Efficient 2PC for Constant Round Secure Equality Testing and Comparison

Tianpei Lu∗†, Xin Kang†‡, Bingsheng Zhang∗§, Zhuo Ma†§, Xiaoyuan Zhang*, Yang Liu† and Kui Ren*

∗The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
†Xidian University, China
lutianpei@zju.edu.cn, kangxin@stu.xidian.edu.cn,bingsheng@zju.edu.cn, mazhuo@mail.xidian.edu.cn,
zhangxiaoyuan@zju.edu.cn, bcds2018@foxmail.com, kuiren@zju.edu.cn

Abstract

Secure equality testing and comparison are two important primitives that have been widely used in many secure computation scenarios, such as privacy-preserving machine learning, private set intersection, secure data mining, etc. In this work, we propose new constant-round two-party computation (2PC) protocols for secure equality testing and secure comparison. Our protocols are designed in the online/offline paradigm. Theoretically, for 32-bit integers, the online communication for our equality testing is only 76 bits, and the cost for our secure comparison is only 384 bits. Our benchmarks show that (i) our equality is 9× faster than the Guo et al. (EUROCRYPT 2023) and 15× of the garbled circuit scheme (EMP-toolkit). (ii) our secure comparison protocol is 3× faster than Guo et al. (EUROCRYPT 2023), 6× faster than both Rathee et al. (CCS 2020) and garbled circuit scheme.

†Tianpei Lu and Xin Kang contributed equally to this work.
§Corresponding author: Bingsheng Zhang and Zhuo Ma.
## CONTENTS

I **Introduction** 3

I-A Related work 5

II **Preliminaries** 6

III **Equality Testing** 10

III-A One-round equality testing 10

III-B Two-round equality testing 12

IV **Secure Comparison** 16

IV-A Protocol Overview 16

IV-B Realize $F_{eq}$ 22

V **Performance Evaluation** 25

V-A Experiment Setting 25

VI **Conclusion** 26

References 26

Appendix A: Other building block 28

A-A OLE protocol 28

A-B The functionality of oblivious short list zero check 28

A-C Oblivious short-list zero check with OLE 29

Appendix B: Other benchmarks 30

B-A Offline of Equality Testing and Secure Comparison 30

B-B 32-bit Secure comparison 30

B-C Running time in different input length 31

B-D Running time in different input length 32
I. INTRODUCTION

Secure multiparty computation (MPC) [7], [25], [49] enables multiple untrusted parties to perform joint computations without revealing their private data. In the early stages, general-purpose MPC protocols [25], [29], [49] were widely studied and significantly improved in performance. Recently, researchers have focused on specific functions that benefit from tailor-made protocol designs and achieve performance far beyond general-purpose implementations [37], [42], [52]. The secure comparison and equality testing problem as typical cases have been considered, in which the parties joint calculate \( a > b \) or \( a = b \) for the private input \( a, b \) without disclosing them. Secure comparison and equality testing enjoy numerous applications as fundamental building blocks for various primitives in privacy computing, such as federated learning, privacy-preserving machine learning, advertising bidding systems, biometric authentication, and so on. We provide thereafter a non-exhaustive list of applications for secure comparison or equality testing.

- **Privacy-preserving machine learning.** Secure comparison is an important component for privacy-preserving machine learning [12], [16], especially for non-linear functions such as ReLU, MaxPool, and so on [52]. Furthermore, a series of works [29], [31], [36] evaluate arbitrary functions, including Sigmod, GeLU, softmax, etc., through piecewise function fitting based on comparison and polynomial evaluation.

- **Private set intersection.** Private Set Intersection (PSI) [14], [32], [44] is a widely used protocol that enables two parties to securely compute a function over the intersected part of their shared datasets and has been a significant research focus over the years. Currently, in PSI instantiations, equality testing accounts for more than 50% of the total communication cost of the protocol [41]. Therefore, optimizing the communication cost of equality testing is of great importance for PSI.

- **Secure Data Mining.** Secure data mining [27], [39] can facilitate the identification of the most relevant items or patterns without exposing raw data. In secure data mining, secure comparison is used in data mining tasks such as identifying the top-\( k \) items [23], outlier detection [46], and other analytics, where comparisons are necessary to draw insights from distributed datasets without compromising data privacy. Therefore, optimizing the performance and efficiency of secure comparison can drive the development and application of data mining technologies.

In general, the performance improvement of secure comparison can benefit a wide range of MPC applications. A series of secure comparison protocols [37], [52] in multi-party scenarios have achieved significant improvements resulting in efficient secure comparison/equality testing. Nevertheless, in two-party computation (2PC) scenarios, secure comparison and equality testing remain major performance bottlenecks in practice. The state-of-the-art (SOTA) [14], [29], [40] unanimously lead to the conclusion that secure 2PC comparison/equality-testing is magnitude slower than other secure operations, e.g., secure multiplication.

A sequence of efforts [19], [40] has been made to optimize the communication of comparison or equality testing. In contrast, in these works, the benefit of communication volume inevitably makes sacrifices on the communication rounds, i.e., logarithmic rounds, suffered a poor performance in the high network delay. The other approaches focus on the constant-round protocols. The typical solutions are based on the garbled circuit [45], [50] scheme or
the function secret sharing (FSS) scheme. The garbled circuit scheme requires massive communication and computation in the circuit evaluation (in the online phase), leading to inferior practical performance to the protocols with logarithmic rounds. FSS gains prior online communication compared to the garbled circuit scheme. In contrast, its online computation cost is close to garbled circuit (GC). Only considering the online phase, to the best of our knowledge, FSS is the most efficient solution for both equality testing and secure comparison. However, conventional FSS is performed on the three-party scenario, which requires the third party to generate the correlated keys. Migrating to the 2PC, the correlated keys should be performed under MPC (Equivalent to running massive PRGs under MPC), which is completely beyond the practical. Recently, a line of works design correlated keys generation protocols that move the PRGs evaluation to the local. As a trade-off, its computation cost is exponential to the data size \( n \). Considering a large \( n \), it is even impossible to output the result.

So far as we know, there doesn’t exist a practical constant round secure comparison or equality testing protocol while holding very efficient online phase performance with a practical offline phase.

**Our Result.** In this work, we focus on secure equality testing and comparison in the two-party computation setting, i.e., Alice and Bob hold the secret input respectively, and look for the shared result of the comparison or equality testing on the inputs. We design low (constant) communication rounds protocols with relatively efficient offline communication volume, ultimately improving overall performance. We show that our protocols are secure against passive adversarial in the universal composability framework of Canetti.

**2-round Equality testing.** We first propose a dimension reduction scheme for \( a \) and \( b \), reducing them to \( a' \) and \( b' \) such that \( a' = b' \) if and only if \( a = b \). The length of \( a' \) and \( b' \) is \( \log n \). Subsequently, we construct a look-up table with linear communication. Both parties sample random numbers \( \varepsilon_0 \) and \( \varepsilon_1 \) and then share a look-up table \( \vec{T} \) such that only the \((\varepsilon_0 + \varepsilon_1)\)th value is 1, this is \( t_{\varepsilon_0 + \varepsilon_1} = 1 \), and all other values are 0 in the offline phase. The parties obtain the result of the equality testing by locally selecting the sharing of \( t_{\varepsilon_0 + \varepsilon_1} + a' - b' \) in the online phase.

**Secure comparison.** We propose a novel 3-round secure comparison protocol \( \Pi_{cmp} \) in the semi-honest setting. Intuitively, we start from the high bit and check the first different bit of \( a \) and \( b \), where the value of \( a \) on the position of different bit corresponds to the result of the comparison. We construct a secret shared list \( \{s\}_n \) that highlights such a position. Consequently, we let \( P_0 \) guess the comparison result \( \Delta \), and open all the possible positions of such comparison result, namely, the position \( \zeta_i \) for \( a_{\zeta_i} = \Delta \), to \( P_1 \). \( P_1 \) checks if there exists \( \zeta_i \), such that \( s_{\zeta_i} \) is highlighted. If \( P_0 \) guess wrong (no highlighted \( s_{\zeta_i} \)), \( P_1 \) set output \( z_1 = 1 \) to flap \( \Delta \). Otherwise \( P_1 \) outputs \( z_1 = 0 \).

**Performance.** Table I depicts the communication comparison between our protocols and SOTA 2PC solutions. Our equality testing protocol requires 2 rounds of \( O(n) \) bits communication in the online phase, which is close to function secret sharing scheme \([20], [26]\), where our computation cost is much slighter than FSS (without invoking any PRF) leading to a faster online phase, i.e. the running time of FSS is over \( 7 \times \) more than ours, in LAN/MAN/WAN setting. Moreover, compared to FSS, the offline of our protocol is magnitude efficient, i.e. over \( 1000 \times \) faster than it. For the other baseline – garbled circuit \([49], [50]\), its online phase communication is \( 200 \times \) higher than ours. Specifically, our benchmark shows that in the MAN setting, our protocol achieves \( 15 \times \) prior
performance.

Our secure comparison protocol requires 3 round of \(2n + 2n \log n\) bits communication in the online phase. Similar to equality testing, our protocol outperforms the SOTA protocols. Compared to FSS [26], our protocol achieves over \(3 \times\) online phase performance improvement, and over \(1000 \times\) in the offline phase. Compared to the SOTA comparison CrypTflow2 [43], our protocol achieves over \(6 \times\) improvement in both MAN and WAN settings.

**Paper Organization.** Section II introduces the preliminary including notations and the primitives to construct our protocols. The rest of the paper is organized as follows. In Section III we propose our equality testing protocol involving one-round and two-round construction. In Section IV we introduce our three-round secure comparison protocol. Section V conducts the performance evaluation of our equality testing and secure comparison protocols.

**A. Related work**

The concept of secure comparison was first proposed by Yao [49], a.k.a, millionaire’s problem. Subsequently, equality testing called socialist millionaires’ problem [30] has been successively proposed. The research in the areas has experienced rapid and consistent development. Due to the primitive similarities between secure protocols for equality tests and comparisons, we provide a unified representation. We categorize the works into five types based on the involved fundamental building blocks: GC-based-CMP/EQ, HE-based-CMP/EQ, OT-based-CMP/EQ, FSS-based-CMP/EQ, and Generic Two-Party Computation. In the following, we let \(n\) denote the input length.

**GC-based-CMP/EQ.** The secure comparison and equality testing protocols were initially constructed by Yao circuits [49]. Kolesnikov et al. [35] proposed a protocol for constructing universal circuits almost exclusively composed of XOR gates, which relies on the random oracle (RO) assumption. Then, they [34] optimize the assumption by allowing one party to garble circuits containing comparison gates, achieving secure comparison through AND gates. Zahur et al. [51] introduced an approach to garbling AND gates using two ciphertexts and XOR gates using zero ciphertexts concurrently, resulting in half the communication cost to compute AND gates. Despite the constant round complexity protocol realized, their communication amount is usually significant.

**HE-based-CMP/EQ.** The beginning of solving the millionaire problem from homomorphic encryption (HE) can be traced back to the protocol proposed by Blake et al. [9]. Subsequently, Garay et al. [24] proposed a secure comparison scheme based on threshold homomorphic encryption. However, the comparison can only be performed by a trusted third party. Cheon et al. [17] proposed a comparison scheme based on HE by using a composite polynomial approximation to obtain an approximate comparison result. However, this scheme is unable to achieve equality testing.

**OT-based-CMP/EQ.** When multiple instances of secure comparison or equality testing are needed, the approach based on oblivious transfer extension is commonly used. The method requires a constant number of public key operations and only inexpensive symmetric operations for each invocation. Couteau [18] proposed a scheme that relied on oblivious transfer (OT) to securely perform a bitwise comparison with \(n\) AND gates. Rathee et al. proposed a framework named CrypTflow2 [43], which recursively equated the comparison of two integers to the comparison of sub-integers of length \((m \leq n)\). The sub-integer comparison was facilitated by 1-out-of-\(2^m\) OT. Therefore, the
comparison could be implemented through \( n/m - 1 \) AND gates. Subsequently, Chandran et al. \([14]\) extends the idea to equality testing. Huang et al. \([29]\) further optimized communication cost in CrypTflow2 \([43]\) by replacing the OT with VOLE-type OT.

**FSS-based-CMP/EQ.** Function secret sharing (FSS) \([10], [11]\) allows two parties to evaluate a secure function with correlated keys locally, and output a shared result, whereas the typical solution requires a third party to generate the corresponding keys. The distributed point function (DPF) \([11]\) can be used to realize the equality test directly and the distributed comparison function (DCF) \([10]\) can be used to realize secure comparison. The correlated keys generation scheme \([20], [26]\) employs FSS on the two parties’ computation.

**Generic Two-Party Computation.** Generic two-party computation techniques enable secure computation of functions expressed as boolean circuits. Demmler et al. \([19]\) presented a framework named ABY that efficiently combines arithmetic sharing, Boolean sharing, and Yao’s garbled circuits to perform secure two-party computation. Secure comparison and equality testing could be efficiently instantiated by ABY. The process involved initially converting the secret input from arithmetic to Boolean form (A2B), followed by conducting bitwise comparisons, and finally reversing the transformation (B2A). Patra et al. \([40]\) optimized multiplication computations in ABY2.0 by depending on function precomputation, reducing the communication cost during the online phase to half of that in ABY.

### II. Preliminaries

**Notation.** The frequently used notations are shown in Table II. Let \( \mathcal{P} := \{P_0, P_1\} \) be the two MPC parties. We denote a vector \( \{a_0, \ldots, a_{n-1}\} \) as \( \vec{A} \), and \( a_i \) be the \( i \)-th element of \( \vec{A} \). We denote \( [n] \) as the index set \( \{0, \ldots, n-1\} \), and \( [1, n] \) as the index set \( \{1, \ldots, n-1\} \). Let \( 1 \{b\} \) denote the indicator function that is 1 when \( b \) is true and 0 when \( b \) is false. Let \( (1, n) \)-OT denote the 1-out-of-\( n \) OT. We define \( \text{shift}(\vec{X}, i) \) as the operation of shifting the column vector \( \vec{X} \) down by \( i \) positions. In additional, we define \( [\cdot]^p \) over finite field \( \mathbb{Z}_p \) as \( [x]^p := ([x]_1 \in \mathbb{Z}_p, [x]_2 \in \mathbb{Z}_p) \) where \( x = [x]_1 + [x]_2 \mod p \). \( P_i \) for \( i \in \{0, 1\} \) hold share \( [x]_i \). We denote the matrix \( M \) as \( \mathbf{M} \), and the element in the \( i \)-th row and \( j \)-th column of \( M \) as \( m_{i,j} \).

