RUFFLE: RAPID 3-PARTY SHUFFLE PROTOCOLS

Cryptography Project Report

Yash Maniya
B20CS033

Contents

1 Introduction **3**
- 1.1 Scope . . . 3
- 1.2 Introduction . . . 3

2 Preliminaries **3**
- 2.1 Secret Sharing Semantics . . . 3
- 2.2 Shared Key Setup . . . 4
- 2.3 Random Permutation . . . 4
- 2.4 Collision Resistant Hash Function . . . 4
- 2.5 Commitment Scheme . . . 5
- 2.6 Joint Message Passing . . . 5
- 2.7 Output reconstruction [29] . . . 5

3 3-Party Computation Shuffle **6**

4 Ruffle **6**
- 4.1 Generation of $[\pi(\alpha_T)]$: . . . 6
- 4.2 Generation of $[\beta_{T_o} = \pi(\beta_T) \oplus R]$ where $\pi = \pi_{12} \circ \pi_{01} \circ \pi_{02}$: . . . 7
- 4.3 Generating β_{T_o} towards P1 : . . . 7
- 4.4 Generating β_{T_o} towards P2 : . . . 7
- 4.5 Generating β_{T_o} towards P0 : . . . 7
- 4.6 Generation of $[\alpha_{T_o}] = [\pi(\alpha_T)] \oplus [R]$: . . . 7
- 4.7 Guarantees for Output Delivery . . . 8

5 Protocol Ruffle **9**

6 Applications **10**
- 6.1 Anonymous Broadcast . . . 11
- 6.2 GraphSC Paradigm . . . 11

7 Benchmarks **12**

8 Future Scope and Extensions on given Protocols **14**

9 Conclusion **14**

10 Proof for Security of Protocol **14**

11 References **15**

1 Introduction

1.1 Scope

The selected paper presents a new approach for performing a shuffle operation among three parties in a distributed system. The authors propose a protocol called Ruffle, which is designed to achieve rapid shuffling of inputs with strong security guarantees. The paper discusses the motivation behind the development of Ruffle, the technical details of the protocol, and its security analysis. The scope of the paper also includes the evaluation of the performance of the Ruffle protocol against other state-of-the-art shuffle protocols, demonstrating its efficiency and effectiveness. Overall, this paper contributes to the field of secure computation by presenting an efficient and secure protocol for three-party shuffling.

1.2 Introduction

The purpose of this paper is to explore the technique of shuffle and its relevance in privacy-preserving protocols. Shuffle involves randomly rearranging the elements of an ordered set while keeping both the permutation and the elements themselves concealed. Secure shuffle is widely used in various domains, including secure sorting, electronic voting, anonymous broadcast, and GraphSC paradigm, among others. The report primarily focuses on the application of secure shuffle in anonymous broadcast, where clients can broadcast messages anonymously. It highlights the importance of quick response time in such applications and explains how the preprocessing paradigm can minimize the response time of secure shuffle protocols.

The report also examines the design choices made to enhance the efficiency of secure shuffle protocols. These include working in the small-party setting and using the cryptographic technique of secure multiparty computation (MPC). It discusses prior works on shuffle and the pros and cons of using MPC in secure shuffle protocols. Finally, the report concludes by summarizing the contributions of this study, which primarily involve developing efficient and fast secure shuffle protocols, especially in the context of anonymous broadcast. The report also points out future research directions in this field. Overall, this report aims to provide a comprehensive and detailed overview of secure shuffle protocols and their applications while focusing on improving efficiency and response time for time-critical applications.

2 Preliminaries

We focus on 3-party protocols, where three parties ($\mathcal{P}$ = P0, P1, P2) are connected through pairwise private and authentic channels. The protocols are designed to be secure against a probabilistic, polynomial-time adversary ($\mathcal{A}$) who can corrupt at most one party. The security is proven in the MPC standalone simulation-based security model using the real-world/ideal-world simulation paradigm. The parties use a one-time key setup to establish common random keys for a Pseudo-Random Function (PRF) and can non-interactively sample a common random bit string. They also have access to a collision-resistant hash function and a non-interactive commitment scheme (mentioned in detail below).

2.1 Secret Sharing Semantics

We would use the following two secret sharing semantics :

- $[\ \cdot\ ]$-sharing : A value v $\in \mathbb{Z}_2$ is said to be (3, 1) replicated secret shared (RSS) or $[\ \cdot\ ]$-shared, if there exists $[\mathrm{v}]_{01}, [v]_{02}, [v]_{12} \in \mathbb{Z}_2$ such that $v = [v]_{01} \oplus [v]_{02} \oplus [v]_{12}$, and each $[v]_{ij} \in \{[v]_{01}, [v]_{02}, [v]_{12}\}$ is held by $\mathrm{P}_i, P_j \in \mathcal{P}$

- [[·]]-sharing : A value v $\in \mathbb{Z}_2$ is [[·]]-shared among P, if there exists $\alpha_v \in \mathbb{Z}_2$ that is [·]-shared, and there exists $\beta_v \in \mathbb{Z}_2$ such that $\beta_v = v\alpha_v$ which is held by all parties in P.

