Malicious Secure Oblivious Shuffling
from Beneš Network

Qiuhong Anna Wei
Joint work with Gayathri Garimella, Anna Lysyanskaya, Peihan Miao

Department of Computer Science, Brown University
May, 2024
1 Introduction

Oblivious shuffle allows two parties to jointly permute a vector and learn additive secret shares of the result. In particular, one party provides the input vector $X$ while the other party provides the permutation $\Pi$, and neither party learns any information about the other’s input. As a result, neither party can independently construct $\Pi(X)$.

Oblivious shuffle is a fundamental building block in many tools in secure multi-party computation and has a great number of applications. Here we briefly discuss a few example use cases.

Private Set Intersection. Private set intersection enables two parties, $P_1$ and $P_2$, with respective input sets $X$ and $Y$ to compute the intersection $X \cap Y$ without revealing any information about items not in the intersection. PSI protocols have become incredibly efficient over the last decade, but they reveal the intersection in the clear, while many real-world applications require that only partial or aggregate information about the intersection should be revealed. To keep $X \cap Y$ secret from both parties, a natural way to achieve this [GMR+21] is to encode items in the set and shuffle them before comparing for equality, so that neither party can associate equality test results with the original input items.

A notable real-world deployment of such techniques is Google’s measurement of ad conversion rate and ad revenue. In particular, Google compares the set of online ad viewers and the set of offline purchasers, which are held by the advertiser and merchants respectively, and computes the cardinality of the intersection as well as the total revenue from customers in the intersection [IKN+20, MPR+20].

Anonymous Communication. With broad applications in broadcasting, messaging, voting, and others, anonymous communication protects the privacy of the participants. While end-to-end encrypted communication has become common, systems that can additionally hide communication patterns (e.g., sender of messages) remain a subject of ongoing research. Many such systems rely on shuffling to break the connection between a message and its sender. A line of work pioneered by Chaum [Cha81], which provides the first solution for sender-anonymous communication, employs mixnets where each server conducts a verifiable shuffle of messages. Another line of work [EB21, LK23] proposes that servers hold secret shares of the data, collaboratively shuffle it, and conduct integrity checks.

Collaborative Data Analytics. In many applications, mutually distrustful parties seek to perform computations on combined datasets while preserving the privacy of their respective data. While many methods exist to protect data confidentiality, it is also important to protect access patterns. In order to disassociate data operations from the affected data items, a common approach is to shuffle the dataset beforehand [MRR20, JSZ+22, AFO+21, AHI+22, ACD+21, ZWR+16]. A notable application of this is machine learning [LWR+21, MZ17].

Most prior works of oblivious shuffle [GMR+21, CGP20] are secure against semi-honest adversaries, who cannot deviate from the protocol but may try to learn as much information as possible from the protocol execution. However, shuffling is frequently employed in critical applications such as health data analysis or anonymous communication for voting. This renders the presence of mali-
cious adversaries, who may deviate arbitrarily from the protocol and take any actions to obtain information of the other parties’ private input, a more realistic model and often a necessary consideration.

In light of this, recent works [EB21, Lau21] attempted to develop malicious-secure shuffle protocols, but a subsequent work [SYB+23] identified concrete attacks that can compromise their claimed security. Additionally, the MP-SPDZ library [Kel20] provides a malicious-secure shuffle protocol, but it is shown to be less efficient than a later construction [SYB+23].

The malicious-secure shuffle protocol proposed in [SYB+23] is the primary work we will compare against. It builds on top of the semi-honest-secure secret-shared shuffle (SSS) protocol [CGP20], which utilizes permutation-correlated pseudorandom correlations generated in an offline phase to facilitate an efficient online shuffle phase. To obtain malicious security, Song et al. proposed correlation checks to defend against incorrect pseudorandom correlations and a leakage-reduction mechanism to defend against selective failure attacks, including ones introduced by the correlation checks.

In this work, we propose an alternative framework to achieve malicious-secure oblivious shuffle. Instead of permutation-correlated pseudorandom correlations, we build on a semi-honest-secure construction based on oblivious switching network and oblivious transfer [GMR+21]. To make it secure against malicious adversaries, at a high level, we employ commitments for verification and leverage redundancy through cut-and-choose and fault-tolerant gates.

Compared to prior work, our construction achieves comparable efficiency with an intuitive and relatively simple construction.

2 Preliminary

In this section, we present the building blocks used in the construction of the malicious secure oblivious shuffling.