**Threat model and security.** Our equality testing and comparison protocols ensure security within the standard semi-honest setting. In this scenario, the adversary may attempt to extract private information from legitimate messages but must adhere strictly to the protocol’s procedure. The security proof is based on the Universal Composability (UC) framework \([13]\), which follows the simulation-based security paradigm. In the UC framework, protocols are executed across multiple interconnected machines. The network adversary \( \mathcal{A} \) is allowed to partially control the communication tapes of all uncorrupted machines, observing messages sent to/from uncorrupted parties and influencing message sequences. Then, a protocol \( \Pi \) is considered UC-secure in realizing a functionality \( F \) if, for every probabilistic polynomial-time (PPT) adversary \( \mathcal{A} \) targeting an execution of \( \Pi \), there exists another PPT adversary known as a simulator \( \mathcal{S} \) attacking the ideal execution of \( F \) such that the executions of \( \Pi \) with \( \mathcal{A} \) and that of \( F \) with \( \mathcal{S} \) are indistinguishable to any PPT environment \( \mathcal{Z} \).

**The idea world execution** \( \text{Ideal}_{F,S,Z}(1^\lambda) \). In the ideal world, the parties \( \mathcal{P} := \{P_0, P_1\} \) only communicate with the ideal functionality \( F^I_{2pc} \) with the executed function \( f \). Both parties send their share to \( F^I_{2pc} \), and \( F^I_{2pc} \) calculates and output the result to \( P_0 \) and \( P_1 \).
TABLE I: Comparison with the state-of-the-art secure comparison and equality testing protocols. \( \lambda \) is the computational security parameter; \( \mu \) is ECC group representation length and \( \mu = 256 \); \( n \) is the length of the element to be compared.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Protocol</th>
<th>Offline Communication</th>
<th>Online Communication</th>
<th>Round</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Equality Testing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC-based-EQ</td>
<td>Yao [48], [49]</td>
<td>-</td>
<td>4( \lambda n )</td>
<td>2</td>
</tr>
<tr>
<td>Generic Two-Party Computation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABY [19]</td>
<td>6( \lambda n + n )</td>
<td>2( \lambda n + 6n )</td>
<td>( \log n + 5 )</td>
<td></td>
</tr>
<tr>
<td>ABY2.0 [40]</td>
<td>5( \lambda n + 2n )</td>
<td>( \lambda n + 6n )</td>
<td>( \log n + 4 )</td>
<td></td>
</tr>
<tr>
<td>FSS-based-EQ</td>
<td>Half-Tree [26]</td>
<td>(( n + 2 )( \lambda^* ))</td>
<td>2( n )</td>
<td>1</td>
</tr>
<tr>
<td>DPF [11]</td>
<td>4( n(\lambda + 1) + \lambda + n^\dagger )</td>
<td>2( n )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>OT-based-EQ</td>
<td>CO [18]</td>
<td>3( \lambda n )</td>
<td>2( n + 2 \log n + 10 )</td>
<td>( \log^* n + 1 )</td>
</tr>
<tr>
<td></td>
<td>CGS [14]</td>
<td>( \frac{3}{2} \lambda n + 8n )</td>
<td>5( n - 4 )</td>
<td>( \log n + 4 )</td>
</tr>
<tr>
<td></td>
<td>( \Pi_{eq} ) (Section III)</td>
<td>( \lambda \log n + 2n )</td>
<td>2( n + 2 \log n + 2 )</td>
<td>2</td>
</tr>
<tr>
<td><strong>Secure Comparison</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC-based-CMP</td>
<td>Yao [48], [49]</td>
<td>-</td>
<td>4( \lambda n )</td>
<td>2</td>
</tr>
<tr>
<td>HE-based-CMP</td>
<td>GSV [24]</td>
<td>-</td>
<td>18( \mu n + 8\mu )</td>
<td>9</td>
</tr>
<tr>
<td>Generic Two-Party Computation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABY [19]</td>
<td>6( \lambda n + 17\lambda + n )</td>
<td>2( \lambda n + 20\lambda )</td>
<td>( \log n + 5 )</td>
<td></td>
</tr>
<tr>
<td>ABY2.0 [40]</td>
<td>5( \lambda n + 17\lambda + 2n )</td>
<td>( \lambda n + 9\lambda )</td>
<td>( \log n + 4 )</td>
<td></td>
</tr>
<tr>
<td>FSS-based-CMP</td>
<td>Half-Tree [26]</td>
<td>(( n + 2 )( \lambda^* ))</td>
<td>2( n )</td>
<td>1</td>
</tr>
<tr>
<td>DCF [10]</td>
<td>4( n(\lambda + 1) + \lambda + n^\dagger )</td>
<td>2( n )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>OT-based-CMP</td>
<td>CO [18]</td>
<td>6( \lambda n )</td>
<td>6( n + 4 \log n )</td>
<td>4( \log^* \lambda + 5 )</td>
</tr>
<tr>
<td>Cryptflow2 [43]</td>
<td>( \lambda n + 16n )</td>
<td>10( n - 8 )</td>
<td>( \log n + 4 )</td>
<td></td>
</tr>
<tr>
<td>Cheetah [29]</td>
<td>( \lambda n + 11n )</td>
<td>10( n - 8 )</td>
<td>( \log n + 4 )</td>
<td></td>
</tr>
<tr>
<td>( \Pi_{cmp}(\text{Section IV}) )</td>
<td>( n\lambda \log n + n \log n + n )</td>
<td>2( n + 2n \log n )</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

* \( \log^* \) represents the iterated logarithm.

* Under correlated keys generation scheme which performs \( O(2^n) \) times Hash locally.

† Under a trusted third-party dealer.

The real world execution \( \text{Real}_{\Pi, A, Z}(\lambda^\dagger) \). In the real world, the parties \( P := \{ {P_0}, {P_1} \} \) communicate with each other, it executes the protocol \( \Pi \). Our protocols work in the pre-processing model, but we analyze the offline and online protocols together as a whole.
TABLE II: Notations

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vec{A} )</td>
<td>The vector ( \vec{A} := {a_0, \ldots, a_{n-1}} ).</td>
</tr>
<tr>
<td>( a_i )</td>
<td>The ( i^{th} ) element of vector ( \vec{A} ), when the context is clear, we abuse ( a_i ) as the ( i^{th} ) bit of value ( a ).</td>
</tr>
<tr>
<td>([n])</td>
<td>The index set ( {0, \ldots, n-1} ).</td>
</tr>
<tr>
<td>([1, n])</td>
<td>The index set ( {1, \ldots, n-1} ).</td>
</tr>
<tr>
<td>( ([p], [T]) )</td>
<td>The algorithm shares over ( \mathbb{Z}_p ) owned by ( P_0 ) and ( P_1 ).</td>
</tr>
<tr>
<td>( \mathbf{1} {b} )</td>
<td>The indicator function, which evaluates to 1 when ( b ) is true and 0 when ( b ) is false.</td>
</tr>
<tr>
<td>( (m, n))-OT</td>
<td>( m )-out-of-( n ) OT.</td>
</tr>
<tr>
<td>( \text{shift}(\vec{X}, i) )</td>
<td>Shift the column vector ( \vec{X} ) down by ( i ) positions.</td>
</tr>
<tr>
<td>( M )</td>
<td>The matrix ( M ).</td>
</tr>
<tr>
<td>( m_{(i,j)} )</td>
<td>The element in the ( i^{th} ) row and ( j^{th} ) column of the matrix ( M ).</td>
</tr>
</tbody>
</table>

**Definition 1.** We say protocol \( \Pi \) UC-secure realizes functionality \( F \) if for all PPT adversaries \( A \) there exists a PPT simulator \( S \) such that for all PPT environment \( Z \), it holds:

\[
\text{Real}_{\Pi, A, Z}(1^\lambda) \approx \text{Ideal}_{F, S, Z}(1^\lambda)
\]

**Oblivious transfer.** For an instance of \((1, 2)\)-OT \([21], [22]\), the sender’s inputs to the \( F_{(1,2)}\)-OT are the strings \( m_0 \) and \( m_1 \in \{0, 1\}^l \), and the receiver’s input is a bit \( i \in \{0, 1\} \). The receiver obtains \( m_i \) from the \( F_{(1,2)}\)-OT and the sender receives no output. Random OT (ROT) \([8]\) is a special case of OT where there is no input. The sender receives two random strings \( r_0 \) and \( r_1 \in \{0, 1\}^l \), while the receiver obtains a bit \( i \in \{0, 1\} \) and \( m_i \). The \( F_{(n-1,n)}\)-ROT \([15]\) makes the sender obtains \( \{m_0, \ldots, m_{n-1}\} \), while the receiver obtains a element \( b \in [n] \) and \( \{m_i\} \) for \( i \in [n]\backslash\{b\} \). ROT is input-independent, which can be executed in the offline phase and OT can be constructed with linear communication in the online phase. The \((1, n)\)-OT \([38], [47]\) is a generalization of \((1,2)\)-OT. The
sender’s inputs to the $F \in OLE$. Oblivious Linear Evaluation (OLE) \cite{6}, \cite{33} is a foundational component in various secure computation protocols \cite{28}, \cite{42}, \cite{44}. In the standard OLE protocol \cite{6}, involves constructing a VOLE \cite{44}, the parties have no input. (Note that $P_1$ does not have the $i_1$th column of $M_i$.)

2) $P_0$ and $P_1$ generate the binary matrix $M \in \{0,1\}^{N \times N}$ by using $m_i$ as the binary column vectors for $i \in [N]$, locally. (Note that $P_1$ does not have the $i_1$th column of $M_i$.)

3) $P_0$ and $P_1$ left cycle shift the $i$th row of $M$ by $i$ positions locally for $i \in [N]$.

4) $P_0$ computes $v_i = \bigoplus_{j=0}^{N-1} m_i(j,i)$ and $u_i = \bigoplus_{j=0}^{N-1} m_i(j,i)$ for $i \in [N]$, and denotes $\vec{V} := \{v_0, \ldots, v_{N-1}\}$ and $\vec{U} := \{u_0, \ldots, u_{N-1}\}$. 

5) $P_1$ computes $w_i = v_i \oplus u_{i+1}$, and denotes $\vec{W} := \{w_0, \ldots, w_{N-1}\}$. 

6) $P_0$ sends $\vec{S}' = \vec{T}' \oplus \vec{U}$ to $P_1$ and sets $\vec{T}_0 := \vec{V}$. 

7) $P_1$ computes $\vec{T}_1 := \text{shift}(\vec{S}', \epsilon_1) \oplus \vec{W}$.

Fig. 2: The Vector Oblivious Shift Evaluation Protocol.

sender’s inputs to the $F_{(1,n)}$-OT are the $n$ strings $\{m_0, \ldots, m_{n-1}\}$, and the receiver’s input is a choose number $i \in [n]$. The receiver obtains $m_i$ from the $F_{(1,n)}$-OT and the sender receives no output. The $(1, n)$-OT can be constructed via $(1, 2)$-OT.

**Oblivious Linear Evaluation.** Oblivious Linear Evaluation (OLE) \cite{6}, \cite{33} is a foundational component in various secure computation protocols \cite{28}, \cite{42}, \cite{44}. In the standard OLE protocol \cite{6}, $P_0$ receives random values $a$ and $b$, while $P_1$ receives a random value $u$ and $w = au + b$. A symmetric variant of OLE \cite{33}, known as product sharing \cite{6}, involves $P_0$ and $P_1$ each sampling $a$ and $b$, respectively. After the protocol, $P_0$ obtains $u$ and $P_1$ obtains $v$, satisfying the condition $ab = u + v$. This set of random values $(a, b, u, v)$ constitutes an OLE tuple. In vector OLE (VOLE) \cite{44}, the parties have no input. $P_0$ obtains a random value $u$ and a random vector $\vec{B}$. $P_1$ obtains a random vector $\vec{A}$ and the vector $\vec{V} = \vec{A}u + \vec{B}$, where $v_i = a_iu + b_i$.

**Secure permutation.** The secure permutation \cite{15} is a protocol that allows two parties, one of the parties holds the permutation and the other party holds the list, to jointly permute the list and obtain additive secret shares of the permuted list. Although this problem could be addressed using generic MPC, the most efficient implementation \cite{15} currently is constructed by OT. We define the functionality $F_{\text{permute}}$ as follow: the $P_0$ inputs a permutation $\pi$, and $P_1$ inputs a list $\vec{X} := \{x_0, \ldots, x_{n-1}\}$. After the protocol, they obtain the secret shares of the permuted list $\{x_{\pi(0)}, \ldots, x_{\pi(n-1)}\}$. 

---

Protocol $\Pi_{\text{orse}}^N(\vec{T}')$

Input : $P_0$ inputs a binary vector $\vec{T}' \in \mathbb{Z}_2^N$.

Output : $P_0$ receives a share vector $\vec{T}_0$, $P_1$ receives an offset $\epsilon_1 \in [N]$ and $\vec{T}_1$, where $\vec{T}_0 \oplus \vec{T}_1 = \text{shift}(\vec{T}', \epsilon_1)$.

**Execution**

1) $P_0$ and $P_1$ invoke $F_{(N-1,N)}$-ROT:
   - $P_0$ receives $\{m_i | i \in [N], m_i \in \mathbb{Z}_2^N\}$.
   - $P_1$ receives $\epsilon_1$ and $\{m_i | i \in [N] \setminus \{\epsilon_1\}, m_i \in \mathbb{Z}_2^N\}$.