  An l-bit value v $\in {\mathbb{Z}_2}^l$ is said to be [[·]]-shared ([·]-shared) if each bit in v is [[·]]-shared ([·]-shared). Henceforth, we use shares and secret-shares interchangeably

Non-interactively generating [·]-shares of a common $v \in {\mathbb{Z}_2}^l$ held by P_l, P_m. To generate [v], parties need to define three shares $[v]_{01}, [v]_{02}, [v]_{12} \in {\mathbb{Z}_2}^l$ such that $v = [v]_{01} \oplus [v]_{02} \oplus [v]_{12}$, where each $[v]_{ij}$ is held by parties $P_i, P_j \in \mathcal{P}$. Observe that this can be done non-interactively by setting the share $[v]_{lm} = v$ and the other two [.] -shares of v as 0.

2.2 Shared Key Setup

A PRF - $\mathcal{F}$ is used to establish shared keys between the parties. The input to the PRF is a pair of binary strings of length k, and the output is a binary string of length l. The set of binary strings of length l is denoted as X, where $X = {\mathbb{Z}_2}^l$. Each pair of parties P_i and P_j know a common key k_{ij}, and all parties in $\mathcal{P}$ know kP. To sample a common random value r non-interactively, P_i and P_j can compute $F_{k_{ij}}(id_{ij})$, where id_{ij} is a counter maintained by P_i and P_j. The counter id_{ij} is updated after every PRF invocation to ensure that a fresh value is obtained for each sampling.

Functionality 2.1. $\mathcal{F}_{setup}$ interacts with the parties in $\mathcal{F}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{setup}$ picks random keys k_{ij} for $i, j \in \{0, 1, 2\}, i < j$, and $k_{\mathcal{P}}$. Let y_x denote the keys corresponding to party P_x . Then

$$y_x = (k_{01},\ k_{02} \text{ and } k_{\mathcal{P}}) \text{ when } P_x = P_0.$$
$$y_x = (k_{01},\ k_{12} \text{ and } k_{\mathcal{P}}) \text{ when } P_x = P_1.$$
$$y_x = (k_{02},\ k_{12} \text{ and } k_{\mathcal{P}}) \text{ when } P_x = P_2.$$

Output : Send (Output, y_x) to every $P_x \in \mathcal{P}$.

2.3 Random Permutation

Let $\mathbb{N} = \{1, 2, ..., N\}$, and permutation $\pi : \mathbb{N} \to \mathbb{N}$ is bijective. The set S_N contains all possible rearrangements of elements in N (i.e. π) and hence comprises N! permutations. S_N satisfies group properties of closure, associativity, and presence of identity. Two parties, P_i and P_j, can non-interactively generate a random $\pi \in S_N$ using the shared key established via $\mathcal{F}_{setup}$.

2.4 Collision Resistant Hash Function

H : $\mathcal{K} \times \mathcal{L} \to \mathcal{Y}$ is a hash function family, where $\mathcal{K}$ is the set of keys, $\mathcal{L}$ is the set of messages, and $\mathcal{Y}$ is the set of hash values. The hash function H is said to be collision-resistant if, for all probabilistic polynomial-time adversaries $\mathcal{A}$: Given the description of H_k for a random $k \in \mathcal{K}$, there exists a negligible function negl(·) such that :

$$Pr[(x_1, x_2) \leftarrow \mathcal{A}(k) : (\mathrm{x}_1 \neq x_2) \wedge H_k(x_1) = H_k(x_2)] \leqslant negl(k), \text{ where } x_1, x_2 \in \mathcal{L}.$$

In simpler terms, a hash function H is considered collision-resistant if it is computationally infeasible for an attacker, given the hash function H and any input message x_1, to find a different message x_2 such that $H(x_1) = H(x_2)$. This means that the probability of finding two different messages with the same hash value should be negligible. The security of the hash function depends on the choice of the key k, which should be chosen at random from the set $\mathcal{K}$.

2.5 Commitment Scheme

The commitment scheme $Com(x)$ has two properties: hiding and binding. Hiding property ensures privacy of the value x given its commitment $Com(x)$. Binding property prevents a corrupt party from opening the commitment to a different value $x' \neq x$. Providing an incorrect opening for a commitment results in outputting a $\perp$. The commitment is defined as $Com(x;r) = H(x||r)$, where $H(\cdot)$ is a hash function. $Com(x;r)$ is a commitment scheme where x is the value being committed to and r is a random value used to generate the commitment. The opening of the commitment is defined as $(x||r)$.

2.6 Joint Message Passing

It allows two parties to send a message to a third party by having one party send the message and the other send its hash. If there is a discrepancy in the received messages, a TTP is identified to perform the required computation and ensure delivery of the output.