2.1 Oblivious Switching Network

Beneš Network $\mathbb{B}[N]$ [Ben64] is a dynamic non-blocking switching network that takes in $N$ inputs and outputs a permutation of the $N$ inputs. It can realize any permutation and is useful for telephone networks, multiprocessor systems, parallel computers, radio communication between robots, and others.

A Beneš Network $\mathbb{B}[N]$ is constructed out of $2 \times 2$ switches that take two inputs $(x_0, x_1)$ and outputs either $(x_0, x_1)$ or $(x_1, x_0)$ depending on a single control bit. It has a defined structure for each $N$ based on recursively stacking these switches (Figure 1). A $\mathbb{B}[2]$ consists of a single $2 \times 2$ switch. A $\mathbb{B}[4]$ stacks two $\mathbb{B}[2]$ nets in the middle, prepend an input layer, and append an output layer. A $\mathbb{B}[8]$ stacks two $\mathbb{B}[4]$ nets and adds two outer layers and so on. A $\mathbb{B}[N]$ has $N/2$ switches at each vertical layer and $2 \log_2(N) - 1$ layers of switches (width), which forms $O(N \log N)$ switches overall.

An oblivious switching network protocol takes in a permutation $\Pi$ from a receiver, vector $\vec{x}$
from a sender and outputs through a switching network additive secret shares of $\Pi(\vec{x})$. This is the key component of the semi-honest-secure oblivious shuffle construction [GMR+21] we build upon.

Mohassel & Sadeghian [MS13] introduced the first semi-honest oblivious switching network protocol based on oblivious transfers (OT). In this protocol, the receiver selects control bits of switches based on the permutation $\Pi$ while the sender selects random masks for the input and output wires of all switches. For each wire in the network, the receiver always learns its logical value blinded with the random mask for that wire. At the input layer of the network, instead of $x$, the sender sends $x \oplus A'$ where $A'$ is the mask for the first layer input wires. At the output layer, the sender knows the mask $B'$ of the output wires, and the receiver learns $\Pi(x) \oplus B'$, forming additive secret shares of the logical output $\Pi(\vec{x})$.

In order to achieve obliviousness such that the receiver does not learn $\vec{x}$ and the sender does not learn $\Pi$, the parties use a 1-out-of-2 OT for each switch. For a $2 \times 2$ switch with input masks $(A_0, A_1)$ and output masks $(B_0, B_1)$, the receiver’s input to the OT is the control bit $b$, while the sender prepares $m_0 = (A_0 \oplus B_0 | A_1 \oplus B_1)$ (for no swap) and $m_1 = (A_1 \oplus B_0 | A_0 \oplus B_1)$ (for a swap). The receiver learns $m_b$ and uses it to reapply the random masks as the evaluation progresses through the layers of the network. This ensures that the receiver always sees $x$ blinded with certain random masks while the sender does not learn the control bits of the switches. Much of our construction involves how to maintain this property against malicious sender and receiver.

### 2.2 Cut-and-Choose

A malicious sender may prepare badly formed OT messages, but the receiver cannot verify these messages in an OT and may unintentionally leak information about their control bits as a result. The standard way to address this is cut-and-choose, a technique dating back to at least Chaum [Cha83] and often used for Yao’s Garbled Circuit [Yao86]. At a high level, the sender generates many independent OTs, and the receiver checks a random subset of them by asking the sender to ‘open’ both messages. If any opened OTs are incorrectly formed, the receiver can abort. If all opened OTs are correctly formed, the receiver knows that most of the unopened OT instances are also correct and continues with the protocol using the unopened instances.
2.3 Committed Oblivious Transfer

To enable the receiver to verify whether the OT messages are prepared correctly by the sender and to allow the sender to verify the difference between the control bits of two OTs (its use will become clear in Section 3), we introduce the functionality of the committed oblivious transfer, presented in Figure 2.

Parameter: message length $t$.

Choose:
1. Receive $(\text{choose}, id, b_{id})$ from the receiver, where $b \in \{0, 1\}$.
2. If no message of the form $(\text{choose}, id, .)$ is present in memory, store $(\text{choose}, id, b_{id})$ and send $(\text{choose}, id)$ to the sender.

Transfer:
1. Receive $(\text{transfer}, id, m_0, m_1)$ from the sender, where $m_0, m_1 \in \{0, 1\}^t$.
2. If no messages of the form $(\text{transfer}, id, m_0, m_1)$ is present in memory and a message of the form $(\text{choose}, id, b_{id})$ is stored, send $(\text{transfer}, id, m_b)$ to the receiver.