2) $P_0$ and $P_1$ generate the binary matrix $M \in \{0,1\}^{N \times N}$ by using $m_i$ as the binary column vectors for $i \in [N]$, locally. (Note that $P_1$ does not have the $i_1$th column of $M_i$.)

3) $P_0$ and $P_1$ left cycle shift the $i$th row of $M$ by $i$ positions locally for $i \in [N]$.

4) $P_0$ computes $v_i = \bigoplus_{j=0}^{N-1} m_i(j,i)$ and $u_i = \bigoplus_{j=0}^{N-1} m_i(j,i)$ for $i \in [N]$, and denotes $\vec{V} := \{v_0, \ldots, v_{N-1}\}$ and $\vec{U} := \{u_0, \ldots, u_{N-1}\}$.

5) $P_1$ computes $w_i = v_i \oplus u_{i+1}$, and denotes $\vec{W} := \{w_0, \ldots, w_{N-1}\}$.

6) $P_0$ sends $\vec{S}' = \vec{T}' \oplus \vec{U}$ to $P_1$ and sets $\vec{T}_0 := \vec{V}$.

7) $P_1$ computes $\vec{T}_1 := \text{shift}(\vec{S}', \epsilon_1) \oplus \vec{W}$.
III. EQUALITY TESTING

In the equality testing, $P_0$ inputs an integer $a \in \mathbb{Z}_2^n$ and $P_1$ inputs an integer $b \in \mathbb{Z}_2^n$. After the protocol, both parties receive boolean shares of $1 \{a = b\}$, which equals 1 if and only if $a = b$, and 0 otherwise.

In this section, we first design a toy protocol for equality testing with one round of communication in the online phase. However, this design results in an $O(2^n)$ communication complexity in the offline phase. To further balance the communication cost between the online and offline phases, we propose a dimension reduction scheme to optimize the toy protocol. This optimization allows for two rounds of communication in the online phase while ensuring that the $O(n)$ communication complexity in the offline phase.

A. One-round equality testing

We first describe a strawman example for the equality testing, then we design a one-round equality testing protocol.

A strawman example. A strawman example for equality testing is that $P_0$ generates a binary vector $\vec{T}$ as the look-up table, such that only the $a^{th}$ value of $\vec{T}$ is 1, while the value of other positions are 0. $P_1$ then uses $b$ to privately select $t_b$. Clearly, $t_b = 1$ if and only if $a = b$. Note that to enumerate all strings of length $n$, the size of $\vec{T}$ is $2^n$. For convenience, we define $N = 2^n$. However, the current design doesn’t guarantee the privacy of the result of the equality testing, as the result is public to $P_1$. Keeping $t_b$ secret, a simple approach is as follows: $P_0$ first samples a bit $s$ and then computes $t'_i = s \oplus t_i$ for $i \in [N]$ to generate $\vec{T}'$, and $P_1$ privately select $t'_b$ instead of $t_b$. Obviously, $s \oplus t'_b = 1$ if and only if $a = b$. Nevertheless, the process of $P_1$ privately selecting $t'_b$ requires an instance of $F_{(1,N)\cdot OT}$, which involves two rounds of communication and incurs an $O(2^n)$ communication complexity in the online phase.

One-round equality testing. Our goal is to design an equality testing protocol that achieves one round of communication and $O(n)$ communication complexity in the online phase. The protocol is described in Figure 4 and the overview is shown as follows. In the offline phase, $P_0$ and $P_1$ respectively pick offsets $\epsilon_0$ and $\epsilon_1$, and then they
generate a shared binary vector $\vec{T} := \{t_0, \ldots, t_{N-1}\}$, where only $t_{\epsilon_0+\epsilon_1} = 1$. In other words, $P_0$ picks an offset $\epsilon_0$ and generate a binary vector $\vec{T}'$ with only $t'_{\epsilon_0} = 1$. As the result, $P_0$ obtains $\vec{T}'_0$ and $P_1$ obtains $\epsilon_1$ and $\vec{T}'_1$, where $\vec{T}'_0$ and $\vec{T}'_1$ are the shares of $\vec{T}' := \text{shift}(\vec{T}', \epsilon_1)$, such that $[t_j]_0 \oplus [t_i]_1 = t_i$. In the online phase, $P_0$ and $P_1$ reveal the value $w = \epsilon_0 + \epsilon_1 + a - b$, and then select $[t_w]_0$ and $[t_w]_1$ locally as the result of equality testing. Specifically, $P_0$ computes $w_0 = \epsilon_0 + a$ and sends it to $P_1$. At the same round, $P_1$ computes $w_1 = \epsilon_1 - b$ and sends it to $P_0$. Subsequently, $P_0$ and $P_1$ can reveal $w$ locally. Clearly, $[t_w]_0 \oplus [t_w]_1 = 1$ if and only if $a = b$.

We construct our offline phase based on a primitive – Vector Oblivious Shift Evaluation (VOSE). Before introducing VOSE, we propose the random VOSE.

**Random Vector Oblivious Shift Evaluation.** In the random VOSE, $P_0$ receives two random binary vectors $\vec{U}$ and $\vec{V}$, while $P_1$ receives the offset $\epsilon_1$ and a vector $\vec{W}$, such that $\vec{W} = \text{shift}(\vec{U}, \epsilon_1) \oplus \vec{V}$. The random VOSE can be built from $\mathcal{F}_{(N-N)\text{-ROT}}$. Specifically, we describe the process as follows.

- $P_0$ and $P_1$ invoke an instance of $\mathcal{F}_{(N-N)\text{-ROT}}$. After the protocol, $P_0$ receives $N$ messages $\{m_0, \ldots, m_{N-1}\}$ and $m_i \in \mathbb{Z}_2^N$. $P_1$ receives $\epsilon_1$ and all messages except for $m_{\epsilon_1}$. We view each message as a $N$-dimension binary vector and denote the binary matrix consisting of $N$ column vectors as $\mathbf{M}$. Therefore, $P_0$ obtains the complete $\mathbf{M}$, while $P_1$ can obtain the $\mathbf{M}$ except for the $\epsilon_1$th column.
- $P_0$ and $P_1$ lift cycle shift the $i$th row of $\mathbf{M}$ by $i$ positions for $i \in [N]$, and denote the new matrix as $\mathbf{M}'$.
- $P_0$ computes $v_i = \bigoplus_{j=0}^{N-1} m_{(i,j)}$ and $u_i = \bigoplus_{j=0}^{N-1} m_{(j,i)}$ for $i \in [N]$ to generate $\vec{V}$ and $\vec{U}$. Obviously, $v_i$ is the XOR value of the $N$ bits in the $i$th row of $\mathbf{M}'$ and $u_i$ is the value of the $i$th column.
- $P_1$ computes $w_i = v_i \oplus u_{\epsilon_1+i} \mod N$ to generate $\vec{W} = \{w_0, \ldots, w_{N-1}\}$. Although $P_1$ cannot obtain $m_{(i,\epsilon_1+i)}$, both $v_i$ and $u_{\epsilon_1+i}$ include $m_{(i,\epsilon_1+i)}$. Therefore, $P_1$ can correctly compute $w_i$ without $m_{(i,\epsilon_1+i)}$.

Through steps 2 and 3, $P_1$ obtains $w_i = v_i \oplus u_{\epsilon_1+i}$ for $i \in [N]$, while $P_0$ obtains $v_i$ and $v_i$. Therefore, the vectors

![Protocol $\Pi_{eq}^N(a, b)$](image-url)
Based on the random VOSE, we elaborate on the implementation of VOSE in Vector Oblivious Shift Evaluation.

For the correctness, $\vec{T}$ as a result, we have $\vec{T} = \vec{T}_1 \oplus \vec{T}_0 \vec{V}$.

Vector Oblivious Shift Evaluation. Based on the random VOSE, we elaborate on the implementation of VOSE in the following three steps. The protocol is shown in Figure 2.

- $P_0$ and $P_1$ invoke the random VOSE. After the protocol, $P_0$ receives $\vec{U} \in \mathbb{Z}_2^N$ and $\vec{V} \in \mathbb{Z}_2^N$, while $P_1$ receives the offset $\vec{e}_1$ and a vector $\vec{W} \in \mathbb{Z}_2^N$, such that $\vec{W} = \text{shift}(\vec{U}, \vec{e}_1) \oplus \vec{V}$.
- $P_0$ sends $\vec{S}' = \vec{T}' \oplus \vec{U}$ to $P_1$ and sets $\vec{T}_0 = \vec{V}$.
- $P_1$ computes $\vec{T}'_1 = \text{shift}(\vec{S}', \vec{e}_1) \oplus \vec{W}$.

For the correctness, $\vec{T}_1 = \text{shift}(\vec{T}', \vec{e}_1) \oplus \text{shift}(\vec{U}, \vec{e}_1) \oplus \vec{V} = \text{shift}(\vec{T}', \vec{e}_1) \oplus \vec{V} = \vec{T} \oplus \vec{V}$ and $\vec{T}_0 = \vec{V}$.

As a result, we have $\vec{T}_0 \oplus \vec{T}_1 = \vec{T}$.

Efficiency. In the offline phase, $P_0$ and $P_1$ invoke one time of $\mathcal{F}_{(N-1, N)}$ and $P_0$ send $\vec{S}' \in \mathbb{Z}_2^N$. Therefore, the communication cost is $n \lambda + N$. In the online phase, the communication cost is $2n$ bits.

B. Two-round equality testing

We observe that, in the aforementioned protocol, the communication in the offline phase is closed to $N$ bits, namely, $2^n$, which is impractical in real-world applications. The overview is shown in Figure 3. To reduce the communication cost in the offline phase, we introduce a dimension reduction protocol that can diminish the overall communication, i.e., $O(n)$ bits communication. The overview of the protocol is shown in Figure 3.

Dimension reduction. The dimension reduction protocol is designed to reduce the integers $(a \in \mathbb{Z}_2^n, b \in \mathbb{Z}_2^n)$ to $(a' \in \mathbb{Z}_2^{\log n}, b' \in \mathbb{Z}_2^{\log n})$ such that $a' = b'$ if and only if $a = b$. The start point of generating $a'$ and $b'$ is that,
By filling in detailed descriptions, we complete our protocol, which is described in Figure 6.

Fig. 6: Two-Round Equality Testing.

At step 1, $P_0$ and $P_1$ invoke $n$ times of $\Pi_{\text{seq}}$ for $a_i$ and $b_i$ simultaneously. Then, they receive $s_i$ and $t_i$ for $i \in [n]$, such that $s_i + t_i = a_i \oplus b_i$.

At step 2, $P_0$ computes $[d]_0 = \sum_{i=0}^{n-1} s_i$ and $P_1$ computes $[d]_1 = \sum_{i=0}^{n-1} t_i$, where it holds that $d = \sum_{i=0}^{n-1} a_i \oplus b_i$. 

### Protocol $\Pi_{\text{seq}}^n (a, b)$

Input: $P_0$ inputs $a \in \{0, 1\}^n$ and $P_1$ inputs $b \in \{0, 1\}^n$.

Output: $P_0$ receives $[s]_0^n$ and $P_1$ receives $[s]_1^n$, where $[s]_0^n \oplus [s]_1^n = 1 \{a = b\}$.

**Execution:**

1. For $i \in [n]$, $P_0$ and $P_1$ invoke $\{s_i, t_i \in \mathbb{Z}_p\} \leftarrow \Pi_{\text{conv}}^{2-p}(a_i, b_i)$, where $s_i + t_i = a_i \oplus b_i$.
2. $P_0$ computes $[d]_0 = \sum_{i=0}^{n-1} s_i$, and $P_1$ computes $[d]_1 = \sum_{i=0}^{n-1} t_i$ locally.
3. $P_0$ and $P_1$ invoke $([e]_0^n, [e]_1^n) \leftarrow \Pi_{\text{seq}}^n ([d]_0, -[d]_1)$.

$d = \sum_{i=0}^{n-1} (a_i \oplus b_i) = 0$ if and only if $a = b$. Notice that the entropy-bit of $d$ is $\lfloor \log n + 1 \rfloor$. Thus, the arithmetic sharing of $d$, denoted as $[d]_0$ and $[d]_1$ where $d = [d]_0 + [d]_1$, are the expectional $a'$ and $b'$ such that $a' = [d]_0$ and $b' = -[d]_1$. The correctness is that $a' - b' = [d]_0 + [d]_1 = d$.

To generate $[d]$, our approach is to convert the boolean sharing of $a_i$ and $b_i$ into arithmetic sharing $s_i$ and $t_i$, such that $s_i + t_i = a_i \oplus b_i$. Consequently, $P_0$ and $P_1$ obtain the sharing of $d$ by computing $[d]_0 = \sum_{i=1}^{n} s_i$ and $[d]_1 = \sum_{i=1}^{n} t_i$, respectively. We refer to the above conversion process as sharing conversion. Formally, in an instance of sharing conversion, $P_0$ and $P_1$ input the boolean sharing $[u]_0^n$ and $[u]_1^n$. After the protocol, they receive the arithmetic sharing $[v]_0^n$ and $[v]_1^n$, satisfying $[v]_0^n + [v]_1^n = [u]_0^n \oplus [u]_1^n$. Here, $p > n$ is a prime.

An instance of sharing conversion can be easily constructed based on the $F_{\{1, 2\}}$-OT. In particular, $P_0$ samples $[s]_0^n$, and inputs $n_0 = [s]_0^n - [u]_0^n$ and $n_1 = [s]_0^n - (1 - [u]_0^n)$, $P_1$ inputs the selection bit $[u]_1^n$ and receives $z$, where $z = [s]_0^n - (|[u]_0^n \oplus [u]_1^n|)$. Consequently, $P_0$ sets $[v]_0^n = [s]_0^n$ and $P_1$ sets $[v]_0^n = -z$. For correctness, we have $[v]_0^n + [v]_1^n = [s]_0^n - z = [s]_0^n - [s]_0^n - (|[u]_0^n \oplus [u]_1^n|) = [u]_0^n \oplus [u]_1^n$ as required. However, all computations are currently performed in the online phase, resulting in the communication complexity is $O(n^2)$ and the round is 2.