Protocol 2.2. $\Pi_{jmp}(v, P_i, P_j, P_k)$ **:**
$\rightarrow SendPhase$: P_i sends $msg_i = v$ to P_k
$\rightarrow VerifyPhase$: P_j sends $msg_j = H(v)$ to P_k who checks if the hash is consistent with the value sent by P_i. If the values are not consistent, parties proceed as follows to identify a TTP.
$\rightarrow \mathrm{P}_k$ broadcasts $(Accuse, P_i, P_j, msg_i, msg_j)$

- If $H(msg_i) == msg_j$ then parties set $TTP = P_i$
- If msg_i is different from the value sent by P_i then P_i broadcasts $(Accuse, P_k)$ and parties set $TTP = P_j$.
- If msg_j is different from the value sent by $\mathrm{P}_j then P_j broadcasts$(Accuse, P_k) and parties set $TTP = P_i$.
- If both parties $\mathrm{P}_i and P_j broadcasts$(Accuse, P_k) then parties set $TTP = P_i$.
- If none of the parties P_i and P_j accuse then parties set $TTP = P_k$.

2.7 Output reconstruction [29]

To enable reconstruction of a $[[\cdot]]$-shared value $v \in {\mathbb{Z}_2}^l$, parties proceed as follows :
Preprocessing Phase - Generate commitments on each of the $[\ \cdot\ ]$-shares of $[\alpha_v]$ using common randomness.

- Each pair $P_i, P_j \in \mathcal{P}$ computes $Com([\alpha_v]_{ij})$ on the value $[\alpha_v]_{ij}$ using the common randomness.
- P_i, P_j jmp $Com([\alpha_v]_{ij})$ to P_k
- If a malicious party is identified, subsequent computation proceeds via the TTP.

Online Phase - To reconstruct v :

- P_i, P_j send the opening of $Com([\alpha_v]_{ij})$ to P_k.
- If the malicious party sends an incorrect opening, P_k is guaranteed to receive the correct opening from the honest party, as the commitment scheme outputs for incorrect ones.
- Party P_k uses the correct opening to obtain the missing share $[\alpha_v]_{ij}$.

- Reconstruct v as $v = \beta_v \oplus [\alpha_v]_{ij} \oplus [\alpha_v]_{ij} \oplus [\alpha_v]_{ij}$.
- Reconstruction will not fail even if a malicious party tries to disrupt it by sending an incorrect message, resulting in robust reconstruction.

3 3-Party Computation Shuffle

The 3PC shuffle is an operation that shuffles a table T consisting of N rows, where each row is an l-bit string. (The J·K shares are computed using a secret sharing scheme, where J parties hold K shares each.)
Ideal functionality for shuffle is defined below :

Functionality 3.1. $\mathcal{F}_{setup}$
Parties involved: Set of parties $\mathcal{P}$ and adversary $\mathcal{S}$
Input : $[[\cdot]]$-shares of the input table T from all parties and $[[\cdot]]$-shares of T_o from $\mathcal{S}$, i.e., β_{T_o}, $[\alpha_{T_o}]_{ic}$, and $[\alpha_{T_o}]_{jc}$ where P_i, P_j, P_c denote parties in $\mathcal{P}$.
Output: $[[\cdot]]$-shares of the shuffled table T_o for each party in $\mathcal{P}$.

$\mathcal{F}_{setup}$ Algorithm :

- Reconstruct input T using $[[\cdot]]$-shares of the honest parties.
- Sample a random permutation π from the space of all permutations, S_N, and generate $T_o = \pi(T)$.
- Set $[\alpha_{T_o}]_{ij} = T_o \oplus \beta_{T_o} \oplus [\alpha_{T_o}]_{ic} \oplus [\alpha_{T_o}]_{jc}$. Let $[[T_o]]_x$ denote the $[[\cdot]]$-share of T_o for $P_x \in \mathcal{P}$.
- Send (Output, $[[T_o]]_x$) to each party in $\mathcal{P}$.

4 Ruffle

The input table T is $[[\cdot]]$-shared, with $_T$ and β_T satisfying $\beta_T = T \oplus \alpha_T$ and $\alpha_T = [\alpha_T]_{01} \oplus [\alpha_T]_{02} \oplus [\alpha_T]_{12}$. To shuffle T randomly, a random permutation π is sampled. To prevent leakage of π, $\pi(\beta_T)$ is masked with randomness R to obtain β_{T_o}. α_{T_o} is then defined as $\pi(\alpha_T) \oplus R$ to ensure $T_o = \beta_{T_o} \oplus \alpha_{T_o}$. The goal is to generate $[\cdot]$-shares of α_{T_o} and ensure all parties hold β_{T_o}. To do this, $[\pi(\alpha_T)]$ is first generated, followed by the steps to generate β_{T_o} and then [R]. Finally, α_{T_o} is defined as $\pi(\alpha_T) \oplus R$.

4.1 Generation of $[\pi(\alpha_T)]$:

- α_T is generated during a preprocessing phase.
- Protocol of [6] is used to generate $[\cdot]$-shares of $\alpha = \pi(\alpha_T)$, where π is a random secret permutation.
- The protocol of [6] employs the semi-honest 3PC shuffle protocol from [35] to shuffle the input table using three invocations of Shuffle-Pair protocol.
- Each invocation of Shuffle-Pair is followed by a Set-Equality protocol to verify the correctness of the semi-honest shuffle.
- The output of the shuffle protocol is guaranteed to be correct if all instances of Shuffle-Pair are verified to be correct.
- Let π_{12}, π_{01}, π_{02} denote the three permutations used in the three Shuffle-Pair instances, where π_{ij} is held by $P_i, P_j \in \mathcal{P}$.
- The protocol of [6] outputs $[\pi(\alpha_T)]$ and a flag indicating correctness of the output.

