information Security and Cryptology*, 2010, pp. 245–257.

[10] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "Crypten: Secure multi-party computation meets machine learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 4961–4973, 2021.

[11] Y. Wang, G. E. Suh, W. Xiong, B. Lefaudeux, B. Knott, M. Annavaram, and H.-H. S. Lee, "Characterization of MPC-based private inference for transformer-based models," in *Proc. IEEE ISPASS*, 2022, pp. 187–197.

[12] F. Fu, H. Xue, Y. Cheng, Y. Tao, and B. Cui, "Blindfl: Vertical federated machine learning without peeking into your data," in *Proc. ACM SIGMOD*, 2022, pp. 1316–1330.

[13] H. Wu, W. Fang, Y. Zheng, J. Ma, J. Tan, and L. Wang, "Ditto: quantization-aware secure inference of transformers upon MPC," in *Proc. ICML*, 2024.

[14] W. Lu, Z. Huang, Z. Gu, J. Li, J. Liu, C. Hong, K. Ren, T. Wei, and W. Chen, "Bumblebee: Secure two-party inference framework for large transformers," in *Proc. NDSS*, 2025.

[15] K. Cheng, Y. Xia, A. Song, J. Fu, W. Qu, Y. Shen, and J. Zhang, "Mosformer: Maliciously secure three-party inference framework for large transformers," in *Proc. ACM CCS*, 2025, pp. 4124–4138.

[16] H. Morita, E. Pohle, K. Sadakane, P. Scholl, K. Tozawa, and D. Tschudi, "MAESTRO: multi-party AES using lookup tables," in *Proc. USENIX Security*, 2025, pp. 1965–1984.

[17] T. Nishide and K. Ohta, "Multiparty computation for interval, equality, and comparison without bit-decomposition protocol," in *Proc. PKC*, 2007, pp. 343–360.

[18] T. I. Reistad, "Multiparty comparison-an improved multiparty protocol for comparison of secret-shared values," in *Proc. International Conference on Security and Cryptography*, vol. 1, 2009, pp. 325–330.

[19] E. Makri, D. Rotaru, F. Vercauteren, and S. Wagh, "Rabbit: Efficient comparison for secure multi-party computation," in *Proc. Financial Cryptography and Data Security*, 2021, pp. 249–270.

[20] C. Ning and Q. Xu, "Multiparty computation for modulo reduction without bit-decomposition and a generalization to bit-decomposition," in *Proc. ASIACRYPT*, 2010, pp. 483–500.

[21] S. Laur, J. Willemson, and B. Zhang, "Round-efficient oblivious database manipulation," in *Proc. Information Security Conference*, 2011, pp. 262–277.

[22] P. Laud and M. Pettai, "Secure multiparty sorting protocols with covert privacy," in *Proc. Nordic Conference on Secure IT Systems*, 2016, pp. 216–231.

[23] K. Hamada, D. Ikarashi, K. Chida, and K. Takahashi, "Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation," *Cryptology ePrint Archive*, 2014.

[24] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, N. Kiribuchi, and B. Pinkas, "An efficient secure three-party sorting protocol with an honest majority," Cryptology ePrint Archive, Paper 2019/695, 2019.

[25] J. Boyar and M. G. Find, "Multiplicative complexity of vector valued Boolean functions," *Theor. Comput. Sci.*, vol. 720, no. C, pp. 36–46, Apr. 2018.

[26] N. Attrapadung, H. Morita, K. Ohara, J. C. Schuldt, and K. Tozawa, "Memory and round-efficient MPC primitives in the pre-processing model from unit vectorization," in *Proc. ACM AsiaCCS*, 2022, pp. 858–872.

[27] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *Proc. TCC*, 2006, pp. 285–304.

[28] T. Toft, "Constant-rounds, almost-linear bit-decomposition of secret shared values," in *Proc. Cryptographers' Track at the RSA Conference*, 2009, pp. 357–371.

[29] T. Veugen, "Linear round bit-decomposition of secret-shared values," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 3, pp. 498–506, 2015.