Open Message:
1. Receive $(\text{open}, id)$ from the sender.
2. Output stored message $(\text{transfer}, id, m_0, m_1)$ to the receiver.

Check Control Bits:
1. Receive $(\text{open}, id_p, id_q)$ from the sender.
2. Output stored message $(\text{transfer}, id_p, id_q, b_{p} \oplus b_{q})$ to the receiver.

Figure 2: Ideal functionality $F_{\text{COT}}$

2.4 Homomorphic Commitment Scheme

We make use of a homomorphic commitment scheme to enable the parties to verify values and to ensure that values are not changed upon verification. The homomorphic commitment functionality is presented in Figure 3. It allows a sender and receiver to commit and open messages. The functionality is xor homomorphic, enabling it to reveal xor of any two committed values within a session.

2.5 Fault Tolerant Gate

Recall that cut-and-choose enables the receiver to have confidence that most of the unopened oblivious transfer instances are correct. In our construction, the receiver can in fact partially verify the correctness of the unopened OT instances. As the OT messages consist of the decommitments of the xors of a specific combination of random masks, which the sender commits to with $F_{\text{HCom}}$ publicly, the receiver may verify their received OT messages using $F_{\text{HCom}}$.

It is possible that a number of incorrectly formed OTs are undetected by cut-and-choose and the receiver needs to work with them in the rest of the protocol, where oblivious switching network evaluation takes place. For reasons discussed in Section 3.3, the receiver cannot simply abort when they detect faulty unopened OTs; instead, they need to successfully evaluate the switching network.
\( \mathcal{F}_{\text{Hcom}} \) interacts with a sender \( P_s \), a receiver \( P_r \), and an adversary \( S \), and it has the following phases:

**Commit:**
1. Upon receiving a message \((\text{com}, sid, idx, P_s, P_r, m)\) from \( P_s \), where \( m \in \{0, 1\}^* \), record the tuple \((idx, P_s, P_r, m)\) and send the message \((\text{receipt}, sid, idx, P_s, P_r)\) to \( P_r \) and \( S \).
2. Ignore any future commit messages with the same \( idx \) from \( P_s \) to \( P_r \).

**Open:**
1. On receiving a message \((\text{reveal}, sid, idx)\) from \( P_s \): If a tuple \((idx, P_s, P_r, m)\) was previously recorded, then send message \((\text{ok}, sid, idx, P_s, P_r, m)\) to \( P_r \) and \( S \). Otherwise, ignore.
2. On receiving a message \((\text{reveal}, sid, idx_0, idx_1)\) from \( P_s \): If tuples \((idx_0, P_s, P_r, m_0), (idx_1, P_s, P_r, m_1)\) were previously recorded, then send message \((\text{ok}, sid, idx_0, idx_1, P_s, P_r, (m_0 \oplus m_1))\) to \( P_r \) and \( S \). Otherwise, ignore.
3. If a message \((\text{aborts}, sid, idx)\) is received from \( S \), the functionality halts.

Figure 3: Ideal functionality \( \mathcal{F}_{\text{HCom}} \)

in the presence of these faulty OT instances, which is guaranteed by cut-and-choose to be low in number with high probability. To enable this, we introduce fault tolerant gate.

A fault tolerant gate has the same functionality as a switch: it takes two inputs and may either swap them or not depending on a single control bit. In our protocol, the Beneš network uses fault tolerant gates in place of switches. Internally, a fault tolerant gate consists of a fixed number of OTs, and it may function correctly as long as at least one OT shares the same choice bit as the gate, one of which is designated as the anchor OT of the gate, and as long as at least one OT in the gate is correctly formed. This fault tolerance relies on correct soldering, discussed in more detail in Section 3. At a high level, during evaluation, the receiver will solder in a specific way the input wires to the gate with the input wires of one correct OT, which may or may not share the gate’s control bit and may or may not be the anchor OT, and the output wires of the OT with the output wires of the gate.

3 Malicious Secure Oblivious Shuffling

In this section, we present our construction of malicious secure oblivious shuffling based on Beneš network.

3.1 Construction Overview

The ideal functionality is presented in Figure 4. The protocol, presented in detail in Figure 5, involves six stages, starting with the sender Alice and the receiver Bob (1) committing to relevant values. Then, the parties (2) invoke oblivious transfer with \( \mathcal{F}_{\text{COT}} \) and (3) engage in Cut-and-Choose on the generated OTs. If no opened OTs are badly formed, then Alice and Bob jointly solder (4) within and (5) between fault tolerant gates. Finally, Bob (6) evaluates the switching network and the parties obtain additive secret shares of the permuted input vector.
Functionality:

1. Receive input vector $X = \{x_1, \ldots, x_n\}$ of size $n$ from Alice.
2. Receive permutation $\Pi$ from Bob.
3. [output] Send $X_a$ to Alice and $X_b$ to Bob, where $\Pi(X) = X_a \oplus X_b$.