- $\pi = \pi_{12} \circ \pi_{01} \circ \pi_{02}$ where $\circ$ denotes composition operation.

4.2 Generation of $[\beta_{T_o} = \pi(\beta_T) \oplus R]$ where $\pi = \pi_{12} \circ \pi_{01} \circ \pi_{02}$:

- Observations : The table to be shuffled is held by all three parties in clear, while the permutation π is still private. Each party misses exactly one permutation that is held by the other two parties.
- Paper uses these observations to design an efficient shuffle protocol for the online phase.

4.3 Generating β_{T_o} towards P1 :

- Recall that P1 misses π_{02}.
- To prevent leakage of permutation π_{02}, we provide P1 with $\pi_{02}(\beta_T) \oplus R$, where R masks π_{02}.
- π_{02} is held by P0 and P2, who sample a random $R_{02} \in \mathbb{Z}_{2^l}^N$, and compute and send $\pi_{02}(\beta_T) \oplus R_02$ to P1. $\pi_{02}(R_{02})$ serves as the random mask R.
- P1 computes β_{T_o} using the received value and the knowledge of permutations $\pi_{12}, \pi 01$, and $\pi 02(R_{02})$ serves as a mask to hide π from P1.
- Similar masks are required in β_{T_o} to keep π hidden from P2 and P0. This results in introducing random masks $\pi_{12}(\pi_{01}(R_{01}))$ and $\pi_{12}(R_{12})$.
- β_{T_o} is defined as

$$\beta_{T_o} = \pi_{12}(\pi_{01}(\pi_{02}(\beta_T \oplus R_{02}) \oplus R_{01}) \oplus R_{12}) = \pi(\beta_T) \oplus \pi(R_{02}) \oplus \pi_{12}(\pi_{01}(R_{01})) \oplus \pi_{12}(R_{12}) \quad (1)$$

$R_{12}, R_{01} \in \mathbb{Z}_{2^l}^N$, and R_{ij} is jointly sampled by $P_i, P_j \in \mathcal{P}$. P1 can compute β_{T_o} using above Eq. since it holds $R_{12}, R_{01}, \pi_{12}, \pi_{01}, and \pi_{02}(\beta_T \oplus R_{02})$.

4.4 Generating β_{T_o} towards P2 :

- P0 computes $\delta_{12} = \pi_{01}(\pi_{02}\beta_T \oplus R_{02}) \oplus R_{01})$ and sends it to P2
- P1 computes and sends the hash of $\pi_{01}(\pi_{02}\beta_T \oplus R_{02}) \oplus R_{01})$ using the value $\delta_{02} = \pi_{02}(\beta_T \oplus R_{02})$ received in the first round.
- P2 verifies the correctness of the received values and then uses Eq. (1) to compute β_{T_o}.

4.5 Generating β_{T_o} towards P0 :

- To generate β_{T_o} towards P0; P1 and P2 can send it to P0 (one sends the value, the other sends the hash).
- $R_{12} is used as a mask while computing \beta_{T_o}$ to prevent leakage of π_{12} to P0.
- P2 receives δ_{12} required for computing β_{T_o} in the first round itself, but it can compute the correct β_{T_o} only after the second round.
- Communication of β_{T_o} from P1, P2 towards P0 can happen in the second round.

A pictorial view of the messages exchanged is given in Fig. 1

4.6 Generation of $[\alpha_{T_o}] = [\pi(\alpha_T)] \oplus [R]$:

- Generate $[\pi_{12}(R_{12})]$ non-interactively.
- Compute $[\pi_{02}(\alpha_T \oplus R_{02})]$ non-interactively by invoking Shuffle-Pair with π_{02} on $[\alpha_T \oplus R_{02})]$.
- Compute $[\pi_{02}(R_{01})]$ non-interactively by invoking Shuffle-Pair with π_{02} on $[R_{01}]$.
- Compute $[\pi_{02}(\alpha_T \oplus R_{02}) \oplus R_{01}]$ by XORing $[\pi_{02}\ (\alpha_T \oplus R_{02})]$ and $[R_{01}]$.

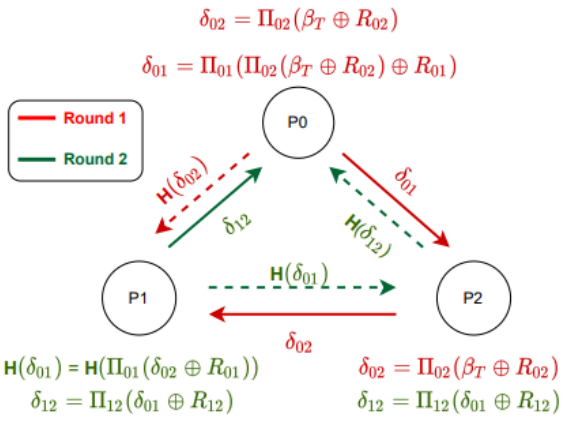


Figure 1: Online phase of Ruffle

- Compute $[\pi_{12}(\pi_{01}(\pi_{02}(\alpha_T \oplus R_{02})) \oplus R_{01}))]$ by invoking Shuffle-Pair with π_{01}, π_{12} on $[\pi_{02}(\alpha_T \oplus R_{02})) \oplus R_{01})]$.
- Compute $\gamma byXORing[\pi_{12}(\pi_{01}(\pi_{02}(\alpha_T \oplus R_{02})) \oplus R_{01}))]$ and $[\pi_{12}(\pi_{01}(\pi_{02}(R_{02})))] \oplus [\pi_{12}(\pi_{01}(R_{01}))]$. - Compute $[\alpha_{T_o}]$ by XORing $[\pi(\alpha_T)]$ and $[\gamma]$.