### Optimization of share conversion

We attempt to shift a significant portion of expensive operations to the offline phase, resulting in only a small amount of computation and communication in the online phase. The specific description is illustrated in Figure 5. In the offline phase, $P_0$ and $P_1$ generate a random sharing conversion pair, i.e. $P_0$ receives $([r]_0^n, [t]_0^n)$ and $P_1$ receives $([r]_1^n, [t]_1^n)$, such that $[t]_0^n + [t]_1^n = [r]_0^n \oplus [r]_1^n$. In the online phase, $P_0$ computes $[w]_0^n = [a]_0^n \oplus [r]_0^n$ and sends it to $P_1$, while $P_1$ computes $[w]_1^n = [a]_1^n \oplus [r]_1^n$ and sends it to $P_0$. Subsequently, $P_0$ and $P_1$ compute the public value $w = [w]_0^n \oplus [w]_1^n$. Finally, $P_0$ sets $[v]_0^n = w + [t]_0^n - 2w[t]_0^n$ and $P_1$ sets $[v]_1^n = [t]_1^n - 2w[t]_1^n$ locally.

### Protocol description

By filling in detailed descriptions, we complete our protocol, which is described in Figure 6. Next, we will explain our protocol step by step as follows.

- At step 1, $P_0$ and $P_1$ invoke $n$ times of $\Pi_{\text{conv}}$ for $a_i$ and $b_i$ simultaneously. Then, they receive $s_i$ and $t_i$ for $i \in [n]$, such that $s_i + t_i = a_i \oplus b_i$.
- At step 2, $P_0$ computes $[d]_0 = \sum_{i=0}^{n-1} s_i$ and $P_1$ computes $[d]_1 = \sum_{i=0}^{n-1} t_i$, where it holds that $d = \sum_{i=0}^{n-1} a_i \oplus b_i$. 


At step 3, $P_0$ and $P_1$ invoke $[e] \leftarrow \Pi_{eq_1}^A([d]_0, [d]_1)$. Then, they output $[e]$ as the shared result of $\Pi\{a = b\}$.

**Efficiency.** In the offline phase, $P_0$ and $P_1$ invoke $n$ times of $\mathcal{F}_{(1,2)\cdot OT}$ and one times of $\mathcal{F}_{(p-1,p)\cdot ROT}$. In addition, $P_0$ send $\vec{S}' \in \mathbb{Z}_2^n$. The corresponding communication cost is $\lambda \log p + 2p + \lambda = \lambda [\log(n + 1)] + 2n + 3$ bits. In the online phase, $P_0$ and $P_1$ send $n$ bits to each other in the share conversion $\Pi_{\text{convert}}$, and send $p = [\log(n + 1)]$ bits to each other in the $\Pi_{eq_1}^P$. Therefore, the rounds are 2 and the communication cost is $2n + 2 \log n + 2$ bits.

**Security.** We define the functionality $\mathcal{F}_{eq}$ for the equality testing as an instance of $\mathcal{F}_{2\text{PC}}$, where $\mathcal{F}_{eq}$ receives $a$ from honest $P_0$ or $S$ and $b$ from honest $P_1$ or $S$, calculates $|e|^2_0 \oplus |e|_1^2 = 1\{a = b\}$ and sends $|e|^2_0$ to $P_0$ and $|e|^2_1$ to $P_1$. Next, we prove our protocol $\Pi_{eq_2}$ UC-realizes functionality $\mathcal{F}_{eq}$.

**Theorem 1.** The protocol $\Pi_{eq_2}$ as shown in Fig. 3 UC realizes $\mathcal{F}_{eq}$ in the $\{\mathcal{F}_{(1,2)\cdot OT}, \mathcal{F}_{(1,N)\cdot OT}\}$-hybrid model against semi-honest PPT adversaries with statical corruption.

**Proof.** To prove Theorem 1 we construct a PPT simulator $\mathcal{S}$, such that no non-uniform PPT environment $Z$ can distinguish between the ideal world Ideal$_{\mathcal{F}_{eq}, S, Z}(1^\lambda)$ and the real world Real$_{\Pi_{eq_1}, A, Z}(1^\lambda)$. We consider the following cases:

Case 1: $P_0$ is corrupted. We construct the simulator $\mathcal{S}$ which internally runs $A$, forwarding messages to/from $Z$ and simulates the interface of honest $P_1$.

- **Upon receiving (Input, sid) from $\mathcal{F}_{eq}$, $\mathcal{S}$ starts simulation.**
- **For the simulation of $i$th times of $\Pi_{\text{convert}}$, $i \in [n]$.**
  - **Upon receiving (Input, sid) from $\mathcal{F}_{eq}$, $\mathcal{S}$ starts simulation.**
  - $\mathcal{S}$ picks random $|r|^2 \in \mathbb{Z}_2$ and emulates $\mathcal{F}_{(1,2)\cdot OT}$ with input $|r|^2$;
  - When corrupted $P_0$ inputs $(m_{0,i}, m_{1,i})$ to $\mathcal{F}_{(1,2)\cdot OT}$, $\mathcal{S}$ records $(m_{0,i}, m_{1,i})$.
  - $\mathcal{S}$ calculates $s_i$ and $t_i$ with $m_{0,i}, m_{1,i}$.
  - $\mathcal{S}$ picks $|w_i|^2 \in \mathbb{Z}_2^2$ and acts as $P_1$ to send it to $P_0$.
  - Upon receiving $|w_i|^2$ from $P_0$, $\mathcal{S}$ calculate $w_i = |w_i|^2_0 \oplus |w_i|^2_1$ and $v_i = w_i + t_i - 2w_t_i$.

- $\mathcal{S}$ calculate $d_0 = \sum_{i=0}^{n-1} v_i$.
- **For the simulation of $\Pi_{eq_1}$,**
  - $\mathcal{S}$ emulates $\mathcal{F}_{(N-1,N)\cdot ROT}$ and forward the output $m_i \in \mathbb{Z}_2^n$ for $i \in [p]$ to $P_0$.
  - $\mathcal{S}$ generate the binary matrix $M$ by using the $\{m_i\}_{i \in [p]}$ as the binary column vectors, and left cycle shift the $i$th row of $M$ by $i$ positions locally for $i \in [p]$.
  - $\mathcal{S}$ computes $v_i = \bigoplus_{j=0}^{p-1} m_{i,j}$ to generate $\vec{T}_0$, such that $[t_{i}]_0 = v_i$; $\mathcal{S}$ computes $u_i = \bigoplus_{j=0}^{p-1} m_{(i,j)}$ to generate $\vec{U}$.
  - Upon receiving $\vec{S}'$ from $P_0$, $\mathcal{S}$ picks $w_1 \in \mathbb{Z}_p$ and acts as $P_1$ to send it to $P_0$. In addition, $\mathcal{S}$ computes $\vec{T}^0 = \vec{S}' \oplus \vec{U}$ with only $t'_0 = 1$ and extract $e_0$.
  - Upon receiving (Output, $[e]_0$) from $\mathcal{F}_{eq}$, $\mathcal{S}$ pick a random index $\rho$ satisfying $[t_{\rho}]_0 = [e]_0$.
  - $\mathcal{S}$ computes $w_1 = \rho - (e_0 + d_0)$ and acts as $P_0$ to send it to $P_1$.
  - Upon receiving $\vec{S}'$ from $P_0$, $\mathcal{S}$ picks $w_1 \in \mathbb{Z}_p$ and acts as $P_1$ to send it to $P_0$. 

14
Claim 1. If PRF_{Z_p}^{z}, PRF_{Z_p}^{z_p} and PRF_{Z_p}^{z_p} are the secure pseudorandom functions with adversarial advantage Adv_{PRF_{Z_p}^{z}}(1^\lambda, A), Adv_{PRF_{Z_p}^{z}}(1^\lambda, A) and Adv_{PRF_{Z_p}^{z}}(1^\lambda, A), then the ideal world Ideal_{eq,S,Z}(1^\lambda) and the real world Real_{eq,z,A,Z}(1^\lambda) are indistinguishable with advantage \( \epsilon = n \cdot (Adv_{PRF_{Z_p}^{z}}(1^\lambda, A) + Adv_{PRF_{Z_p}^{z}}(1^\lambda, A)) + p \cdot Adv_{PRF_{Z_p}^{z}}(1^\lambda, A) \).

Proof. In the ideal world for simulating \( \Pi_{convert} \), \([w_i]_1^2\) are picked random rather than calculated by \( b_i \oplus [r_i]_1^2 \).
Therefore, the advantage in invoking \( \Pi_{convert} \) is \( \epsilon_0 = n \cdot Adv_{PRF_{Z_p}^{z}}(1^\lambda, A) \). In the ideal world for simulating \( \Pi_{eq_i} \), \( m_i \) for \( i \in [p] \) are the output of \( F_{(N-1,N).ROT} \). In addition, \( w_1 = \rho - (\varepsilon_0 + d_0) \) are picked random rather than calculated by \( \varepsilon_1 - d_1 \), where the \( \rho \) is random. Therefore, the advantage in invoking \( \Pi_{eq_i} \) is \( \epsilon_1 = Adv_{PRF_{Z_p}^{z}}(1^\lambda, A) + p \cdot Adv_{PRF_{Z_p}^{z}}(1^\lambda, A) \). Therefore, the ideal world Ideal_{eq,S,Z}(1^\lambda) and the real world Real_{eq,z,A,Z}(1^\lambda) are indistinguishable with advantage \( \epsilon = \epsilon_0 + \epsilon_1 \).

Case 2: \( P_1 \) is corrupted. We construct the simulator \( S \) which internally runs \( A \), forwarding messages to/from \( Z \) and simulates the interface of honest \( P_0 \).

- Upon receiving (Input, sid) from \( F_{eq} \), \( S \) starts simulation.
  - For the simulation of \( \Pi_{convert} \), \( i \in [n] \),
    - \( S \) picks random \([r_i]_1^2 \in Z_2 \), \([s_i]_p \in Z_p \) and emulates \( F_{(1,2).OT} \) with input \( m_0 = [s_i]_0 - [r_i]_0 \), \( m_1 = [s_i]_0 - ([r_i]_0 - 1) \);
    - When corrupted \( P_1 \) inputs \([r_i]_1^2 \) to \( F_{(1,2).OT} \), \( S \) records \([r_i]_1^2 \) and sends \([r_i]_1^2 \) to \( P_1 \).
    - \( S \) calculates \( t_i \) with \( m_i \).
    - \( S \) picks \([w_i]_0 \in Z_2 \) and acts as \( P_0 \) to send it to \( P_1 \).
    - Upon receiving \([w_i]_1^2 \) from \( P_1 \), \( S \) calculates \( w_i = [w_i]_0 \oplus [w_i]_1^2 \) and \( v_i = t_i - 2w_i \).
  - \( S \) calculate \( d_1 \) = \( \sum_{i=0}^{n-1} v_i \).
  - For the simulation of \( \Pi_{eq_i} \),
    - \( S \) picks \( \varepsilon_1 \in Z_p \) and \( m_i \in Z_2^p \) for \( i \in [p]\{\varepsilon_1\} \), and acts as \( F_{(N-1,N).ROT} \) to send them to \( P_1 \).
    - \( S \) generate the binary matrix \( M \) by using the \{\( m_i \)\}_{i\in[p]\{\varepsilon_1\}} \) as the binary column vectors, and left cycle shift the \( i \)th row of \( M \) by \( i \) positions locally for \( i \in [p] \).
    - \( S \) computes \( w_i = v_i \oplus u_{\varepsilon_1+i} \) to generate \( \hat{W} \), where \( v_i = (\bigoplus_{j=0}^{p-1} m_{(i,j)}) \oplus (\bigoplus_{j=\varepsilon_1} m_{(i,j)}) \) and \( u_i = (\bigoplus_{j=0}^{p-1} m_{(i,j)}) \oplus (\bigoplus_{j=\varepsilon_1} m_{(i,j)}) \).
    - \( S \) picks \( \hat{S}_i \in Z_2^p \) and acts as \( P_0 \) to send it to \( P_1 \).
    - \( S \) computes \( \hat{T}_1 := \text{shift}(\hat{S}_i, \varepsilon_1) \oplus \hat{W} \).
    - Upon receiving (Output, \([e]_1\)) from \( F_{eq} \), \( S \) picks a random \( p \) satisfying \([t_i]_1 = [e]_1 \).
    - \( S \) computes \( w_0 = \rho - (\varepsilon_1 + d_1) \) and acts as \( P_0 \) to send it to \( P_1 \).

Claim 2. If PRF_{Z_p}^{z}, PRF_{Z_p}^{z_p} and PRF_{Z_p}^{z_p} are the secure pseudorandom functions with adversarial advantage Adv_{PRF_{Z_p}^{z}}(1^\lambda, A), Adv_{PRF_{Z_p}^{z}}(1^\lambda, A), then the ideal world Ideal_{eq,S,Z}(1^\lambda) and the real world Real_{eq,z,A,Z}(1^\lambda) are indistinguishable with advantage \( \epsilon = n \cdot (Adv_{PRF_{Z_p}^{z}}(1^\lambda, A)) + (n+1) \cdot Adv_{PRF_{Z_p}^{z}}(1^\lambda, A) + Adv_{PRF_{Z_p}^{z}}(1^\lambda, A) \).

Proof. In the ideal world for simulating \( \Pi_{convert} \), \([s_i]_1^2 \) is picked random rather than calculated by \( [s_i]_0 - ([r_i]_0 \oplus [r_i]_1^2) \).
Therefore, the advantage in this step is \( \epsilon_2 = n \cdot Adv_{PRF_{Z_p}^{z}}(1^\lambda, A) \) and the \( \epsilon_0 \) is the same as in case 1. In additional,
For the integers \( a \) held by \( P_0 \) and \( b \) held by \( P_1 \), the result of comparison \( 1 \{ a > b \} \) can be obtained by bitwise comparing \( a \) and \( b \) from the big-endian. Formally, it is denoted by \( 1 \{ a > b \} = a_\rho \), where the position \( \rho \) correspond to the first different bit between \( a \) and \( b \). Observe that in the case \( a = b \) of which there is no different bits between \( a \) and \( b \), we append 1 to the end of \( b \) and 0 to the end of \( a \) ( In contrast, we append 1 to \( a \) and 0 to \( b \) for \( 1 \{ a \geq b \} \)).