Figure 4: Ideal functionality $F_{\text{shuffle}}$ for malicious-secure shuffling, where Alice and Bob learn additive shares of shuffled input vector.

3.2 Protocol

The protocol is presented in detail in Figure 5.

3.3 Discussion

In this section, we highlight aspects of the protocol that are important for malicious security.

3.3.1 Defense against Bob

Learning the other soldering xor

In stage 4 of the protocol, if Bob can obtain the soldering xor from Alice for $\text{flip}_i = \text{decom}(b_1 \oplus b_i) = b$ while the real flip is $1 - b$, then Bob can combine it with the OT messages he received in stage 2 to learn both permutations for the fault tolerant gate. More concretely, he can obtain the outputs for both $b = 0$ and $b = 1$ for this particular gate. Assuming the logical value input to this gate is $(v_0, v_1)$, and denoting the gate’s output wire labels as $(B_0, B_1)$, then Bob learns both $(v_0 \oplus B_0, v_1 \oplus B_1)$ and $(v_1 \oplus B_0, v_0 \oplus B_1)$. This will allow him to extract all four values directly and thus learn about Alice’s private input $X$, compromising Alice’s privacy.

The above may happen if Bob can lie about flip without being caught or if Alice always solders as if flip is a fixed value. The former is countered with ‘Check Control Bits’ in $F_{\text{COT}}$ (Figure 2), which allows Alice to verify the xor of OT choice bits and ensure the soldering xor she sends always matches the real flip and does not leak any information that Bob should not learn. The latter is solved by introducing stage (4) to the protocol, where Alice sends different soldering xor messages based on the flip.

Learning both OT messages

Note that the implementation of $F_{\text{COT}}$ uses malicious-secure OT as if only semi-honest OTs are employed, a malicious Bob may learn both OT messages, consisting of four different xor values of four random masks. This will allow Bob to learn the random masks in the clear, and thus also information about $X$, compromising Alice’s privacy.
Parameters:
- Committed Oblivious Transfer (see Figure 2) implemented with malicious OT.
- Additively homomorphic commitment scheme ah-com() (see Figure 3).
- The number of fault tolerant gates $q$, the number of OTs in a fault tolerant gate $\ell$, the total number of committed OT instances $k = (q\ell)/(1 - \alpha)$ where $\alpha$ is the fraction of all OTs to check in cut-and-choose.

Inputs:
- Receive input vector $X = \{x_1, \ldots, x_n\}$ of size $n$ from the sender Alice, where $x_i \in \{0, 1\}^t$.
- Receive permutation $\Pi$ from the receiver Bob.

Protocol:
1. (Commit)
   (a) Alice commits to input masks $A_0^i, A_1^i$ and output masks $B_0^i, B_1^i$ for each of $i \in [k]$ OT instances and sends $\text{ah-com}(A_0^i), \text{ah-com}(A_1^i), \text{ah-com}(B_0^i), \text{ah-com}(B_1^i)$ to Bob.
   (b) Bob commits to
      - a subset of $\alpha k$ OT indices open-indices. Bob sends $\text{com(open-indices)}$ to Alice.
      - a random $\Pi': [q\ell] \rightarrow [q\ell]$ that arranges $k - \alpha k = q\ell$ OT instances into a Beneš network. Bob sends $\text{com(\Pi')} = \Pi$ to Alice.
   (c) Bob chooses the choice bits $b_i$ based on $\Pi'$ and $\Pi$.

2. (Oblivious Transfer) The parties invoke $k$ (parallel) instances of oblivious transfer using $\mathcal{F}_{\text{COT}}$. In the $i$-th instance:
   - Alice is the sender with input $(m_0^i = \text{decom}(A_0^i \oplus B_0^i)||\text{decom}(A_1^i \oplus B_1^i), m_1^i = \text{decom}(A_1^i \oplus B_0^i)||\text{decom}(A_0^i \oplus B_1^i))$.
   - Bob is the receiver with choice bit $b_i$. He obtains output $m_i^b$.