$[\cdot]$ denotes a secret shared value.
π_{ij} denotes the secret permutation known by parties P_i and P_j.
Shuffle-Pair([T], π_{ij}) denotes the application of π_{ij} on [T] to obtain a secret share $[\rho]$, where $\rho = \pi_{ij}$(T).
Set-Equality([T1], [T2]) outputs flag = 0 if Shuffle-Pair([T1], π_{ij}) = $[\rho]$ and Shuffle-Pair($[\rho]$, π_{ij}^{-1}) = [T2], and flag = 1 otherwise.

4.7 Guarantees for Output Delivery

The current solution for random shuffle in a distributed setting can fail due to adversarial behavior. A trusted third party (TTP) based approach is proposed to attain GOD (Guarantee of Delivery) and ensure output delivery. The TTP approach involves robust reconstruction of the input table and shuffle operation on the clear table.
Identifying TTP during Preprocessing Phase :

- Preprocessing phase involves sequential invocations of Shuffle-Pair protocol followed by a Set-Equality protocol to verify correctness.
- If any invocation of Set-Equality indicates that Shuffle-Pair fails, it must be due to misbehavior of one of the two communicating parties.
- Since at most one party is malicious, the non-communicating residual party is honest and can be designated as TTP.

Identifying TTP during Online Phase :

- Each of the three messages exchanged in the online phase involve two senders and a receiver.
- If the received message and hash do not match at the receiver, it broadcasts a complaint accusing the senders and sends the received messages.
- The senders then broadcast a complaint against the receiver if the latter's broadcast message is inconsistent with the senders' sent message.

- Based on the publicly available complaints, parties can unanimously determine a pair of conflicting parties, one of which is corrupt.
- Due to at most one malicious corruption, the third party not involved in the conflict is honest and can be designated as TTP.

Conclusion :

- The TTP approach provides GOD and ensures output delivery even in the presence of adversarial behavior.
- The approach involves identifying an honest party as TTP during preprocessing and online phases.

5 Protocol Ruffle

Considering that you have read all the preliminaries and generations in section 4 properly, I think it would be quite easy to now understand the overall protocol.

Protocol 5.1. $\Pi_{Ruffle}([[(T)]])$
Preprocessing :
– Each pair of parties $P_i, P_j \in \mathcal{P}$ non-interactively sample $R_{ij} \in \mathcal{Z}_{2^l}^{(N)}$ and random permutations π_{ij}.
– P_i, P_j compute $\pi_{12}(R_{12})$, and parties generate its $[\cdot]$-shares, noninteractively.
– Parties in $\mathcal{P}$ generate $[\cdot]$-shares of R_{01}, R_{02}, non-interactively.
– Parties in $\mathcal{P}$ follow the steps in Fig. 2 to generate $[\alpha_{T_o}]$. (by the way we have already discussed this earlier in detail under 4.6)
– Identifying TTP when shuffle fails: If $flag_{ij}$ indicates a failure, all parties set TTP to be the non-communicating party in the corresponding Shuffle-Pair protocol. When multiple $flag_{ij}$ indicates failure, break tie deterministically and use one $flag_{ij}$.
Online :

- Shuffle (Round 1) :
  - P0, P2 compute $\delta_{02} = \pi_{02}(\beta_T \oplus R_{02})$. P2 sends δ_{02} to P1. P0 sends $H(\delta_{02})$ to P1, where H is a collision-resistant hash function.
  - P0 computes and sends $\delta_{01} = \pi_{01}(\pi_{02}(\beta_T \oplus R_{02}) \oplus R_{01})$ to P2
- Shuffle (Round 2) :
  - P1 computes and sends $H(\delta_{01}) = H(\pi_{01}(\delta_{02} \oplus R_{01}))$ to P2.
  - P1, P2 compute $\delta_{12} = \pi_{12}(\delta_{01} \oplus R_{12})$.
  - P1 sends δ_{12} and P2 sends H(δ_{12}) to P0.
- Verification (Round 3) :
  For each receiver $P_i \in \mathcal{P}$, let P_j, P_k denote the senders. Let P_j send the message and P_k send its hash. P_i checks if the received values are consistent. If not, it broadcasts ("accuse", P_j, P_k, c_j, c_k), where $c_j = H(x)$, such that x and c_k are the values sent by P_j and P_k, respectively
- Verification and TTP Identification (Round 4) :
  Consider the first instance when a party P_i broadcasts ("accuse", P_j, P_k, c_j, c_k).
  - If $c_j = c_k$, set TTP = P_j .
  - Else if c_j is different from the hash of the value sent by P_j to P_i, then P_j broadcasts ("accuse", P_i). Set TTP = P_k . The above steps follow analogously for P_k . - Else if $c_j \neq c_k$ and neither P_j nor P_k accuses P_i , set TTP = P_i .
- One-time computation through TTP :
  If TTP is set, all parties robustly reconstruct the input table towards the TTP, who randomly shuffles the input and sends the shuffled table to all parties.