Fig. 9 illustrates the overview of our secure comparison protocol. In the first step, we locate the position \( \rho \). In the second step, we design a protocol to make two parties securely obtain the corresponding bit \( a_\rho \) which implies the comparison result.

**First different bit detection.** At first step, we view \( a \) and \( b \) as the bitwise-XOR share of \( m \). Namely, \( m_i = a_i \oplus b_i \) for \( i \in [n] \). The position \( \rho \) corresponds to the first non-zero bit of \( m \). We introduce a transformation \( \{ s_i \}_{i \in [n]} = \)

\[
\text{Algorithm 2: \text{ShortListZeroCheck}}
\]

- **Input:** \( (a, b, n) \)
- **Execution:**
  - 1. Compute \( m = a \oplus b \)
  - 2. If \( m = 0 \), return \( \rho = 0 \)
  - 3. Otherwise, \( \rho \) is the first non-zero bit of \( m \)

This concludes the proof.

### IV. Secure Comparison

In this section, we propose a novel secure comparison protocol where \( P_0 \) inputs \( a \) and \( P_1 \) inputs \( b \), receiving the shared result \( 1 \{ a > b \} \). We first give an overview of our protocol which is constructed by a new primitive – oblivious short-list zero check (OZC). Then we propose a two-round OZC protocol as the build block.

**A. Protocol Overview**

For the integers \( a \) held by \( P_0 \) and \( b \) held by \( P_1 \), the result of comparison \( 1 \{ a > b \} \) can be obtained by bitwise comparing \( a \) and \( b \) from the big-endian. Formally, it is denoted by \( 1 \{ a > b \} = a_\rho \), where the position \( \rho \) correspond to the first different bit between \( a \) and \( b \). Observe that in the case \( a = b \) of which there is no different bits between \( a \) and \( b \), we append 1 to the end of \( b \) and 0 to the end of \( a \) ( In contrast, we append 1 to \( a \) and 0 to \( b \) for \( 1 \{ a \geq b \} \)).

Fig. 9 illustrates the overview of our secure comparison protocol. In the first step, we locate the position \( \rho \). In the second step, we design a protocol to make two parties securely obtain the corresponding bit \( a_\rho \) which implies the comparison result.

**First different bit detection.** At first step, we view \( a \) and \( b \) as the bitwise-XOR share of \( m \). Namely, \( m_i = a_i \oplus b_i \) for \( i \in [n] \). The position \( \rho \) corresponds to the first non-zero bit of \( m \). We introduce a transformation \( \{ s_i \}_{i \in [n]} = \)
Let \( P \) be the prefix sum of \( m_i \). Specifically, \( s'_i := \sum_{j=0}^{i-1} m_j \) for \( i \in [n] \). We define \( s_i := \phi(m_i) := s'_i - 2m_i + 1 \). Obviously, when \( i < \rho \), it holds that \( m_i = s'_i = 0 \), therefore, we have \( s_i = 1 \); when \( i = \rho \), it holds that \( s'_i = m_i = 1 \), therefore, \( s_i = 0 \); when \( i > \rho \), it holds that \( s'_i = m_i + 1 \), therefore, \( s_i = 2 - m_i \geq 1 \). In general, \( s_i = 0 \) if and only if \( i = \rho \). For instance, if \( a = 10010 \) and \( b = 10101 \), we have \( m = a \oplus b = 00111 \), and then \( s' = 00123 \) and \( s = 11012 \). Analogously, it holds that \( s_i \leq n \) (The maximum \( s_i \) takes \( n \) when \( s'_{n-1} = n - 1 \) and \( m_i = 0 \)). To avoid extra 0 caused by wrapping round, \( \phi \) should be performed on \( \mathbb{Z}_p \) where \( p > n \), w.r.t. \([m_i]^p\) instead of \([m_i]^2\). We apply the sharing conversion protocol \( \Pi_{conver} \) in Sec. [III] to expand \([m_i]^2 \in \mathbb{Z}_2\) to \([m_i]^p \in \mathbb{Z}_p\).

Now we have shared list \([s_i]^p \in \mathbb{Z}^{|n|}\), where its zero element position \( \rho \) corresponds to the comparison result of \( a \) and \( b \), this is, \( a_\rho = 1 \{ a > b \} \). The second challenge is how \( P_0 \) and \( P_1 \) can obliviously obtain \([a_\rho]\) from \([s_i]^p \in \mathbb{Z}^{|n|}\) and \( a \). To address this issue, we introduce a new primitive – Oblivious Short-list Zero Check (OZC).

**Oblivious Short-listed Zero Check.** The OZC scheme checks if a shared list contains zero on a subsequence. We formalize its functionality in Fig. [14] In particular, an OZC scheme allows \( P_0 \) input \( k \)-dimension selective index set \( I := \{\zeta_0, \ldots, \zeta_{k-1}\} \), \( P_0 \) and \( P_1 \) input shared list \([x_i]_i \in [n]\). For \( x_i = [x_i][0] + [x_i][1] \), it checks if \([x_{\zeta_i}]_i \in [k]\) contains zero and sends the check result to \( P_1 \).

We construct our secure comparison protocol with OZC. At a high level, we let \( P_0 \) toss a coin \( \Delta \in \{0,1\} \) and input all the position \( \{\zeta_i \}_i \in \{k\} \), where \( a_{\zeta_i} = \Delta \), as the indices of \( \mathcal{F}_{ozc} \) (We assume there are \( k \) bits in \( a \) equal to \( \Delta \)). \( P_0 \) and \( P_1 \) input aforementioned \([s_i]^p \in \mathbb{Z}^{|n|}\) as the shared list of \( \mathcal{F}_{ozc} \). \( P_1 \) will receive the zero check result \( z \). For the case \( z = 0 \), it indicates that all the bits of \( a_{\zeta_i} = \Delta \) do not lay on the position \( \rho \) for \( s_\rho = 0 \), which implies \( a_\rho = \Delta \oplus 1 \). For the case \( z = 1 \), \( P_0 \) successfully guesses the correct result \( a_\rho = \Delta \). Obviously, it holds that \( a_\rho = \Delta \oplus z \oplus 1 \). We let \( P_0 \) output the result \([c][0] = \Delta \) and \( P_1 \) output \([c][1] = z \oplus 1 \).

**Dummy queries.** The number of queries \( k \) will leak the hamming weight of \( a \) to \( P_1 \). To avoid this leakage, we introduce dummy queries which pad the overall queries to the maximum possible number of queries. Firstly, we let \( P_0 \) and \( P_1 \) generate non-zero share \([s_n]^p \). We let \( P_0 \) perform extra \( n - k \) queries using index \( n \). Namely, for \( i \in \{k, \ldots, n-1\} \), \( P_0 \) sets \( \zeta_i = n \) and all parties invoke \( \mathcal{F}_{ozc} \) with \( n \) dimention indices and \( (n + 1) \) dimension shared list \([s_i]^p \in \mathbb{Z}^{|n+1|}\). Consequently, the overall queries are \( n \).

**Protocol description.** The full description of our secure comparison protocol is depicted in Figure [8] Next, we explain our protocol step by step as follows.

- At step 1, \( P_0 \) and \( P_1 \) invoke \( \Pi_{conver}^{(a_i,b_i)} \) for each bit \( a_i \) and \( b_i \), receiving \([m_i][0]\) and \([m_i][1]\) respectively, such that \([m_i][0] + [m_i][1] = a_i \oplus b_i \).
- At step 2, \( P_0 \) and \( P_1 \) append 0 to \( a \) and 1 to \( b \) for dealing with \( a = b \).
- At steps 3, \( P_0 \) and \( P_1 \) compute \([s_i][0] = \sum_{j=0}^{i-1} x_j - 2x_i + 1 \) and \([s_i][1] = \sum_{j=0}^{i} y_j - 2y_i + 1 \), respectively. It holds that \( s_\rho = 0 \), where \( \rho \) denotes the position of the first differing bit between \( a \) and \( b \).
- At steps 4, \( P_0 \) and \( P_1 \) sets \([s_{n+1}][0] = [s_{n+1}][1] = 1 \) for dummy queries.
Input: $P_0$ inputs $a \in \mathbb{Z}_{2^n}$; $P_1$ inputs $b \in \mathbb{Z}_{2^n}$.

Output: $P_0$ receives $[c]_0^2 \in \mathbb{Z}_2$; $P_1$ receives $[c]_1^2 \in \mathbb{Z}_2$; it holds that $[c]_0^2 \oplus [c]_1^2 = 1 \{a < b\}$.

Execution:
1) Let $p := \lceil \log(n) \rceil$, for $i \in [n]$, $P_0$ and $P_1$ invoke $[m_i] \leftarrow \Pi_{\text{convex}}^p(a_i, b_i)$, and then $P_j$ holds $[m_i] \in \mathbb{Z}_p$, $P_1$ holds $y_i \in \mathbb{Z}_p$.
2) $P_0$ sets $a_n = [m_n]_0 = 0$; $P_1$ sets $b_n = [m_n]_1 = 1$.
3) For $i \in [n + 1]$, $P_0$ computes $[s_i]_0$, where $[s_i]_0 = \sum_{j=0}^{i}[m_j]_0 - 2 \cdot [m_i]_0 + 1$, and $P_1$ computes $[s_i]_1$, where $[s_i]_1 = \sum_{j=0}^{i}[m_j]_1 - 2 \cdot [m_i]_1 + 1$.
4) $P_0$ and $P_1$ sets $[s_{n+1}]_0 = [s_{n+1}]_1 = 1$.
5) $P_0$ picks $\Delta \leftarrow \{0, 1\}$.
6) $P_0$ sets $I := \{\zeta_j\}_{j \in \mathbb{Z}_2} = \{i|a_i = \Delta, i \in \mathbb{Z}_{n+1}\}$, where we assume the size of $I$ is $k$.
7) For $j \in \{k, \ldots, n\}$, $P_0$ appends $\zeta_n = n$ to get $n + 1$-dimension vector $\mathcal{I}^*$.  
8) $P_0$ and $P_1$ invoke $F_{\text{fozc}}[n+1, n+2]$ with index list $\mathcal{I}$, shared list $\{[s_i]_0\}_{i \in [n+2]}$ and $\{[s_i]_1\}_{i \in [n+2]}$; $P_1$ receives $z \in \{0, 1\}$.
9) $P_1$ sets $[c]_1^2 = z \oplus 1$.
10) $P_0$ set $[c]_0^2 = \Delta$.

Fig. 8: The Comparison Protocol

Fig. 9: The Overview of Secure Comparison

- At step 5-6, $P_0$ picks random $\Delta$, records all indices $i$ where $a_i = \Delta$, and denotes the set of these indices as $I$. We assume the size of the set $I$ is $k$, namely, $I = \{\zeta_j\}_{j \in \mathbb{Z}_k}$.
- At step 7, to prevent the leakage of the hamming weight of $a$, $P_0$ pads the size of $I$ to $n + 1$. Therefore, $P_0$
ppends $\zeta_j = n + 1$ for $j \in n + 1$.

- At step 8, $P_0$ and $P_1$ invoke $F_{ozc}$. Specifically, $P_0$ inputs the index list $I = \{\zeta_j\}_{j \in [n+1]}$ and the shared list $\{(s_i)_0\}_{i \in [n+1]}$, and $P_1$ inputs the shared list $\{(s_i)_0\}_{i \in [n+2]}$. After the protocol, $P_1$ receives $z = 1 \{0 \in \{s_{G_0}, \ldots, s_{Z_{k-1}}\}$.

- At steps 9-10, $P_1$ sets output $[c]_1 = z \oplus 1$ and $P_0$ set output $[c]_0 = \Delta$.

Our secure comparison protocol $\Pi_{comp}^{n}$ requires 1-round communication of $2n$ bits in the online phase for the $n$ times invoking of $\Pi_{convert}$ and one time $F_{ozc}[n + 1, n + 2]$.

**Security.** We define the functionality $F_{cmp}$ for secure comparison as an instance of $F_{2PC}$, where $F_{cmp}$ receives $a$ and $[c]_0^2 \in \{0, 1\}$ from honest $P_0$ or $S$, receives $b$ from honest $P_1$ or $S$, calculates $[c]_1^2 = 1\{a > b\} \oplus [c]_0^2$ and sends to $P_1$. Next, we prove our protocol $\Pi_{comp}$ realizes functionality $F_{cmp}$.

**Theorem 2.** The protocol $\Pi_{comp}$ as depicted in Fig. 8 UC realizes $F_{cmp}$ in the $(F_{1,2}^\cdot OT, F_{ozc})$-hybrid model against semi-honest PPT adversaries with statical curroption.