3. (Cut-and-Choose)
   (a) Bob sends $\text{decom(open-indices)}$ to Alice. Alice ‘opens’ both messages of the committed OT instances corresponding to open-indices.
   (b) If any OTs are badly formed, Bob aborts.
   (c) Otherwise, Bob sends $\text{decom}(\Pi')$ to Alice and discards the $\alpha k$ OTs corresponding to open-indices. Among the remaining $q\ell$ OT instances, re-index the OTs so that $[j * \ell, (j + 1) * \ell)$ denote the instances belonging to fault tolerant gate $j \in [0, q)$. Let $j * \ell$ denote the anchor OT of gate $j$.

4. (Soldering within fault tolerant gates) For each fault tolerant gate $j \in [0, q)$ and for each OT $i$ in it:
   (a) Bob sends $\text{flip}_i = \text{decom}(b_i \oplus b_i)$ to Alice. Note $b_i$ corresponds to masks $A_0^{j*\ell}$ and $A_1^{j*\ell}$.
   (b) If $\text{flip}_i = 0$ then Alice sends $\text{decom}(A_0^{j*\ell} \oplus A_0^{j*\ell+i}), \text{decom}(A_1^{j*\ell} \oplus A_1^{j*\ell+i})$. Else, $\text{flip}_i = 1$ then Alice sends $\text{decom}(A_1^{j*\ell} \oplus A_0^{j*\ell+i}), \text{decom}(A_0^{j*\ell} \oplus A_1^{j*\ell+i})$. Alice also sends $\text{decom}(B_0^{j*\ell+i} \oplus B_0^{j*\ell}), \text{decom}(B_1^{j*\ell+i} \oplus B_1^{j*\ell})$.
   (c) If Alice’s messages are badly formed, Bob aborts.

5. (Soldering between fault tolerant gates) For each fault tolerant gate $j \in [0, q)$ not in the last layer:
   - To connect its output wires with appropriate input wires in the next layer, Alice sends $\text{decom}(B_0^{j*\ell}) \oplus \text{decom}(A_i^{c*\ell})$ and $\text{decom}(B_1^{j*\ell}) \oplus \text{decom}(A_i^{c''*\ell})$, where $j', j''$ are the corresponding gates and $c', c''$ are the corresponding input wire indices. Note the correspondence is defined by Beneš network’s structure.

6. (Evaluation) Let $A'$ denote the input wire labels of the first layer and $B'$ denote the output wire labels of the last layer.
   (a) Alice sends $X \oplus A'$ to Bob. Bob computes $\Pi(X \oplus A')$.
   (b) Bob evaluates the switching network with information from stages 1-5 to compute $\Pi(A') \oplus B'$.
   (c) Bob computes $\Pi(X \oplus A') \oplus (\Pi(A') \oplus B') = \Pi(X) \oplus B'$. Alice has output wire label $B'$ of the last layer.

Figure 5: Protocol realizing malicious secure oblivious shuffling
3.3.2 Defense against Alice

Learning Bob’s choice bit through selective failure attack on OT messages

In stage 1 of the protocol, Alice commits to her OT masks and sends them to Bob. In stage 2, Bob may partially verify each OT. As he knows his control bit and as the OT messages have a fixed structure, he may compute the commitments of the xors of masks he expects to receive (from the homomorphic commitments of the individual masks), and he can check it against the actual message he received from the OT. If Bob detects inconsistencies, it is critical that he does not abort or signal. This is because if Alice performs a selective failure attack where she only corrupts one of the two messages, signaling errors will leak Bob’s choice bit.

As such, in stage 2, Bob keeps quiet about any faulty OTs he notices and proceeds with the switching network evaluation using fault tolerant gates. In order to successfully evaluate the switching network in the presence of faulty OTs, there needs to be at least one correct OT in each fault tolerant gate. This is ensured with high probability by cut-and-choose from stage 3, which guards against maliciously constructed OTs from Alice.

Note that during cut-and-choose, Bob may freely abort as Alice is asked to ‘open’ both messages with no dependence on Bob’s control bit.

Learning through selective failure attack on soldering xor

In stage 4, Bob aborts if any of Alice’s messages (decommitments of the xor of masks) are badly formed. This does not incur any privacy loss as all information involved in verification is public.

3.4 Performance

For input vector size $n$, our construction achieves $O(n \log n)$ computation and communication. Most of the computation and communication costs are incurred for each OT, and there are $k = (q\ell)/(1 - \alpha)$ OT instances in total where $\ell$ and $\alpha$ are constants, and $q$ is $O(n \log n)$, the number of switches in a Beneš network for $n$ inputs. It is comparable to the malicious-secure shuffle protocol proposed in [SYB+23] based on the SSS protocol [CGP20]. We leave concrete optimizations for future work.
References