6 Applications

The paper discusses two applications of shuffle - Anonymous broadcast and the GraphSC paradigm. Anonymous broadcast requires a shuffle protocol that handles independent shuffles, while the GraphSC paradigm requires a shuffle protocol that caters to composed shuffles. Paper gives Ruffle-1, which is an extension of Ruffle and can handle independent shuffles, but it is not efficient for composed shuffles due to sequential dependence in the preprocessing phase. Hence, it further introduces Ruffle-2, a shuffle protocol that is tailor-made (as written in the paper by the IISC authors) to handle composed shuffles.

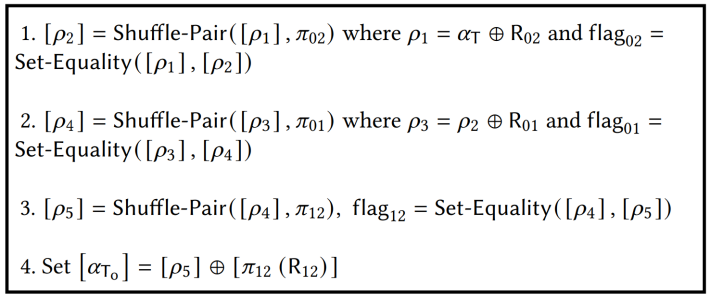


Figure 2: Generation of $[\alpha_{T_o}]$ by parties in $\mathcal{P}$

6.1 Anonymous Broadcast

Anonymous broadcast is a system where N clients securely shuffle their input messages. This system requires multiple sequential invocations of shuffle, which is called Independent-Shuffles scenario. In this scenario, multiple independent shuffles are required with the constraint that they are invoked sequentially.

Ruffle, which is designed to handle a single shuffle invocation, is extended to handle Independent-Shuffles and is called Ruffle-1. Ruffle-1 leverages the independence of the M shuffles in Independent-Shuffles to perform the necessary preprocessing steps in parallel.

The paper claims that using Ruffle-1 allows for a more efficient shuffle-based anonymous broadcast system. Additionally, the new system guarantees censorship resistance, which ensures that a malicious server cannot discard an honest client's message by claiming it to be malformed.

Mathematically, the scenario involves M ordered sets T1, T2, ..., Tm that are required to be shuffled under random secret permutations, π_1, π_2, ..., π_m, respectively. The shuffles are performed sequentially, such that $\pi_{i+1}(T_{i+1})$ is invoked after $\pi_i(T_i)$, and T_{i+1} is independent of $\pi_i(T_i)$. The protocol Ruffle-1 is designed to handle this scenario efficiently by performing necessary preprocessing steps in parallel.

6.2 GraphSC Paradigm

The GraphSC paradigm provides an efficient and scalable solution for securely evaluating graph algorithms. By using Ruffle-2, improvements can be achieved in this paradigm. The shuffle protocols developed in this paper also allow for the reusing and inverting of the underlying secret permutation, which is a crucial property required in applications such as the GraphSC paradigm.

The Composed-Shuffles scenario requires a composition of m shuffles, where the output of one shuffle is fed as the input to the next shuffle. This generates a sequence of intermediate shuffled sets. In this scenario, Ruffle-1 is not suitable as it leverages the independence of shuffles to facilitate parallel preprocessing, which is not possible in the case of Composed-Shuffles. Therefore, a new protocol called Ruffle-2 is designed to specifically cater to this scenario by breaking the sequential dependence on shuffles in the preprocessing phase. This enables parallel preprocessing for the m shuffles.

In the Composed-Shuffles scenario, the shuffles are dependent, and hence the design of Ruffle-2 for breaking the dependency in the preprocessing comes at the cost of slightly increased preprocessing communication compared to the preprocessing of Ruffle-1. However, Ruffle-2 can still be used in the Independent-Shuffles scenario, but it may not be as efficient as Ruffle-1 for this scenario.

Mathematically, in the Composed-Shuffles scenario, the composition of m shuffles generates a sequence of

intermediate shuffled sets, where the ith ordered set is denoted as $T_i = \pi_i(...\pi_1(T))$, and $T_m = \pi_m(\pi_{m1}(...\pi_1(T)))$. The shuffles are dependent in this scenario, and hence Ruffle-1 is not suitable. Instead, Ruffle-2 is designed to break the sequential dependence on shuffles in the preprocessing phase. The shuffle protocols developed in this paper enable the reusing and inverting of the underlying secret permutation, which is an essential property in applications such as the GraphSC paradigm.