**Proof.** To prove Thm. 2 we construct a PPT simulator $S$, such that no non-uniform PPT environment $Z$ can distinguish between the ideal world $\text{Ideal}_{F_{cmp}, S, z}(1^\lambda)$ and the real world $\text{Real}_{\Pi_{comp}, A, Z}(1^\lambda)$. We consider the following cases:

- **Case 1:** $P_0$ is corrupted. We construct the simulator $S$ which internally runs $A$, forwarding messages to/from $Z$ and simulates the interface of honest $P_1$.
  - Upon receiving $(\text{Input}, \text{sid})$ from $F_{cmp}$, $S$ starts simulation.
  - For the simulation of $i^{th}$ times of $\Pi_{convert}$, $i \in [n]$,
    - $S$ picks random $[r_i]^2 \in \mathbb{Z}_2$ and emulates $F_{(1,2)\cdot OT}$ with input $[r_i]^2$;
    - When corrupted $P_0$ inputs $(m_{0,i}, m_{1,i})$ to $F_{(1,2)\cdot OT}$, $S$ records $(m_{0,i}, m_{1,i})$.
    - $S$ calculates $r_i$ and $s_i$ with $m_{0,i}, m_{1,i}$;
    - $S$ picks $[w_i]^2 \in \mathbb{Z}_2$ and acts as $P_1$ to send it to $P_0$.
    - Upon receiving $[w_i]^2$ from $P_0$, $S$ calculate $a_i = [w_i]^2 \oplus [r_i]^2$
  - $S$ emulates $F_{ozc}$ with random list $\{(s_i)_i\}_{i \in \mathbb{Z}_{n+1}}$.
  - When $P_0$ input $I$ to $F_{ozc}$, $S$ records $I$ and calculates $\Delta := a_i$ for $i \in I$ and $i \neq n + 1$.
  - If $a = 0$ or $a = 2^n - 1$ and $I := \{n + 1, \ldots, n + 1\}$, set $\Delta = 1 \oplus a_0$.
  - $S$ sends $(\text{Input}, \text{sid}, a, \Delta)$ to external $F_{cmp}$.

Observe that $P_0$ locally set $[c]_0 = \Delta$, so that the output of ideal execution keeps consistent with the real execution. We show that the incoming message of $P_0$ in the ideal world is indistinguishable from the real world.

**Claim 3.** If $\text{PRF}_{\mathbb{Z}_2}$ is the secure pseudorandom functions with adversarial advantage $\text{Adv}_{\text{PRF}_{\mathbb{Z}_2}}(1^\lambda, A)$, then the ideal world $\text{Ideal}_{F_{cmp}, S, z}(1^\lambda)$ and the real world $\text{Real}_{\Pi_{comp}, A, Z}(1^\lambda$) are indistinguishable with advantage $\varepsilon = \text{Adv}_{\text{PRF}_{\mathbb{Z}_2}}(1^\lambda, A)$.

**Proof.** In the ideal world, $[w_i]^2$ are picked random rather than calculated by $[a]^2_i \oplus [r_i]^2$, which replace $n$ $\text{PRF}_{\mathbb{Z}_2}$ outputs to uniformly random; therefore, the overall advantage is $\varepsilon = n \cdot \text{Adv}_{\text{PRF}_{\mathbb{Z}_2}}(1^\lambda, A)$.\qed
Case 2: $P_1$ is corrupted. We construct the simulator $S$ to simulates the interface of honest $P_0$

- Upon receiving $(\text{Input}, \text{sid})$ from $F_{\text{emp}}$, $S$ picks $a$.
- For the simulation of $i$th times of $\Pi_{\text{convert}}, i \in [n],$
  - $S$ picks random $[r_i]^2 \in \mathbb{Z}_2$, $[s_i]^p \in \mathbb{Z}_p$ and emulates $F_{(1,2), \text{OT}}$ with input $m_0 = [s_i]^p - [r_i]^2, m_1 = [s_i]^p - (1 - [r_i]^2)$;
  - When corrupted $P_0$ inputs $[r_i]^2$ to $F_{(1,2), \text{OT}}, S$ records $[r_i]^2$ and sends $m_{[r_i]^2}$ to $P_1$.
  - Upon receiving $[w_i]^2$ from $P_1$, $S$ calculate $b_i = [w_i]^2 \oplus [r_i]^2$
  - $S$ calculates $[w_i]^2 = [r_i]^2 \oplus a_i$ and acts as $P_1$ to send it to $P_0$.
- For simulation of $\Pi_{\text{comp}},$
  - $S$ calculates $\{s_i\}_{i \in [n+1]}$ with $\phi((a||0) \oplus (b||1)), it holds that \( s_\rho = 0 \) and $\rho < n + 1$.
  - Upon receiving $[c]^2$ from $F_{\text{emp}}, S$ does:
    * if $[c]^2 = 1$, set $\mathcal{I} := \{n + 1, n + 1, \ldots, n + n, n + 1\}$ with $n + 1$ dimension.
    * if $[c]^2 = 0$, set $\mathcal{I} := \{n + 1, n + 1, \ldots, n + n, n, \rho\}$ with $n + 1$ dimension.
  - $S$ emulates $F_{\text{oxc}}$ with random list $\{s_i\}_{i \in [n+2]}$ and selection list $\mathcal{I}$

We first show that in the ideal world, $P_1$ receives same output as the real world: If $\mathcal{I} := \{n + 1, n + 1, \ldots, n + 1, n + 1\}, F_{\text{oxc}}$ will output all positive value $\{\beta_i \cdot s_{n+1}\}_{i \in [n/2]}$ to $P_1$ induce $P_1$ to output $[c]^2 = 1$. On the contrary, if $\mathcal{I} := \{n + 1, n + 1, \ldots, n + n, n, \rho\}, F_{\text{oxc}}$ will output zero-contained list to $P_1$ such that $P_1$ output $[c]^2 = 0$. Next, we show that the incoming message of $P_1$ in the ideal world is indistinguishable with the real world.

Claim 4. For two sets of list ($\mathcal{I}, X, Y$) and ($\mathcal{I}', X', Y'$), where

- $\mathcal{I} := \{\zeta_i\}_{i \in [k]} \in \mathbb{Z}_n^k, \mathcal{I}' := \{\zeta'_i\}_{i \in [k]} \in \mathbb{Z}_n^k$;
- $X := \{x_i\}_{i \in [n]} \in \mathbb{Z}_p^n, X' := \{x'_i\}_{i \in [n]} \in \mathbb{Z}_p^n$;
- $Y := \{y_i\}_{i \in [n]} \in \mathbb{Z}_p^n, Y' := \{y'_i\}_{i \in [n]} \in \mathbb{Z}_p^n$;

If it have
- $\mathcal{M} := \{x_i + y_i\}_{i \in [n]}$ and $\mathcal{M}' := \{x'_i + y'_i\}_{i \in [n]}$ only contains one 0 (denoted by $m_0$ and $m'_0$), and contains non-zero value in the other position.
- The number of $\rho$ contained in $\mathcal{I}$ and $\mathcal{I}'$ are both $\ell \in \{0, 1\}$.

It holds that

$$\Pr[A(F_{\text{oxc}}(X_0, X_B, Y_B), \{\mathcal{I}, X, Y\}_{i \in [2]}) = b] < \frac{1}{2} + \text{negl}$$

Claim 5. If $\text{PRF}_{\text{Z}_2}^{1, \lambda}$ is the secure pseudorandom functions with adversarial advantage $\text{Adv}_{\text{PRF}_{\text{Z}_2}^{1, \lambda}}$, then the ideal world $\text{Ideal}_{F_{\text{emp}}, \text{Z}_2}^{1, \lambda}$ and the real world $\text{Real}_{\Pi_{\text{emp}}, \text{Z}_2}^{1, \lambda}$ are indistinguishable with advantage $\epsilon = \text{Adv}_{\text{PRF}_{\text{Z}_2}^{1, \lambda}}$.

Proof. In the ideal world, $[w_i]^2$ are calculated by randomly picked $a$, which replace random value to $n$ to output of where the advantage is $\text{Adv}_{\text{PRF}_{\text{Z}_2}^{1, \lambda}}$. For the list $\{z_i\}_{i \in [n+1]}$, from Claim 4, it is indistinguishable between the ideal world and real world.

This concludes the proof.
Protocol $\Pi_{\text{oszc}}^{k,n,p}(\mathcal{I}, X, Y)$

**Input:** Index list $\mathcal{I} := \{\xi_i\}_{i \in [k]}$ input by $P_0$, which contains $k - t$ non-repeating items, and last $t$ indices equal to $n$; list $X := \{x_i\}_{i \in [n]}$ input by $P_0$; list $Y := \{y_i\}_{i \in [n]}$ input by $P_1$.

**Output:** $P_1$ receives $z_i = (x_{\xi_i} + y_{\xi_i}) \cdot \beta_{\xi_i}$ for the random value $\beta_{\xi_i}$, which is unknown to $P_1$.

**Offline:**
- $P_0$ and $P_1$ invoke:
  - $(\beta_i, r_i, u_i, v_i) \leftarrow \mathcal{F}_{\text{ake}}(p)$, for $i \in [n]$.
  - $(\beta_j)_{j \in [k-1]}, r, \{u_j\}_{j \in [k-1]}, \{v_j\}_{j \in [k-1]} \leftarrow \mathcal{F}_{\text{ake}}(p, k-1)$
- $P_1$ concatenates $(\beta_j)_{j \in [k-1]}, r, \{u_j\}_{j \in [k-1]}, \{v_j\}_{j \in [k-1]}$ with $\beta_i, r_i, u_i, v_i$ where copy $k - 2$ copies of $r$ as alignment.
- $P_0$ picks random permutation $\pi : S_{n+k-1} \mapsto S_{n+k-1}$.
- $P_0$ and $P_1$ invoke $\mathcal{F}_{\text{permute}}$:
  - $P_0$ inputs the permutation $\pi$, and $P_1$ inputs the list $\{v_i\}_{i \in [n+k-1]}$.
  - $P_0$ receives the sharing list $\{[w_{\pi(i)}]_0\}_{i \in [n+k-1]}$ and $P_1$ receives $\{[w_{\pi(i)}]_1\}_{i \in [n+k-1]}$, respectively.
- $P_0$ sets $[w_i]_0 = [v_i]_0 + u_{\pi(i)}$; $P_1$ sets $[w_i]_1 = [v_i]_1$.

**Online:**
- $P_1$ sets $y_i' = y_i + r_i$ for $i \in [n]$ and sends the list $Y' := \{y_0', \ldots, y_n'\}$ to $P_0$.
- $P_0$ sets
  - $t_i = \beta_{\xi_i} \cdot (x_{\xi_i} + y_{\xi_i}') - [w_{\pi(-\xi_i)}]_0$ for $i \in [k-t]$;
  - $s_i = \pi^{-1}(\xi_i)$ for $i \in [k-t]$;
  - $t_i = \beta_{n+i-k} \cdot (x_{n+i} + y_{n+i}') - [w_{\pi(-(n+i-k))}]_0$ for $i \in [k-t, k]$;
  - $s_i = \pi^{-1}(n+i-k)$ for $i \in [k-t, k]$;
- $P_0$ sends $\{t_i\}_{i \in [k]}$ and $\{s_i\}_{i \in [k]}$ to $P_1$.
- $P_1$ calculates $z_i = t_i - [w_{s_i}]$ for $i \in [k-t]$.
- $P_1$ outputs $z = 1 \{0 \in \{z_0, \ldots, z_{k-1}\}$.

**Fig. 10:** The Oblivious Selective Multiplication Protocol

**Fig. 11:** The running time of equality testing protocol $\Pi_{\text{oszc}}$ compare with ABY [19], GC scheme implemented in EMP-toolkits [48] and DPF [26] in LAN/MAN/WAN setting. All benchmarks take the data length $n = 32$. 

![Graphs showing running time comparison](image-url)
B. Realize $\mathcal{F}_{ozc}$

We propose a naive construction of the OZC protocol, which requires heavy communication in the online phase. After that, we optimize the communication of the online phase by introducing the permutation tuples in the offline phase.

**OLE-based implement.** Recall the functionality $\mathcal{F}_{ozc}$ accepts list $X := \{x_0, \ldots, x_{n-1}\}$, $Y := \{y_0, \ldots, y_{n-1}\}$ and a index list $\mathcal{I} := \{i_0, \ldots, i_{k-1}\}$. $\mathcal{F}_{ozc}$ sends the information of whether there exists $x_i + y_i = 0$ for $i \in \mathcal{I}$ to $P_1$. The naive approach to implementing OZC is to scale all selected items $x_{i_\xi} + y_{i_\xi}$ with non-zero random value $\beta_i$ and directly reveal to $P_1$, namely, $c_i = \beta_i \cdot (x_{i_\xi} + y_{i_\xi})$. $P_1$ check whether there exist $c_i = 0$ for $i \in [k]$ to verify $x_i + y_i = 0$. To hide the index $i_\xi$, we let $P_0$ first take $y_{i_\xi}$ using $(1, n)$-OT, then $P_0$ picks $\beta_i$ and calculates $c_i = \beta_i \cdot (x_{i_\xi} + y_{i_\xi})$. Avoiding reveal $y_{i_\xi}$ to $P_0$, we employ $P_1$ generate $r$ to mask $y_{i_\xi}$ and rewrite $z$ as $[z] = \beta \cdot (x_{i_\xi} + y_{i_\xi} + r) - [\beta \cdot r]$. $P_0$ takes $y'_{i_\xi} = y_{i_\xi} + r$ from OT instead of $y_{i_\xi}$. For the part of $[\beta \cdot r]$, it can be produced by OLE tuple generation protocol with random $\beta \in \mathbb{Z}_p$ and $r \in \mathbb{Z}_p$, where $P_0$ holds $\{\beta, [\beta \cdot r]\}_0$ and $P_1$ holds $\{r, [\beta \cdot r]\}_1$. At present, $P_1$ can locally calculate $[c_i]_0 = \beta_i \cdot (x_{i_\xi} + y'_{i_\xi}) - [\beta \cdot r]_0$ and $P_1$ calulate $[c_i]_1 = -[\beta \cdot r]_1$. When $P_0$ reveal $[c_i]_0$ to $P_1$ for reconstruction $c_i$. The naive approach is illustrated in Fig. 15 (CF. Appendix. A-C).

**Remark.** To avoid the 0 caused by wrapping round $\beta_i \cdot (x_i + y_i)$ with non-zero $x_i + y_i$, $\beta_i$ and $p$ should be coprime. We exclude such cases by taking $p$ as prime and $\beta_i \in \mathbb{Z}_p^*$.