[30] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for MPC over mixed arithmetic-binary circuits," in *Proc. CRYPTO*, 2020, pp. 823–852.

[31] H. Chen, H. Li, Y. Wang, M. Hao, G. Xu, and T. Zhang, "Privdt: An efficient two-party cryptographic framework for vertical decision trees," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1006–1021, 2023.

[32] P. Xing, H. Li, M. Hao, H. Chen, J. Hu, and D. Liu, "Distributed function secret sharing and applications," in *NDSS*, 2025.

[33] D. Rathee, M. Rathee, R. K. Kiran Goli, D. Gupta, R. Sharma, N. Chandran, and A. Rastogi, "Sirnn: A math library for secure rnn inference," in *Proc. IEEE S&P*, 2021, pp. 1003–1020.

[34] B. Zhang, "Generic constant-round oblivious sorting algorithm for MPC," in *Proc. International Conference on Provable Security*, 2011, pp. 240–256.

[35] A. Agarwal, E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, M. Kelkar, and Y. Ma, "Secure sorting and selection via function secret sharing," in *Proc. ACM CCS*, 2024, pp. 3023–3037.

[36] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Computer Conf.*, 1968, pp. 307–314.

[37] K. V. Jönsson, G. Kreitz, and M. Uddin, "Secure multi-party sorting and applications," *Cryptology ePrint Archive*, 2011.

[38] M. T. Goodrich, "Randomized shellsort: A simple data-oblivious sorting algorithm," *J. ACM*, vol. 58, no. 6, pp. 1–26, Dec. 2011.

[39] G. Wang, T. Luo, M. T. Goodrich, W. Du, and Z. Zhu, "Bureaucratic protocols for secure two-party sorting, selection, and permuting," in *Proc. ACM AsiaCCS*, 2010, pp. 226–237.

[40] G. Asharov, T. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Bucket oblivious sort: An extremely simple oblivious sort," in *Proc. SOSA*, 2020, pp. 8–14.

[41] M. T. Goodrich, "Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in O(n log n) time," in *Proc. ACM STOC*, 2014, pp. 684–693.

[42] M. Ajtai, J. Komlós, and E. Szemerédi, "An O(n log n) sorting network," in *Proc. ACM STOC*, 1983, pp. 1–9.

[43] K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi, "Practically efficient multi-party sorting protocols from comparison sort algorithms," in *Proc. International Conference on Information Security and Cryptology*, 2013, pp. 202–216.

[44] D. Bogdanov, S. Laur, and R. Talviste, "A practical analysis of oblivious sorting algorithms for secure multi-party computation," in *Proc. Nordic Conference on Secure IT Systems*, 2014, pp. 59–74.

[45] G. Asharov, K. Hamada, D. Ikarashi, R. Kikuchi, A. Nof, B. Pinkas, K. Takahashi, and J. Tomida, "Efficient secure three-party sorting with applications to data analysis and heavy hitters," in *Proc. ACM CCS*, 2022, pp. 125–138.

[46] M. Keller and P. Scholl, "Efficient, oblivious data structures for MPC," in *Proc. ASIACRYPT*, 2014, pp. 506–525.

[47] M. Chase, E. Ghosh, and O. Poburinnaya, "Secret-shared shuffle," in *Proc. ASIACRYPT*, 2020, pp. 342–372.

[48] J. Gao, Y. Zhang, and S. Zhong, "Multiparty shuffle: Linear online phase is almost for free," Cryptology ePrint Archive, Paper 2024/1936, 2024.

[49] J. Bar-Ilan and D. Beaver, "Non-cryptographic fault-tolerant computing in constant number of rounds of interaction," in *Proc. ACM PODC*, 1989, pp. 201–209.

[50] J. Launchbury, I. S. Diatchki, T. DuBuisson, and A. Adams-Moran, "Efficient lookup-table protocol in secure multiparty computation," *SIGPLAN Not.*, vol. 47, no. 9, pp. 189–200, Sep. 2012.