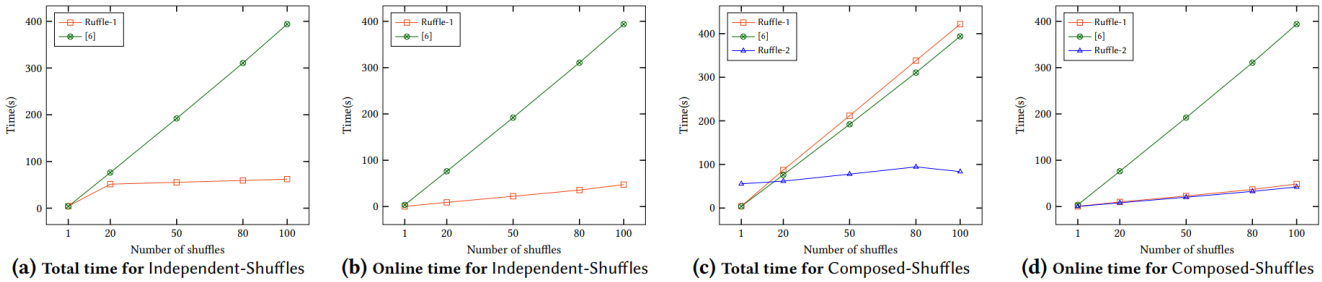
7 Benchmarks

The authors empirically evaluate the performance of their shuffle protocols under various parameters and application scenarios and compare them against state-of-the-art counterparts. The benchmarking is done over a LAN using n1-standard instances of Google Cloud with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors and 240 GB of RAM, with a bandwidth of 16Gbps. The protocols are implemented in Python, including those of [20] and [6], and account for multi-threading with 64 threads. The communication layer between parties is instantiated using the PyTorch library, and the Crypto library is used for AES and hashlib for generating SHA256 hash.

We make the following observations from the results displayed below :
1. The cost of [6] is the same for Independent-Shuffles and Composed-Shuffles.
2. Ruffle-1 (and Ruffle-2) outperforms [6] by up to 10× in terms of online complexity.
3. For a single shuffle invocation, Ruffle-1 and Ruffle-2 have a slightly higher run time than [6], but Ruffle-1 begins to outperform [6] starting from two invocations for Independent-Shuffles due to its faster online phase, while [6] continues to outperform Ruffle-1 for Composed-Shuffles due to the sequential nature induced by the composition of shuffles in Ruffle-1's preprocessing phase.
4. To capture the effect of both total run time and total communication, the monetary cost in Fig. 5 is reported, which is the price paid for performing the secure shuffle computation.

|T|	Protocol	Time (s)	Comm. (MB)
10^3	Ruffle	0.062	0.323
	[6]	0.056	0.258
	[20]	0.079	0.427
10^4	Ruffle	0.504	3.232
	[6]	0.434	2.318
	[20]	0.794	4.272
10^5	Ruffle	4.211	32.074
	[6]	3.959	22.919
	[20]	8.012	42.724
10^6	Ruffle	55.559	320.465
	[6]	49.577	228.912
	[20]	98.576	427.246

Figure 3: Total complexity of shuffle for varying table sizes for single shuffle invocation.



(a) Total time for Independent-Shuffles **(b) Online time for** Independent-Shuffles **(c) Total time for** Composed-Shuffles **(d) Online time for** Composed-Shuffles

Figure 4: Comparison of Ruffle-1, Ruffle-2, [6] in terms of online and total time for scenario of Independent-Shuffles and Composed-Shuffles for varying number of shuffle invocations and table size of 105.

Number of shuffles	Protocol	Online			Total		
		Time(s)	Comm.(MB)	TP (per min)	Time(s)	Comm.(MB)	Monetary cost (USD)
1	Ruffle-1(Independent-Shuffles)	0.38	9.16	120.00	4.21	32.06	0.020
	Ruffle-1(Composed-Shuffles)	0.38	9.16	120.00	4.21	32.06	0.020
	Ruffle-2(Composed-Shuffles)	0.46	9.16	120.00	4.35	41.21	0.022
	[6]	3.81	22.91	59.93	3.81	22.91	0.016
2	Ruffle-1(Independent-Shuffles)	0.92	18.31	60.00	5.16	64.13	0.030
	Ruffle-1(Composed-Shuffles)	0.92	18.31	60.00	10.21	64.13	0.044
	Ruffle-2(Composed-Shuffles)	0.98	18.31	60.00	5.46	82.44	0.035
	[6]	7.58	45.81	29.95	7.62	45.81	0.032
25	Ruffle-1(Independent-Shuffles)	10.30	228.88	4.80	53.72	801.56	0.349
	Ruffle-1(Composed-Shuffles)	10.30	228.88	4.80	105.25	801.56	0.491
	Ruffle-2(Composed-Shuffles)	11.18	228.88	4.80	61.68	1030.44	0.429
	[6]	95.24	572.68	2.39	95.74	572.68	0.408
50	Ruffle-1(Independent-Shuffles)	20.50	457.76	2.40	55.31	1603.33	0.556
	Ruffle-1(Composed-Shuffles)	20.50	457.76	2.40	211.82	1603.33	0.986
	Ruffle-2(Composed-Shuffles)	20.53	457.76	2.40	77.76	2060.74	0.733
	[6]	192.06	1145.55	1.20	194.29	1145.55	0.817
100	Ruffle-1(Independent-Shuffles)	40.18	960.01	1.20	62.09	3206.66	0.978
	Ruffle-1(Composed-Shuffles)	40.18	960.01	1.20	421.76	3206.66	1.968
	Ruffle-2(Composed-Shuffles)	42.29	960.01	1.20	83.60	3206.66	1.037
	[6]	393.82	2402.40	0.59	395.91	2291.11	1.660

Figure 5: Comparison of Ruffle-1, Ruffle-2, [6] with respect to the scenario of Independent-Shuffles and Composed-Shuffles for varying number of shuffle invocations and table size of 105. Note that the cost of [6] remains the same for both the scenarios.