**Online Phase Communication Optimization.** For $k$ indices, $\Pi_{ozc}$ requires invoking $k$ times 1-out-of-$n$ OT in the online phase, which is a huge communication cost. We optimize the online phase communication through the oblivious permutation. Our starting point is that $y'_{i_\xi}$ in $\Pi_{ozc}$ can be masked with different $r_i$ and directly reveal to $P_0$. Instead of OT, $P_0$ can directly select $y'_{i_\xi}$ and calculate $\beta_{i_\xi}(x_{i_\xi} + y_{i_\xi}) = \beta_{i_\xi}(x_{i_\xi} + y_{i_\xi} + r_{i_\xi})$. The challenge is how to cancel $\beta_{i_\xi} \cdot r_{i_\xi}$ when $P_1$ doesn’t know $i_\xi$. We introduce permutation tuples to address this issue. In particular, the permutation tuple $(\{\beta_i, r_i, [w_i]_0, [w_i]_1\}_{i \in [n]}, \pi)$ is generated in the offline phase, where it holds that,

- $\pi$ is a random permutation held by $P_0$ (we use $\pi(i)$ to denote the permuted result of $i$);
- $\beta_i \cdot r_i = [w_{\pi(i)}]_0 + [w_{\pi(i)}]_1$ are the permuted OLE tuples, where $P_0$ holds $(\{\beta_i, [w_i]_0\}_{i \in [n]})$ and $P_1$ holds $(\{r_i, [w_i]_1\}_{i \in [n]})$.

Considering $z_i = \beta_{i_\xi}(x_{i_\xi} + y_{i_\xi} + r_{i_\xi}) - \beta_{i_\xi} \cdot r_{i_\xi}$, we can replace $\beta_{i_\xi} \cdot r_{i_\xi}$ with $[w_{\pi(i)}]_0 + [w_{\pi(i)}]_1$. Namely, $z_i = \beta_{i_\xi}(x_{i_\xi} + y_{i_\xi} + r_{i_\xi}) - [w_{\pi(i)}]_0 - [w_{\pi(i)}]_1$. $P_0$ hold $\beta_{i_\xi}, x_{i_\xi}, y'_{i_\xi} = y_{i_\xi} + r_{i_\xi}$, $\pi$ and $[w_{\pi(i)}]_0$ so that it can calculate $t_i = \beta_{i_\xi}(x_{i_\xi} + y_{i_\xi} + r_{i_\xi}) - [w_{\pi(i)}]_0$. Since $\pi$ is a uniformly random permutation, $\pi(i)$ can be revealed to $P_1$ directly without information leakage about $i_\xi$. Consequently, $P_1$ calculates $z_i = t_i - [w_{\pi(i)}]_1$ which is equal to $\beta_{i_\xi}(x_{i_\xi} + y_{i_\xi})$ and checks if there exists $z_i = 0$ for $i \in [k]$.

**Privacy on dummy queries.** The foregoing version of the protocol can not deal with the duplicated indices. Because the same index $i_\xi$ will obtain the same permuted index $\pi(i_\xi)$ which can not be directly revealed to $P_1$, leading to an incompatible with the original dummy queries approach. Our solution is to generate another $k - 1$ dimension VOLE permutation tuple $(\{\beta_i, [w_i]_0, [w_i]_1\}_{i \in [k-1]}, r)$, where it holds

- $\beta_i \cdot r = [w_{\pi(i)}]_0 + [w_{\pi(i)}]_1$ are the permuted VOLE tuples;
The VOLE tuple is concatenated with the original OLE tuples and the \( \pi : \mathbb{Z}_p^{n+k-1} \rightarrow \mathbb{Z}_p^{n+k-1} \) is performed on the overall tuples, namely, \( (\{\beta_i, r_i, [w_i]_0, [w_i]_1\}_{i \in \{n+k-1\}}, \pi) \) where \( r_n = r_{n+1} \ldots = r_{n+k} \) corresponds to the \( r \) of VOLE tuple. We utilize VOLE tuples to deal with the duplicated indices. In particular, assume the last \( t \) items of \( \mathcal{T} \) is duplicated indices, i.e. \( \zeta_i = \eta \) for \( i \in [k-t, k] \). \( P_0 \) sets \( t_i = \beta_{n+i-k+t} \cdot (x_\eta + y_\eta + r_\eta) - [w_{\pi(n+i-k+t)}]_0 \) and sends \( t_i \) and \( \pi(n+i-k+t) \) to \( P_0 \). Analogously, \( P_1 \) can recover \( z_i = \beta_{n+i-k+t} \cdot (x_\eta + y_\eta) \) for the duplicated index.

**Offline tuples generation.** We generate the offline tuples with three primitives: \( \mathcal{F}_{ole}, \mathcal{F}_{vole}, \mathcal{F}_{permute} \). We set \( \mathcal{F}_{ole} \) and \( \mathcal{F}_{vole} \) generate the OLE tuples and VOLE tuple for dummy queries, denote them as \( \{\beta_i, r_i, u_i, v_i\} \) where \( \beta_i \cdot r_i = u_i + v_i \). We let \( P_0 \) input random permutation \( \pi \) and \( P_1 \) input list \( \{v_i\}_{i \in [n]} \) to functionality \( \mathcal{F}_{permute} \). After that \( P_0 \) and \( P_1 \) receive \([v_{\pi(i)}]\) and calculate \([w_i] = [v_{\pi(i)}] + [u_{\pi(i)}]\). Now we have \( \beta_i \cdot r_i = [w_{\pi(i)}]_0 + [w_{\pi(i)}]_1 \) for \( i \in [n] \). In our benchmark, we use the SOTA protocol to realize \( \mathcal{F}_{ole}[33], \mathcal{F}_{vole}[44], \mathcal{F}_{permute}[15] \).

Our complete protocol design is illustrated in Figure. [10]. Our oblivious short-list zero check protocol \( \Pi_{\text{ozc}}^{k,n,p} \) requires 2-round communication of \( 2 \cdot k \cdot p \) bits in the online phase. In the offline phase, it requires \( n \) times invoking of \( \mathcal{F}_{ole}[p] \), one time invoking of \( \mathcal{F}_{vole}[k-1,p] \) and one time invoking of \( \mathcal{F}_{permute}[n+k-1] \).

**Theorem 3.** The protocol \( \Pi_{\text{ozc}} \) as depicted in Fig. [10] UC realizes \( \mathcal{F}_{ozc} \) against semi-honest PPT adversaries who can statically corrupt up to one party.

**Proof.** To prove Thm. [3] we construct a PPT simulator \( S \), such that no non-uniform PPT environment \( Z \) can distinguish between the ideal world \( \text{Ideal}_{\text{ozc}, S, Z}(1^\lambda) \) and the real world \( \text{Real}_{\Pi_{\text{ozc}}, \mathcal{A}, Z}(1^\lambda) \). We consider the following cases:

**Case 1:** \( P_0 \) is corrupted. We construct the simulator \( S \) which internally runs \( \mathcal{A} \), forwarding messages to/from \( Z \) and simulates the interface of honest \( P_1 \).

- \( S \) emulates \( \mathcal{F}_{ole} \), outputs \( \{\beta_i, r_i, u_i, v_i\} \) for \( i \in [n] \) and sends \( \{\beta_i, u_i\} \) to \( P_0 \).
- \( S \) emulates \( \mathcal{F}_{vole} \), outputs \( \{\{\beta_j, r, u_j, v_j\}_{j \in [k-1]}\} \) and sends \( \{\beta_j, u_j\} \) to \( P_0 \).
- \( S \) emulates \( \mathcal{F}_{permute} \) with input \( v_i \) and record \( \pi \).
- \( S \) picks random list \( \{y_i\}_{i \in [n]} \) and acts as \( P_1 \) to send it to \( P_0 \).
- Upon receiving \( \{t_i\}_{i \in [k]} \) and \( \{s_i\}_{i \in [k]} \), \( S \) does
  - calculate \( \zeta_i = \pi(s_i) \) for \( i \in [k] \).
  - calculate \( x_\zeta = \beta_\zeta(t_i + [w_{\pi(-\zeta_i)}]) \)
  - set \( x_j \leftarrow \mathbb{Z}_p \) for \( j \in [n] \setminus \{\zeta_i\}_{i \in [k]} \)
  - send (Input, sid, \{\zeta_i\}_{i \in [k]}, \{x_j\}_{j \in [n]} \) to \( \mathcal{F}_{ozc} \).

Observe that \( \mathcal{F}_{ozc} \) will the output each items \( z_i = (x_\zeta + y_\zeta) \cdot \beta_i \) to \( P_1 \), which equals to \( z_i \) in the real world. We show that the incoming message of \( P_0 \) in the ideal world is indistinguishable with the real world.

**Claim 6.** If \( \text{PRF}_p \) is the secure pseudorandom functions with adversarial advantage \( \text{Adv}_{\text{PRF}_p}(1^\lambda, \mathcal{A}) \), then the ideal world \( \text{Ideal}_{\text{omp}, S, Z}(1^\lambda) \) and the real world \( \text{Real}_{\Pi_{\text{omp}}, \mathcal{A}, Z}(1^\lambda) \) are indistinguishable with advantage \( \epsilon = n \cdot \)

23
Proof. In the ideal world, \( \{y'_i\}_{i \in [n]} \) are uniformly random, which replace \( n \) value of \( \text{PRF} \) output, where the advantage is \( n \cdot \text{Adv}_{\text{PRF}_n}(1^\lambda, \mathcal{A}) \).

Case 2: \( P_1 \) is corrupted. We construct the simulator \( S \) to simulate the interface of honest \( P_0 \)

- In the offline phase, \( S \) does,
  - emulates \( F_{\text{ole}} \), outputs \((\beta_i, r_i, u_i, v_i)\) for \( i \in [n] \) and sends \((r_i, v_i)\) to \( P_1 \).
  - emulates \( F_{\text{vole}} \), outputs \( \{\beta_j, r, u_j, v_j\}_{j \in [k-1]} \) and sends \((r, v_j)\) to \( P_1 \).
  - picks random permutation \( \pi \) and emulates \( F_{\text{permute}} \) with input \( \pi \).
  - Record \( v_i \) when \( P_1 \) input it to \( F_{\text{permute}} \).

- Upon receiving \( \{y'_i\}_{i \in [n]} \) from \( P_1 \), \( S \) does,
  - calculate \( y_i = y'_i - r_i \) for \( i \in [n] \).
  - send \((\text{Input}, \text{sid}, \{y_i\}_{i \in [n]})\) to \( F_{\text{oec}} \).

- Upon receiving \( z \) from \( F_{\text{oec}} \), \( S \) does,
  - pick random list \( \{r_0, \ldots, r_{k-1}\} \in (\mathbb{Z}_p^*)^k \).
  - pick random set \( \mathcal{I}_1 := \{s_0, \ldots, s_{k-1}\} \in \mathbb{Z}_{n+k-1}^k \).
  - for \( i \in [k] \), set \( t_i = [w_s] + r_i \);
  - if \( b = 0 \), pick \( \eta \leftarrow \mathcal{I}_1 \) and set \( t_i = t_i - r_i \).
  - act as \( P_0 \) to send \( \{t_i\}_{i \in [k]} \) and \( \mathcal{I}_1 \) to \( P_1 \).

---

Claim 7. If \( \text{PRF}_n^{2k} \) are the secure pseudorandom functions with adversarial advantage \( \text{Adv}_{\text{PRF}_n}(1^\lambda, \mathcal{A}) \), then the ideal world \( \text{Ideal}_{\Pi_{\text{cmp}}, S, Z}(1^\lambda) \) and the real world \( \text{Real}_{\Pi_{\text{cmp}}, A, Z}(1^\lambda) \) are indistinguishable with advantage \( \epsilon = \text{Adv}_{\text{PRF}_n}(1^\lambda, \mathcal{A}). \)

Proof. In the ideal world, \( \{s_i\}_{i \in [k]} \) and \( \{t_i\}_{i \in [k]} \) are randomly generated instead of calculated by \((X, Y)\) and \((\pi, \mathcal{I})\). Obviously, the advantage between random set \( \mathcal{I}_1 \) and \( \{s_i\}_{i \in [k]} \) which is calculated by random permutation \( \pi \) in the...
real world is \( \text{Adv}_{\text{PRF}}^Z(1^\lambda, A) \). For \( \{t_i\}_{i \in [k]} \), due to the random mask \( \beta_i \) and \( \{w_i\} \) are generated from ole and vole, \( \{t_i\}_{i \in [k]} \) in the real world and ideal world are indistinguishable. Therefore, the overall advantage is \( \text{PRF}_k^Z \).

This concludes the proof.

V. PERFORMANCE EVALUATION

In this section, we respectively implement our equality test (Section III) and secure comparison (Section IV), and compare their performance with the CrypTFlow2 [43], ABY [19], GC [3], FSS [26].

A. Experiment Setting

We implement our protocols in C++. For the \( \mathcal{F}_{\text{OT}} \), we utilize the OT library – libOTe [4]. For FSS, we implement the keys correlated generation scheme for benchmark [1]. For the garbled circuit, we utilize EMP-toolkits [3], which is integrated half-gate [51]. The source code of our protocol can be obtained from the anonymous GitHub repository [5]. For ABY and CrypTFlow, we utilize their open-source code [2]. Our experiments are performed in a local area network, using traffic control in Linux to simulate three network settings: (1) local-area settings (LAN): 20Gbps bandwidth with 0.01 ms round-trip latency (RTT). (2) metropolitan-area setting (MAN): 400 Mbps bandwidth with 20 ms round-trip. (3) wide-area setting (WAN): 10Mbps bandwidth with 100 ms round-trip. Our benchmark setting is deployed on the server running Ubuntu 18.04.2 LTS with Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, 48 CPUs, 128 GB Memory. In our benchmark, we set the security parameter \( \lambda = 128 \).