8 Future Scope and Extensions on given Protocols

The paper showcase that the customized protocols not only provide a fast response time, but also provide improved overall run time for multiple shuffle invocations. However, these are confined to 3PC setting. The task of extending these protocols to the arbitrary N-party setting is a quite a bit challenging, as it would lead to exponential blow-up in the number of permutations held at each party, which would affect communication and round complexity adversely. Hence, future research should focus on designing efficient solutions for the n-party setting. Overall, the study contributes to the development of more efficient and secure shuffle protocols, which could have practical applications in various domains.

9 Conclusion

In this study, the authors have presented new secure shuffle protocols that are not only fast but also improve the overall runtime of sequential shuffle invocations. The study demonstrates the significant improvement achieved in Anonymous broadcast and Secure BFS computation via the GraphSC paradigm. The authors suggest that these protocols can be used to improve various other applications that use secure shuffle. It would be interesting to see how our shuffle protocols can be used to improve therein since secure shuffle is a crucial component of many applications. Going forward, I think it's crucial to design secure shuffle protocols for the arbitrary n-party setting because applications might require a different number of parties. But as mentioned earlier, the paper lacks here being limited to 3PC setting. However there are future scopes for the same for improvements.

10 Proof for Security of Protocol

The *LemmaF*.1 states that in the shuffle protocol, whenever a TTP is identified, it is always a honest party. The proof explains that during the preprocessing phase, the TTP is identified when a Shuffle-Pair fails due to an incorrect message sent by one of the parties. Since only one of the parties can be malicious, the identified TTP is honest. During the online phase, if a receiver broadcasts an accusation message, the TTP identified in each corruption scenario will be an honest party.

$\mathcal{PROOF}$:
The proof involves showing that the real-world execution of the Ruffle shuffle protocol and the ideal-world execution are indistinguishable.

To do this, we consider two adversaries: $\mathcal{A}$, who operates in the real-world, and $\mathcal{S}_{\Pi_{Ruffle}}$, who operates in the ideal-world. $\mathcal{S}_{\Pi_{Ruffle}}$ starts by emulating the $\mathcal{F}_{setup}$ phase, which establishes common keys between $\mathcal{A}$ and $\mathcal{S}_{\Pi_{Ruffle}}$. This allows $\mathcal{S}_{\Pi_{Ruffle}}$ to know all the randomness used by $\mathcal{A}$ and extract $\mathcal{A}$'s input (specifically, $[\alpha_T]$ and β_T). $\mathcal{S}_{\Pi_{Ruffle}}$ can also verify the correctness of messages sent by $\mathcal{A}$. Then, $\mathcal{S}_{\Pi_{Ruffle}}$ simulates the steps of the shuffle protocol, with specific simulation steps for a corrupt player P0. $\mathcal{S}_{\Pi_{Ruffle}}$ can simulate the corruption of other players (P1, P2) in an analogous manner.

In the real-world, during the preprocessing phase, $\mathcal{A}$ receives masked messages computed as part of Shuffle-Pair and Set-Equality, where the messages are masked using some randomness to hide the missing permutation and information regarding the missing share at $\mathcal{A}$. During the online phase, $\mathcal{A}$ receives $\delta_{12}, H(\delta_{12})$ where δ_{12} is a randomized using the random mask R_{12}.

In the ideal world, $\mathcal{A}$ receives messages that are sampled randomly from the uniform distribution, and SRuffle can verify the correctness of the messages sent by $\mathcal{A}$. $\mathcal{S}_{\Pi_{Ruffle}}$ can also simulate all the accuse messages as done in the real-world. Thus, real-world and ideal-world executions are indistinguishable.

11 References

[1] https://en.wikipedia.org/wiki/Secure_multi-party_computation
[2] https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing
[3] https://en.wikipedia.org/wiki/Secure_two-party_computation
[4] https://inpher.io/technology/what-is-secure-multiparty-computation/
[5] https://www.youtube.com/watch?v=YvDmGiNzV5E
[6] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin,
[20] Saba Eskandarian and Dan Boneh. 2022. Clarion: Anonymous Communication
[29] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Superfast and Robust Privacy-Preserving Machine Learning. In USENIX Security.
[35] Sven Laur, Jan Willemson, and Bingsheng Zhang. 2011. Round-efficient oblivious database manipulation. In ISC.

The specific numbers (6, 20, 29, 35) are intentionally included as these were used in the paper itself and are important to be included.