**Equality testing.** The equality testing running time of the online phase (for \( n = 64 \)) is depicted in Fig. 11. Compared with other equality testing implementations, our protocol realizes multiple performance improvements for the online phase. The communication cost of our protocol is close to FSS [26], while the computation cost of our protocol is more subtle than FSS, leading to a significant performance superiority. In general, considering appropriate data size, the efficiency of our equality-testing is (i) over 2× of the garbled circuit, over 7× of the FSS, and over 40× of ABY in the LAN setting; (ii) over 9× of the FSS, over 15× of garble circuit and over 50× of ABY in both MAN and WAN settings. Fig. 16(a) depicts the offline running time compared to FSS (with the correlated keys generation) and ABY. (Considering \( \mathcal{O}(2^n) \) computation complexity of FSS, we take \( n = 16 \).) The offline running time of our protocol is over 1000× faster than FSS and over 5× faster than ABY.

**Secure comparison.** Fig. 17 depicts the online phase running time of secure comparison compared to ABY [19], GC [48], DCF [26] and CrypTflow2 [43] (Due to CrypTflow2 only support 64 bits, our benchmarks perform on \( n = 64 \)). In most cases, our protocol outperforms other protocols in the online phase. In particular, the efficiency of our protocol is (i) over 3× of the FSS/CrypTflow2/GC, and over 20× of the ABY in the LAN setting; (ii) over 3× of the FSS, over 6× of GC/CrypTflow2 and over 15× of ABY in WAN settings. When the network is worse and the data volume is large enough, our protocol efficiency will be slightly lower than FSS (WAN setting and \( > 10^5 \) number of comparisons). Fig. 16(b) depicts the offline running time. The offline phase performance of our protocol is 1000× of FSS. As a trade-off, our offline phase is slightly slower than ABY.

For more benchmarks, we refer readers to Appendix. [B]
VI. CONCLUSION

We propose constant-round equality testing and secure comparison protocols, where each of our protocols enjoys a low communication round and volume in the online phase. Our benchmarks show that the performance of our protocols is several times better than that of SOTA, both in the equality testing and secure comparison.

REFERENCES


This section gives other building blocks such as the OLE and the oblivious short-list zero check.

### A. OLE protocol

In the OLE, both parties have no input initially, and then $P_0$ receives $(a, c)$ and $P_1$ receives $(b, d)$ such that $ab = c + d$. The OLE can be implemented by invoking $p$ times $\mathcal{F}(\frac{1}{2})_{\text{OT}}$. Specifically, $P_1$ picks $a \in \mathbb{Z}_p$ and $\{r_j\}_{j \in \mathbb{Z}_p}$, while $P_1$ picks $b \in \mathbb{Z}_p$. Subsequently, for each invoking of $\mathcal{F}(\frac{1}{2})_{\text{OT}}$, $P_0$ sets $m_0 = -r_j$ and $m_1 = a \cdot 2^j - r_j$, and as the sender inputs $(m_0, m_1)$ to $\mathcal{F}(\frac{1}{2})_{\text{OT}}$: $P_1$ as the receiver inputs the chooes bit $b_j$ and receives output $z_j$. Finally, $P_0$ computes $c = \sum_{j=1}^{p} r_j$, and $P_1$ computes $d = \sum_{j=1}^{p} z_j$. Clearly, $c + d = \sum_{j=1}^{p} r_j + \sum_{j=1}^{p} z_j = \sum_{j=1}^{p} a \cdot 2^j = ab$.

![Protocol $P^\text{ole}$](image.png)

**Input:** $P_0$ and $P_1$ have no input.

**Output:** $P_0$ receives $a \in \mathbb{Z}_p$ and $c \in \mathbb{Z}_p$, while $P_1$ receives $b \in \mathbb{Z}_p$ and $d \in \mathbb{Z}_p$, where $a \cdot b = c + d$.

**Execution:**

- $P_0$ samples $a \in \mathbb{Z}_p$ and $\{r_j\}_{j \in \mathbb{Z}_p}$.
- $P_1$ samples $b \in \mathbb{Z}_p$.
  - For $j \in \mathbb{Z}_p$, $P_0$ and $P_1$ invoke $\mathcal{F}(\frac{1}{2})_{\text{OT}}$:
    - $P_0$ inputs $m_0 = -r_j$ and $m_1 = a \cdot 2^j - r_j$.
    - $P_1$ inputs the chooes bit $b_j$ and receives output $z_j$.
  - $P_0$ computes $c = \sum_{j=1}^{p} r_j$, $P_1$ computes $d = \sum_{j=1}^{p} z_j$.

**APPENDIX A**

**OTHER BUILDING BLOCK**

**B. The functionality of oblivious short list zero check**

In this section, we define the functionality of oblivious short-list zero checks. In particular, an OZC scheme allows $P_0$ input k-dimension selective index set $\mathcal{I} := \{\zeta_0, \ldots, \zeta_{k-1}\}$, $P_0$ and $P_1$ input shared list $\{[x_i]\}_{i \in [n]}$. For $x_i = [x_i]_0 + [x_i]_1$, it checks if $\{x_{\zeta_i}\}_{i \in [k]}$ contains zero and sends the check result to $P_1$. 


Functionality $F_{\text{ocz}}[n, k, p]$

$F_{\text{ocz}}$ interacts with the parties in $P$ and the adversary $S$.

**Input:**
- Upon receiving $(\text{Input}, \text{sid}, I, X)$ from $P_0 \in P$, record $(I, X)$ and send $(\text{Input}, \text{sid}, P_0)$ to $S$, where
  - $X := \{x_0, \cdots, x_{n-1}\} \in \mathbb{Z}_p^n$;
  - $I \in \mathbb{Z}_k^n$;
- Upon receiving $(\text{Input}, \text{sid}, Y)$ from $P_1 \in P$, record $Y$ and send $(\text{Input}, \text{sid}, P_1)$ to $S$, where
  - $Y = \{y_0, \cdots, y_{n-1}\} \in \mathbb{Z}_p^n$.

**Execution:**
- If $I, X$ and $Y$ are recorded, $F_{\text{ocz}}$ does:
  - set $b = 1$ if $\exists i \in I$, $x_\zeta_i + y_\zeta_i = 0$.
  - set $b = 0$ otherwise.
- Send $(\text{Output}, \text{sid}, b)$ to $P_1$.

Fig. 14: The Ideal Functionality $F_{\text{ocz}}$.

Protocol $\Pi_{\text{ocz}}^{k,n,p}(I, X, Y)$

Input: Index list $I := \{\zeta_i\}_{i \in [k]}$ input by $P_0$ which contains $k - t$ non-repeating items, and last $t$ indices equal to $n$; list $X := \{x_i\}_{i \in [n]}$ input by $P_0$; list $Y := \{y_i\}_{i \in [n]}$ input by $P_1$;

Output: $P_1$ receives $z_i = (x_\zeta_i + y_\zeta_i) \cdot \beta_\zeta_i$ for the random value $\beta_\zeta_i$ which is unknown to $P_1$.

**Offline:**
- $P_0$ and $P_1$ invoke $n$ times $\{\beta_i, r_i, [t_i]_{p_0}, [t_i]_{p_1}\} \leftarrow \Pi_{\text{ole}}$, where $P_0$ holds $\{\beta_i, [t_i]_{p_0}\}$, $P_1$ holds $\{r_i, [t_i]_{p_1}\}$.

**Execution:**
- For $i \in [k]$:
  - $P_1$ set $y'_j = y_j + r_i$ for $j \in [n]$;
  - $P_0$ and $P_1$ invoke $\mathcal{F}_{(1,n)}$-$\mathcal{OT}$:
    - $P_1$ as a sender inputs a set $Y' := \{y'_0, \cdots, y'_{n-1}\}$;
    - $P_0$ as a receiver inputs select index $\zeta_i$ and receives $y'_\zeta_i$;
  - $P_0$ calculates $[z_i]_0 = \beta_1 \cdot (x_\zeta_i + y'_\zeta_i) - [t_i]_{p_0}$;
  - $P_1$ sets $[z_i]_1 = -[t_i]_{p_1}$;
  - $P_0$ and $P_1$ reveal $z_\zeta_i$ to $P_1$.

Fig. 15: The Oblivious Short-List Zero Check with OLE Protocol.

C. Oblivious short-list zero check with OLE

We describe the implementation of the oblivious short-list zero check with OLE in Figure 15.
APPENDIX B
OTHER BENCHMARKS

In this section, we give more benchmarks.

A. Offline of Equality Testing and Secure Comparison

Figure 16 shows the running time in the offline phase for the equality testing and secure comparison protocol compared with ABY [19] and DPF [26] in the LAN setting. The running time of our equality testing protocol in the offline phase is entirely superior to the DPF [26], outperforming ABY [19] when the batch size exceeds 1000. Similarly, our secure comparison protocol is also based entirely on the DPF [26]. Although it is slower than ABY [19], the offline performance loss is acceptable for the overall protocol as it achieves a $15 \times$ improvement in running time over ABY during the online phase. In addition, to provide a more detailed comparison of the efficiency between our protocols and ABY, we present the offline running time of our protocols compared to ABY under LAN/MAN/WAN settings in the table III. The running time is given in milliseconds. The results indicate that the higher the bandwidth, the more significant the performance advantage of our protocol.

![Fig. 16: The running time of offline phase on equality testing protocol $\Pi_{eq}$ and secure comparison protocol $\Pi_{cmp}$ compare with ABY [19] and DPF [26] in LAN setting.](image)

(a) Equality testing.  
(b) Secure comparison.

B. 32-bit Secure comparison

Due to Cryptflow2 [43] only supporting the 64-bit secure comparison, we benchmark the running time of the secure comparison protocol $\Pi_{cmp}$ compared with ABY [19], the GC scheme implemented in EMP-toolkits [48], and DCF [26] in LAN/MAN/WAN settings, where takes the elements size $n = 32$ in Figure 17. All benchmarks assume an input length of $n = 32$. The results show that our protocol achieves the best running time across all network settings and batch sizes.
TABLE III: Offline running time of our protocols compared to ABY, under LAN/MAN/WAN settings. The running time is given in ms.

<table>
<thead>
<tr>
<th>Batch size</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10Mbps</td>
<td>100ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our equality testing</td>
<td>1457</td>
<td>2527</td>
<td>4221</td>
<td>15634</td>
</tr>
<tr>
<td>ABY equality testing</td>
<td>213</td>
<td>1477</td>
<td>4574</td>
<td>14434</td>
</tr>
<tr>
<td>Our secure comparison</td>
<td>3510</td>
<td>4344</td>
<td>10678</td>
<td>64996</td>
</tr>
<tr>
<td>ABY secure comparison</td>
<td>129</td>
<td>778</td>
<td>4056</td>
<td>14613</td>
</tr>
<tr>
<td></td>
<td>400Mbps</td>
<td>20ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our equality testing</td>
<td>326</td>
<td>626</td>
<td>818</td>
<td>2649</td>
</tr>
<tr>
<td>ABY equality testing</td>
<td>46</td>
<td>361</td>
<td>1126</td>
<td>3891</td>
</tr>
<tr>
<td>Our secure comparison</td>
<td>809</td>
<td>1065</td>
<td>3175</td>
<td>22873</td>
</tr>
<tr>
<td>ABY secure comparison</td>
<td>45</td>
<td>321</td>
<td>1093</td>
<td>3440</td>
</tr>
<tr>
<td></td>
<td>20Gbps</td>
<td>0.01ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our equality testing</td>
<td>41</td>
<td>51</td>
<td>137</td>
<td>818</td>
</tr>
<tr>
<td>ABY equality testing</td>
<td>4</td>
<td>38</td>
<td>237</td>
<td>2233</td>
</tr>
<tr>
<td>Our secure comparison</td>
<td>66</td>
<td>312</td>
<td>1991</td>
<td>18773</td>
</tr>
<tr>
<td>ABY secure comparison</td>
<td>5</td>
<td>41</td>
<td>258</td>
<td>1764</td>
</tr>
</tbody>
</table>

C. Running time in different input length

Table IV exhibits the online running time of our protocols for different input lengths and batch sizes, provided in milliseconds. The results show that under a WAN setting of 10Mbps, the online running time of our equality testing protocol remains nearly constant at less than 0.5s for batch sizes below 10000, and is approximately 1s for a batch size of 100000. For the secure comparison protocol, the running time is only 5s when the batch size is 100000. The performance is even better in other network environments.

Fig. 17: The running time of secure comparison protocol $\Pi_{eq_2}$ compare with ABY \[19\], GC scheme implemented in EMP-toolkits \[48\], DCF \[26\] and CrypFlow2 \[43\] in LAN/MAN/WAN setting. All benchmarks take the input length $n = 32$. CF2 refers to CrypFlow2. EMP refers to EMP-toolkits.
TABLE IV: The online running time of our protocols in different input lengths and batch sizes, which is given in ms.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Secure comparison</th>
<th>Equality testing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size 16 32 64 128</td>
<td>Size 16 32 64 128</td>
</tr>
<tr>
<td>Batch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAN 10Mbps 100ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>401</td>
<td>401</td>
</tr>
<tr>
<td>1000</td>
<td>405</td>
<td>410</td>
</tr>
<tr>
<td>10000</td>
<td>449</td>
<td>492</td>
</tr>
<tr>
<td>100000</td>
<td>1006</td>
<td>1935</td>
</tr>
<tr>
<td>MAN 400Mbps 20ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>81</td>
<td>81</td>
</tr>
<tr>
<td>1000</td>
<td>83</td>
<td>84</td>
</tr>
<tr>
<td>10000</td>
<td>98</td>
<td>109</td>
</tr>
<tr>
<td>100000</td>
<td>215</td>
<td>404</td>
</tr>
<tr>
<td>LAN 20Gbps 0.01ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>&lt;1</td>
<td>&lt;1</td>
</tr>
<tr>
<td>1000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>10000</td>
<td>14</td>
<td>19</td>
</tr>
<tr>
<td>100000</td>
<td>72</td>
<td>146</td>
</tr>
</tbody>
</table>

D. Running time in different input length

Table IV exhibits the online running time of our protocols for different input lengths and batch sizes, provided in milliseconds. The results show that under a WAN setting of 10Mbps, the online running time of our equality testing protocol remains nearly constant at less than 0.5s for batch sizes below 10000, and is approximately 1s for a batch size of 100000. For the secure comparison protocol, the running time is only 5s when the batch size is 100000. The performance is even better in other network environments.